Clase 04

Rumbo al Proyecto final Full Stack Python con objetos y API REST

Cuarta etapa:

Implementar una API

Necesitamos realizar modificaciones en el código existente para que pueda ser publicado en el servidor de PythonAnywhere, y crear una API con Flask para interactuar desde un frontend. Para lograr esto, necesitamos realizar varios cambios. El nuev código, entre otras, incluye las siguientes modificaciones:

- Se importa el módulo Flask y jsonify de la biblioteca Flask para facilitar la creación de la API.
- Se agregan decoradores de Flask a cada método para definir las rutas de la API.
- Se utilizan los métodos jsonify para devolver las respuestas en formato JSON.
- Se reciben los datos del frontend mediante el objeto request.json.
- Se crea una instancia de la clase Inventario y Carrito en cada ruta según sea necesario.

Instalar el módulo Flask:

Para comenzar, necesitaremos instalar el módulo Flask en Python. Puedes seguir estos pasos:

1 - Abre una terminal o línea de comandos en tu sistema operativo.

2 - Asegúrate de tener Python instalado en tu computadora. Puedes verificarlo ejecutando el comando **python --version** en la terminal. Si Python está instalado, verás la versión de Python que tienes.

3 - Ejecuta el siguiente comando en la terminal para instalar Flask utilizando **pip**, el administrador de paquetes de Python:

```
pip install flask
```

Si estás utilizando Python 3, es posible que necesites usar **pip3** en lugar de **pip**:

```
pip3 install flask
```

4 - Espera a que pip descargue e instale Flask y sus dependencias.

5 - Una vez completada la instalación, puedes comenzar a utilizar Flask en tus aplicaciones Python importando el módulo flask y la clase Flask, como se muestra en el código de más arriba.

Recuerda que es recomendable utilizar un entorno virtual para tus proyectos de Python. Puedes crear un entorno virtual utilizando herramientas como **venv** o **virtualenv**, y luego instalar Flask dentro del entorno virtual. Esto ayuda a mantener las dependencias de cada proyecto separadas y evita conflictos entre versiones.

La clase Flask es la piedra angular de una aplicación Flask y se utiliza para crear una instancia de la aplicación.

Código de la aplicación

Y escribimos el código necesario para importar los módulos necesarios, crear la instancia de Flask y establecer la conexión con la base de datos. Como antes, se utiliza Flask y SQLite para crear una aplicación web con una base de datos SQLite.

```
import sqlite3
from flask import Flask,  jsonify, request

# Configurar la conexión a la base de datos SQLite
DATABASE = 'inventario.db'

def get_db_connection():
    conn = sqlite3.connect(DATABASE)
    conn.row_factory = sqlite3.Row
    return conn

# Crear la tabla 'productos' si no existe
def create_table():
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS productos (
            codigo INTEGER PRIMARY KEY,
            descripcion TEXT NOT NULL,
            cantidad INTEGER NOT NULL,
            precio REAL NOT NULL
        )
    ''')
    conn.commit()
    cursor.close()
    conn.close()

# Verificar si la base de datos existe, si no, crearla y crear la tabla
def create_database():
    conn = sqlite3.connect(DATABASE)
    conn.close()
    create_table()

# Crear la base de datos y la tabla si no existen
create_database()
```

En esta parte, el código es prácticamente el mismo que vimos en la etapa anterior. Solo hemos agregado una linea. Por comodidad, repetimos la explicación del código:

En la primera línea, importamos la clase Flask del módulo flask para crear una instancia de la aplicación Flask.

from flask import Flask, jsonify, request importa clases y funciones del módulo Flask que necesitamos en nuestro proyecto.

Luego, se establece el nombre de la base de datos SQLite en la variable DATABASE. En este caso, se utiliza el archivo '**inventario.db**' como base de datos.

La función **get_db_connection()** se define para establecer la conexión con la base de datos SQLite utilizando la biblioteca sqlite3. Esta función devuelve una conexión a la base de datos que se puede utilizar para realizar consultas y transacciones.

La función **create_table()** se define para crear la tabla 'productos' en la base de datos SQLite si no existe. Utiliza la conexión obtenida mediante **get_db_connection()** y el método **execute()** para ejecutar una sentencia SQL que crea la tabla con las columnas 'codigo', 'descripcion', 'cantidad' y 'precio'. Luego, se realiza una confirmación (**commit()**) para guardar los cambios y se cierra el cursor y la conexión.

Finalmente, se llama a **create_table()** para asegurarse de que la tabla 'productos' exista en la base de datos antes de continuar con el resto del código.

En resumen, se establece la conexión a la base de datos SQLite y crea la tabla 'productos' si no existe. Esto asegura que la base de datos esté lista para almacenar los datos de los productos antes de que la aplicación Flask comience a interactuar con ella.

