Autenticación y autorización, sistema distribuido con gRPC

Alumnos:

- Felipe Dioguardi - 16211/4
- Leonardo Germán Loza Bonora - 16181/7

Índice

Introducción

En la web, constantemente ocurren procesos de autenticación y autorización. Pero… ¿De qué tratan estos procesos? ¿Cómo se podrían implementar?.

La autenticación es el acto de probar la identidad de un usuario en un sistema informático, mientras que la autorización es la función de especificar los derechos o privilegios de acceso a los recursos. Existen diversos métodos y estándares para la implementación de estos procesos.

En este trabajo se desarrollará una aplicación distribuida en dónde se generarán dos servicios diferentes para autenticar y autorizar a un cliente empleando el estándar JWT, utilizando las tecnologías de gRPC y Python.

Problema planteado

Generar dos servicios, y un cliente para que interactúe con los dos primeros. En la Figura 1, se muestra el flujo de comunicaciones que se deben realizar.

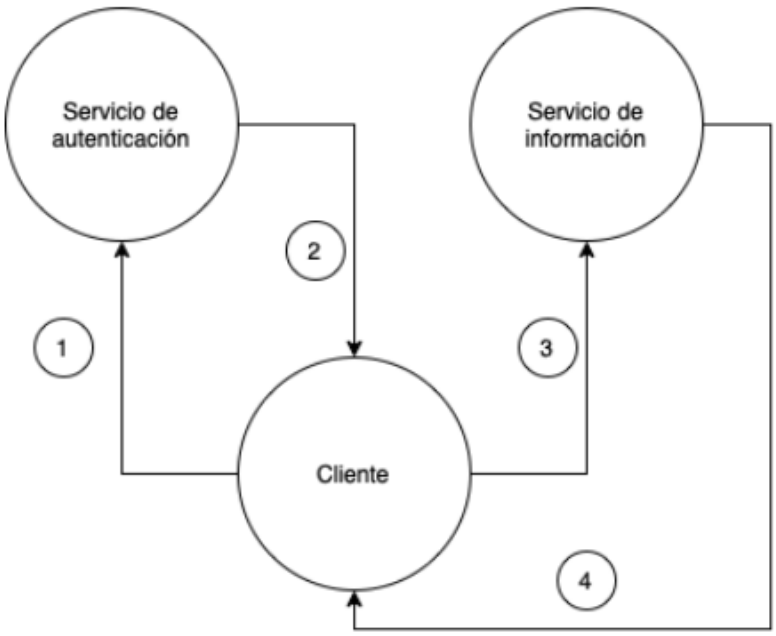

Figura 1. Flujo de comunicación entre servidores y el cliente.

Los pasos de comunicación son los siguientes:

1. El cliente pide al *servicio de autenticación* un JWT con su email/password.
2. El *servicio de autenticación* retorna un token válido al cliente. En caso de que la combinación sea errónea, retorna el error.
3. Con el JWT, *el cliente* pide información al *servicio de información* para que le retorne datos personales del usuario. En caso de que el token no sea válido, debe retornar error.

Procesos a implementar:

1. Un servidor que provea de tókenes (JWT).
2. Un servidor que provea información sensible al usuario. Por ejemplo, datos personales.
3. Un cliente que consuma el servicio para obtener el token y consulte, con dicho token, su información personal.

gRPC

gRPC es un framework moderno de llamadas a procedimientos remotos (RPC) de código abierto y alto rendimiento creado por Google que puede ejecutarse en cualquier entorno [10]. Puede conectar servicios eficientemente en y a través de los centros de datos con soporte para el balanceo de carga, rastreo, chequeo de correctitud y autenticación.

Se utiliza principalmente para:

- Conectar eficazmente servicios políglotas en una arquitectura de estilo de microservicios.
- Conectar dispositivos móviles, clientes de navegador a servicios de backend.

- Generación de bibliotecas de clientes eficientes.

