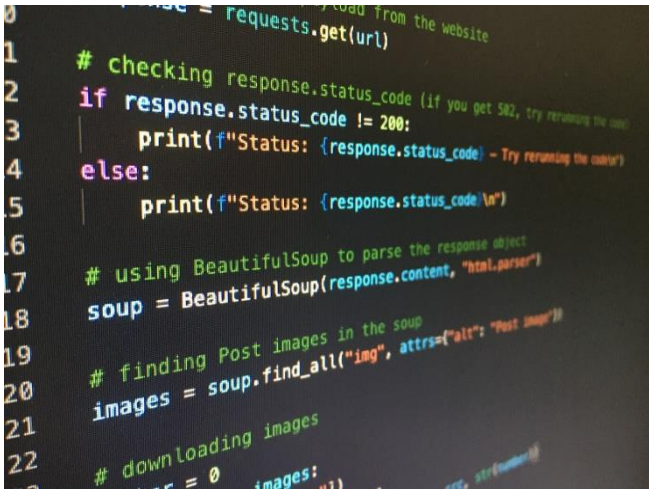


PROYECTO CRUD

TPO FINAL

HTML - CSS - JAVASCRIPT - VUE
BASES DE DATOS - PYTHON - FLASK

- Desarrollo de clases y objetos
- Creación de la base de datos
- Implementación de la API en Flask
- Despliegue en servidor PythonAnywhere
- Codificación del Front-End

Codo a Codo
2023

Agencia de Aprendizaje a lo largo de la vida

INDICE

PROYECTO CRUD - TPO FINAL

HTML - CSS - JAVASCRIPT - VUE

BASES DE DATOS - PYTHON - FLASK

DESCRIPCIÓN GENERAL DEL PROYECTO

El proyecto se trata de un sistema que permite administrar una base de datos de productos, implementado una API en Python utilizando el framework Flask. Las operaciones que se realizarán en este proyecto es dar de alta, modificar, eliminar y listar los productos, operaciones que podrán hacer los usuarios a través de una página Web.
Como ejemplo trabajaremos con una empresa de venta de **artículos de computación** que ofrece sus productos a través de una Web.

Aquí hay un resumen de las principales características y funcionalidades del proyecto:

Gestión de productos:

- Agregar un nuevo producto al catálogo.
- Modificar la información de un producto existente en el catálogo.
- Eliminar un producto del catálogo.
- Consultar información de un producto por su código.

Persistencia de datos:

- Los datos de los productos se almacenan en una base de datos SQL.
- Se utiliza una conexión a la base de datos para realizar operaciones CRUD en los productos.
- El código proporcionado incluye las clases Catálogo, que representa la estructura y funcionalidad relacionada con el catálogo de productos. Además, se define una serie de rutas en Flask para manejar las solicitudes HTTP relacionadas con la gestión de productos.

Se implementan desde cero el backend y el frontend. En el caso del backend, el proyecto va "evolucionando", comenzando en el desarrollo de las funciones que se necesitan para manipular los productos utilizando arreglos en memoria, luego se modifica para utilizar objetos, más tarde se gestiona la persistencia de los datos utilizando una base de datos SQL, después se implementa la API en Flask y se aloja el script Python en un servidor, y por último se crea un frontend básico para interactuar con los datos desde el navegador, a través dela API creada.

El proyecto se divide en seis etapas:

1) **Desarrollo de arreglos y funciones:** Implementar un CRUD de productos utilizando arreglos y funciones.
2) **Conversión a clases y objetos:** Convertir las funciones vistas en objetos y clases.
3) **Creación de la base de datos SQL:** Utilizar como almacenamiento una base de datos SQL.
4) **Implementación de la API en Flask:** A través de este framework implementar una API que permita ser consumida desde el front.
5) **Codificación del Front-End:** Vistas que permitan realizar las operaciones del CRUD.
6) **Despliegue en servidor PythonAnywhere:** Hosting para aplicaciones web escritas en Python.

ETAPA 1: DESARROLLO DE ARREGLOS Y FUNCIONES

Objetivo: Implementar un CRUD de productos utilizando arreglos y funciones.

Funciones para gestionar un arreglo con datos de productos

Escribimos una serie de funciones para crear una pequeña app que maneje un arreglo que contenga diccionarios con los datos de los productos.
Los diccionarios tienen las siguientes claves:

- **codigo:** un número entero que sirve como identificación única para cada producto.
- **descripcion:** una cadena de texto que describe el producto, como su nombre o modelo.
- **cantidad:** otro número entero que indica cuántas unidades de este producto están disponibles en el inventario.
- **precio:** un número decimal que representa el precio de venta del producto.
- **imagen:** un texto que contiene el nombre de la imagen relacionada con el producto (esto podría ser útil en una aplicación de comercio electrónico).
- **proveedor:** un número entero que identifica al proveedor del producto.

Nuestras funciones harán lo siguiente:
✓ Agregar un producto al arreglo
✓ Consultar un producto a partir de su código
✓ Modificar los datos de un producto a partir de su código
✓ Obtener un listado de los productos que existen en el arreglo.
✓ Eliminar un producto del arreglo.

Antes de definir las funciones haremos lo siguiente:

```
# Definimos una lista de diccionarios para almacenar los productos.
productos = []
```

A continuación, explicaremos en detalle cada una de las funciones:

agregar_producto(codigo, descripcion, cantidad, precio, imagen, proveedor)

Esta función tiene la tarea de agregar un nuevo producto a una lista llamada **productos**, siempre que no exista un producto con el mismo código previamente. Esto asegura la unicidad de productos en función de su código numérico. Veamos cómo funciona:

Parámetros:
- **codigo:** int, código numérico del producto.
- **descripcion:** str, descripción alfabética del producto.
- **cantidad:** int, cantidad en stock del producto.
- **precio:** float, precio de venta del producto.
- **imagen:** str, nombre de la imagen del producto.
- **proveedor:** int, número de proveedor del producto.

Retorna:
- **Valor booleano: True** si el producto se agregó exitosamente al arreglo y **False** si ya existe un producto con el mismo código y no se agrega el nuevo producto.

```python
def agregar_producto(codigo, descripcion, cantidad, precio, imagen,
proveedor):

    if consultar_producto(codigo):
        return False

    nuevo_producto = {
        'codigo': codigo,
        'descripcion': descripcion,
        'cantidad': cantidad,
        'precio': precio,
        'imagen': imagen,
        'proveedor': proveedor
    }
    productos.append(nuevo_producto)
    return True
```

Explicación del código:

1. **Verificamos su existencia:** Verificamos si ya existe un producto con el mismo código, utilizando la función **consultar_producto**. Si un producto con el mismo código se encuentra en el arreglo, se evita agregar el nuevo producto y se retorna ***False***. Si no se encuentra un producto con el mismo código, se procede a crear un diccionario que contiene los datos del nuevo producto.
2. **Creamos un Producto:** En caso de no existir el nuevo producto la función toma estos parámetros y crea un nuevo producto. Esta representación del producto es un "diccionario" de datos[1]. Las claves de este diccionario son '**codigo**', '**descripcion**', '**cantidad**', '**precio**', '**imagen**' y '**proveedor**'. Cada clave se asocia a su valor correspondiente, que se toma de los parámetros de la función.
3. **Agregamos el Producto:** Una vez que tenemos el diccionario del producto con todos los detalles, la función agrega este producto a la lista llamada **productos**, que almacena todos los productos disponibles.
4. **Valor de Retorno:** La función devuelve ***True*** como resultado. Esto es una señal para indicar que el proceso de agregar el producto se completó con éxito.

En resumen: La función verifica la existencia del producto y, en caso de no existir en la lista, toma la información pasada por parámetro, la organiza en un formato específico (el diccionario) y agrega ese producto a la lista.

consultar_producto(codigo)

Esta función se encarga de **buscar y recuperar** la información de un producto de acuerdo a su código. Esto es especialmente útil en una aplicación de gestión de inventario para encontrar detalles específicos de un producto, además de que le permite a la función **agregar_producto** verificar si un producto no fue agregado previamente a la lista, evitando así la duplicación de registros. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto. Se utiliza para identificar un producto en particular. La función buscará un producto que coincida con ese código.

[1] Un diccionario es una estructura de datos que permite almacenar varios valores asociados a claves.

Retorna:

- Si el producto fue encontrado retorna un **diccionario con los datos del producto**. Si no se encontró retorna el valor booleano ***False***.

```
def consultar_producto(codigo):
    for producto in productos:
        if producto['codigo'] == codigo:
            return producto
    return False
```

Explicación del código:

Proceso de Búsqueda con bucle: La función comienza recorriendo **productos,** la lista que almacena todos los productos disponibles en la aplicación. El bucle ***for*** examina cada producto, verificando si el valor de la clave '**codigo**' en el diccionario del producto coincide con el valor proporcionado en el parámetro codigo. Esto es fundamental para identificar el producto correcto. Este proceso de búsqueda condiuce a dos resultados:

a. **Producto Encontrado:** Si se encuentra un producto con el código coincidente, la función regresa el diccionario que representa ese producto. Esto significa que la función proporciona todos los detalles de ese producto en particular, como su descripción, cantidad en stock, precio, imagen, y proveedor. Este diccionario se puede utilizar posteriormente en el programa para mostrar o manipular la información del producto.
b. **Producto no Encontrado:** Si el bucle finaliza sin encontrar el producto deseado, la función regresa ***False***. Esto sirve como indicador de que no se ha encontrado un producto con el código proporcionado.

En resumen: Esta función es un mecanismo esencial para recuperar información detallada de un producto utilizando su código único como referencia. A través de un bucle y una comparación de códigos, la función encuentra el producto correcto y devuelve todos sus detalles en forma de un diccionario. De esta manera, los usuarios pueden acceder a la información de un producto en base a su código, lo que facilita la gestión y visualización de datos en la aplicación.

modificar_producto(codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor)

Esta función se encarga de actualizar los datos de un producto existente en función de su código. Esto es fundamental en una aplicación de gestión de inventario, ya que permite realizar cambios en la descripción, cantidad en stock, precio, imagen y proveedor de un producto. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto. Se utiliza para identificar el producto que se desea modificar.
- **nueva_descripcion:** str, contiene la nueva descripción que se asignará al producto.
- **nueva_cantidad:** int, indica la nueva cantidad en stock del producto.
- **nuevo_precio:** float, representa el nuevo precio de venta del producto.
- **nueva_imagen:** str, contiene el nombre de la nueva imagen del producto.
- **nuevo_proveedor:** int, identifica al nuevo proveedor del producto.

Retorna:

- **Valor booleano: True** si el producto fue modificado exitosamente y **False** si no pudo realizarse la modificación.

```python
def modificar_producto(codigo, nueva_descripcion, nueva_cantidad,
nuevo_precio, nueva_imagen, nuevo_proveedor):
    for producto in productos:
        if producto['codigo'] == codigo:
            producto['descripcion'] = nueva_descripcion
            producto['cantidad'] = nueva_cantidad
            producto['precio'] = nuevo_precio
            producto['imagen'] = nueva_imagen
            producto['proveedor'] = nuevo_proveedor
            return True
    return False
```

Explicación del código:
Proceso de Modificación con bucle: La función comienza recorriendo **productos,** la lista que almacena todos los productos disponibles en la aplicación. El bucle ***for*** examina cada producto, verificando si el valor de la clave '**codigo**' en el diccionario del producto coincide con el valor proporcionado en el parámetro codigo. Esto es fundamental para identificar el producto correcto que se va a modificar. Este proceso de búsqueda condiuce a dos resultados:

a. **Producto Encontrado y salida exitosa:** Si se encuentra un producto con el código coincidente, la función procede a actualizar los valores de sus claves en el diccionario. Los valores proporcionados en los parámetros nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen y nuevo_proveedor se asignan a las claves correspondientes en el diccionario del producto, esto actualiza efectivamente los datos del producto. Después de realizar la modificación, la función devuelve ***True*** como indicación de que la operación ha tenido éxito.
b. **Producto no Encontrado:** Si el bucle finaliza sin encontrar el producto deseado, la función regresa ***False***. Esto significa que el producto no existe en la lista y, por lo tanto, no se puede modificar.