Configuración y rutas de la API Flask

La sección "Configuración y rutas de la API Flask" del código es completamente nueva.
Este código, como puede verse en el código adjunto (clase_04.py), se coloca despues de las definiciones de las clases, justo antes de implementar los decoradores y funciones de nuestra API. Recordemos que este texto no analiza el codigo fuente de forma lineal, sino que lo hace explicando cada sección importante.

1) Importación de los módulos y creación de la aplicación Flask:

```
app = Flask(__name__)

carrito = Carrito()         # Instanciamos un carrito
inventario = Inventario()   # Instanciamos un inventario
```

Importamos el módulo Flask y creamos una instancia de la clase Flask llamada app. Esta instancia representa nuestra aplicación Flask.
Tambien instanciamos las clases Carrito e Inventario para crear sendos objetos.

2) Decorador **@app.route('/productos/<int:codigo>', methods=['GET'])** y función **obtener_producto(codigo)**:

En la ruta llamada /productos/<int:codigo> podemos obtener los datos de un producto según su código.

```
# Ruta para obtener los datos de un producto según su código
@app.route('/productos/<int:codigo>', methods=['GET'])
def obtener_producto(codigo):
    #inventario = Inventario()
    producto = inventario.consultar_producto(codigo)
    if producto:
        return jsonify({
            'codigo': producto.codigo,
            'descripcion': producto.descripcion,
            'cantidad': producto.cantidad,
            'precio': producto.precio
        }), 200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

Veamos cómo funciona:

En la definición de la ruta, utilizamos <int:codigo> para indicar que esperamos un parámetro entero llamado codigo. Esto significa que en la URL de la solicitud, debes proporcionar el código del producto después de /productos/.

- Cuando se recibe una solicitud GET a esta ruta, se llama a la función obtener_producto(codigo).
- Dentro de la función obtener_producto, creamos una instancia de la clase Inventario para acceder a los métodos relacionados con la gestión de productos.
- Luego, llamamos al método consultar_producto(codigo) del objeto inventario para obtener los datos del producto correspondiente al código proporcionado.
- Si el producto existe en la base de datos, se crea un objeto producto con los detalles del producto, como el código, descripción, cantidad y precio.
- Utilizamos la función jsonify() de Flask para convertir el objeto producto en una respuesta JSON.
- Devolvemos la respuesta JSON con un código de estado 200, indicando que la solicitud se procesó correctamente y los datos del producto se encuentran en la respuesta.
- Si el producto no se encuentra en la base de datos, devolvemos un mensaje de error JSON con un código de estado 404, indicando que el producto no fue encontrado.

Con esta funcionalidad, los usuarios podrán obtener los datos de un producto específico utilizando el código del producto como referencia. Esto puede ser útil para mostrar los detalles de un producto en una página de detalles o cualquier otra funcionalidad relacionada con la obtención de datos específicos.

3) Decorador **@app.route('/')** y **función index()**:

```
# Ruta para obtener la lista de productos del inventario
@app.route('/')
def index():
    return 'API de Inventario'
```

El decorador @app.route('/') establece la ruta de la URL ("/") a la cual se asocia la función index(). En este caso, cuando accedemos a la página principal de la API, se ejecuta la función index() y devuelve la cadena de texto "API de Inventario". Aqui podríamos devolver el código HTML de una "pagina principal" que presente la API, o describa sus características. Queda como sugerencia para el alumno.

4) Decorador **@app.route('/productos', methods=['GET'])** y función **obtener_productos()**:

```
# Ruta para obtener la lista de productos del inventario
@app.route('/productos', methods=['GET'])
def obtener_productos():
    return inventario.listar_productos()
```

El decorador **@app.route('/productos', methods=['GET'])** establece la ruta de la URL (**"/productos"**) y el método HTTP permitido (en este caso, solo permitimos el método GET) para la función **obtener_productos()**. Cuando se realiza una solicitud GET a la ruta "/productos", se ejecuta esta función.

Dentro de la función obtener_productos(), se llama al método listar_productos() del objeto de la clase Inventario, que devuelve una lista de productos. Luego, utilizamos la función **jsonify()** para convertir la lista de productos en un objeto JSON y lo devolvemos como respuesta.

5) Decorador **@app.route('/productos', methods=['POST'])** y **función agregar_producto()**:

```
# Ruta para agregar un producto al inventario
@app.route('/productos', methods=['POST'])
def agregar_producto():
    codigo = request.json.get('codigo')
    descripcion = request.json.get('descripcion')
    cantidad = request.json.get('cantidad')
    precio = request.json.get('precio')
    return inventario.agregar_producto(codigo, descripcion, cantidad, precio)
```

El decorador **@app.route('/productos', methods=['POST'])** establece la ruta de la URL ("**/productos**") y el método HTTP permitido (en este caso, solo permitimos el método POST) para la función agregar_producto(). Cuando se realiza una solicitud POST a la ruta "/productos", se ejecuta esta función.

Dentro de la función **agregar_producto()**, utilizamos **request.get_json()** para obtener los datos enviados en el cuerpo de la solicitud POST como un objeto JSON. Luego, extraemos los valores de los campos "codigo", "descripcion", "cantidad" y "precio" del objeto JSON.

A continuación, llamamos al método **agregar_producto()** del objeto de la clase Inventario para agregar un nuevo producto. Dependiendo del resultado de la operación, devolvemos una respuesta JSON con un mensaje correspondiente y un código de estado HTTP 200 (éxito) o 400 (error).

6) Decorador **@app.route('/productos/<int:codigo>', methods=['PUT'])** y función **modificar_producto()**:

```
# Ruta para modificar un producto del inventario
@app.route('/productos/<int:codigo>', methods=['PUT'])
def modificar_producto(codigo):
    nueva_descripcion = request.json.get('descripcion')
    nueva_cantidad = request.json.get('cantidad')
    nuevo_precio = request.json.get('precio')
    return inventario.modificar_producto(codigo, nueva_descripcion, nueva_cantidad, nuevo_precio)
```

El decorador **@app.route('/productos/<int:codigo>', methods=['PUT'])** establece la ruta de la URL ("**/productos/<codigo>**") y el método HTTP permitido (en este caso, solo permitimos el método PUT) para la función modificar_producto(). El <int:codigo> indica que se espera un parámetro entero llamado "codigo" en la URL. Cuando se realiza una solicitud PUT a la ruta "/productos/<codigo>", se ejecuta esta función.

Dentro de la función **modificar_producto()**, utilizamos **request.get_json()** para obtener los datos enviados en el cuerpo de la solicitud PUT como un objeto JSON. Luego, extraemos los valores de los campos "descripcion", "cantidad" y "precio" del objeto JSON.

Llamamos al método **modificar_producto()** del objeto de la clase Inventario para modificar el producto correspondiente al código recibido. Devolvemos una respuesta JSON con un mensaje de éxito y un código de estado HTTP 200.

7) Decorador **@app.route('/productos/<int:codigo>', methods=['DELETE'])** y función **eliminar_producto()**:

```
# Ruta para eliminar un producto del inventario
@app.route('/productos/<int:codigo>', methods=['DELETE'])
def eliminar_producto(codigo):
    return inventario.eliminar_producto(codigo)
```

El decorador **@app.route('/productos/<int:codigo>', methods=['DELETE'])** establece la ruta de la URL ("**/productos/<codigo>**") y el método HTTP permitido (en este caso, solo permitimos el método DELETE) para la función eliminar_producto(). El <int:codigo> indica que se espera un parámetro entero llamado "codigo" en la URL. Cuando se realiza una solicitud DELETE a la ruta "/productos/<codigo>", se ejecuta esta función.

Dentro de la **función eliminar_producto()**, llamamos al **método eliminar_producto()** del objeto de la clase Inventario para eliminar el producto correspondiente al código recibido. Devolvemos una respuesta JSON con un mensaje de éxito y un código de estado HTTP 200.

8) Decorador **@app.route('/carrito', methods=['POST'])** y función **agregar_carrito()**:

```
# Ruta para agregar un producto al carrito
@app.route('/carrito', methods=['POST'])
def agregar_carrito():
    codigo = request.json.get('codigo')
    cantidad = request.json.get('cantidad')
    inventario = Inventario()
    return carrito.agregar(codigo, cantidad, inventario)
```

Este código implementa una ruta en la API de Flask para agregar un producto al carrito. Define una ruta en la API de Flask que permite agregar un producto al carrito mediante una solicitud HTTP POST a la ruta /carrito. La función agregar_carrito() se encarga de procesar la solicitud, crear el carrito y agregar el producto utilizando la instancia de la clase Carrito.

- El decorador @app.route('/carrito', methods=['POST']) define la ruta /carrito y especifica que solo se permite el método POST para esta ruta. Esto significa que solo se puede realizar una solicitud HTTP POST a esta ruta en particular.
- La función agregar_carrito() se ejecutará cuando se realice una solicitud HTTP POST a la ruta /carrito. Esta función es responsable de agregar un producto al carrito.
- Se extraen los datos necesarios de la solicitud HTTP utilizando request.json.get(). En este caso, se obtiene el valor del parámetro codigo y cantidad del cuerpo de la solicitud JSON. Estos valores representan el código y la cantidad del producto que se va a agregar al carrito.
- Luego, se llama al método agregar() del objeto carrito para agregar el producto al carrito. Se le pasan como argumentos el codigo, la cantidad y el objeto inventario.
- Por último, el resultado de la llamada al método agregar() se devuelve como respuesta HTTP. Dependiendo de cómo esté implementado el método agregar() en la clase Carrito, podría devolver un mensaje de éxito si el producto se agrega correctamente, o un mensaje de error si hay algún problema.

9) Decorador **@app.route('/carrito', methods=['DELETE'])** y función **quitar_carrito()**:

```
# Ruta para quitar un producto del carrito
@app.route('/carrito', methods=['DELETE'])
def quitar_carrito():
    codigo = request.json.get('codigo')
    cantidad = request.json.get('cantidad')
    inventario = Inventario()
    return carrito.quitar(codigo, cantidad, inventario)
```

Este fragmento de código define una ruta en la API de Flask que permite quitar un producto del carrito mediante una solicitud HTTP DELETE a la ruta /carrito. La función quitar_carrito() se encarga de procesar la solicitud, crear el carrito y quitar el producto utilizando la instancia de la clase Carrito.

- El decorador @app.route('/carrito', methods=['DELETE']) define la ruta /carrito y especifica que solo se permite el método DELETE para esta ruta. Esto significa que solo se puede realizar una solicitud HTTP DELETE a esta ruta en particular.
- La función quitar_carrito() se ejecutará cuando se realice una solicitud HTTP DELETE a la ruta /carrito. Esta función es responsable de quitar un producto del carrito.
- Se extraen los datos necesarios de la solicitud HTTP utilizando request.json.get(). En este caso, se obtiene el valor del parámetro codigo y cantidad del cuerpo de la solicitud JSON. Estos valores representan el código y la cantidad del producto que se va a

quitar del carrito.

- Luego, se llama al método quitar() del objeto carrito para quitar el producto del carrito. Se le pasan como argumentos el codigo, la cantidad y el objeto inventario.
- Por último, el resultado de la llamada al método quitar() se devuelve como respuesta HTTP. Dependiendo de cómo esté implementado el método quitar() en la clase Carrito, podría devolver un mensaje de éxito si el producto se quita correctamente, o un mensaje de error si hay algún problema.

10) Decorador **@app.route('/carrito', methods=['GET'])** y función **obtener_carrito()**:
Esta sección del código es una ruta en la API Flask que responde a la solicitud GET en la URL "/carrito" para obtener y devolver el contenido del carrito.

```
# Ruta para obtener el contenido del carrito
@app.route('/carrito', methods=['GET'])
def obtener_carrito():
    return carrito.mostrar()
```

- **@app.route('/carrito', methods=['GET'])** es un decorador de Flask que indica que esta función manejará solicitudes GET en la URL "/carrito".
- **def obtener_carrito():** define la función obtener_carrito que se ejecutará cuando se realice una solicitud GET a "/carrito".
- **return carrito.mostrar()** devuelve el resultado de llamar al método mostrar() en la instancia de la clase Carrito. Este método devuelve el contenido del carrito en formato JSON.

Estos son los decoradores Flask que definen las rutas y los métodos HTTP asociados a cada función. Las funciones manipulan los datos y devuelven respuestas JSON con mensajes y códigos de estado adecuados para la comunicación con el frontend. Ahora estamos listos para analizar los cambios que se han introducido en el código de cada clase.

Clase Producto

La clase Producto sigue sin cambios, recordemos que sus instancias solo se utilizan como parte del inventario:

```
# ----------------------------------------------------------------
# Definimos la clase "Producto"
# ----------------------------------------------------------------
class Producto:
    def __init__(self, codigo, descripcion, cantidad, precio):
        self.codigo = codigo
        self.descripcion = descripcion
        self.cantidad = cantidad
        self.precio = precio

    def modificar(self, nueva_descripcion, nueva_cantidad, nuevo_precio):
        self.descripcion = nueva_descripcion
        self.cantidad = nueva_cantidad
        self.precio = nuevo_precio
```

Clase Inventario

Método init:

Recordemos: La clase Inventario representa un inventario de productos y tiene un constructor init que se llama cuando se crea una instancia de la clase. Al crear una instancia de la clase Inventario, se establece una conexión a la base de

datos y se crea un cursor que se puede utilizar para interactuar con la base de datos. Esto permite que la clase Inventario realice operaciones de base de datos, como agregar, modificar, consultar y eliminar productos.

```
def __init__(self):
    self.conexion = get_db_connection()
    self.cursor = self.conexion.cursor()
```

En el constructor init, se realiza lo siguiente:

self.conexion = get_db_connection(): Se establece la conexión a la base de datos utilizando la función **get_db_connection()**. Esta función devuelve una conexión a la base de datos SQLite.

self.cursor = self.conexion.cursor(): Se crea un objeto cursor a partir de la conexión a la base de datos. Un cursor es utilizado para ejecutar consultas y obtener resultados de la base de datos.

Método agregar_producto:

Este método permite agregar un nuevo producto al inventario. Recibe como parámetros el código, descripción, cantidad y precio del producto a agregar. En primer lugar, se realiza una verificación para determinar si ya existe un producto con el mismo código en la base de datos. Si se encuentra un producto con el mismo código, se imprime un mensaje de error indicando que ya existe un producto con ese código. En caso contrario, se crea una instancia de la clase Producto con los datos proporcionados y se inserta en la tabla productos de la base de datos utilizando una consulta SQL. Por último, se imprime un mensaje de éxito indicando que el producto ha sido agregado correctamente.