Sus características principales son:

- Librerías de cliente en 11 lenguajes.
- Alta eficiencia en la arquitectura y con un sencillo framework de definición de servicios.
- Transmisión bidireccional con transporte basado en HTTP/2.
- Autenticación, rastreo, balanceo de carga y chequeo de correctitud.

En gRPC, un cliente puede llamar directamente a un método de un servidor en una máquina diferente como si fuera un objeto local, lo que facilita la creación de aplicaciones y servicios distribuidos. Como en muchos sistemas RPC, gRPC se basa en la idea de definir un servicio, especificando los métodos que pueden ser llamados remotamente con sus parámetros y tipos de retorno. En el lado del servidor, este implementa la interfaz y levanta un servicio gRPC para manejar las llamadas del cliente. En el lado del cliente, él posee un stub que proporciona los mismos métodos que el servidor [11].

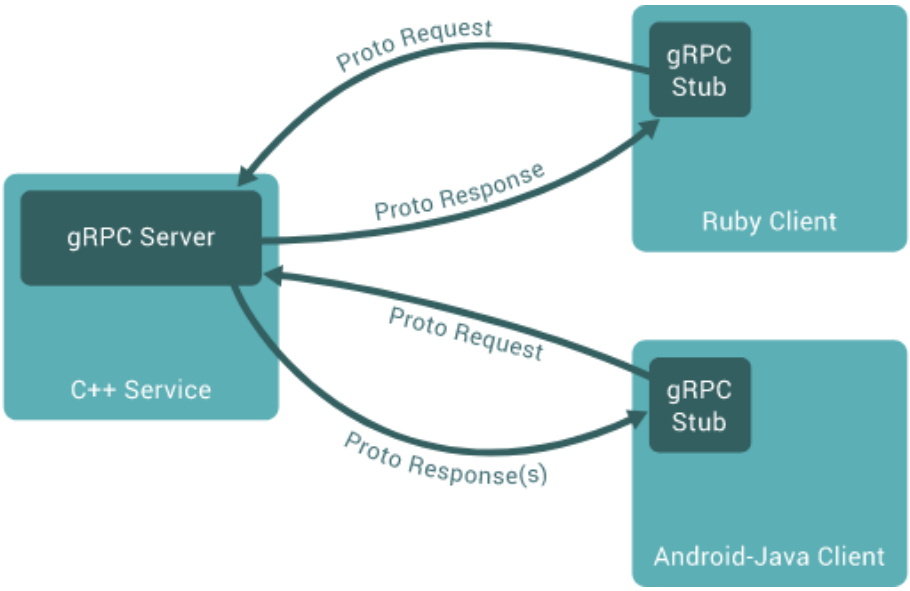

Figura 2. Flujo común de una aplicación gRPC.

Por defecto, gRPC utiliza `Protocol Buffers`, que son el mecanismo libre de Google para serializar datos de manera estructurada. Estos se definen en un archivo de extensión `.proto` donde se especifican en forma de “mensajes”, que son pequeños registros lógicos que contienen una serie de pares nombre-valor llamados “campos”. En estos archivos también se definen los servicios gRPC junto a sus métodos RPC, cuyos parámetros y tipos de retorno se especifican como “mensajes”. Un ejemplo:

```
// Definición del servicio Saludador.
service Saludador {
  // Envía un saludo
  rpc Saludar (SaludoSolicitud) returns (SaludoRespuesta) {}
}
```

```
// Mensaje de solicitud que contiene el nombre del usuario
message SaludoSolicitud {
  string nombre = 1;
}

// Mensaje de respuesta que contiene los saludos
message SaludoRespuesta {
  string mensaje = 1;
}
```

Una vez definido el archivo `.proto`, se puede emplear el compilador `protoc` para generar clases de accesos a datos en el lenguaje que se quiera.

gRPC permite definir cuatro tipos de métodos en los servicios [12]:

- **Unary RPCs** en los que el cliente envía una única petición al servidor y obtiene una única respuesta de vuelta, al igual que una llamada a una función normal.

```
rpc Saludar(SaludoSolicitud) returns (SaludoRespuesta);
```

- **Server streaming RPCs** en los que el cliente envía una petición al servidor y obtiene un flujo para leer una secuencia de mensajes de vuelta. El cliente lee del flujo devuelto hasta que no haya más mensajes. gRPC garantiza el orden de los mensajes dentro de una llamada RPC individual.

```
rpc MuchasRespuestas(SaludoSolicitud) returns (stream SaludoRespuesta);
```

- **Client streaming RPCs** en los que el cliente escribe una secuencia de mensajes y los envía al servidor, utilizando también un flujo proporcionado. Una vez que el cliente termina de escribir los mensajes, espera a que el servidor los lea y devuelva su respuesta. De nuevo, gRPC garantiza el orden de los mensajes dentro de una llamada RPC individual.

```
rpc MuchosSaludos(stream SaludoSolicitud) returns (SaludoRespuesta);
```

- **Bidirectional streaming RPCs** en los que ambas partes envían una secuencia de mensajes mediante un flujo de lectura y escritura. Los dos flujos funcionan de manera independiente, por lo que los clientes y los servidores pueden leer y escribir en el orden que deseen: por ejemplo, el servidor podría esperar a recibir todos los mensajes del cliente antes de escribir sus respuestas, o podría leer alternativamente un mensaje y luego escribir un mensaje, o alguna otra combinación de lecturas y escrituras. El orden de los mensajes en cada flujo se mantiene.

```
rpc SaludoBidireccional(stream SaludoSolicitud) returns (stream SaludoRespuesta);
```

JWT

JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autocontenida de transmitir información de forma segura entre las partes como un objeto JSON [13]. Esta información es confiable porque está firmada digitalmente, por lo que puede ser verificada. Los JWT pueden ser firmados usando un “secreto” (con el algoritmo HMAC) o un par de claves públicas/privadas usando RSA o ECDSA.

Los tokens firmados sirven para validar la integridad de las afirmaciones contenidas en ellos, mientras que los tokens cifrados permiten ocultar esas afirmaciones a otras partes. Cuando los tókenes se firman utilizando pares de claves públicas/privadas, la firma también certifica que sólo la parte que posee la clave privada es la que lo ha firmado.

JWT se usa para autenticar y autorizar solicitudes. Proporciona una forma conveniente de identificar a un usuario entre diversos sistemas y permite que el usuario acceda a recursos restringidos sin tener que volver a ingresar sus credenciales.

JWT está compuesto por tres partes:

- Header: contiene la información del algoritmo de cifrado y el tipo de token (JWT).

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

- Payload: contiene los datos codificados. Hay tres tipos de “afirmaciones” (claims) que puede tener:
    - Registered claims: aquellas que no son mandatorias pero sí recomendadas, tales como: **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), etc.
    - Public claims: definidas a voluntad. Para evitar colisiones deben estar definidas en el *IANA JSON Web Token Registry* o estar definidos en una URI que cotenga un espacio de nombre que evite superposiciones.
    - Private claims: son personalizadas. Creadas para compartir información entre partes que aceptan utilizarlas y que no son ni “registered” ni “public”

```
{
  "sub": "1",
  "name": "Juan Carlos Bodoque",
  "admin": true
}
```

- Signature: contiene la firma digital que sirve para corroborar que el token no haya sido modificado. Se crea concatenando el header codificado, un punto, el payload codificado, un secreto, y a todo eso se lo encripta con el algoritmo definido en el header.

Desarrollo

Para simular la interacción propuesta, desarrollamos dos servidores y un cliente gRPC en Python.