En resumen: Esta función es una parte esencial de la aplicación de gestión de inventario, ya que permite a los usuarios realizar cambios en los datos de los productos existentes. Al proporcionar un código de producto, junto con los nuevos valores para descripción, cantidad, precio, imagen y proveedor, los usuarios pueden mantener actualizado su inventario de productos de manera efectiva y eficiente.

listar_productos()

Esta función tiene como objetivo mostrar en pantalla un listado de los productos almacenados en la aplicación de gestión de inventario. Resulta de utilidad para que los usuarios puedan ver una vista general de todos los productos disponibles y sus respectivos detalles. Veamos cómo funciona:

Parámetros: No requiere.
Retorna: No retorna valores.

```python
def listar_productos():
    print("-" * 50)
    for producto in productos:
        print(f"Código.....: {producto['codigo']}")
        print(f"Descripción: {producto['descripcion']}")
        print(f"Cantidad...: {producto['cantidad']}")
```

```
        print(f"Precio.....: {producto['precio']}")
        print(f"Imagen.....: {producto['imagen']}")
        print(f"Proveedor..: {producto['proveedor']}")
        print("-" * 50)
```

Explicación del código:

1. **Separador visual de inicio:** La función comienza imprimiendo una línea de guiones (-) repetida 50 veces, lo que crea una especie de separación visual en la pantalla para distinguir entre los productos.
2. **Bucle de Productos y visualización de datos:** Luego, utiliza un bucle ***for*** para recorrer la lista **productos**, que contiene todos los productos de la aplicación. Por cada iteración del bucle, se procesa un producto y se imprimen sus detalles en la pantalla:
   - **Código**: código numérico único de identificación del producto.
   - **Descripción**: descripción alfabética que contiene una breve información sobre el producto.
   - **Cantidad**: cantidad en stock del producto, ejemplares disponibles.
   - **Precio**: precio de venta del producto en formato decimal.
   - **Imagen**: nombre de la imagen asociada al producto, si está disponible.
   - **Proveedor**: número identificador del proveedor del producto.
3. **Separador visual de cierre:** Después de mostrar los datos de cada producto, se imprime nuevamente una línea de guiones (-). Esto crea una división visual entre los productos y facilita la lectura y comprensión de la lista.

En resumen: Esta función es esencial para la aplicación de gestión de inventario, ya que permite a los usuarios obtener una vista completa de todos los productos almacenados. Al proporcionar detalles clave, como código, descripción, cantidad y precio, los usuarios pueden tomar decisiones informadas sobre la gestión y compra de productos. La función es especialmente útil cuando se necesita una visión general rápida de todo el inventario disponible.

eliminar_producto(codigo)

Esta función tiene la responsabilidad de eliminar un producto específico de la lista de productos en la aplicación de gestión de inventario. Esto puede ser necesario cuando un producto ya no está disponible o cuando se comete un error al ingresar información incorrecta. Veamos cómo funciona:

Parámetros:

- **codigo:** int, código numérico del producto.

Retorna:

- **Valor booleano: True** si el producto se eliminó exitosamente del arreglo y **False** si no fue posible eliminar el producto.

```
def eliminar_producto(codigo):
    for producto in productos:
        if producto['codigo'] == codigo:
            productos.remove(producto)
            return True
    return False
```

Explicación del código:

1. **Búsqueda de Producto:** La función comienza tomando un parámetro, codigo, que representa el código numérico del producto que se desea eliminar. Este código es utilizado para identificar de manera única el producto que se desea eliminar.
2. **Recorrido de la Lista de Productos:** Luego, se inicia un bucle ***for*** que recorre la lista de productos, que contiene todos los productos almacenados en la aplicación.
3. **Verificación de Código:** En cada iteración del bucle, se verifica si el código del producto actual (contenido en el diccionario **producto**) coincide con el código proporcionado como argumento. Si se encuentra un producto con el código correspondiente, se procede a la eliminación.
4. **Eliminación del Producto:** Cuando se encuentra un producto con el código coincidente, se utiliza el método ***remove*** para eliminar ese producto específico de la lista de productos. Esto se logra al pasar el objeto producto como argumento al método ***remove***.
5. **Finalización del Bucle:** Dado que los códigos son únicos y no deberían haber duplicados, una vez que se elimina el producto, la función sale del bucle de búsqueda utilizando ***return True***. Esto indica que se ha encontrado y eliminado un producto con éxito.
6. **Producto no Encontrado:** Si el bucle de búsqueda se completa sin encontrar un producto que coincida con el código proporcionado, la función regresa ***False***. Esto indica que no se ha eliminado ningún producto porque no se encontró un producto con el código dado.

En resumen: Esta función es útil para mantener actualizada la lista de productos y permitir a los usuarios eliminar productos no deseados o incorrectos de la aplicación de gestión de inventario. Al eliminar productos, se asegura que la información sea precisa y refleje con precisión el inventario disponible.

Ejemplo del uso de las funciones implementadas

```
# Agregamos productos a la lista:
agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500, 'teclado.jpg',
101)
agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg', 102)
agregar_producto(3, 'Monitor LCD 22 pulgadas', 15, 52500,
'monitor22.jpg', 103)
agregar_producto(4, 'Monitor LCD 27 pulgadas', 25, 78500,
'monitor27.jpg', 104)
agregar_producto(5, 'Pad mouse', 5, 500, 'padmouse.jpg', 105)
agregar_producto(3, 'Parlantes USB', 4, 2500, 'parlantes.jpg', 105) # No
es posible agregarlo, mismo código que el producto 3.

# Listamos todos los productos en pantalla
listar_productos()

# Consultar un producto por su código
cod_prod = int(input("Ingrese el código del producto: "))
producto = consultar_producto(cod_prod)
if producto:
    print(f"Producto encontrado: {producto['codigo']} -
{producto['descripcion']}")
else:
    print(f'Producto {cod_prod} no encontrado.')
```

```
# Modificar un producto por su código
modificar_producto(1, 'Teclado mecánico 62 teclas', 20, 34000,
'tecladomecanico.jpg', 106)

# Listamos todos los productos en pantalla
listar_productos()

# Eliminamos un producto del inventario
eliminar_producto(5)

# Listamos todos los productos en pantalla
listar_productos()
```

ETAPA 2: CONVERSIÓN A CLASES Y OBJETOS

El objetivo de esta etapa es convertir las funciones vistas en la etapa anterior en objetos y clases. Para ello vamos a adaptar el código desarrollado antes al paradigma de **objetos** en Python. Para ello, crearemos una **clase** llamada **Catálogo** que encapsulará los datos y las operaciones relacionadas con los productos, trabajadas anteriormente mediante funciones.
Las clases nos permitirán crear objetos para nuestro proyecto.

Clase CATÁLOGO

Definiremos una clase llamada `Catalogo`, que se utiliza para administrar un catálogo de productos. Cada producto en el catálogo se representa como un diccionario que contiene información sobre el producto, como su código, descripción, cantidad en stock, precio, imagen y proveedor. La clase `Catalogo` ofrece métodos para agregar, consultar, modificar, listar y eliminar productos en el catálogo, así como para mostrar los detalles de un producto específico.
El código para inicializar la clase es el siguiente:

```
class Catalogo:
    productos = []
```

Esta clase posee un **atributo de clase** llamado **productos**, una lista que almacena los productos. Al ser un atributo de clase es compartido por todas las instancias de la clase `**Catalogo**`. Esta lista se inicializa vacía y se llena luego con diccionarios que representan productos.
Veamos los métodos de la Clase `Catalogo`:

agregar_producto(self, codigo, descripcion, cantidad, precio, imagen, proveedor)

Este método permite agregar productos al catálogo asegurándose de que no haya duplicados en función de su código. Si el producto no existe previamente, se crea un nuevo diccionario para representar el producto y se agrega a la lista de productos. El método devuelve ***True*** si el producto se agrega exitosamente y ***False*** si ya existe un producto con el mismo código. Veamos cómo funciona:
Parámetros:

- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo**: Un número entero que representa el código numérico del producto.

- **descripcion**: Una cadena de texto que proporciona una descripción alfabética del producto.
- **cantidad**: Un número entero que indica la cantidad en stock del producto.
- **precio**: Un número decimal (punto flotante) que representa el precio de venta del producto.
- **imagen**: Una cadena de texto que especifica el nombre de la imagen del producto.
- **proveedor**: Un número entero que identifica al proveedor del producto.

Retorna:

- **Valor booleano: True** si el producto se agregó exitosamente al arreglo y **False** si ya existe un producto con el mismo código y no se agrega el nuevo producto.

```
    def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        if self.consultar_producto(codigo):
            return False

        nuevo_producto = {
            'codigo': codigo,
            'descripcion': descripcion,
            'cantidad': cantidad,
            'precio': precio,
            'imagen': imagen,
            'proveedor': proveedor
        }

        self.productos.append(nuevo_producto)
        return True
```

Explicación del código:

1. **Verificación de Duplicados:** Antes de agregar un nuevo producto, el método realiza una verificación para asegurarse de que no exista ya un producto con el mismo código. Esto se hace llamando al método **self.consultar_producto(codigo)**.
2. **Método `consultar_producto`:** Este método se utiliza para verificar si ya existe un producto en el catálogo con el mismo código. Si se encuentra un producto con el mismo código, esto significa que no se debe agregar el nuevo producto y se devuelve ***False*** para indicar que no se realizó la operación. De lo contrario, se continúa con la adición del nuevo producto.
3. **Creación del Nuevo Producto:** Si no se encuentra un producto con el mismo código, se procede a crear un nuevo producto en forma de diccionario. Los atributos del nuevo producto, como 'codigo', 'descripcion', 'cantidad', 'precio', 'imagen' y 'proveedor', se establecen según los valores proporcionados en los parámetros del método.
4. **Agregar el Nuevo Producto al Catálogo:** Una vez creado el diccionario que representa el nuevo producto, se agrega a la lista de productos del catálogo (**self.productos**) utilizando el método **append**. Esto aumenta la lista de productos con el nuevo producto.
5. **Valor de Retorno:** Finalmente, el método devuelve ***True*** para indicar que la operación se completó con éxito y que el producto se agregó al catálogo.

consultar_producto(self, codigo)

Este método se encarga de **buscar y recuperar** la información de un producto en el catálogo de acuerdo a su código. Si se encuentra un producto con el código proporcionado, se devuelve un diccionario con los datos del producto. Si no se encuentra ningún producto con ese código, se devuelve ***False*** para indicar que el producto no está en el catálogo. Este método le permite al

método **agregar_producto** verificar si un producto no fue agregado previamente al catálogo, evitando así la duplicación de registros. Veamos cómo funciona:

Parámetros:
- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se quiere consultar.

Retorna:
- Si el producto fue encontrado retorna un **diccionario con los datos del producto**. Si no se encontró retorna el valor booleano ***False***.

```
    def consultar_producto(self, codigo):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                return producto
        return False
```

Explicación del código:
1. **Búsqueda en el Catálogo**: El método recorre la lista de productos en el catálogo (**self.productos**) utilizando un bucle ***for***. Para cada producto en la lista, comprueba si el valor de la clave '**codigo**' en el diccionario del producto coincide con el código proporcionado como parámetro.
2. **Coincidencia de Códigos:** Si se encuentra un producto cuyo código coincide con el código proporcionado, se devuelve ese producto en forma de diccionario utilizando la instrucción ***return producto***.
3. **Producto No Encontrado:** Si el bucle de búsqueda finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False*** para indicar que el producto no se encontró en el catálogo.

modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor)

Este método permite actualizar los datos de un producto existente en el catálogo utilizando su código. Si se encuentra un producto con el código proporcionado, se actualizan sus datos y se devuelve ***True*** para indicar una modificación exitosa. Si no se encuentra ningún producto con ese código, se devuelve ***False*** para indicar que el producto no existe en el catálogo. Veamos cómo funciona:

Parámetros:
- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea modificar.
- **nueva_descripcion:** Una cadena de texto que representa la nueva descripción alfabética del producto.
- **nueva_cantidad:** Un número entero que representa la nueva cantidad en stock del producto.
- **nuevo_precio:** Un número de punto flotante que representa el nuevo precio de venta del producto.
- **nueva_imagen:** Una cadena de texto que representa la nueva imagen del producto.

- **nuevo_proveedor:** Un número entero que representa el nuevo número de proveedor del producto.

Retorna:
- **Valor booleano: True** si el producto fue modificado exitosamente y **False** si no pudo realizarse la modificación.

```
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                producto['descripcion'] = nueva_descripcion
                producto['cantidad'] = nueva_cantidad
                producto['precio'] = nuevo_precio
                producto['imagen'] = nueva_imagen
                producto['proveedor'] = nuevo_proveedor
                return True
        return False
```

Explicación del código:
1. **Búsqueda en el Catálogo**: El método recorre la lista de productos en el catálogo (**self.productos**) utilizando un bucle ***for***. Para cada producto en la lista, comprueba si el valor de la clave '**codigo**' en el diccionario del producto coincide con el código proporcionado como parámetro.
2. **Modificación de Datos:** Si se encuentra un producto cuyo código coincide con el código proporcionado, se actualizan los datos del producto con los nuevos valores proporcionados para descripción, cantidad, precio, imagen y proveedor.
3. **Resultado de la Modificación:** El método devuelve ***True*** para indicar que los datos del producto se han modificado con éxito.
4. **Producto No Encontrado:** Si el bucle de búsqueda finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False*** para indicar que el producto no existe en el catálogo y, por lo tanto, no se pueden realizar modificaciones.

listar_productos(self)

Este método permite mostrar en pantalla un listado detallado de todos los productos que se encuentran en el catálogo. Veamos cómo funciona:

Parámetros: No requiere.
Retorna: No retorna valores.

```
    def listar_productos(self):
        print("-" * 50)
        for producto in self.productos:
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 50)
```

Explicación del código:

1. **Separador visual de inicio:** Cuando el método se invoca, inmediatamente muestra una línea divisoria hecha de guiones para separar visualmente los distintos productos en el listado. Esto mejora la presentación en pantalla.
2. **Bucle de recorrido y visualización de datos:** Luego, utiliza un bucle ***for*** para recorrer la lista de productos en el catálogo (***self.productos***). Para cada producto en la lista, realiza las siguientes acciones:
   a. **Muestra de Datos:** El método imprime en pantalla información detallada sobre cada producto, incluyendo los siguientes datos:
      - **Código**: El código numérico del producto (`producto['codigo']`).
      - **Descripción:** La descripción alfabética del producto (`producto['descripcion']`).
      - **Cantidad:** La cantidad en stock del producto (`producto['cantidad']`).
      - **Precio:** El precio de venta del producto (`producto['precio']`).
      - **Imagen:** El nombre de la imagen asociada al producto (`producto['imagen']`).
      - **Proveedor:** El número de proveedor del producto (`producto['proveedor']`).
   b. **Separador visual de cierre:** Después de mostrar los datos de cada producto, se imprime nuevamente una línea de guiones (-). Esto crea una división visual entre los productos y facilita la lectura y comprensión de la lista.