```
def agregar_producto(self, codigo, descripcion, cantidad, precio):
    producto_existente = self.consultar_producto(codigo)
    if producto_existente:
        return jsonify({'message': 'Ya existe un producto con ese código.'}), 400

    nuevo_producto = Producto(codigo, descripcion, cantidad, precio)
    self.cursor.execute("INSERT INTO productos VALUES (?, ?, ?, ?)", (codigo, descripcion, cantidad, precio))
    self.conexion.commit()
    return jsonify({'message': 'Producto agregado correctamente.'}), 200
```

Usaremos la función fetch desde el frontend para acceder a los métodos de la API que están alojados en, por ejemplo, **PythonAnywhere**.

Recordemos que la función fetch es una API de JavaScript que permite realizar solicitudes HTTP desde el navegador web. Puedes utilizarla para enviar solicitudes POST al método agregar_producto de la API y enviar los datos del producto al servidor.

Veamos un ejemplo de cómo podríamos usar fetch para realizar la llamada al método **agregar_producto** desde el frontend:

```
const url = 'https://tu-domino-en-pythonanywhere.com/productos';
const data = {
  codigo: 4,
  descripcion: 'Producto 4',
  cantidad: 20,
  precio: 49.99
};

fetch(url, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data)
})
  .then(response => {
    if (response.ok) {
      console.log('Producto agregado correctamente');
    } else {
      console.log('Error al agregar el producto');
    }
  })
  .catch(error => {
    console.error('Error de conexión:', error);
  });
```

debes reemplazar '**https://tu-domino-en-pythonanywhere.com'** con la URL de tu aplicación alojada en PythonAnywhere. Asegúrate de que el dominio y la ruta sean correctos según tu configuración.

Al ejecutar este código en el navegador, se realizará una solicitud POST a la URL especificada, enviando los datos del producto en el cuerpo de la solicitud en formato JSON. Luego, la API procesará la solicitud y devolverá una respuesta. Dependiendo de si la operación es exitosa o no, el código en el bloque then manejará la respuesta correspondiente.

Es importante tener en cuenta que, al realizar solicitudes desde el frontend a un servidor externo, debes asegurarte de que la API tenga habilitados los encabezados CORS (Cross-Origin Resource Sharing) para permitir solicitudes desde un dominio diferente al de la API. De lo contrario, es posible que encuentres restricciones de seguridad que bloqueen las solicitudes.

response.ok es una propiedad booleana de la respuesta devuelta por la función fetch. Indica si la solicitud HTTP se realizó con éxito o no.

Cuando se realiza una solicitud HTTP, el servidor devuelve una respuesta con un código de estado HTTP. Un código de estado en el rango de 200 a 299 se considera exitoso, lo que significa que la solicitud se ha realizado correctamente. Por ejemplo, el código de estado 200 indica una respuesta exitosa.

Cuando response.ok es true, significa que la respuesta tiene un código de estado exitoso (en el rango de 200 a 299). En cambio, cuando response.ok es false, significa que la respuesta tiene un código de estado que indica un error, como un código de estado 400 (solicitud incorrecta) o un código de estado 500 (error interno del servidor).

En el ejemplo anterior, se utiliza response.ok para determinar si la solicitud de agregar un producto fue exitosa. Si response.ok es true, se muestra el mensaje "Producto agregado correctamente". De lo contrario, si response.ok es false, se muestra el mensaje "Error al agregar el producto".

Resumiendo, es una forma conveniente de verificar el estado de la respuesta HTTP sin tener que analizar el código de estado directamente.

Método consultar_producto:

Su objetivo es buscar un producto en la base de datos según el código proporcionado y devolver una instancia de la clase Producto si se encuentra, o False si no se encuentra.
Ejecuta una consulta SQL para buscar un producto en la base de datos según el código proporcionado. Si se encuentra, se crea una instancia de la clase Producto con los datos recuperados y se devuelve. De lo contrario, se devuelve False.

```
def consultar_producto(self, codigo):
    self.cursor.execute("SELECT * FROM productos WHERE codigo = ?", (codigo,))
    row = self.cursor.fetchone()
    if row:
        codigo, descripcion, cantidad, precio = row
        return Producto(codigo, descripcion, cantidad, precio)
    return None
```

Descripción del método consultar_producto paso a paso:

- **self.cursor.execute("SELECT * FROM productos WHERE codigo = ?", (codigo,)):** Ejecuta una consulta SQL utilizando el cursor para seleccionar todas las columnas de la tabla "productos" donde el código coincide con el valor proporcionado. La consulta utiliza un parámetro de marcador de posición (?) para evitar problemas de seguridad y el valor del código se pasa como una tupla en el segundo argumento de la función execute.
- **row = self.cursor.fetchone():** Recupera la primera fila de resultados de la consulta realizada. El método **fetchone()** devuelve una tupla que contiene los valores de las columnas seleccionadas.
- Si row es verdadero (es decir, se encontró una fila que coincide con el código), se extraen los valores de codigo, descripcion, cantidad y precio de la tupla row, y luegose crea una instancia de la clase Producto utilizando los valores extraídos y se devuelve como resultado de la función .
- Si no se encontró ninguna fila que coincida con el código, se devuelve False.

Método modificar_producto:

El método modificar_producto también forma parte de la clase Inventario. Su propósito es modificar los datos de un producto existente en la base de datos, así como en la instancia correspondiente de la clase Producto