Para implementar el protocolo utilizamos las librerías `grpcio` y `grpcio-tools`, ambas indicadas en la documentación oficial de gRPC [1]. Para la generación y decodificación rápida y segura de los tókenes JWT nos valemos de la librería `PyJWT` [2], que provee una interfaz sencilla al estándar definido en [3]. Si bien no era la única opción disponible [4, 5, 6], `PyJWT` ofrece la funcionalidad requerida sin el agregado del resto de tecnologías JOSE [7].

A su vez, elegimos MongoDB [8] como base de datos no relacional para el almacenamiento de la información del usuario. Para facilitar su manejo desde Python, hacemos uso de la librería `mongoengine` [9].

Aplicación

El código se encuentra disponible de forma pública en el siguiente repositorio de GitHub:

https://github.com/fdioguardi/UNLP_gRPC_auth

La aplicación desarrollada se estructura de la siguiente forma:

```
.
├── main.py
└── src
    ├── client
    │   └── start_client.py
    ├── server
    │   ├── auth.py
    │   ├── db.py
    │   ├── info.py
    │   └── start_servers.py
    └── service
        ├── __init__.py
        ├── auth_pb2_grpc.py
        ├── auth_pb2.py
        ├── info_pb2_grpc.py
        ├── info_pb2.py
        └── protos
            ├── auth.proto
            └── info.proto
```

En la raíz del proyecto se encuentra el archivo `main.py`, que sirve como un punto de acceso centralizado para iniciar el cliente y los servidores.

Dentro del directorio `src` se encuentran los 3 sectores centrales de la aplicación:

- El código del cliente, que se encarga de comunicarse con los servidores.
    1. Primero, prepara 2 usuarios para consultar a la base de datos, uno correcto, y otro erroneo.

```
USERS = [
    User(username="user1", password="password1"),
    User(username="user2", password="password2"),
]
```

2. Luego, pone en marcha la conexión con los servidores.

```
# Create a channel to the server
auth_channel = grpc.insecure_channel("localhost:5000")

# Connect to the authentication server
auth_stub = auth_grpc.AuthenticatorStub(auth_channel)

# Connect to the information server
info_channel = grpc.insecure_channel("localhost:5001")

# Connect to the information server
info_stub = info_grpc.InformantStub(info_channel)
```

3. Después, para cada usuario, se intenta conectar al servidor de autenticación para obtener el token de acceso. En el caso del segundo usuario, con `username = "user2"` y `password = "password2"`, el servidor no autorizará sus credenciales y devolverá un mensaje de error. Para el primer usuario, el cliente obtendrá el JWT de acceso.

```
auth_request = auth_pb2.Credentials(
    user.username,
    hashed_password=utils.hash_passwd(user.password)
)

try:
    auth_response = auth_stub.authenticate(auth_request)
    print("Authentication successful")
except grpc.RpcError as e:
    print(e.details())
    print("Authentication failed")
    continue
```

4. Por último, el cliente enviará el JWT al servidor de datos, que lo validará y responderá con la información solicitada.

```
info_request = info_pb2.TokenRequest(token=auth_response.token)

try:
    info_response = info_stub.getInfo(info_request)
    print("User information:")
    print(f"\tUsername: {info_response.username}")
    print(f"\tName: {info_response.name}")
    print(f"\tSurname: {info_response.surname}")
    print(f"\tEmail: {info_response.email}")
except grpc.RpcError as e:
    print(e.details())
    print("Failed to get user information")
    continue
```

5. Y como prueba final, el cliente se intentará conectar al servidor de datos enviando un token inválido, y recibirá un error.

```
info_request = info_pb2.TokenRequest(token="invalid_token")
try:
    info_response = info_stub.getInfo(request)
    print("User information:")
    print(f"\tUsername: {info_response.username}")
    print(f"\tName: {info_response.name}")
    print(f"\tSurname: {info_response.surname}")
    print(f"\tEmail: {info_response.email}")
except grpc.RpcError as e:
    print(e.details())
    print("Failed to get user information")
```

