**UNIVERSIDAD CENTRAL DEL ECUADOR**

**FACULTAD DE INGENIERÍA Y CIENCIAS APLICADAS**

****

**CARRERA DE SISTEMAS DE INFORMACIÓN**

**PROGRAMACIÓN DISTRIBUIDA**

**“DOCUMENTACIÓN APLICACIÓN BOOKS Y AUTHORS”**

**GRUPO 3:**

* **MARTINEZ PAUL**
* **MENA JAIRO**
* **PANCHI ARIEL**
* **JULIAN ROSERO**

**BOOKS – APP**

Para la aplicación “books - app” se usó la siguiente configuración:

* Implementación Microprofile 5.0: Open Liberty
* Base de datos NoSQL: MongoDB

La aplicación fue construida de la siguiente manera (observe la ilustración 1):

Interfaz de usuario gráfica, Texto, Aplicación, Chat o mensaje de texto

Descripción generada automáticamente

Ilustración

Primeramente, se define una clase “Book” como se muestra en la ilustración 2, con cinco campos de instancia: "id", "isbn", "title", "author\_id" y "price". Cada campo está decorado con la anotación "@BsonProperty", que indica que se utilizará para serializar/deserializar objetos BSON (Binary JSON). Además, la clase tiene tres constructores: uno sin argumentos (por defecto), uno que toma todos los campos como argumentos y uno que toma solo "id" e "isbn" como argumentos. Los constructores se utilizan para crear instancias de la clase "Book" con valores específicos para sus campos.

La clase también implementa la interfaz "Serializable", lo que significa que sus instancias se pueden convertir en bytes y luego restaurar desde esos bytes. Esto es útil, por ejemplo, cuando se desea guardar objetos de la clase en archivos o transferirlos a través de la red. La anotación "@Data" proviene de la biblioteca Lombok y agrega automáticamente métodos de acceso (getters y setters) para los campos de la clase, así como los métodos "equals", "hashCode" y "toString". Estos métodos son útiles para trabajar con objetos de la clase "Book" de manera más conveniente y legible. Las anotaciones "@AllArgsConstructor" y "@NoArgsConstructor" también provienen de Lombok y generan constructores con argumentos para todos los campos de la clase (en el caso de "@AllArgsConstructor") y un constructor sin argumentos (en el caso de "@NoArgsConstructor"). Estos constructores son útiles para crear instancias de la clase de manera más conveniente.



Ilustración : clase “Book”

Se definió una clase DbConfig en la cual se la utiliza para configurar la conexión a una base de datos MongoDB. La clase utiliza la anotación "@ApplicationScoped", lo que indica que solo se creará una instancia de esta clase por aplicación como se observa en la ilustración 3. La clase utiliza la inyección de dependencias para obtener los valores de las propiedades de configuración de la base de datos, que se definen en un archivo de propiedades externo “microprofile-config.properties” (observe ilustración 4). Las propiedades que se inyectan en la clase son "dbHost", "dbName" y "puerto". La clase también define un método llamado "mongoDatabase()" que se utiliza para obtener una instancia de la colección "book" en la base de datos. El método utiliza la biblioteca de MongoDB para crear una instancia de "MongoClient" utilizando el host y el puerto de la base de datos que se han inyectado. Luego, se obtiene la instancia de la base de datos MongoDB utilizando el nombre de la base de datos que se ha inyectado. Finalmente, se obtiene la instancia de la colección "book" en la base de datos y se devuelve como resultado del método. La anotación "@Produces" indica que el método "mongoDatabase()" se utiliza para producir una instancia de la colección "book" y que esta instancia se inyectará en otras partes de la aplicación que necesiten acceder a la base de datos. La anotación "@ApplicationScoped" indica que la instancia de la colección "book" producida por este método se mantendrá durante toda la vida de la aplicación.

Texto

Descripción generada automáticamente

Ilustración : clase “DbConfig”

Texto

Descripción generada automáticamente

Ilustración : archivo externo “microprofile-config.properties”

Se definió dos clases de servicios. Una clase “ServicioBookImp” que es una implementación de la interfaz “ServicioBook” que proporciona métodos para realizar operaciones CRUD (observe las ilustraciones 5 a 10) en una colección de libros almacenada en una base de datos MongoDB y también interactúa con un servicio web externo utilizando una clase de proxy llamada “AuthorRestProxy”.