El método ***listar_productos*** ofrece una representación visual clara de todos los productos presentes en el catálogo. Para cada producto, muestra sus datos clave en un formato estructurado. Esto facilita la revisión y gestión de los productos almacenados en el catálogo.

eliminar_producto(self, codigo)

Este método permite eliminar un producto específico del catálogo según su código. Veamos cómo funciona:

Parámetros:

- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea eliminar.

Retorna: No retorna valores.

```
    def eliminar_producto(self, codigo):
        for producto in self.productos:
            if producto['codigo'] == codigo:
                self.productos.remove(producto)
                return True
        return False
```

Explicación del código:

1. **Bucle de recorrido y comparación de códigos:** El método utiliza un bucle ***for*** utiliza para recorrer la lista de productos en el catálogo (***self.productos***). En cada iteración del bucle, compara el código del producto actual (`**producto['codigo']**`) con el código proporcionado como argumento (`**codigo**`).
2. **Eliminación del Producto:** Si el código del producto coincide con el código proporcionado, significa que se ha encontrado el producto que se desea eliminar. En este caso, el método elimina el producto de la lista del catálogo utilizando el método `**remove**` de las listas de Python.

3. **Valor de Retorno:** Una vez que se ha eliminado el producto, el método devuelve ***True*** para indicar que la operación se completó con éxito. Esto permite saber que el producto se eliminó correctamente.
4. **Producto No Encontrado:** Si el bucle de recorrido finaliza sin encontrar un producto con el código proporcionado, el método devuelve ***False***. Esto indica que el producto no existe en el catálogo y, por lo tanto, no se pudo eliminar.

El método ***eliminar_producto*** proporciona una forma efectiva de gestionar la eliminación de productos en el catálogo. Al especificar el código del producto que se desea eliminar, se elimina de la lista y se confirma el éxito de la operación mediante el valor de retorno. Esto facilita la gestión del catálogo y la eliminación de productos no deseados.

mostrar_producto(self, codigo)

Este método tiene como objetivo mostrar los datos de un producto específico del catálogo según su código. Veamos cómo funciona:

Parámetros:
- **self**: Este es un parámetro especial que hace referencia a la instancia de la clase **Catalogo**. Permite acceder a los atributos y otros métodos de la instancia.
- **codigo:** Un número entero que representa el código numérico del producto que se desea eliminar.

Retorna:
- **Valor booleano: True** si el producto se eliminó exitosamente del arreglo y **False** si no fue posible eliminar el producto.

```
    def mostrar_producto(self, codigo):
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 50)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 50)
        else:
            print("Producto no encontrado.")
```

Explicación del código:
1. **Consulta de Producto:** El método utiliza la función **consultar_producto(codigo)** de la misma clase Catalogo para buscar un producto con el código proporcionado. El resultado se almacena en la variable **producto**.
2. **Verificación de Existencia del Producto:** Se verifica si **producto** es diferente de *False*, lo que significa que se ha encontrado un producto con el código especificado. Si se encontró el producto, el método procede a mostrar sus detalles.
3. **Mostrar Datos del Producto:** Si se encuentra el producto, se muestra un bloque de información detallada sobre el producto en la pantalla. Esto incluye su código, descripción,

cantidad en stock, precio, imagen y proveedor. La presentación se realiza utilizando declaraciones `print`.

4. **Producto No Encontrado:** Si no se encuentra ningún producto con el código proporcionado, se muestra un mensaje que indica "Producto no encontrado".

El método ***mostrar_producto*** es útil para visualizar los detalles de un producto en el catálogo. Si el producto existe, se muestran todos sus atributos. Si no se encuentra el producto, se informa al usuario que el producto no se encuentra en el catálogo. Este método es una herramienta eficaz para acceder a información específica sobre productos en el catálogo de una manera organizada y legible.

Ejemplo del uso de las funciones implementadas

```
catalogo = Catalogo()
catalogo.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500,
'teclado.jpg', 101)
catalogo.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg',
102)
print()
print("Listado de productos:")
catalogo.listar_productos()
print()
print("Datos de un producto:")
catalogo.mostrar_producto(1)
catalogo.eliminar_producto(1)
print()
print("Listado de productos:")
catalogo.listar_productos()
```

ETAPA 3: CREACIÓN DE LA BASE DE DATOS SQL

Sistema gestor de bases de datos MySQL (persistencia de datos)

La clase "**Catalogo**" proporciona una interfaz sencilla para administrar productos en una base de datos MySQL. Cada método realiza una operación específica, como agregar, consultar, modificar, listar o eliminar productos. Esta implementación es útil para aquellos que deseen crear una aplicación de gestión de inventario o catálogo de productos utilizando Python y MySQL.
Es importante tener en cuenta que se requiere la instalación del paquete **mysql-connector-python** para utilizar esta clase y que se deben proporcionar credenciales válidas para acceder a una base de datos MySQL.
El paquete **mysql-connector-python** es un conector oficial de MySQL para Python. Se utiliza para conectarse y comunicarse con bases de datos MySQL desde aplicaciones Python.

Instalación de MySQL:

Para instalar **mysql-connector-python**, puedes utilizar **pip**, que es el administrador de paquetes de Python. Abre una terminal o línea de comandos y ejecuta el siguiente comando:

```
pip install mysql-connector-python
```

Este comando descargará e instalará el paquete **mysql-connector-python** y todas sus dependencias en tu entorno de Python.
Una vez instalado, puedes utilizar este paquete para conectar tu aplicación Python a una base de datos MySQL y realizar operaciones de base de datos, como consultas SQL, inserciones, actualizaciones y eliminaciones de registros.
Es importante mencionar que debes proporcionar credenciales válidas para conectarte a la base de datos, como el nombre de usuario, la contraseña, la dirección del servidor de la base de datos y el nombre de la base de datos a la que deseas acceder. El paquete **mysql-connector-python** facilita la conexión y la interacción con bases de datos MySQL desde Python, lo que lo hace útil para desarrollar aplicaciones que requieran almacenar y gestionar datos de manera persistente en bases de datos MySQL.

Introducción

El código proporcionado es una implementación de una clase llamada "Catalogo", que se utiliza para administrar un catálogo de productos almacenados en una base de datos MySQL. Este documento detalla la funcionalidad de cada método de la clase. Para acceder a los datos se utilizan algunos elementos que debemos conocer.

Conector:
Un "conector" se refiere a un conjunto de herramientas o una biblioteca que facilita la interacción entre un lenguaje de programación (como Python) y un sistema de gestión de bases de datos (como MySQL), manejando aspectos como la conexión, la ejecución de consultas y el manejo de transacciones.
En el contexto de la programación y las bases de datos, un "**conector**" es un software o una biblioteca que permite a un programa (como una aplicación escrita en Python) conectarse y comunicarse con un sistema de gestión de bases de datos (como MySQL). En nuestro código, el término "conector" se refiere específicamente a la biblioteca ***mysql.connector***, que proporciona funcionalidades para interactuar con bases de datos MySQL desde Python. Otras características son:

1. **Interfaz entre Python y MySQL:** ***mysql.connector*** es un controlador que proporciona una interfaz entre Python y una base de datos MySQL. Permite que tu aplicación Python ejecute operaciones de base de datos como consultas, actualizaciones y transacciones.
2. **Establecimiento de la Conexión:** El conector se utiliza para establecer una conexión con la base de datos MySQL. Esto se hace especificando los detalles necesarios como el host, el nombre de usuario, la contraseña y el nombre de la base de datos. En tu código, esto se realiza a través de:

```
self.conn = mysql.connector.connect(
    host=host,
    user=user,
    password=password,
    database=database
)
```

3. **Manejo de Sesiones de Base de Datos:** Una vez establecida la conexión, el conector administra la sesión de la base de datos, permitiéndote realizar operaciones como ejecutar comandos SQL, manejar transacciones y cerrar la conexión.
4. **Compatibilidad y Estándares:** Los conectores, como ***mysql.connector*** para Python, generalmente son compatibles con los estándares de la industria, como el API de Base de

Datos de Python (DB-API). Esto hace que sea más fácil para los desarrolladores trabajar con diferentes bases de datos de manera consistente.

5. **Gestión de Recursos y Errores:** El conector también gestiona aspectos importantes como el pooling de conexiones, el manejo de errores y excepciones, y la conversión entre tipos de datos de Python y SQL.
6. **Ejecución de Consultas y Recuperación de Resultados:** Aunque el trabajo de ejecutar consultas y recuperar resultados se realiza a través de los cursores, el conector es responsable de crear estos cursores y mantener la conexión que los respalda.

Cursor:
En el contexto de bases de datos, particularmente en nuestro código que utiliza MySQL con Python, un **"cursor"** es un objeto que se utiliza para interactuar con el sistema de gestión de la base de datos. Es fundamental para ejecutar consultas SQL y recuperar datos de la base de datos. Esto es lo que debes saber sobre los cursores:

1. **Intermediario entre Python y la Base de Datos:** El cursor actúa como un intermediario entre tu programa Python y la base de datos MySQL. Permite ejecutar comandos SQL (como SELECT, INSERT, UPDATE, DELETE) desde Python.
2. **Ejecución de Consultas:** Utilizas el cursor para ejecutar consultas SQL. Por ejemplo, *cursor.execute("SELECT * FROM tabla")* ejecuta una consulta SQL para seleccionar todos los registros de una tabla.
3. **Recuperación de Datos:** Después de ejecutar una consulta SELECT, el cursor puede usarse para obtener los resultados. Puedes iterar sobre el cursor o usar métodos como *fetchone()*, *fetchall()* para recuperar filas de la base de datos.
4. **Manejo de Transacciones:** El cursor es utilizado para manejar transacciones con la base de datos. Por ejemplo, después de insertar o actualizar datos, necesitas hacer un ***commit*** de la transacción para que los cambios se guarden permanentemente en la base de datos. En tu código, esto se hace mediante *self.conn.commit()*.
5. **Configuración del Cursor:** En tu código, el cursor se crea con la opción ***dictionary=True***, lo que significa que los resultados de las consultas se devolverán como diccionarios de Python en lugar de tuplas, permitiendo un acceso más fácil y legible a los datos por nombre de columna, en lugar de solo por índice.
6. **Cierre del Cursor:** Es una buena práctica cerrar el cursor con ***cursor.close()*** cuando ya no se necesita, para liberar recursos del sistema de gestión de la base de datos.

El **cursor** es esencial para ejecutar consultas SQL, manejar resultados y controlar transacciones con la base de datos MySQL.

Definición de la base de datos

Los datos de productos se almacenan en la base de datos MySQL que definimos a continuación.

Tabla de Productos (productos):

- **codigo:** Un número único que identifica cada producto.
- **descripcion:** Una cadena de texto que describe el producto.
- **cantidad:** Un número que representa la cantidad de productos disponibles en el inventario.
- **precio:** Un número que representa el precio del producto.
- **imagen_url:** Una cadena de texto que almacena la URL de la imagen del producto.
- **proveedor:** Un número que representa al proveedor del producto.