Al finalizar la ejecución del código del cliente, se verá el siguiente output, correspondiente a la ejecución correcta del usuario 1:

```
-----------------------------------------------------
Authentication successful
User information:
  Username: user1
  Name: Roberto
  Surname: Carlos
  Email: rc@email.com
-----------------------------------------------------
```

Seguido de este mensaje de error producido por el servidor de autenticación, al no validar la existencia del usuario 2 en la base de datos:

```
-----------------------------------------------------
Invalid username or password
Authentication failed
-----------------------------------------------------
```

Y terminando con el mensaje de error correspondiente al intento de acceso al servidor de información sin otorgar un JWT válido:

```
-----------------------------------------------------
Invalid token
Failed to get user information
-----------------------------------------------------
```

- Los archivos `.proto` dentro de `src/service/protos/`, que definen el esqueleto de los servicios gRPC; y los archivos `.py` correspondientes en `src/service/`.

- El archivo `auth.proto` contiene la especificación del Protocol Buffer correspondiente al servidor de autenticación. Define un servicio `Authenticator` que sabe responder al método `authenticate`, de tipo *Unary gRPC*. Elegimos este tipo de método porque no es necesario mantener un flujo continuo de datos, y de esta forma se facilita la comunicación.

```
// The authentication service definition.
service Authenticator {
  // Authenticates the user with the given credentials.
  rpc authenticate (Credentials) returns (TokenResponse) {}
}
```

  Además, estructura los mensajes `Credentials` y `TokenResponse`, que empaquetan respectivamente la `request` y la `response` de `authenticate`.

```
// The request message containig the username and password to authenticate.
message Credentials {
  string username = 1;
  string password = 2;
}

// The response message containing the authentication token.
message TokenResponse {
  string token = 1;
}
```

- Análogamente, el archivo `info.proto` especifica el Protocol Buffer correspondiente al servidor de información. Éste define un servicio `Informant` que sabe responder al método `getInfo`.

```
// The information service definition.
service Informant {
  // Returns the information of the user.
  rpc getInfo (TokenRequest) returns (Information) {}
}
```

  Y, estructura los mensajes `TokenRequest` e `Information`, que representan respectivamente la `request` y la `response` de `authenticate`.

```
// The request message containing the token to verify the user's identity.
message TokenRequest {
  string token = 1;
}

// The response message containing the user's information.
message Information {
  string username = 1;
  string name = 2;
  string surname = 3;
  string email = 4;
}
```

- Y finalmente el código de los servidores, que contiene la implementación de los mensajes especificados en los `.proto` junto con el acceso a la base de datos, la generación y validación de tókenes JWT, y el hasheo de contraseñas de los usuarios.
    - El archivo `auth.py` implementa el `Authenticator` definido en el `auth.proto`. En la clase `AuthenticatorServicer` se encuentra el código correspondiente al método `authenticate`, que verifica que el nombre de usuario y contraseña recibidos se encuentren en la base de datos, y retorna un tóken de acceso JWT o un error según corresponda.

```
def authenticate(self, request: Credentials, context: grpc.RpcContext) -> TokenResponse:
        """
        Authenticates a user and returns a JWT token.

        :param request: The request containing the username and
            password.
        :param context: The context of the request.
        :return: A token response containing the JWT token.
        """
        user: User | None = self.db.authenticate(request.username, request.password)

        if user is None:
            context.set_code(grpc.StatusCode.UNAUTHENTICATED)
            context.set_details("Invalid username or password")
            return TokenResponse()

        return TokenResponse(token=user.generate_token())
```

    - A su vez, en el archivo `info.py` se encuentra el `InformantServicer`, servidor que se comunica con la base de datos para obtener los datos de un usuario a partir de un token JWT en la función `getInfo`. En caso de recibir un JWT inválido, arroja un error acorde.

```
def getInfo(self, request: TokenRequest, context: grpc.RpcContext) -> Information:
        """
        Get information about a user.

        :param request: A TokenRequest object containing the JWT token.
        :param context: The context of the request.
        :return: An Information object containing the user's data.
        """
        user: User | None = self.db.get_user_by_token(request.token)
        if user is None:
            context.set_code(grpc.StatusCode.UNAUTHENTICATED)
            context.set_details("Invalid token")
            return Information()

        return Information(
            username=user.username,
            name=user.name,
            surname=user.surname,
            email=user.email,
        )
```