* **findAll():** Este método devuelve una lista de todos los libros almacenados en la base de datos MongoDB. Itera sobre los documentos recuperados de la colección MongoCollection y crea un objeto Book para cada uno. Luego, los objetos Book se agregan a una lista y se devuelve.

Texto

Descripción generada automáticamente

Ilustración : clase “ServicioBookImpl – método findAll”

* **findById(String id):** Este método busca un libro por su ID en la base de datos MongoDB. Recupera el documento correspondiente de la colección MongoCollection utilizando el filtro Filters.eq("\_id", oid), donde oid es el ObjectId del libro. Luego, se crea un objeto Book a partir del documento y se devuelve.

Texto

Descripción generada automáticamente

Ilustración : clase “ServicioBookImpl – método findById”

* **insert(Book book)**: Este método inserta un libro en la base de datos MongoDB. Crea un nuevo documento y lo inserta en la colección MongoCollection.

Texto

Descripción generada automáticamente

Ilustración : clase “ServicioBookImpl – método insert”

* **update(Book book, String id):** Este método actualiza un libro existente en la base de datos MongoDB. Recupera el documento correspondiente de la colección MongoCollection utilizando el filtro new Document("\_id", oid), donde oid es el ObjectId del libro. Luego, crea un nuevo documento con los datos actualizados del libro y utiliza findOneAndReplace(filterByGradeId,document) para reemplazar el documento existente con el nuevo documento.

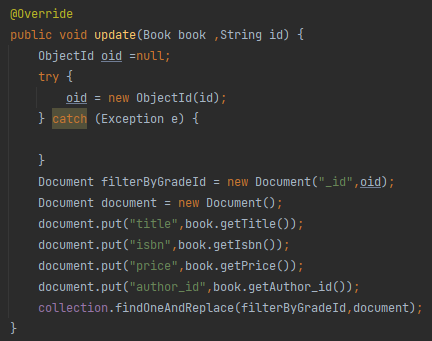


Ilustración : clase “ServicioBookImpl – método update”

* **delete(String id):** Este método elimina un libro de la base de datos MongoDB. Recupera el documento correspondiente de la colección MongoCollection utilizando el filtro new Document("\_id", oid), donde oid es el ObjectId del libro. Luego, utiliza deleteOne(filterByGradeId) para eliminar el documento.

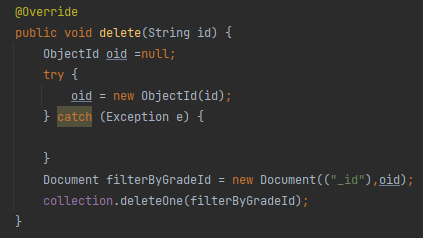


Ilustración : clase “ServicioBookImpl – método delete”

* **findAllCompleto():** Este método devuelve una lista de todos los libros almacenados en la base de datos MongoDB, junto con el nombre completo del autor correspondiente. Itera sobre los documentos recuperados de la colección MongoCollection y crea un objeto Book para cada uno. Luego, utiliza el AuthorRestProxy para buscar el autor correspondiente del libro utilizando el author\_id del libro. Por último, se crea un objeto BookDto que incluye la información del libro y el nombre completo del autor y se devuelve como una lista.

Texto

Descripción generada automáticamente

Ilustración : clase “ServicioBookImpl – método findAllCompleto”

Se definió una clase “BookDto” observe la ilustración 11 que se utiliza para devolver información detallada de un libro, incluido el nombre del autor del libro, a partir de la información básica del libro almacenada en la base de datos. Esta clase utiliza las anotaciones de Lombok @Data, @AllArgsConstructor y @NoArgsConstructor para generar automáticamente los métodos getter, setter, toString, hashCode y equals para los campos de la clase. Además, utiliza la anotación @Access para especificar que se accederá a los campos directamente en lugar de través de métodos de acceso.



Ilustración : clase “BookDto”

También se definió una clase tipo interfaz “BookService” observe la ilustración 12, la cual define los métodos que se deben implementar para poder interactuar con la entidad "Book" en el sistema.

* **findAll():** devuelve una lista de todos los libros en el sistema sin incluir el nombre del autor.
* **findAllCompleto():** devuelve una lista de todos los libros en el sistema incluyendo el nombre del autor.
* **findById(String id):** busca un libro por su identificador único y lo devuelve.
* **insert(Book book):** agrega un nuevo libro al sistema.
* **update(Book book ,String id):** actualiza los detalles de un libro existente en el sistema.
* **delete(String id):** elimina un libro del sistema por su identificador único.