Crear la base de datos y sus tablas

Para crear la base de datos y las tablas en **XAMPP**, primero debes asegurarte de que el **servidor MySQL** esté en funcionamiento. Luego, puedes utilizar una herramienta como **phpMyAdmin** o ejecutar comandos SQL directamente en la consola de MySQL.
Puedes (casi siempre) acceder a phpMyAdmin desde http://127.0.1.1/phpmyadmin/
Aquí tienes un script SQL que puedes utilizar para crear la base de datos y las tablas, asegurándote de que las claves primarias se definan correctamente y que las tablas se creen si no existen:

```
-- Crear la base de datos si no existe
CREATE DATABASE IF NOT EXISTS miapp;
-- Usar la base de datos
USE miapp;
-- Crear la tabla de Productos si no existe
CREATE TABLE IF NOT EXISTS productos (
codigo INT,
descripcion VARCHAR(255) NOT NULL,
cantidad INT(4) NOT NULL,
precio DECIMAL(10, 2) NOT NULL,
imagen_url VARCHAR(255),
proveedor INT(2));
```

Clase Catalogo

La clase **Catalogo** tiene como utilidad principal gestionar un catálogo de productos almacenados en una base de datos MySQL. Proporciona una interfaz para realizar operaciones comunes en un catálogo de productos, como agregar nuevos productos, consultar información sobre productos, modificar productos existentes, listar todos los productos en el catálogo, eliminar productos y mostrar detalles de un producto en particular.
Esta clase facilita la administración de productos en una base de datos MySQL a través de los siguientes métodos:

Constructor: def __init__(self, host, user, password, database):

Este método es el constructor de la clase. Inicializa una instancia de **Catalogo** y crea una conexión a la base de datos. Toma cuatro argumentos: `host`, `user`, `password`, y `database`, que se utilizan para establecer una conexión con la base de datos.
Dentro del constructor, se crea una conexión a la base de datos MySQL y se configura un cursor para que devuelva resultados en forma de diccionarios.
Luego, se verifica si la tabla **"productos"** existe en la base de datos. Si no existe, se crea la tabla con las columnas necesarias.

Argumentos (Args):

- **host (str):** La dirección del servidor de la base de datos.
- **user (str):** El nombre de usuario para acceder a la base de datos.
- **password (str):** La contraseña del usuario.
- **database (str):** El nombre de la base de datos.

Este es el código completo de la clase **Catálogo**, hasta el momento:

```
import mysql.connector

class Catalogo:

    def __init__(self, host, user, password, database):

        self.conn = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            database=database
        )

        self.cursor = self.conn.cursor(dictionary=True)
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
            codigo INT,
            descripcion VARCHAR(255) NOT NULL,
            cantidad INT(4) NOT NULL,
            precio DECIMAL(10, 2) NOT NULL,
            imagen_url VARCHAR(255),
            proveedor INT(2))''')
        self.conn.commit()
```

Método Agregar Producto: def agregar_producto(self, codigo, descripcion, cantidad, precio, imagen, proveedor):

Este método tiene como objetivo principal agregar un nuevo producto a una base de datos. Para llevar a cabo esta tarea, se requieren varios parámetros que describen las características del producto a agregar. A continuación, se detalla el funcionamiento de este método:

Argumentos (Args):

- **codigo (int):** El código del producto. Debe ser un número entero.
- **descripcion (str):** La descripción del producto. Se espera una cadena de texto que describa el producto de manera clara.
- **cantidad (int):** La cantidad en stock del producto. Debe ser un número entero que representa la cantidad de unidades disponibles.
- **precio (float):** El precio del producto. Este valor es un número en formato decimal que refleja el costo del producto.
- **imagen (str):** La URL de la imagen del producto. Debe ser una cadena que contenga una URL válida que apunte a la imagen del producto.
- **proveedor (int):** El código del proveedor del producto. Se asume que es un número entero que identifica al proveedor en la base de datos.

Retorno (Returns):

- **bool:** El método retorna un valor booleano. Si el producto se agrega con éxito a la base de datos, devuelve True. En caso de que ya exista un producto con el mismo código en la base de datos, el método retorna False.

```
    def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        # Verificamos si ya existe un producto con el mismo código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        producto_existe = self.cursor.fetchone()
        if producto_existe:
            return False

        # Si no existe, agregamos el nuevo producto a la tabla
        sql = f"INSERT INTO productos \
               (codigo, descripcion, cantidad, precio, imagen_url,
proveedor) \
               VALUES \
               ({codigo}, '{descripcion}', {cantidad}, {precio},
'{imagen}', {proveedor})"
        self.cursor.execute(sql)
        self.conn.commit()
        return True
```

Descripción del Funcionamiento:

1. El método comienza verificando si ya existe un producto con el mismo código en la base de datos. Esto se realiza mediante una consulta SQL que busca productos en la tabla productos que tengan el mismo código que el proporcionado como argumento. Si se encuentra un producto con el mismo código, se almacena esta información en la variable ***producto_existe***.
2. Luego, se verifica el valor de ***producto_existe***. Si esta variable contiene información (lo que significa que ya existe un producto con el mismo código), el método retorna ***False***, indicando que no se puede agregar un nuevo producto con un código duplicado.
3. En caso de que no exista un producto con el mismo código, se procede a agregar el nuevo producto a la base de datos. Se construye una consulta SQL que inserta una nueva fila en la tabla productos con los detalles del producto, como código, descripción, cantidad, precio, imagen y proveedor.
4. Después de crear la consulta SQL, se ejecuta mediante el método ***execute*** en el cursor (***self.cursor***). La base de datos guarda el nuevo producto en la tabla productos.
5. Finalmente, se confirma la transacción con ***self.conn.commit()*** para asegurar que los cambios se guarden de manera permanente en la base de datos. Si la operación de inserción fue exitosa, el método retorna ***True*** para indicar que el producto se agregó con éxito.

En resumen: este método permite agregar nuevos productos a una base de datos MySQL después de verificar que no existan productos con el mismo código. En caso de duplicados, retorna ***False***, y en caso de éxito en la inserción, retorna ***True***.

Probaremos si estos primeros dos métodos funcionan a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo**:

```
# Programa principal
catalogo = Catalogo(host='localhost', user='root', password='',
database='miapp')

# Agregamos productos a la tabla
```

```
catalogo.agregar_producto(1, 'Teclado USB 101 teclas', 10, 4500,
'teclado.jpg', 101)
catalogo.agregar_producto(2, 'Mouse USB 3 botones', 5, 2500, 'mouse.jpg',
102)
catalogo.agregar_producto(3, 'Monitor LED', 5, 25000, 'monitor.jpg', 102)
```

Si todo ha funcionado bien, en phpMyAdmin debería aparecer algo como esto:

codigo	descripcion	cantidad	precio	imagen_url	proveedor
1	Teclado USB 101 teclas	10	4500.00	teclado.jpg	101
2	Mouse USB 3 botones	5	2500.00	mouse.jpg	102
3	Monitor LED	5	25000.00	monitor.jpg	102

Método Consultar Producto: def consultar_producto(self, codigo):

Este método tiene como propósito principal consultar un producto específico en la base de datos a partir de su código. Aquí se explica en detalle cómo funciona:

Argumentos (Args):

- **codigo (int):** El código del producto a consultar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns):

- **dict:** El método retorna un diccionario que contiene la información del producto consultado. Si el producto se encuentra en la base de datos, el diccionario contendrá los detalles del producto, como código, descripción, cantidad, precio, URL de la imagen y proveedor. Si no se encuentra ningún producto con el código proporcionado, el método retorna **False**.

```
    def consultar_producto(self, codigo):
        # Consultamos un producto a partir de su código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        return self.cursor.fetchone()
```

Descripción del Funcionamiento:

1. El método inicia ejecutando una consulta SQL en la base de datos para buscar un producto específico. La consulta se realiza en la tabla productos y se seleccionan todos los campos de la fila que coincidan con el código proporcionado como argumento.
2. La ejecución de la consulta se lleva a cabo mediante el cursor de la base de datos (***self.cursor***). La base de datos busca un producto con el código especificado y recupera sus datos.
3. El resultado de la consulta se almacena en un diccionario, que contiene la información del producto. Este diccionario se genera automáticamente en formato clave-valor, donde las claves son los nombres de las columnas en la tabla productos, y los valores son los datos correspondientes al producto consultado.
4. Finalmente, el método retorna el diccionario que contiene la información del producto consultado. Si no se encuentra ningún producto con el código especificado, el método retorna ***False***, lo que indica que no se encontró ningún producto con ese código en la base de datos.

En resumen: este método es utilizado para recuperar la información detallada de un producto en la base de datos, dada su identificación única (código). Si el producto se encuentra, se devuelve un ***diccionario*** con sus detalles; de lo contrario, se retorna ***False***.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos que permitían agregar los productos). Se puede comprobar el funcionamiento agregando un valor que existe y otro que no existe.

```
# Consultamos un producto y lo mostramos
cod_prod = int(input("Ingrese el código del producto: "))
producto = catalogo.consultar_producto(cod_prod)
if producto:
    print(f"Producto encontrado: {producto['codigo']} -
{producto['descripcion']}")
else:
    print(f'Producto {cod_prod} no encontrado.')
```

Método Modificar Producto: def modificar_producto(self, codigo, nueva_descripcion, nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):

Este método tiene la función de actualizar los datos de un producto específico en la base de datos a partir de su código. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto que se va a modificar. Debe ser un número entero que identifica de manera única al producto en la base de datos.
- **nueva_descripcion (str):** La nueva descripción que se asignará al producto.
- **nueva_cantidad (int):** La nueva cantidad en stock del producto.
- **nuevo_precio (float):** El nuevo precio del producto.
- **nueva_imagen (str):** La nueva URL de la imagen del producto.
- **nuevo_proveedor (int):** El nuevo código del proveedor.

Retorno (Returns):

- **bool:** El método retorna True si la modificación se realizó con éxito. Si no se encontró ningún producto con el código proporcionado, el método retorna False.

```
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        # Modificamos los datos de un producto a partir de su código
        sql = f"UPDATE productos SET \
                    descripcion = '{nueva_descripcion}', \
                    cantidad = {nueva_cantidad}, \
                    precio = {nuevo_precio}, \
                    imagen_url = '{nueva_imagen}', \
                    proveedor = {nuevo_proveedor} \
                WHERE codigo = {codigo}"
        self.cursor.execute(sql)
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Descripción del Funcionamiento:

1. El método inicia construyendo una consulta SQL para actualizar un producto en la base de datos. La consulta se construye mediante una cadena de formato (f-string) que incluye todos los datos que deben ser actualizados. Se utiliza la información proporcionada como argumentos para actualizar la descripción, cantidad, precio, URL de la imagen y el código del proveedor del producto identificado por su código.
2. La consulta se ejecuta utilizando el cursor de la base de datos (***self.cursor***). La base de datos busca un producto con el código especificado y aplica las modificaciones definidas en la consulta SQL.
3. Después de la ejecución de la consulta, se realiza una confirmación de la transacción con **self.conn.commit()**. Esta confirmación asegura que los cambios se almacenan de manera permanente en la base de datos.
4. El método verifica el número de filas afectadas por la actualización a través de **self.cursor.rowcount**. Si se encontró un producto con el código especificado y se realizaron modificaciones, **self.cursor.rowcount** será mayor que 0, y el método retorna ***True***. De lo contrario, si no se encontró el producto, se retorna ***False***.

En resumen: este método permite actualizar los detalles de un producto en la base de datos a través de su código. Devuelve ***True*** si se realizó la modificación con éxito y ***False*** si no se encontró ningún producto con el código especificado.

Método Mostrar Producto: def mostrar_producto(self, codigo):

Este método tiene como objetivo mostrar en la consola los datos de un producto a partir de su código. Aquí está una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto a mostrar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns): este método no retorna valores.

```
    def mostrar_producto(self, codigo):
        # Mostramos los datos de un producto a partir de su código
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 40)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen_url']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 40)
        else:
            print("Producto no encontrado.")
```

Descripción del Funcionamiento:

1. El método toma un único argumento, que es el código del producto que se desea mostrar. El código se pasa como el parámetro **codigo**.

2. En la primera parte del método, se utiliza el método **consultar_producto(codigo)** para obtener un diccionario con la información del producto que corresponde al código proporcionado. Esto se hace llamando al método **consultar_producto** que ya hemos explicado previamente.
3. Se verifica si el producto fue encontrado en la base de datos. Si producto es un diccionario (lo que significa que se encontró un producto con el código especificado), se procede a mostrar los detalles del producto.
4. Si el producto se encontró, se muestra un encabezado visualizado con guiones (-) para separar claramente los detalles del producto. Luego, se imprimen en la consola los siguientes datos del producto:
   - Código
   - Descripción
   - Cantidad en stock
   - Precio
   - URL de la imagen
   - Código del proveedor
5. Después de mostrar los detalles del producto, se imprime otro encabezado de guiones para una mejor visualización.
6. Si no se encuentra ningún producto con el código proporcionado, se imprime **"Producto no encontrado"** en la consola.

En resumen: Este método permite mostrar de manera detallada la información de un producto específico en la consola. Si el producto con el código especificado se encuentra en la base de datos, sus detalles se muestran en la consola. Si no se encuentra ningún producto con ese código, se muestra un mensaje indicando que el producto no fue encontrado. Este método es útil para obtener información detallada sobre un producto específico en el catálogo.

Comprobaremos el funcionamiento de los dos métodos para modificar el producto y mostrarlo a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Modificamos un producto y lo mostramos
catalogo.mostrar_producto(1)
catalogo.modificar_producto(1, 'Teclado mecánico', 20, 34000,
'tecmec.jpg', 106)
catalogo.mostrar_producto(1)
```

Método Listar Productos: def listar_productos(self):

Este método tiene la finalidad de mostrar en pantalla un listado de todos los productos almacenados en la tabla de la base de datos. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args): Este método no requiere argumentos.
Retorno (Returns): Este método no tiene valor de retorno.

```
    def listar_productos(self):
        # Mostramos en pantalla un listado de todos los productos en la
tabla
        self.cursor.execute("SELECT * FROM productos")
        productos = self.cursor.fetchall()
        return productos
```

Descripción del Funcionamiento:

1. El método comienza ejecutando una consulta SQL en la base de datos para seleccionar todos los registros de la tabla "productos" mediante la instrucción **self.cursor.execute("SELECT * FROM productos")**.
2. Luego, utiliza **self.cursor.fetchall()** para recuperar todos los resultados de la consulta. Estos resultados se almacenan en la variable **productos**, que es una lista de diccionarios. Cada diccionario representa un producto y contiene información detallada sobre el mismo.
3. Finalmente, el método devuelve todos los registros de la tabla "productos".