- El archivo `db.py` es el que contiene la implementación de dos objetos que sirven para facilitar la comunicación con la base de datos de MongoDB, y (de)codificar los tókenes JWT de los usuarios.

  En primer lugar, se define la función `hash_passwd`, que utiliza el algoritmo SHA-256 para encriptar strings. En este caso solo se usará para las contraseñas de los usuarios.

  ```
  def hash_passwd(password: str) -> str:
      """Hash a string using the SHA-256 algorithm."""
      return hashlib.sha256(bytes(password, "utf-8")).hexdigest()
  ```

  Luego se define una lista de los usuarios que estarán precargados en la base de datos. Cabe aclarar que se tuvo en cuenta que las contraseñas siempre se guarden cifradas, y no en texto plano.

  ```
  USERS: list[dict[str, str]] = [
      {
          "username": "user1",
          "hashed_password": hash_passwd("password1"),
          "name": "Roberto",
          "surname": "Carlos",
          "email": "rc@email.com",
      },
      {
          "username": "admin",
          "hashed_password": hash_passwd("admin"),
          "name": "Admin",
          "surname": "Admin",
          "email": "admin@email.com",
      },
  ]
  ```

  A continuación se crea la implementa `Database`, cuyas instancias iniciaran una conección con la base de datos al momento de ser creadas, cargando la información de los usuarios en caso de ser necesario.

  ```
  class Database:
      """A class to manage the Users' database."""

      def __init__(self):
          """Initialize the database."""

          mongoengine.connect("users")

          if not User.objects.count():
              for user in USERS:
                  User(**user).save()
  ```

  En la misma clase se define el método `authenticate`, llamado por la función homónima para del `AuthenticationServicer` para obtener al usuario cuyas credenciales fueron recibidas, o `None` en caso contrario.

```
def authenticate(self, username: str, password: str) -> User | None:
        """
        Authenticate a user.

        :param username: the username of the user.
        :param password: the password of the user.
        :return: The user if exists, None otherwise.
        """
        user: User | None = User.objects(username=username).first()

        if user is None:
            return None

        if user.is_password(password):
            return user

        return None
```

Y allí también se encuentra el método `get_by_user_token`, que devuelve un usuario según el tóken JWT que se haya recibido. Aquí es donde se efectúa la decodificación y validación del tóken, acciones facilitadas por la librería `PyJWT`.

```
def get_user_by_token(self, token: str) -> User | None:
        """
        Get the user from the token.

        :param token: the token.
        :return: the user if exists, None otherwise.
        """
        try:
            username: str = jwt.decode(token, SECRET, algorithms=["HS256"])["username"]
        except jwt.InvalidTokenError:
            return None

        return User.objects(username=username).first()
```

Además, este archivo contiene la clase `User`, que modela un usuario en la base de datos y provee métodos auxiliares para tratar con ellos.

El primero es el `generate_token`, donde se codifica un tóken JWT correspondiente a un usuario.

```
def generate_token(self) -> str:
        """
        Generate a JWT token for the user.

        :return: the JWT token.
        """
        return jwt.encode({"username": self.username}, SECRET, algorithm="HS256")
```

Y el segundo es el `is_password`, utilizado para verificar que la contraseña recibida sea efectivamente la que le corresponde al usuario indicado.

```
def is_password(self, password: str) -> bool:
        """
        Check if the password is correct.

        :param password: the password.
        :return: True if the password is correct, False otherwise.
        """
        return self.hashed_password == hash_passwd(password)
```