Texto

Descripción generada automáticamente

Ilustración : clase “ServicioBook”

Se definió una clase “BookRest” que implementa los endpoints para exponer los servicios del CRUD de Book (observe las ilustraciones 13 a 18). Aquí se utilizan anotaciones de JAX-RS para definir las rutas y métodos HTTP que se utilizan en cada uno de los endpoints.

* En el método findAll(), se llama al método findAll() del servicio correspondiente para obtener una lista de todos los libros y devolverla en formato JSON.

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Ilustración

* En el método findAllCompleto(), se llama al método findAllCompleto() del servicio correspondiente para obtener una lista de todos los libros con información adicional de autor y devolverla en formato JSON.

Interfaz de usuario gráfica

Descripción generada automáticamente

Ilustración

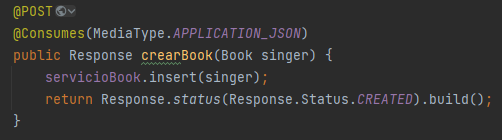
* En el método findById(), se llama al método findById(String id) del servicio correspondiente, pasando el ID del libro como parámetro y devolviendo el libro correspondiente en formato JSON.

Interfaz de usuario gráfica

Descripción generada automáticamente

Ilustración

* En el método crearBook(), se utiliza la anotación @POST para indicar que se trata de una solicitud POST. Se espera que se envíe un objeto JSON que se convierte en un objeto Book y se pasa al método insert(Book book) del servicio correspondiente para agregar el nuevo libro a la base de datos.



Ilustración

* En el método borrarBook(), se utiliza la anotación @DELETE para indicar que se trata de una solicitud DELETE. Se espera que se proporcione el ID del libro que se va a borrar como parte de la ruta. Este ID se pasa al método delete(String id) del servicio correspondiente para eliminar el libro correspondiente de la base de datos.

Texto

Descripción generada automáticamente

Ilustración

* En el método modificar(), se utiliza la anotación @PUT para indicar que se trata de una solicitud PUT. Se espera que se envíe un objeto JSON que se convierte en un objeto Book y se pasa junto con el ID del libro como parámetros al método update(Book book ,String id) del servicio correspondiente para actualizar el libro correspondiente en la base de datos.

Texto

Descripción generada automáticamente

Ilustración

La clase tipo interfaz “AuthorRestProxy” utiliza anotaciones JAX-RS para especificar los métodos HTTP y las rutas para interactuar con un servicio RESTful que maneja autores (observe la ilustración 19). La anotación @Path("/authors") especifica que todas las rutas definidas en la interfaz estarán bajo el contexto "/authors".

La anotación @Produces(MediaType.APPLICATION\_JSON) especifica que la interfaz producirá objetos JSON y la anotación @Consumes(MediaType.APPLICATION\_JSON) especifica que la interfaz aceptará objetos JSON en las solicitudes.

La anotación @RegisterRestClient(configKey = "author") especifica que esta interfaz será registrada como un cliente REST en Quarkus y que utilizará la configuración "author" en el archivo application.properties.

* findById(Long id): este método realiza una solicitud GET al endpoint "/authors/{id}" para obtener la información de un autor específico identificado por el valor del parámetro "id".
* findAll(): este método realiza una solicitud GET al endpoint "/authors" para obtener la lista de todos los autores.
* insert(AuthorsCliente obj): este método realiza una solicitud POST al endpoint "/authors" para insertar un nuevo autor. El objeto AuthorsCliente que se pasa como parámetro se serializa como JSON en el cuerpo de la solicitud.
* update(AuthorsCliente obj, Long id): este método realiza una solicitud PUT al endpoint "/authors/{id}" para actualizar la información de un autor específico identificado por el valor del parámetro "id". El objeto AuthorsCliente que se pasa como parámetro se serializa como JSON en el cuerpo de la solicitud.
* delete(Long id): este método realiza una solicitud DELETE al endpoint "/authors/{id}" para eliminar un autor específico identificado por el valor del parámetro "id".