En resumen: este método recopila información sobre todos los productos de la base de datos y la muestra de manera organizada en la salida. Esto facilita la visualización y la gestión de todos los productos en el catálogo.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Listamos todos los productos
catalogo.listar_productos()
```

Método Eliminar Producto: def eliminar_producto(self, codigo):

Este método tiene como objetivo eliminar un producto de la tabla de la base de datos a partir de su código. A continuación, se proporciona una descripción detallada de cómo funciona este método:

Argumentos (Args):

- **codigo (int):** El código del producto a eliminar. Se espera que sea un número entero que identifica de manera única al producto en la base de datos.

Retorno (Returns):

- **bool:** True si se eliminó el producto con éxito, False si no se encontró el producto.

```
    def eliminar_producto(self, codigo):
        # Eliminamos un producto de la tabla a partir de su código
        self.cursor.execute(f"DELETE FROM productos WHERE codigo =
{codigo}")
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Descripción del Funcionamiento:

1. El método toma un único argumento, que es el código del producto a eliminar. Este código se pasa como el parámetro **codigo**.

2. En la primera parte del método, se ejecuta una consulta SQL utilizando **self.cursor.execute(f"DELETE FROM productos WHERE codigo = {codigo}")**. Esta consulta se encarga de eliminar el registro de la tabla "productos" que coincida con el código proporcionado. En otras palabras, elimina el producto que tiene el código especificado.
3. Luego, se llama a **self.conn.commit()** para confirmar los cambios en la base de datos. Esto es importante porque las modificaciones en la base de datos, como la eliminación de registros, no se hacen efectivas hasta que se confirman.
4. Finalmente, el método evalúa si la eliminación fue exitosa. Lo hace revisando el valor de **self.cursor.rowcount**. Si es mayor que 0, **significa que se eliminó al menos un registro de la base de datos y el método devuelve True**, indicando que la eliminación se realizó con éxito. Si self.cursor.rowcount es igual a 0, significa que no se encontró ningún registro con el código proporcionado y el método devuelve **False**.

En resumen: Este método permite eliminar un producto de la base de datos a partir de su código. Si el producto con el código especificado existe en la base de datos y se elimina correctamente, el método devuelve ***True***. Si no se encuentra ningún producto con ese código, devuelve ***False***. Este método es útil para gestionar la base de datos y mantener actualizado el catálogo de productos, permitiendo la eliminación de productos que ya no están disponibles.

Comprobaremos el funcionamiento de este método a través del siguiente código, que incorporaremos al final, por fuera de la clase **Catalogo** (se aconseja comentar la invocación de los métodos anteriores):

```
# Eliminamos un producto
catalogo.eliminar_producto(2)
catalogo.listar_productos()
```

Conclusiones

Hemos explorado detalladamente la funcionalidad de la **clase Catalogo** y sus métodos para administrar un catálogo de productos almacenados en una base de datos MySQL. Cada uno de los métodos de esta clase tiene un propósito específico y proporciona una funcionalidad esencial para trabajar con productos. A continuación, destacamos algunas conclusiones clave:

- **Facilidad de Uso:** La clase Catalogo se ha diseñado para ser accesible y fácil de usar, incluso para personas con conocimientos básicos de programación en Python. Los métodos están bien estructurados y cuentan con documentación que describe claramente sus propósitos y cómo usarlos.
- **Administración de Productos:** Los métodos *agregar_producto*, *consultar_producto*, *modificar_producto*, *listar_productos*, *eliminar_producto*, y *mostrar_producto* brindan un conjunto completo de funcionalidades para agregar, consultar, actualizar, eliminar y mostrar productos. Esto permite una gestión completa del catálogo de productos.
- **Mecanismo de Base de Datos:** La clase utiliza una base de datos MySQL para almacenar y recuperar información sobre los productos. Esta elección de base de datos es escalable y segura, lo que la hace adecuada para la gestión de productos a nivel empresarial.
- **Evitar Duplicados:** El método *agregar_producto* verifica si ya existe un producto con el mismo código antes de agregarlo a la base de datos. Esto garantiza que no haya duplicados en el catálogo.
- **Detalles de Productos:** El método *mostrar_producto* permite ver en detalle la información de un producto específico, lo que puede ser útil para inspeccionar detalles o tomar decisiones de compra.

En resumen: la clase **Catalogo** proporciona una solución sólida y amigable para administrar un catálogo de productos a través de una base de datos MySQL. Los métodos ofrecen una gama completa de funcionalidades y están diseñados para ser fáciles de entender y utilizar, lo que la hace valiosa tanto para principiantes como para aquellos con experiencia en Python y bases de datos.
La aplicación de estos conceptos y métodos brinda una plataforma robusta para la administración eficiente de inventarios y catálogos de productos en entornos empresariales y otros.

ETAPA 4, PARTE I: INTRODUCCIÓN A FLASK

Antes de comenzar a trabajar con la API veremos algunas características básicas sobre Flask.

¿Qué es Flask?

Flask es un microframework para el desarrollo de aplicaciones web en Python. Es una herramienta extremadamente útil y versátil para crear desde simples páginas web hasta aplicaciones web complejas y APIs RESTful.
Permite crear aplicaciones de forma sencilla y rápida. Es decir, un acelerador de tareas que funciona con pocas líneas de código y que ejecuta las aplicaciones rápidamente.
Vamos a explorar en detalle qué es Flask y para qué se utiliza:

- **Es un "Microframework":** Es conocido como un microframework porque es ligero y ofrece lo esencial para desarrollar aplicaciones web, sin imponer dependencias o estructuras de proyecto específicas. A diferencia de otros frameworks más pesados como Django, Flask te proporciona las herramientas básicas y te deja la libertad de usar extensiones o herramientas adicionales según tus necesidades.
- **Está escrito en Python:** Esto lo hace accesible y fácil de usar para cualquier persona familiarizada con este lenguaje. Además, se beneficia de la simplicidad y la potencia de Python, haciéndolo una opción popular para desarrolladores de todos los niveles de habilidad.

Usos de Flask

- **Aplicaciones Web Sencillas y Complejas:** Puedes usar Flask para crear desde una simple página web hasta aplicaciones web completas y complejas.
- **APIs RESTful:** Flask es una opción popular para desarrollar APIs RESTful debido a su simplicidad y flexibilidad. Permite manejar fácilmente las solicitudes y respuestas en formatos como JSON.
- **Microservicios:** Debido a su ligereza, Flask es ideal para crear microservicios, pequeñas aplicaciones independientes que trabajan juntas para formar sistemas más grandes.
- **Desarrollo Rápido y Prototipado:** Flask es muy adecuado para el desarrollo rápido y el prototipado gracias a su simplicidad y flexibilidad.

Características Principales de Flask

Routing:
Flask permite definir rutas y funciones (*endpoints*) para manejar las solicitudes HTTP (GET, POST, PUT, DELETE, etc.). Cada función puede devolver una respuesta específica, lo que facilita la creación de una interfaz de usuario y de una API.

Profundicemos sobre este concepto, que es fundamental para comprender cómo se construyen aplicaciones web y APIs con este framework:

¿Qué es el Routing en Flask?
El "Routing" se refiere al proceso de dirigir una solicitud HTTP a la función específica de Python que debe manejarla. En Flask, esto se hace asociando URLs con funciones de Python, a las cuales se les denomina "rutas" o "endpoints".

Definición de Rutas
En Flask, las rutas se definen utilizando el decorador **@app.route()**, donde **app** es una instancia de la clase **Flask**.
El decorador **@app.route()** se utiliza para vincular una función con una URL específica. Así, cuando se realiza una solicitud HTTP a esa URL, Flask invoca automáticamente la función asociada.

Ejemplo Básico

```
@app.route('/')
def home():
    return 'Hello, World!'
```

Aquí, **@app.route('/')** indica que la función **home()** se asociará con la raíz del sitio web (es decir, la URL `/`). Cuando un usuario accede a esta URL, Flask ejecuta **home()** y devuelve la respuesta 'Hello, World!'.

Manejo de Diferentes Métodos HTTP
Flask permite especificar qué métodos HTTP puede aceptar una ruta. Los métodos comunes incluyen GET, POST, PUT y DELETE.
Por defecto, las rutas aceptan solicitudes GET. Si quieres manejar otros métodos, debes especificarlos explícitamente.

```
@app.route('/post', methods=['POST'])
def post_method():
    return 'You sent a POST request'
```

Aquí, la ruta `/post` solo aceptará solicitudes POST.

Variables en Rutas
Flask permite capturar valores en las rutas utilizando reglas variables. Estos valores pueden ser pasados a la función de la ruta.

```
@app.route('/user/<username>')
def show_user_profile(username):
    return f'User {username}'
```

En este ejemplo, **<username>** es una variable. Cuando alguien visita **/user/Alice**, la función **show_user_profile** se llama con **username** establecido en **'Alice'**.

Respuestas
Las funciones de ruta pueden devolver diferentes tipos de respuestas, incluyendo cadenas simples, HTML, JSON, y objetos de respuesta de Flask.

En resumen: el routing en Flask es un mecanismo poderoso que conecta las URLs con el código Python que debe ejecutarse cuando se accede a esas URLs. Permite definir cómo se debe responder a diferentes solicitudes HTTP y facilita la creación de interfaces de usuario y APIs al permitir un mapeo claro y flexible entre las URLs y la lógica de backend.

Templates:
Utiliza el motor de plantillas Jinja2, que permite generar páginas HTML dinámicamente. Las plantillas Jinja2 son archivos HTML con marcadores de posición y estructuras de control, que son rellenados y controlados por tu código Python. Profundizaremos sobre estos conceptos, fundamentales para generar páginas HTML dinámicas:

¿Qué es el motor de Plantillas Jinja2 en Flask?
Jinja2 es un motor de plantillas para Python utilizado en Flask para generar contenido HTML de manera dinámica. Las plantillas son una parte fundamental en la creación de aplicaciones web con Flask, ya que permiten una separación clara entre la lógica de la aplicación y la presentación del contenido.

Características de las Plantillas Jinja2
- **Sintaxis Familiar:** Jinja2 utiliza una sintaxis similar a HTML, lo que facilita su aprendizaje y uso, especialmente para quienes ya están familiarizados con HTML.
- **Herencia de Plantillas:** Jinja2 permite la herencia de plantillas. Esto significa que puedes tener una plantilla base que define una estructura general (como cabecera, pie de página, etc.), y luego crear plantillas específicas que extienden esta estructura base, redefiniendo solo las partes que cambian.
- **Marcadores de Posición:** Las plantillas pueden incluir marcadores de posición, normalmente encerrados entre llaves dobles, que son reemplazados por datos reales cuando se renderiza la plantilla. Estos marcadores pueden representar variables simples, expresiones, incluso llamar a funciones.
- **Estructuras de Control:** Jinja2 permite usar estructuras de control como bucles ***for*** y sentencias ***if*** directamente en la plantilla.
  Esto es útil para generar contenido dinámico basado en la lógica de la aplicación, como listar elementos de una base de datos.

Ejemplo de Uso de Plantillas
Supongamos que tienes una plantilla HTML con Jinja2 llamada **template.html**:

```
<html>

<head>
    <title>{{ title }}</title>
</head>

<body>
    <h1>{{ heading }}</h1>
    {% for item in items %}
        <li>{{ item }}</li>
    {% endfor %}
</body>

</html>
```

Aquí, **title**, **heading** e **item** son marcadores de posición para variables que serán proporcionadas por Flask cuando se renderice la plantilla.
{% for item in items %} es un bucle que iterará sobre una lista **items** proporcionada por Flask.

Renderización de Plantillas en Flask
Para renderizar esta plantilla en Flask, usarías una función como esta:

```
from flask import render_template

@app.route('/')
def home():
    return render_template('template.html', title='Home Page',
heading='Welcome!', items=['Item 1', 'Item 2', 'Item 3'])
```

render_template es una función de Flask que toma el nombre de una plantilla y las variables que quieres pasar a ella.
title, **heading**, e **items** son pasados a la plantilla y reemplazan los marcadores de posición correspondientes.