```
def modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio):
    producto = self.consultar_producto(codigo)
```

```
    if producto:
        producto.modificar(nueva_descripcion, nueva_cantidad, nuevo_precio)
        self.cursor.execute("UPDATE productos SET descripcion = ?, cantidad = ?, precio = ? WHERE codigo = ?",
                            (nueva_descripcion, nueva_cantidad, nuevo_precio, codigo))
        self.conexion.commit()
        return jsonify({'message': 'Producto modificado correctamente.'}), 200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

El método modificar_producto() verifica la existencia del producto en el inventario, lo modifica tanto en memoria como en la base de datos y devuelve una respuesta JSON con el resultado de la operación.

Método listar_productos:

El método listar_productos recupera todos los productos de la base de datos.

```
def listar_productos(self):
    self.cursor.execute("SELECT * FROM productos")
    rows = self.cursor.fetchall()
    productos = []
    for row in rows:
        codigo, descripcion, cantidad, precio = row
        producto = {'codigo': codigo, 'descripcion': descripcion, 'cantidad': cantidad, 'precio': precio}
        productos.append(producto)
    return jsonify(productos), 200
```

- No recibe ningún parámetro, ya que simplemente obtiene los productos existentes en el inventario.
- Ejecuta la consulta SQL **"SELECT * FROM productos"** para obtener todos los registros de la tabla productos en la base de datos.
- Utiliza el método **fetchall()** para obtener todas las filas resultantes de la consulta.
- Itera sobre cada fila y extrae los valores de codigo, descripcion, cantidad y precio.
- Crea un diccionario producto con los valores extraídos y lo agrega a la lista productos.
- Finalmente, devuelve una respuesta JSON utilizando jsonify(productos), donde la lista de productos se convierte en un objeto JSON. El código de estado HTTP 200 se utiliza para indicar una respuesta exitosa.

En resumen, el método listar_productos() obtiene todos los productos almacenados en el inventario, los transforma en una lista de diccionarios y devuelve esa lista como una respuesta JSON junto con un código de estado HTTP 200.

Método eliminar_producto:

El método eliminar_producto() se utiliza para eliminar un producto del inventario. Recibe el código de un producto, lo elimina de la base de datos si existe y devuelve una respuesta JSON con un mensaje indicando el resultado de la operación y el código de estado HTTP correspondiente.

```
def eliminar_producto(self, codigo):
    self.cursor.execute("DELETE FROM productos WHERE codigo = ?", (codigo,))
    if self.cursor.rowcount > 0:
        self.conexion.commit()
        return jsonify({'message': 'Producto eliminado correctamente.'}), 200
    return jsonify({'message': 'Producto no encontrado.'}), 404
```

- Recibe un parámetro codigo que representa el código del producto que se desea eliminar.
- Llama al método consultar_producto(codigo) para verificar si el producto existe en el inventario.
- Si se encuentra el producto, ejecuta la consulta SQL **"DELETE FROM productos WHERE codigo = ?"** para eliminar el registro correspondiente al código proporcionado.
- Verifica si se ha eliminado algún registro utilizando la propiedad rowcount del cursor. Si se ha eliminado al menos un registro, se muestra el mensaje "Producto eliminado" y se realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit(). En caso contrario, se muestra el mensaje "Producto no encontrado".

- Devuelve una respuesta JSON utilizando jsonify() con un diccionario que contiene el mensaje correspondiente ("Producto eliminado" o "Producto no encontrado") y el código de estado HTTP apropiado: 200 si el producto fue eliminado exitosamente o 404 si el producto no se encontró.

Clase Carrito

Método init:

El método init es el constructor de la clase Carrito. Se llama automáticamente cuando se crea una nueva instancia de la clase Carrito. Su objetivo es inicializar los atributos de la clase. Establece la conexión a la base de datos, crea un cursor y inicializa el atributo items como una lista vacía.

```
def __init__(self):
    self.conexion = get_db_connection()
    self.cursor = self.conexion.cursor()
    self.items = []
```

No posee cambios destacables respecto de la versión anterior.