Texto

Descripción generada automáticamente

Ilustración

Por último, la clase “AuthorsCliente” es una representación de un autor utilizada en el cliente REST para comunicarse con el servicio de autores (observe la ilustración 20). Tiene tres atributos: "id" que representa el identificador del autor, "first\_name" que representa el nombre del autor y "last\_name" que representa el apellido del autor. Los atributos están anotados con las anotaciones @Getter y @Setter para proporcionar métodos de acceso para obtener y establecer los valores de los atributos. También tiene la anotación @ToString para generar automáticamente una representación de cadena de la instancia de la clase.

Texto

Descripción generada automáticamente

Ilustración

Adicionalmente se creó un archivo XML “server.xml”. Este archivo XML contiene la configuración del servidor en el cual se desplegará la aplicación. Algunas de las características que se han habilitado son microProfile-5.0 y persistence-3.0. También se ha configurado un puerto para el servidor y un host que permite la conexión a cualquier dirección IP (observe la ilustración 21).

El archivo también especifica la ubicación de la aplicación web que se va a desplegar, así como la configuración de seguridad para JWT y SSL, especificando la ubicación del keystore y su contraseña.

Además, se ha habilitado la funcionalidad de métricas de MicroProfile, lo que permitirá medir y monitorear el rendimiento y la salud de la aplicación en tiempo de ejecución.

Texto

Descripción generada automáticamente

Ilustración

Se creo un arvrchivo “docker-compose-yaml” que define la configuración de varios servicios que se ejecutan en contenedores Docker (observe las ilustraciones de la 22 a 25). Cada servicio se define como una imagen de Docker que se ejecutará en un contenedor separado.

* El primer servicio es un contenedor que ejecuta la última versión de MongoDB y se expone en el puerto 27017.

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Ilustración

* El segundo servicio es un proxy inverso de Traefik que se ejecutará en el puerto 80 y 8080. Este servicio se encarga de enrutar las solicitudes a los servicios de la aplicación.

Texto

Descripción generada automáticamente

Ilustración

* El tercer servicio es el servicio Books que se ejecuta en el puerto 7080 y tiene una etiqueta que especifica la regla de ruta y el middleware que se utilizará para enrutar las solicitudes.

Texto

Descripción generada automáticamente

Ilustración

* El cuarto servicio es el servicio Authors que se ejecuta en el puerto 5000 y también tiene una etiqueta que especifica la regla de ruta y el middleware que se utilizará para enrutar las solicitudes.

Texto

Descripción generada automáticamente

Ilustración

* Además, el archivo docker-compose también define varias variables de entorno que se utilizan para configurar los servicios y sus dependencias. Por ejemplo, la variable AUTHOR\_URL se utiliza para especificar la URL del servicio Authors en el servicio Books.

**AUTHORS – APP**

Para la aplicación “authors - app” se usó la siguiente configuración:

* Implementación Flask 2.2.3
* Base de datos SQL: Postgres

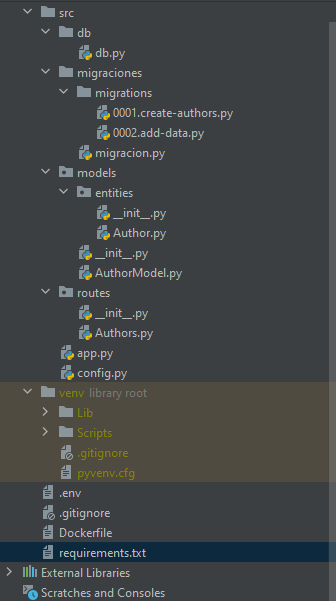
La aplicación fue construida de la siguiente manera (observe la ilustración 1):

Ilustración 1

Primeramente, se define una clase “Author” como se muestra en la ilustración 2, la clase tiene un constructor "init" que inicializa las propiedades de un objeto de tipo "Author" con los parámetros proporcionados.

El constructor tiene tres parámetros: "id", "first\_name" y "last\_name". El parámetro "id" es obligatorio, mientras que los parámetros "first\_name" y "last\_name" son opcionales. Cuando se crea un objeto de tipo "Author", el valor del parámetro "id" debe ser proporcionado, mientras que los valores de los parámetros "first\_name" y "last\_name" pueden ser opcionales.

Además, la clase tiene un método llamado "to\_JSON" que devuelve un diccionario Python que representa los valores de las propiedades del objeto "Author". Este método se utiliza para convertir un objeto "Author" en un formato JSON, que es un formato de intercambio de datos ampliamente utilizado.