En resumen: el uso de plantillas Jinja2 en Flask permite crear páginas HTML dinámicas de manera eficiente y organizada. Permite una separación clara entre la lógica de back-end y la presentación del contenido en el front-end, facilitando la mantenibilidad y escalabilidad de las aplicaciones web.

Desarrollo y Depuración

Ofrece un entorno de desarrollo y depuración integrado que se puede activar con una línea de código. Este entorno incluye un servidor web de desarrollo y una herramienta de depuración interactiva.
El entorno de desarrollo y depuración en Flask es una característica significativa que facilita el proceso de construcción y mantenimiento de aplicaciones web. Esta funcionalidad integrada aporta varias ventajas a los desarrolladores, especialmente durante las fases iniciales del desarrollo y al solucionar problemas. Veamos con más detalle:

Entorno de Desarrollo Integrado

- **Servidor Web de Desarrollo:** Flask viene con un servidor web integrado, que no está destinado a la producción, pero es perfecto para el desarrollo. Este servidor puede ser lanzado fácilmente con una línea de código, y permite ver rápidamente los cambios realizados en el código sin necesidad de configuraciones complejas. El servidor se ejecuta localmente en tu máquina, lo que facilita el acceso y la prueba de tu aplicación durante el desarrollo.
- **Recarga Automática:** Una de las características más útiles del servidor de desarrollo de Flask es la recarga automática. Esto significa que el servidor puede detectar automáticamente cuando se han realizado cambios en el código fuente y reiniciar la aplicación, lo que permite ver los cambios en tiempo real sin necesidad de reiniciar manualmente el servidor.

Herramientas de Depuración

- **Depurador Interactivo:** Flask incluye un depurador interactivo en el navegador, que se activa cuando ocurre un error en la aplicación. Este depurador muestra una traza de la pila de ejecución y permite inspeccionar el estado de la aplicación en el momento del error, lo cual es invaluable para identificar y solucionar problemas.
- **Mensajes de Error Detallados:** El depurador proporciona mensajes de error detallados y sugerencias útiles, lo que facilita el diagnóstico y la corrección de errores en la aplicación.

Activación del Entorno de Desarrollo y Depuración

- **Línea de Código para Activación:** Para activar el entorno de desarrollo y depuración en Flask, simplemente se debe incluir **app.run(debug=True)** en el script de la aplicación. Esto pone en marcha el servidor de desarrollo y activa el modo de depuración.

Importancia en el Desarrollo

- **Rápido Ciclo de Desarrollo:** La combinación del servidor de desarrollo y las herramientas de depuración facilita un rápido ciclo de desarrollo, donde puedes escribir código, probarlo y depurarlo en un entorno eficiente y amigable.
- **Seguridad:** Es importante destacar que estas características están pensadas para ser usadas en un entorno de desarrollo y no en un entorno de producción, debido a cuestiones de seguridad y rendimiento.

En resumen: el entorno de desarrollo y depuración de Flask proporciona un marco de trabajo extremadamente útil para el desarrollo de aplicaciones web. Facilita a los desarrolladores la tarea de escribir, probar y depurar su código de manera eficiente y efectiva, acelerando el proceso de desarrollo y ayudando a asegurar la calidad y la estabilidad de la aplicación.

Extensiones

Aunque Flask es minimalista en su núcleo, se puede extender con una gran variedad de extensiones para añadir funcionalidades como ORM, autenticación, manejo de formularios, etc. Esto lo hace muy adaptable a diferentes tipos de proyectos.
La naturaleza minimalista de Flask es uno de sus mayores atractivos, pero también es lo que hace que las extensiones sean una parte crucial de su ecosistema. Las extensiones de Flask proporcionan una forma de añadir funcionalidades adicionales a las aplicaciones Flask sin sobrecargar el núcleo del framework. A continuación, se detalla cómo funcionan estas extensiones y por qué son importantes:

¿Qué son las Extensiones de Flask?

- **Añaden Funcionalidades:** Las extensiones de Flask son paquetes o módulos que se pueden añadir a una aplicación Flask para proporcionar funcionalidades adicionales que no están incluidas en el núcleo de Flask.
- **Desarrolladas por la Comunidad:** Muchas de estas extensiones son desarrolladas y mantenidas por la comunidad de Flask, lo que significa que hay una amplia gama de extensiones disponibles para casi cualquier necesidad de desarrollo web.

Tipos Comunes de Extensiones

- **ORM (Mapeo Objeto-Relacional)**: Extensiones como SQLAlchemy y Flask-SQLAlchemy proporcionan soporte ORM para facilitar la interacción con bases de datos a través de objetos de Python en lugar de consultas SQL puro.

- **Autenticación:** Extensiones como Flask-Login y Flask-Security añaden funcionalidades para gestionar sesiones de usuario, autenticación y autorización.
- **Manejo de Formularios:** Flask-WTF es una extensión popular que integra Flask con WTForms, simplificando la creación y validación de formularios web.
- **API REST:** Extensiones como Flask-RESTful permiten construir APIs REST de manera más estructurada y simplificada.

Integración de Extensiones

- **Fácil Integración:** Integrar una extensión en Flask generalmente es un proceso sencillo. Después de instalar la extensión (usualmente a través de pip), se importa en el código de la aplicación y se configura según sea necesario.
- **Configuración Personalizable:** Las extensiones suelen ser altamente configurables para adaptarse a las necesidades específicas de cada proyecto.

Ventajas de Usar Extensiones

- **Flexibilidad:** Las extensiones permiten a los desarrolladores agregar solo las funcionalidades que necesitan, manteniendo la aplicación ligera y eficiente.
- **Rápido Desarrollo de Aplicaciones:** Al proporcionar soluciones preconstruidas para problemas comunes, las extensiones aceleran significativamente el proceso de desarrollo de aplicaciones.
- **Mantenimiento y Soporte:** Dado que muchas extensiones son ampliamente utilizadas y mantenidas por la comunidad, generalmente están bien documentadas y actualizadas.

En resumen: las extensiones de Flask hacen que este framework sea extremadamente adaptable y potente, permitiendo a los desarrolladores construir desde aplicaciones web básicas hasta sistemas complejos con diversas funcionalidades. Esta flexibilidad, combinada con la facilidad de integración y la amplia disponibilidad de extensiones, hace de Flask una herramienta versátil y eficaz para el desarrollo web moderno.

Conclusión

Flask es una herramienta poderosa y flexible que puede ser utilizada para una amplia gama de aplicaciones web. Su naturaleza minimalista y extensible lo hace adecuado tanto para proyectos pequeños como para aplicaciones web a gran escala. Además, al estar construido en Python, se beneficia de la simplicidad y la eficiencia de este lenguaje, lo que lo hace accesible para desarrolladores con diferentes niveles de experiencia.

Pequeño ejemplo de muestra de Flask:

1) Instalar flask desde pip install Flask
2) Armar dentro de tu carpeta del proyecto una carpeta ***templates*** y dentro de ella un archivo llamado prueba.html. Pegar este código:

```
<html>
<head>
    <title>{{ title }}</title>
</head>
<body>
    <h1>{{ heading }}</h1>
    {% for item in items %}
        <li>{{ item }}</li>
    {% endfor %}
</body>
```

```
</html>
```

3) Dentro de tu carpeta del proyecto (al mismo nivel que la carpeta **templates**) armar un archivo llamado **app.py**, pegar este código:

```
from flask import Flask, render_template

app = Flask(__name__)
@app.route('/')
def home():
    return render_template('prueba.html', title='Prueba Flask',
heading='Bienvenidos a Flask!', items=['Item 1', 'Item 2', 'Item 3'])

if __name__ == "__main__":
    app.run(debug=True)
```

Python asigna el nombre "_main_" al script cuando se ejecuta. Si el script se importa desde otro script, mantiene su nombre (por ejemplo, hola.py).
En nuestro caso, estamos ejecutando el script, por lo tanto, _name_ será igual a "_main_". Eso significa que se cumple la declaración condicional if y se ejecutará el método app.run(). También estamos configurando el parámetro de depuración en verdadero. Eso imprimirá posibles errores de Python en la página web, ayudándonos a rastrear los errores.

4) Ejecutar desde la terminal, deberá aparecer lo siguiente:

```
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 117-656-217
```

5) Ingresar en http://127.0.0.1:5000 y en el navegador debería aparecer lo siguiente:

ETAPA 4, PARTE II: DESARROLLAR UNA API PARA NUESTRO CRUD

Descripción e Instalación de módulos

Esta guía proporciona una visión general de cómo instalar y para qué se utilizan estos módulos y extensiones en el desarrollo de aplicaciones web con Python.

1. Flask

- **¿Qué es Flask?:** Flask es un microframework para Python utilizado para desarrollar aplicaciones web. Es ligero, fácil de usar y muy flexible, lo que lo hace popular para proyectos pequeños y grandes.
- **Instalación:** Para instalar Flask, usa el gestor de paquetes pip con el siguiente comando en tu terminal o línea de comandos:

```
pip install Flask
```

- **Uso:** Flask se usa para manejar solicitudes web, crear APIs, y renderizar plantillas HTML. Permite definir rutas y funciones para responder a distintos tipos de solicitudes HTTP (GET, POST, etc.).

2. Flask-CORS

- **¿Qué es Flask-CORS?:** Flask-CORS es una extensión para Flask que maneja el intercambio de recursos de origen cruzado (CORS), permitiendo que tu API Flask acepte solicitudes de otros dominios.
- **Instalación:** Para instalar Flask-CORS, utiliza pip con el siguiente comando:

```
pip install flask-cors
```

- **Uso:** Es útil cuando tu frontend y backend están en diferentes dominios y necesitas hacer solicitudes entre ellos. Se usa para añadir encabezados CORS a las respuestas de tu aplicación Flask.

3. MySQL Connector/Python

- **¿Qué es MySQL Connector/Python?:** MySQL Connector/Python es un driver que te permite conectar tu aplicación Python con una base de datos MySQL, permitiendo ejecutar consultas SQL, manejar transacciones, etc.
- **Instalación:** Instálalo con pip utilizando:

```
pip install mysql-connector-python
```

- **Uso:** Se utiliza para establecer una conexión con bases de datos MySQL desde Python, ejecutar consultas SQL, y manejar resultados y transacciones.

4. Werkzeug

- **Qué es Werkzeug?:** Werkzeug es una biblioteca WSGI (*Web Server Gateway Interface*) para Python. Es una de las bases sobre las que se construye Flask.
- **Instalación:** Normalmente, Werkzeug se instala automáticamente con Flask, pero si necesitas instalarlo manualmente, usa:

```
pip install Werkzeug
```

- **Uso:** Una de sus funcionalidades más comunes es **secure_filename**, que se utiliza para asegurar que los nombres de archivos subidos a tu servidor sean seguros.

5. Módulos Estándar de Python (os, time)

- **¿Qué son os y time?: os** y **time** son módulos que forman parte de la biblioteca estándar de Python. No necesitas instalarlos, ya que vienen incluidos con Python.
- Uso: **os** se usa para interactuar con el sistema operativo, como manejar rutas de archivos y variables de entorno. **time** se utiliza para operaciones relacionadas con el tiempo, como pausas y marcas de tiempo.

Importando librerías

Este código representa una aplicación web desarrollada en Python utilizando Flask, que interactúa con una base de datos MySQL para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre productos.

En primer lugar, importaremos todos los módulos necesarios para nuestra aplicación:

```
#-----------------------------------------------------------------------
# Instalar con pip install Flask
from flask import Flask, request, jsonify
from flask import request

# Instalar con pip install flask-cors
from flask_cors import CORS

# Instalar con pip install mysql-connector-python
import mysql.connector

# Si es necesario, pip install Werkzeug
from werkzeug.utils import secure_filename

# No es necesario instalar, es parte del sistema standard de Python
import os
import time
#-----------------------------------------------------------------------
```

A continuación detallaremos cada función y método.

Armando la clase Catálogo

Inicialización de Flask y Habilitación de CORS

```
app = Flask(__name__)
CORS(app)  # Esto habilitará CORS para todas las rutas
```

- **Flask(name):** Crea una instancia de la aplicación Flask. **name** es una variable especial de Python que se utiliza para determinar el nombre del módulo o paquete que se está ejecutando.

- **CORS(app)**: Habilita *Cross-Origin Resource Sharing (CORS)* para todas las rutas de la aplicación Flask. Esto permite que el frontend de la aplicación haga solicitudes a la API desde un origen diferente (dominio, protocolo o puerto).