- Por último, el archivo `start_servers.py` contiene la función `start_servers` que crea un proceso para cada servidor, mas uno para el daemon de MongoDB, y los pone en ejecución.

```
def start_servers():
    auth_process = mp.Process(target=_start_auth_server)
    info_process = mp.Process(target=_start_info_server)
    mp.Process(target=_start_mongodb).start()

    auth_process.start()
    info_process.start()
    auth_process.join()
    info_process.join()
```

Manual de uso

Instalación

Para instalar el entorno donde se ejecutara el programa se debera y contar con Docker. Luego, desde una terminal ubicada en la carpeta del proyecto ejecutar los siguientes comandos de bash:

```
# Descargar imagen oficial de la cátedra
docker pull gmaron/pdytr

# Correr el container con la imagen descargada
docker run -itd -v $(pwd):/pdytr/ -p 27017:27017 -p 5000:5000 -p 5001:5001 --name pdytr gmaron/pdytr

# Abrir terminal de bash en el container
docker container exec -it --user root -w /pdytr pdytr bash

# Configurar el ambiente
make setup
```

Lo que hace el último comando es darle los permisos necesarios al script “setup.sh” y ejecutarlo para poder instalar e iniciar en el container MongoDB, Python 3.7 y dependencias de los programas.

Ejecución

Para ejecutar el experimento es necesario utilizar dos terminales: una que haga el papel de cliente, y otra que levante los servidores. Entonces, se deben abrir dos terminales de bash en

el container instalado previamente, con el comando:

```
docker container exec -it --user root -w /pdytr pdytr bash
```

Luego, en la terminal donde funcionarán los dos servidores de manera concurrente, ejecutar alguna de las siguientes instrucciones:

```
make server

# comando alternativo
python3.7 main.py --server
```

En la terminal restante, donde correrá el cliente, ejecutar alguna de las siguientes instrucciones:

```
make client

# comando alternativo
python3.7 main.py --client
```

Referencias

[1] “gRPC in Python”, *gRPC*. https://grpc.io/docs/languages/python/ (accedido 31 de mayo de 2022).

[2] “PyJWT 2.4.0 documentation”. https://pyjwt.readthedocs.io/ (accedido 31 de mayo de 2022).

[3] M. Jones, J. Bradley, y N. Sakimura, “JSON Web Token (JWT)”, Internet Engineering Task Force, Request for Comments RFC 7519, may 2015. doi: 10.17487/RFC7519.

[4] “python-jose 0.2.0 documentation”. https://python-jose.readthedocs.io/ (accedido 31 de mayo de 2022).

[5] “Authlib: Python Authentication”, *Authlib*. https://docs.authlib.org/ (accedido 31 de mayo de 2022).

[6] “JWCrypto 1.3.1 documentation”. https://jwcrypto.readthedocs.io/ (accedido 31 de mayo de 2022).

[7] M. A. Miller, “Examples of Protecting Content Using JSON Object Signing and Encryption (JOSE)”, Internet Engineering Task Force, Request for Comments RFC 7520, may 2015. doi: 10.17487/RFC7520.

[8] “MongoDB: The Application Data Platform | MongoDB”. https://www.mongodb.com/ (accedido 31 de mayo de 2022).

[9] “MongoEngine 0.24.1 documentation”. https://docs.mongoengine.org/ (accedido 1 de junio de 2022).

[10] “About gRPC | gRPC”. https://grpc.io/about/ (accedido 31 de mayo de 2022).

[11] “Introduction to gRPC”, *gRPC*. https://grpc.io/docs/what-is-grpc/introduction/ (accedido 31 de mayo de 2022).

[12] “Core concepts, architecture and lifecycle”, *gRPC*. https://grpc.io/docs/what-is-grpc/core-concepts/ (accedido 31 de mayo de 2022).

[13] auth0.com, “JSON Web Tokens Introduction”. http://jwt.io/ (accedido 31 de mayo de 2022).