Método agregar:

El método agregar(self, codigo, cantidad, inventario) de la clase Carrito se utiliza para agregar un producto al carrito de compras. Recibe el código y la cantidad de un producto, verifica su disponibilidad en el inventario, actualiza la cantidad en el carrito y en el inventario, y devuelve un valor booleano para indicar si el producto se ha agregado correctamente al carrito o no.

```
def agregar(self, codigo, cantidad, inventario):
    producto = inventario.consultar_producto(codigo)
    if producto is None:
        return jsonify({'message': 'El producto no existe.'}), 404
    if producto.cantidad < cantidad:
        return jsonify({'message': 'Cantidad en stock insuficiente.'}), 400

    for item in self.items:
        if item.codigo == codigo:
            item.cantidad += cantidad
            self.cursor.execute("UPDATE productos SET cantidad = cantidad - ? WHERE codigo = ?",
                                (cantidad, codigo))
            self.conexion.commit()
            return jsonify({'message': 'Producto agregado al carrito correctamente.'}), 200

    nuevo_item = Producto(codigo, producto.descripcion, cantidad, producto.precio)
    self.items.append(nuevo_item)
    self.cursor.execute("UPDATE productos SET cantidad = cantidad - ? WHERE codigo = ?",
                        (cantidad, codigo))
    self.conexion.commit()
    return jsonify({'message': 'Producto agregado al carrito correctamente.'}), 200
```

- Recibe tres parámetros: codigo, que representa el código del producto a agregar, cantidad, que indica la cantidad de unidades del producto a agregar, y inventario, que es una instancia de la clase Inventario utilizada para consultar los productos.
- Llama al método consultar_producto(codigo) del inventario para verificar si el producto existe en el inventario.
- Si el producto no existe, muestra el mensaje "El producto no existe" y devuelve False para indicar que no se pudo agregar el producto al carrito.
- Si el producto existe, verifica si la cantidad solicitada está disponible en el inventario. Si la cantidad en el inventario es menor a la cantidad solicitada, muestra el mensaje "Cantidad en stock insuficiente" y devuelve False.
- Recorre los elementos del carrito de compras (self.items) para verificar si el producto ya está en el carrito.
- Si el producto ya está en el carrito, actualiza la cantidad del producto sumando la cantidad solicitada a la cantidad existente.

- Realiza una consulta SQL para actualizar la cantidad en el inventario, restando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit().
- Devuelve True para indicar que el producto se ha agregado exitosamente al carrito.
- Si el producto no está en el carrito, crea una nueva instancia de la clase Producto con los datos del producto obtenidos del inventario.
- Añade el nuevo producto al carrito (self.items).
- Realiza una consulta SQL para actualizar la cantidad en el inventario, restando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit().
- Devuelve True para indicar que el producto se ha agregado exitosamente al carrito.

Método quitar:

El método quitar(self, codigo, cantidad, inventario) de la clase Carrito se utiliza para quitar un producto del carrito de compras. Recibe el código y la cantidad de un producto, busca el producto en el carrito, actualiza la cantidad en el carrito y en el inventario, y devuelve un valor booleano para indicar si el producto se ha quitado correctamente del carrito o no. Además, maneja la verificación de la cantidad solicitada para quitar, evitando que se quiten más unidades de las disponibles en el carrito.

```
def quitar(self, codigo, cantidad, inventario):
    for item in self.items:
        if item.codigo == codigo:
            if cantidad > item.cantidad:
                return jsonify({'message': 'Cantidad a quitar mayor a la cantidad en el carrito.'}), 400
            item.cantidad -= cantidad
            if item.cantidad == 0:
                self.items.remove(item)
            self.cursor.execute("UPDATE productos SET cantidad = cantidad + ? WHERE codigo = ?",
                                (cantidad, codigo))
            self.conexion.commit()
            return jsonify({'message': 'Producto quitado del carrito correctamente.'}), 200

    return jsonify({'message': 'El producto no se encuentra en el carrito.'}), 404
```

- Recibe tres parámetros: codigo, que representa el código del producto a quitar, cantidad, que indica la cantidad de unidades del producto a quitar, y inventario, que es una instancia de la clase Inventario utilizada para consultar los productos.
- Recorre los elementos del carrito de compras (self.items) para buscar el producto que se desea quitar.
- Si encuentra el producto en el carrito, verifica si la cantidad solicitada para quitar es mayor a la cantidad actual en el carrito. Si es así, muestra el mensaje "Cantidad a quitar mayor a la cantidad en el carrito" y devuelve False.
- Actualiza la cantidad del producto en el carrito restando la cantidad solicitada.
- Realiza una consulta SQL para actualizar la cantidad en el inventario, sumando la cantidad solicitada al producto correspondiente en la base de datos.
- Realiza la confirmación de la transacción en la base de datos mediante self.conexion.commit().
- Devuelve True para indicar que el producto se ha quitado exitosamente del carrito.
- Si no encuentra el producto en el carrito, muestra el mensaje "El producto no se encuentra en el carrito" y devuelve False.