Clase `Catalogo`

Método init:

Este método en la clase **Catalogo** es el constructor de la clase, y su propósito principal es establecer una conexión con la base de datos MySQL y preparar el ambiente para las operaciones subsecuentes con la base de datos. Vamos a desglosar su funcionamiento paso a paso:

1. **Parámetros del Constructor**
   El método **init** se define con los siguientes parámetros:

```
class Catalogo:
    # Constructor de la clase
    def __init__(self, host, user, password, database):
```

   Estos parámetros son esenciales para establecer la conexión con la base de datos MySQL:
   - **host**: La dirección del servidor de la base de datos (por ejemplo, 'localhost').
   - **user**: El nombre de usuario para acceder a la base de datos.
   - **password**: La contraseña del usuario para la base de datos.
   - **database**: El nombre de la base de datos a la que conectarse.

2. **Establecimiento de la Conexión Inicial**

```
        # Primero, establecemos una conexión sin especificar la base
de datos
        self.conn = mysql.connector.connect(
            host=host,
            user=user,
            password=password
        )
```

   Aquí, se utiliza **mysql.connector.connect** para crear una conexión con el servidor MySQL. Inicialmente, la base de datos no se especifica. Esto se hace para manejar la situación en la que la base de datos especificada en **database** no exista aún.

3. **Creación y Manejo del Cursor**

```
        self.cursor = self.conn.cursor()
```

   Se crea un cursor a través del objeto de conexión. El cursor es utilizado para ejecutar comandos SQL en la base de datos. En esta etapa, el cursor se crea sin opciones específicas.

4. **Selección o Creación de la Base de Datos**

```
        # Intentamos seleccionar la base de datos
        try:
            self.cursor.execute(f"USE {database}")
```

```
        except mysql.connector.Error as err:
            # Si la base de datos no existe, la creamos
            if err.errno == mysql.connector.errorcode.ER_BAD_DB_ERROR:
                self.cursor.execute(f"CREATE DATABASE {database}")
                self.conn.database = database
            else:
                raise err
```

Aquí se intenta seleccionar la base de datos usando el comando SQL **USE**. Si la base de datos no existe (lo cual se captura como una excepción **mysql.connector.Error** con el código de error **ER_BAD_DB_ERROR**), entonces se crea una nueva base de datos con el nombre proporcionado usando **CREATE DATABASE**.
Si se encuentra cualquier otro error durante este proceso, el error se propaga hacia arriba con **raise err**.

5. **Creación de la Tabla `productos`**

```
        # Una vez que la base de datos está establecida, creamos la
tabla si no existe
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
            codigo INT AUTO_INCREMENT,
            descripcion VARCHAR(255) NOT NULL,
            cantidad INT NOT NULL,
            precio DECIMAL(10, 2) NOT NULL,
            imagen_url VARCHAR(255),
            proveedor INT)''')
        self.conn.commit()
```

Se ejecuta una consulta SQL para crear una tabla **productos** si no existe ya. Esta tabla incluye varias columnas como `codigo`, `descripcion`, `cantidad`, etc., con sus respectivos tipos de datos y restricciones.
self.conn.commit() asegura que la creación de la tabla se confirme en la base de datos.

6. **Reconfiguración del Cursor**

```
        # Cerrar el cursor inicial y abrir uno nuevo con el parámetro
dictionary=True
        self.cursor.close()
        self.cursor = self.conn.cursor(dictionary=True)
```

Primero, el cursor existente se cierra con **self.cursor.close()**.
Luego, se crea un nuevo cursor, esta vez con el parámetro **dictionary=True**. Esto significa que cualquier resultado de una consulta SQL será devuelto como un diccionario, lo que hace que sea más fácil y claro trabajar con los datos en Python, accediendo a los valores de las columnas por su nombre.

En resumen: El método **init** en la clase **Catalogo** prepara todo lo necesario para interactuar con una base de datos MySQL. Establece la conexión, maneja la creación de la base de datos y de la tabla necesaria, y configura el cursor para su uso en operaciones posteriores con la base de

datos. Todo esto se hace de manera que la clase **Catalogo** esté lista para realizar operaciones de base de datos como agregar, consultar, modificar y eliminar productos.
Hasta ahora, tendremos el siguiente código:

```
class Catalogo:
    # Constructor de la clase
    def __init__(self, host, user, password, database):
        # Primero, establecemos una conexión sin especificar la base de datos
        self.conn = mysql.connector.connect(
            host=host,
            user=user,
            password=password
        )
        self.cursor = self.conn.cursor()

        # Intentamos seleccionar la base de datos
        try:
            self.cursor.execute(f"USE {database}")
        except mysql.connector.Error as err:
            # Si la base de datos no existe, la creamos
            if err.errno == mysql.connector.errorcode.ER_BAD_DB_ERROR:
                self.cursor.execute(f"CREATE DATABASE {database}")
                self.conn.database = database
            else:
                raise err

        # Una vez que la base de datos está establecida, creamos la tabla si no existe
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
            codigo INT AUTO_INCREMENT,
            descripcion VARCHAR(255) NOT NULL,
            cantidad INT NOT NULL,
            precio DECIMAL(10, 2) NOT NULL,
            imagen_url VARCHAR(255),
            proveedor INT(3)''')
        self.conn.commit()

        # Cerrar el cursor inicial y abrir uno nuevo con el parámetro dictionary=True
        self.cursor.close()
        self.cursor = self.conn.cursor(dictionary=True)
```

Creando un catálogo

Comenzaremos con el cuerpo principal del programa de la siguiente manera:

```
#------------------------------------------------------------------
# Cuerpo del programa
```

```
#---------------------------------------------------------------------
# Crear una instancia de la clase Catalogo
catalogo = Catalogo(host='localhost', user='root', password='',
database='miapp')

# Carpeta para guardar las imagenes
ruta_destino = './static/imagenes/'
```

Nota: la ruta donde colocarás las imágenes puede se modificada de acuerdo a tu preferencia.

Creación de una Instancia del Catálogo:

catalogo = Catalogo(host='localhost', user='root', password='', database='miapp'): Esta línea crea una nueva instancia de un objeto llamado **Catalogo**.
Catalogo es una clase que gestiona la conexión y las interacciones con una base de datos.
Se están pasando varios argumentos al constructor de Catalogo:
host='localhost': Especifica el host de la base de datos. En este caso, es **'localhost'**, lo que sugiere que la base de datos se ejecuta en el mismo servidor que la aplicación.
user='root': Es el nombre de usuario utilizado para acceder a la base de datos. Aquí se utiliza el usuario **'root'**, que es el administrador de la base de datos.
password='': Es la contraseña para el usuario de la base de datos. Se deja en blanco aquí, lo cual no es recomendable para entornos de producción debido a razones de seguridad.
database='miapp': Especifica el nombre de la base de datos a la que se conectará, en este caso, una base de datos llamada **'miapp'**.

Definición de la Ruta para Guardar Imágenes:
ruta_destino = './static/imagenes/': Esta línea define una variable **ruta_destino** que almacena una ruta de directorio como un string.
La ruta **./static/imagenes/** implica que hemos creado un directorio imagenes dentro de un directorio static en la misma ubicación que el script Python, en el servidor.
Esta convención de nomenclatura es común en aplicaciones web donde los archivos estáticos (como imágenes, CSS y JavaScript) se almacenan. En nuestro caso, es el lugar donde se almacenarán las imágenes cargadas en la aplicación.

Listar productos: Métodos y rutas

Método listar_productos:

Este método en la clase **Catalogo** está diseñado para recuperar y devolver una lista de todos los productos almacenados en la base de datos. Es una operación de **Leer** en el marco de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).
Este es el código de la función:

```
    def listar_productos(self):
        self.cursor.execute("SELECT * FROM productos")
        productos = self.cursor.fetchall()
        return productos
```

Expliquemos en detalle cómo funciona este método:

1. **Ejecución de la Consulta**

Aquí, el método **listar_productos** ejecuta una consulta SQL **SELECT * FROM productos**, que solicita todos los registros **(*)** de la tabla **productos**.
Esta consulta no tiene parámetros ya que el objetivo es obtener todos los productos, no uno específico.

2. **Recuperación y Devolución de Resultados**
   Después de ejecutar la consulta, se usa **self.cursor.fetchall()** para recuperar todos los registros que coincidan con la consulta. Este método devuelve todos los resultados de la consulta como una lista de diccionarios (debido a que el cursor se configuró con **dictionary=True** en el constructor). Cada diccionario en la lista representa un registro (producto) de la base de datos, donde las claves son los nombres de las columnas y los valores son los datos correspondientes de cada producto.
   Finalmente, la lista de productos se devuelve a quien llamó al método.

En resumen: este método realiza las siguientes tareas:
1. Ejecuta una consulta SQL para obtener todos los registros de la tabla **productos** en la base de datos.
2. Recupera y devuelve todos los productos como una lista de diccionarios, facilitando su acceso y manipulación en el código que consuma este método.

Este método proporciona una manera simple y eficaz de recuperar todos los datos de un conjunto de registros de la base de datos, lo cual es una operación común en muchas aplicaciones web que requieren mostrar listados o colecciones de objetos almacenados en la base de datos.

Ruta Listar Productos (`/productos` - GET)

Este fragmento de código define un endpoint **/productos** en la aplicación web que, cuando se solicita mediante un método GET, ejecuta la función **listar_productos**. Esta función recupera los datos de los productos y los devuelve en formato JSON, lo que facilita su uso en aplicaciones cliente como interfaces de usuario web o móviles.
La ruta Flask **/productos** con el método HTTP **GET** está diseñada para proporcionar una lista de todos los productos almacenados en la base de datos. Esta ruta es parte de una API web y sirve como un punto de acceso para obtener datos de productos.

```python
@app.route("/productos", methods=["GET"])
def listar_productos():
    productos = catalogo.listar_productos()
    return jsonify(productos)
```

Analicemos su funcionamiento detallado:

@app.route("/productos", methods=["GET"]) es un decorador que define una ruta en la aplicación Flask. El decorador **@app.route** asocia la función que sigue (**listar_productos**) con la URL **/productos**. El **methods=["GET"]** especifica que esta ruta responde a solicitudes HTTP **GET**.
"/productos": Es el endpoint o ruta específica en la URL del servidor. Por ejemplo, si tu aplicación se ejecuta en http://localhost:5000, esta ruta sería accesible en http://localhost:5000/productos.
methods=["GET"]: Indica que esta ruta acepta solicitudes HTTP GET. Esto significa que cuando un cliente (como un navegador o una aplicación de frontend) realiza una solicitud GET a esta URL, se ejecutará la función **listar_productos**, asociada a esta ruta.

En la función **listar_productos**, se llama al método **listar_productos** de la instancia **catalogo** de la clase **Catalogo**. Esta instancia debe haber sido creada previamente en el código y está conectada a la base de datos. El método **catalogo.listar_productos()** recupera todos los productos de la base de datos y devuelve una lista de diccionarios, donde cada diccionario representa un producto.
return jsonify(productos): La función **jsonify** de Flask convierte la lista de diccionarios (productos) en una respuesta JSON. JSON (*JavaScript Object Notation*) es un formato estándar para el intercambio de datos, particularmente en APIs web. Esta respuesta JSON es lo que se envía de vuelta al cliente que hizo la solicitud GET a **/productos**.
Devolver los datos en formato JSON es una práctica común en APIs REST, ya que facilita que diferentes clientes (como navegadores web, aplicaciones móviles, etc.) puedan procesar y utilizar estos datos.

En resumen: la ruta **/productos** con el método **GET** en Flask funciona de la siguiente manera:
1. Cuando se recibe una solicitud **GET** a **/productos**, se invoca la función **listar_productos**.
2. Dentro de esta función, se llama al método **listar_productos** de una instancia de **Catalogo** para recuperar todos los productos de la base de datos.
3. La lista de productos se convierte en una respuesta JSON y se devuelve al cliente que realizó la solicitud.

Esta ruta es un ejemplo típico de cómo se pueden exponer datos de una base de datos a través de una API web, permitiendo que clientes como navegadores web, aplicaciones móviles o servicios de terceros recuperen estos datos de manera estructurada y utilizable.

Ejecutar la aplicación

Esta parte del código se asegura de que el servidor web Flask se inicie solo cuando el script se ejecuta directamente (no cuando se importa como módulo) y con el modo de depuración activado si debug está configurado como True. Esto permite que la aplicación Flask atienda las solicitudes HTTP entrantes cuando ejecutas el script.

```
if __name__ == "__main__":
    app.run(debug=True)
```

El fragmento de código **if __name__ == "__main__": app.run(debug=True)** se utiliza para iniciar el servidor web si el script actual se ejecuta como el programa principal. Veamos como funciona este fragmento de código:
if __name__ == "__main__": Esta línea verifica si el script actual se está ejecutando como el programa principal. Cuando un archivo Python se ejecuta directamente, el valor de **name** se establece en **"main"**; sin embargo, cuando el archivo se importa como un módulo en otro script, el valor de name se establece en el nombre del módulo. Por lo tanto, esta condición se cumple solo cuando ejecutas este script específico directamente desde la línea de comandos.
app.run(debug=True): Si la condición **if __name__ == "__main__":** se cumple (es decir, el script se está ejecutando como el programa principal), entonces Flask se inicia mediante **app.run()**. Esto inicia el servidor web de Flask y permite que la aplicación escuche las solicitudes HTTP entrantes.
app.run(debug=True): Ejecuta la aplicación Flask en modo de depuración. El modo de depuración permite ver los errores detalladamente y recarga automáticamente la aplicación cuando se hacen cambios en el código.
debug=True: Cuando debug se establece en **True**, Flask activa el modo de depuración, lo que significa que, si hay errores en el código Python o en la aplicación, se mostrarán mensajes

detallados de error en el navegador web para ayudar a depurar el problema. Esto es útil durante el desarrollo, pero no se debe usar en entornos de producción debido a posibles problemas de seguridad.