El método "to\_JSON" devuelve un diccionario con las siguientes claves y valores:

* **'id':** el valor del atributo "id" del objeto "Author".
* **'first\_name':** el valor del atributo "first\_name" del objeto "Author", o None si este atributo no está definido.
* **'last\_name':** el valor del atributo "last\_name" del objeto "Author", o None si este atributo no está definido.

Texto

Descripción generada automáticamente

Ilustración 2: clase “Author”

En db como se muestra en la ilustración 3 importa los módulos "psycopg2", "DatabaseError" y "config" de las bibliotecas "psycopg2" y "decouple". También define una función llamada "get\_connection" que devuelve una conexión a una base de datos PostgreSQL.

La función "get\_connection" utiliza el módulo "psycopg2" para establecer una conexión a una base de datos PostgreSQL utilizando los valores de configuración definidos en un archivo “. env". La función utiliza la función "config" de la biblioteca "decouple" para obtener los valores de configuración del archivo “. env".

Los valores de configuración que se utilizan para conectarse a la base de datos son:

* **"PGSQL\_HOST":** el nombre o la dirección IP del host de la base de datos PostgreSQL.
* **"PGSQL\_USER":** el nombre de usuario de la cuenta de la base de datos PostgreSQL.
* **"PGSQL\_PASSWORD":** la contraseña de la cuenta de la base de datos PostgreSQL.
* **"PGSQL\_DATABASE":** el nombre de la base de datos PostgreSQL a la que se quiere conectar.
* **"PGSQL\_PORT":** el número de puerto de la base de datos PostgreSQL.

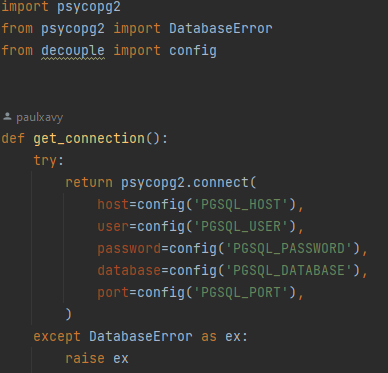
Si la conexión es exitosa, la función "get\_connection" devuelve el objeto de conexión. Si ocurre algún error al intentar establecer la conexión, se captura la excepción "DatabaseError" y se relanza para que sea manejada en otra parte del código.

Ilustración 3: “db”

Se define una clase llamada "AuthorModel", que contiene varios métodos de clase para interactuar con una tabla "authors" en una base de datos PostgreSQL. La clase importa la función "get\_connection" del archivo "db.py" y la clase "Author" del archivo "Author.py".

El método "get\_authors" es un método de clase que devuelve una lista de autores de la tabla "authors" de la base de datos. El método realiza lo siguiente:

* Obtiene una conexión a la base de datos utilizando la función "get\_connection".
* Crea una lista vacía llamada "authors".
* Ejecuta una consulta SQL para obtener todos los registros de la tabla "authors".
* Recorre los registros obtenidos y para cada uno, crea un objeto "Author" utilizando los valores de los campos "id", "first\_name" y "last\_name" y lo agrega a la lista "authors".
* Cierra la conexión a la base de datos.
* Devuelve la lista de autores.

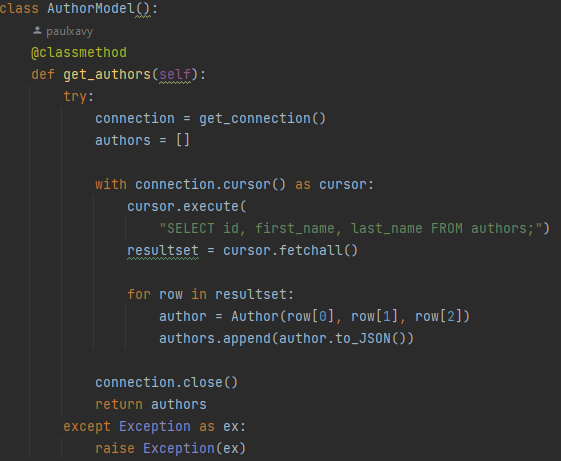


Ilustración 4: clase “AuthorModel – método get\_authors”

El método "get\_author" es otro método de clase que recibe un parámetro "id" y devuelve un objeto "Author" que corresponde al registro de la tabla "authors" con ese "id". El método realiza lo siguiente:

* Obtiene una conexión a la base de datos utilizando la función "get\_connection".
* Ejecuta una consulta SQL para obtener el registro de la tabla "authors" con el "id" especificado.
* Crea un objeto "Author" utilizando los valores de los campos "id", "first\_name" y "last\_name".
* Cierra la conexión a la base de datos.
* Devuelve el objeto "Author" creado.