Método mostrar:

El método mostrar(self) de la clase Carrito se utiliza para mostrar el contenido del carrito de compras. Transforma los productos en el carrito en un formato JSON estructurado y los devuelve como respuesta HTTP junto con el código 200, lo que permite visualizar el contenido del carrito en el frontend de manera legible y fácil de procesar.

```
def mostrar(self):
    productos_carrito = []
    for item in self.items:
        producto = {'codigo': item.codigo, 'descripcion': item.descripcion, 'cantidad': item.cantidad,
                    'precio': item.precio}
        productos_carrito.append(producto)
    return jsonify(productos_carrito), 200
```

- No recibe ningún parámetro.
- Crea una lista vacía llamada productos_carrito para almacenar los productos del carrito en formato JSON.
- Recorre cada elemento del carrito de compras (self.items).
- Para cada producto en el carrito, crea un diccionario con las claves "codigo", "descripcion", "cantidad" y "precio" que representan los atributos del producto.
- Agrega el diccionario del producto a la lista productos_carrito.
- Retorna una respuesta JSON utilizando la función jsonify de Flask, que convierte la lista de productos (productos_carrito) en formato JSON.
- Además, se devuelve el código de estado HTTP 200, indicando que la solicitud se ha procesado correctamente.

Ejecución del código

El código completo que hemos elaborado está estructurado de la siguiente manera:

- Importamos las bibliotecas necesarias, incluyendo Flask y otros módulos relacionados.
- Creamos una instancia de la clase Flask y la asignamos a la variable app. Esta instancia representa nuestra aplicación Flask.
- Definimos las rutas y los métodos asociados a cada ruta utilizando los decoradores @app.route(). Estos decoradores especifican la URL y los métodos HTTP permitidos para cada ruta.
- Implementamos las funciones correspondientes a cada ruta. Estas funciones se ejecutan cuando se accede a las rutas específicas de la API.
- Dentro de estas funciones, realizamos las operaciones necesarias, como agregar, modificar o eliminar productos del inventario o del carrito. Utilizamos las clases Carrito e Inventario que hemos definido para realizar estas operaciones.
- Utilizamos los métodos jsonify() de Flask para convertir los datos en formato JSON y devolverlos como respuesta a las solicitudes.

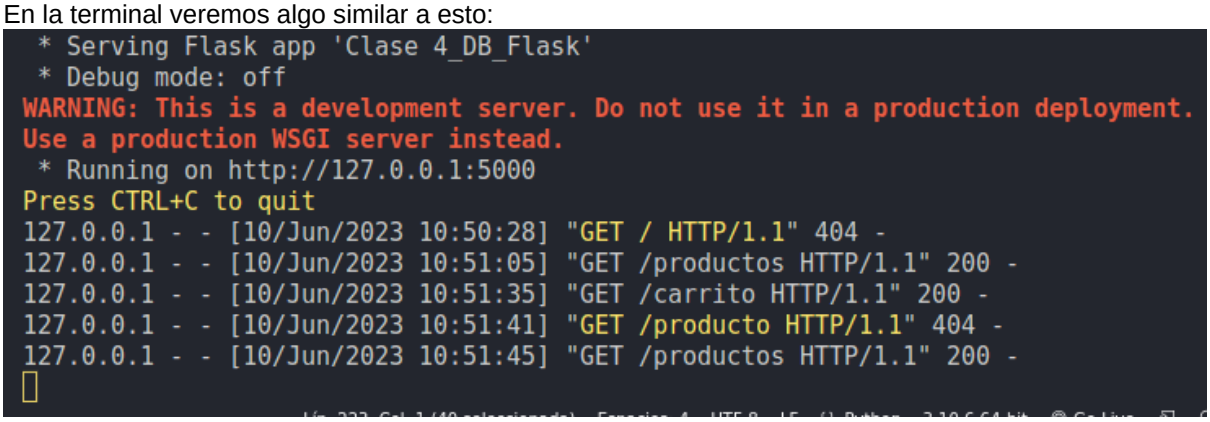
Finalmente, fuera de las funciones de ruta, agregamos las siguientes líneas de código:

```
# Finalmente, si estamos ejecutando este archivo, lanzamos app.
if __name__ == '__main__':
    app.run()
```

Estas líneas se encargan de iniciar el servidor de Flask. El condicional **if name == 'main':** asegura que el servidor solo se ejecute si el archivo se ejecuta directamente (no cuando se importa como módulo).

Cuando ejecutamos el archivo, Flask se pone en marcha y escucha las solicitudes entrantes en el servidor local. El servidor se inicia en la dirección **http://localhost:5000/**, donde localhost es la dirección IP del servidor local y 5000 es el número de puerto por defecto utilizado por Flask. Puedes acceder a la API a través de esta URL en tu navegador o desde el frontend para realizar las solicitudes HTTP y obtener las respuestas correspondientes.

En la terminal veremos algo similar a esto:



Con esto, finalizamos la implementación de la API.
Resta subir la aplicación a un servidor como PythonAnywhere, y luego implementar un frontend que utiluce los servicios de esta API.