Servicio de almacenamiento

Revisión 2

1. Introducción

Se deberá implementar una API REST que permita el almacenamiento de bloques de bytes. El lenguaje de implementación es libre, aunque se recomienda utilizar Python 3. Además, deberá crearse una librería de acceso a la API y un cliente que permita utilizar el servicio sin necesidad de conocer la propia API REST.

2. Implementación del servicio

El servicio debe almacenar los bloques de bytes (o blobs) de manera persistente. Para ello, el alumno decidirá la forma de hacerlo teniendo en cuenta:

- No debe existir límite práctico en el tamaño del blob.
- El contenido del blob debe ser completamente transparente: debe almacenar texto o datos binarios de la misma manera.
- Todos los blobs deben estar identificados de una manera unívoca, teniendo en cuenta que puede haber blobs con el mismo nombre, pero contenido diferente.
- Cada blob contará con una lista de roles que pueden modificar dicho blob.

Se deberá crear una clase que implemente la persistencia de datos, es decir, que abstraiga de todos los detalles del almacenamiento a las demás capas. Se sugieren métodos para:

- Crear un blob.
- Obtener un blob.
- Borrar un blob.
- Modificar un blob (reemplazarlo por otro).

Cuando se intente cualquier operación de acceso sobre un blob que no exista, se lanzará una excepción de usuario BlobNotFound(blob_id).

Además de la capa de persistencia, se creará otra clase que implementará la capa de negocio. Dicha clase se inicializará utilizando una instancia de la clase de persistencia con la que realizará todas las operaciones de acceso a los blobs. Todas las operaciones deben ir autenticadas con un token, de manera que este servicio utilizará la librería de acceso al servicio de tokens (se recomienda usar un mock para el desarrollo). Esta capa implementará las siguientes reglas de acceso:

- La lista de roles que pueden modificar un blob determinado es persistente.
- Los métodos de la capa de negocio que impliquen alteración de blobs (modificación o borrado) deben incluir un argumento más: el rol o roles del usuario dueño del token, para que la función pueda decidir si acepta o no la petición. De igual manera de tratarán las operaciones de modificación de roles de acceso al blob.
- Si el rol (o roles) del usuario no está (o están) en la lista de roles admitidos, se lanzará una excepción de usuario Forbidden(blob_id, rol|roles).
- Si el usuario tiene rol “admin” siempre se permite la acción, esté o no ese rol en la lista de acceso del blob.

Por último, se escribirá la capa de presentación, que consistirá en la API REST descrita a continuación.

3. API REST

A continuación, se indican los recursos admitidos por la API:

- API_ROOT/blob
  - Método PUT
    - Cabeceras obligatorias: AuthToken: <string>
    - Input: {“name”: <string>, “writable_by”: [<string>], “data”: <multipart>}
    - Output:
      - 201 (Created), data: {“blob_id”: <string>}
      - 400 (Bad Request)
      - 401 (Unauthorized)
- API_ROOT/blob/{blob_id}
  - Método DELETE
    - Cabeceras obligatorias: AuthToken: <string>
    - Output:
      - 204 (No content)
      - 401 (Unauthorized)
      - 404 (Not Found)
- API_ROOT/blob/{blob_id}/data
  - Método GET
    - Cabeceras obligatorias: AuthToken: <string>
    - Output:
      - 200 (Ok), data: <multipart>
      - 401 (Unauthorized)
      - 404 (Not Found)
  - Método PATCH, POST
    - Cabeceras obligatorias: AuthToken: <string>
    - Input: <multipart>
    - Output:
      - 204 (No content)
      - 401 (Unauthorized)
      - 404 (Not Found)
- API_ROOT/blob/{blob_id}/roles
  - Método PATCH, POST
    - Cabeceras obligatorias: AuthToken: <string>
    - Input: {“writable_by”: [<string>]}
    - Output:
      - 204 (No Content)
      - 400 (Bad Request)
      - 401 (Unauthorized)
      - 404 (Not Found)
  - Método GET
    - Cabeceras obligatorias: AuthToken: <string>
    - Output:

                - 200 (Ok), data: {“writable_by”: [<string>]}
                - 401 (Unauthorized)
                - 404 (Not Found) // Se puede utilizat 401 siempre
- API_ROOT/blob/{blob_id}/name
    - Método PATCH, POST
        - Cabeceras obligatorias: AuthToken: <string>
        - Input: {“name”: <string>}
        - Output:
            - 204 (No Content)
            - 400 (Bad Request)
            - 401 (Unauthorized)
            - 404 (Not Found)
    - Método GET
        - Cabeceras obligatorias: AuthToken: <string>
        - Output:
            - 200 (Ok), data: {“writable_by”: [<string>]}
            - 401 (Unauthorized)
            - 404 (Not Found)

La capa de presentación debe tener en cuenta los siguientes aspectos:

- El solicitante debe averiguarse utilizando el valor de la cabecera AuthToken. Para ello debe usarse el servicio correspondiente, en caso de no disponer de dicho servicio, podrá utilizarse un mock.
- Una vez obtenido el solicitante, se llamará a la función de la capa de negocio correspondiente.
    - En caso de capturar una excepción se retornará el error 4XX correspondiente.
    - Si la operación no produjo errores, se retornará el estado 2XX indicado.

4. El servidor

El servicio de autenticación se implementará mediante un servidor, que aceptará las siguientes opciones:

- “-p <puerto> o “--port” <puerto>”: establece un puerto de escucha, si no se establece por defecto será el 3003.
- “-l <dirección>” o “--listening <dirección>”: establece una dirección de escucha, por defecto se usará “0.0.0.0”.
- “-s <ruta>” o “--storage <ruta>”: establece la ruta donde se almacenarán todos los datos de la persistencia, por defecto se usará el *current working directory* o CWD.

5. Pruebas

Se crearán una serie de pruebas unitarias que deberán poder ejecutarse de forma automática dentro del entorno virtual apropiado. Por tanto, el proyecto incluirá un archivo *requirements.txt* con los datos necesarios para crear ese entorno. Las pruebas deberán tener las siguientes características:

- Las clases de la capa de negocio y persistencia deberán probarse con un >70% de cobertura.
- El código del servidor y la capa de presentación en >50%.

Para facilitar la ejecución inicial del servicio, si fuera necesario, se creará un script (en bash o python) que se llame “bootstrap” y que incializará la base de datos. Si el servicio requiere de algún archivo adicional, este script debe crearlo. De manera que después de ejecutar el script, el servicio debe estar listo para ejecutarse.