Texto

Descripción generada automáticamente

Ilustración 5: clase “AuthorModel – método get\_author”

Los métodos "add\_author", "delete\_author" y "update\_author" son métodos de clase que agregan, eliminan y actualizan registros en la tabla "authors", respectivamente. Todos estos métodos realizan las siguientes operaciones:

* Obtiene una conexión a la base de datos utilizando la función "get\_connection".
* Ejecuta una consulta SQL para agregar, eliminar o actualizar el registro de la tabla "authors" correspondiente.
* Cierra la conexión a la base de datos.
* Devuelve el número de filas afectadas por la operación.

Texto

Descripción generada automáticamente

Ilustración 6: clase “AuthorModel – método add\_author”



Ilustración 7: clase “AuthorModel – método delete\_author”

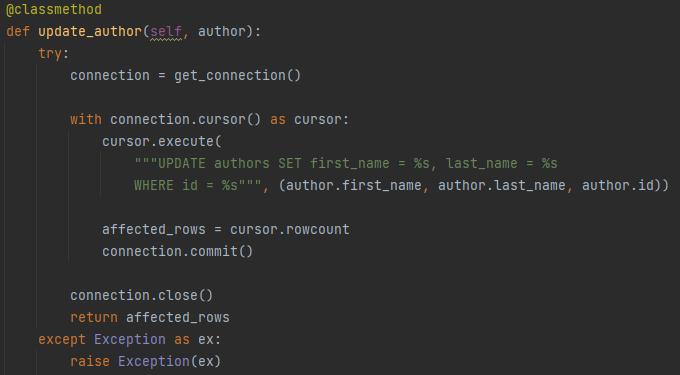


Ilustración 8: clase “AuthorModel – método update\_author”

Se define una API RESTful para manejar la entidad "Author" (autor) en una base de datos. La API utiliza el framework Flask para construir el servidor web.

Se importa los módulos necesarios para construir la API, como Flask, jsonify, request, la clase Author y la clase AuthorModel.

Luego, se crea un objeto Blueprint llamado "author\_blueprint" que se utilizará para agrupar todas las rutas relacionadas con el manejo de autores.

Las rutas definidas en el objeto Blueprint son:

**get\_authors():** Esta ruta se llama cuando se hace una petición GET a "/". Retorna todos los autores almacenados en la base de datos.

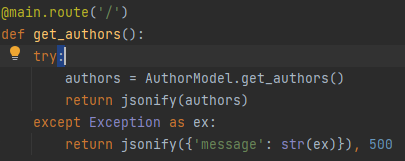


Ilustración 9: “Authors – ruta get\_authors”

**get\_author(id):** Esta ruta se llama cuando se hace una petición GET a "/<id>" donde id es el identificador único del autor. Retorna la información de un autor específico identificado por su ID.

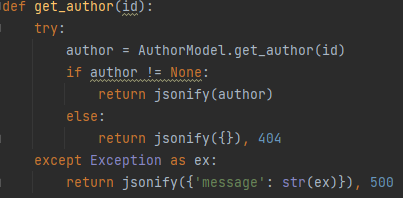


Ilustración 10: “Authors – ruta get\_author”

**add\_author():** Esta ruta se llama cuando se hace una petición POST a "/". Agrega un nuevo autor a la base de datos.

Texto

Descripción generada automáticamente

Ilustración 11: “Authors – ruta add\_author”

**delete\_author(id):** Esta ruta se llama cuando se hace una petición DELETE a "/<id>" donde id es el identificador único del autor. Elimina un autor específico identificado por su ID.

Texto

Descripción generada automáticamente

Ilustración 12: “Authors – ruta delete\_author”

**update\_author(id):** Esta ruta se llama cuando se hace una petición PUT a "/<id>" donde id es el identificador único del autor. Actualiza la información de un autor específico identificado por su ID.

Texto

Descripción generada automáticamente

Ilustración 13: “Authors – ruta update\_author”

En cada ruta, se llama a los métodos correspondientes de la clase AuthorModel para realizar las operaciones en la base de datos.

Si ocurre un error durante la ejecución de cualquiera de las rutas, se devuelve una respuesta HTTP con el código de estado 500 y un mensaje de error en formato JSON.