Hasta el momento, el cuerpo del programa debería haber quedado de esta forma:

```
#------------------------------------------------------------------
# Cuerpo del programa
#------------------------------------------------------------------
# Crear una instancia de la clase Catalogo
catalogo = Catalogo(host='localhost', user='root', password='',
database='miapp')

# Carpeta para guardar las imagenes
ruta_destino = './static/imagenes/'

@app.route("/productos", methods=["GET"])
def listar_productos():
    productos = catalogo.listar_productos()
    return jsonify(productos)

if __name__ == "__main__":
    app.run(debug=True)
```

Mostrar producto: Métodos para consultar/mostrar y rutas

Método consultar_producto:

Este método en la clase **Catalogo** está diseñado para buscar y devolver los detalles de un producto específico de la base de datos, basándose en su código. Es una operación de **Leer** en el marco de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

```
def consultar_producto(self, codigo):
        # Consultamos un producto a partir de su código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        return self.cursor.fetchone()
```

Veamos en detalle cómo funciona este método:

1. **Parámetro del Método**
   Este método toma un único parámetro:
   - **codigo**: El identificador único del producto que se desea consultar.

2. **Ejecución de la Consulta**
   Aquí, el método ejecuta una consulta SQL **SELECT** para buscar un producto en la tabla **productos** que coincida con el **codigo** proporcionado.
   Este tipo de consulta recupera todos los campos **(*)** del producto cuyo **codigo** coincida con el valor dado.

3. **Recuperación y Devolución de Resultados**

Después de ejecutar la consulta, **self.cursor.fetchone()** se utiliza para obtener el primer registro del resultado de la consulta.
Si un producto con el **codigo** especificado existe en la base de datos, este registro **(producto)** se devuelve como resultado.
El resultado es un diccionario (debido a que el cursor se configuró con **dictionary=True** en el constructor) donde las claves corresponden a los nombres de las columnas de la tabla **productos**, y los valores son los datos del producto.
Si no se encuentra ningún producto con el **codigo** proporcionado, **fetchone()** devolverá **None**.

En resumen: este método realiza las siguientes tareas:
1. Ejecuta una consulta SQL para buscar en la base de datos un producto con el **codigo** especificado.
2. Recupera los detalles de este producto si existe.
3. Devuelve los detalles del producto como un diccionario o **None** si el producto no se encuentra.

Este método es un ejemplo típico de una operación de consulta en una aplicación web que interactúa con una base de datos, proporcionando una manera eficiente y segura de recuperar información específica de la base de datos.

Método mostrar_producto:

Imprime en consola la información de un producto específico. Se mantiene solo con propósitos de depuración:

```
    def mostrar_producto(self, codigo):
        # Mostramos los datos de un producto a partir de su código
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 40)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen_url']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 40)
        else:
            print("Producto no encontrado.")
```

Ruta Mostrar Producto (`/productos/<int:codigo>` - GET)

Este código define un endpoint para consultar detalles específicos de un producto basado en su código. Responde a solicitudes GET y devuelve los detalles del producto en formato JSON si se encuentra, o un mensaje de error con un código de estado adecuado si no se encuentra el producto.
La ruta Flask **/productos/<int:codigo>** con el método HTTP **GET** está diseñada para proporcionar los detalles de un producto específico basado en su código. Esta ruta es un ***endpoint*** de una API web y sirve como un punto de acceso para obtener datos detallados de un solo producto.

```
@app.route("/productos/<int:codigo>", methods=["GET"])
```

```
def mostrar_producto(codigo):
    producto = catalogo.consultar_producto(codigo)
    if producto:
        return jsonify(producto)
    else:
        return "Producto no encontrado", 404
```

Veamos en detalle cómo funciona:

@app.route("/productos/<int:codigo>", methods=["GET"]): Este decorador define una ruta en la aplicación Flask. La parte **<int:codigo>** en la ruta es una variable dinámica que representa el código del producto. El **int** indica que esta variable debe ser un entero. Flask automáticamente manejará cualquier valor que se pase en esta parte de la URL como un entero y lo asignará al parámetro **codigo** de la función **mostrar_producto**. Esto permite que la ruta maneje solicitudes para diferentes códigos de producto, como /productos/123.
La función **mostrar_producto** se asocia con esta URL y es llamada cuando se hace una solicitud **GET** a **/productos/** seguido de un número (el código del producto).

producto = catalogo.consultar_producto(codigo): Esta línea llama al método **consultar_producto** de la instancia **catalogo** de la clase **Catalogo**, pasando el **codigo** del producto. Este método busca en la base de datos el producto con el código especificado y devuelve un diccionario con los detalles del producto si lo encuentra, o **None** si no lo encuentra.
Si se encuentra el producto (**if producto:**), los detalles del producto se convierten en una respuesta JSON utilizando **jsonify(producto)** y se envían de vuelta al cliente con un código de estado HTTP 201, que generalmente significa que un recurso se ha creado con éxito
Si el producto no se encuentra (**else:**), la función devuelve un mensaje "Producto no encontrado" con un código de estado **HTTP 404**, indicando que el recurso solicitado no existe.

En resumen: la ruta **`/productos/<int:codigo>`** con el método GET en Flask funciona de la siguiente manera:

1. Cuando se recibe una solicitud GET a esta ruta con un código de producto específico, se invoca la función **mostrar_producto**.
2. La función busca el producto con el código dado en la base de datos.
3. Si el producto se encuentra, se devuelve una respuesta JSON con los detalles del producto.
4. Si el producto no se encuentra, se devuelve un mensaje de error con un código de estado 404.

Esta ruta proporciona una manera eficiente y estructurada de acceder a los detalles de un producto específico en una base de datos a través de una API web, lo que es común en aplicaciones que requieren acceso detallado a registros individuales.

Agregar productos: Métodos y rutas

Método agregar_producto:

Este método en la clase **Catalogo** está diseñado para agregar un nuevo producto a la base de datos. Es un ejemplo de una operación **Crear** en el contexto de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

```
    def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        # Verificamos si ya existe un producto con el mismo código
```

```
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        producto_existe = self.cursor.fetchone()
        if producto_existe:
            return False

        # Si no existe, agregamos el nuevo producto a la tabla
        sql = "INSERT INTO productos (codigo, descripcion, cantidad,
precio, imagen_url, proveedor) VALUES (%s, %s, %s, %s, %s, %s)"
        valores = (codigo, descripcion, cantidad, precio, imagen,
proveedor)
        self.cursor.execute(sql, valores)
        self.conn.commit()
        return True
```

Vamos a desglosar su funcionamiento paso a paso:

1. **Parámetros del Método**
   Este método toma varios parámetros que representan las propiedades del producto a agregar:
   - **codigo:** Un identificador único para el producto.
   - **descripcion:** Descripción textual del producto.
   - **cantidad:** Cantidad del producto disponible.
   - **precio:** Precio del producto.
   - **imagen:** URL o nombre del archivo de imagen asociado con el producto.
   - **proveedor:** Identificador del proveedor del producto.

2. **Verificación de la Existencia del Producto**
   Antes de agregar un nuevo producto, el método verifica si ya existe un producto con el mismo **codigo**. Para ello, ejecuta una consulta SQL **SELECT** en la tabla **productos**.
   self.cursor.fetchone() intenta recuperar el primer registro del resultado de la consulta. Si encuentra un producto con el mismo código, **producto_existe** no será **None**.

3. **Lógica de Agregación del Producto**
   Si un producto con el mismo código ya existe (**producto_existe** no es **None**), el método devuelve **False**, indicando que no se agregó el producto.
   Si no se encuentra un producto con el mismo código, se procede a agregar el nuevo producto a la base de datos.
   Luego se construye una consulta SQL **INSERT INTO** para insertar un nuevo registro en la tabla **productos**.
   Los valores a insertar se pasan de manera segura usando parámetros **%s** para evitar problemas como la inyección SQL. Estos valores se proporcionan en la tupla **valores**.
   self.cursor.execute(sql, valores) ejecuta la consulta SQL con los valores dados.

4. **Confirmación de Cambios y Retorno**
   Después de ejecutar la consulta de inserción, se realiza un **commit** a través de **self.conn.commit()**. Esto asegura que los cambios (la inserción del nuevo producto) se guarden permanentemente en la base de datos.
   Finalmente, el método devuelve **True**, indicando que el producto se ha agregado con éxito.

Resumiendo, el método **agregar_producto** realiza las siguientes tareas:

1. Verifica si ya existe un producto con el mismo código en la base de datos.
2. Si no existe, inserta un nuevo producto con los detalles proporcionados.
3. Confirma los cambios en la base de datos.
4. Devuelve **True** si se agregó el producto con éxito, y **False** si el producto ya existía.

Este método es un ejemplo clásico de cómo manejar la inserción de datos en una base de datos en una aplicación web, asegurando tanto la integridad de los datos como la seguridad contra inyecciones SQL.

Ruta Agregar Producto (`/productos` - POST)

La ruta Flask **`/productos`** con el método HTTP **POST** está diseñada para permitir la adición de un nuevo producto a la base de datos. Esta ruta es un **endpoint** de la API web y actúa como un punto de acceso para crear un nuevo registro de producto en la base de datos.

```
@app.route("/productos", methods=["POST"])
def agregar_producto():
    # Recojo los datos del form
    codigo = request.form['codigo']
    descripcion = request.form['descripcion']
    cantidad = request.form['cantidad']
    precio = request.form['precio']
    proveedor = request.form['proveedor']
    imagen = request.files['imagen']
    nombre_imagen = secure_filename(imagen.filename)

    nombre_base, extension = os.path.splitext(nombre_imagen)
    nombre_imagen = f"{nombre_base}_{int(time.time())}{extension}"
    imagen.save(os.path.join(ruta_destino, nombre_imagen))

    if catalogo.agregar_producto(codigo, descripcion, cantidad, precio,
nombre_imagen, proveedor):
        return jsonify({"mensaje": "Producto agregado"}), 201
    else:
        return jsonify({"mensaje": "Producto ya existe"}), 400
```

Analicemos en detalle cómo funciona:

@app.route("/productos", methods=["POST"]): Este decorador define una ruta en la aplicación Flask que responde a solicitudes HTTP **POST**. En este caso, la URL es **/productos**.
La función **agregar_producto** se asocia con esta URL y es llamada cuando se hace una solicitud **POST** a **/productos**.

La función comienza recuperando los datos del producto enviados en la solicitud. Flask proporciona el objeto **request** para acceder a los datos enviados en la solicitud HTTP. Aquí, se accede a los campos del formulario (**request.form**) y a un archivo cargado (**request.files**).
secure_filename se utiliza para sanitizar el nombre del archivo de la imagen, asegurándose de que sea seguro para guardar en el sistema de archivos, evitando vulnerabilidades de seguridad.

nombre_base, extension = os.path.splitext(nombre_imagen): Separa el nombre del archivo de su extensión.

nombre_imagen = f"{nombre_base}_ int(time.time()){extension}": Genera un nuevo nombre para la imagen usando un timestamp, para evitar sobreescrituras y conflictos de nombres, esta modificación en el nombre del archivo de imagen ayuda a evitar colisiones de nombres de archivos.
Luego, la imagen se guarda en el servidor.

Finalmente, se llama al método **agregar_producto** de la instancia **catalogo** de la clase **Catalogo**, pasando los detalles del producto. Este método intenta agregar el producto a la base de datos.
Si el producto se agrega con éxito, se devuelve una respuesta JSON con un mensaje de éxito y un código de estado **HTTP 201 (Creado)**.
Si el producto ya existe (basado en el código), se devuelve una respuesta JSON con un mensaje de error y un código de estado **HTTP 400 (Solicitud Incorrecta).**

En resumen: la ruta **/productos** con el método **POST** en Flask funciona de la siguiente manera:
1. Recibe una solicitud POST con los datos del producto y la imagen.
2. Extrae y procesa estos datos, incluyendo guardar la imagen en el servidor.
3. Intenta agregar el producto a la base de datos.
4. Devuelve una respuesta JSON indicando el resultado de la operación, ya sea un éxito o un error.

Esta ruta es un ejemplo típico de cómo se manejan las solicitudes de creación de nuevos registros en aplicaciones web a través de una API, permitiendo la interacción del usuario con la base de datos de manera segura y controlada.

Modificar productos: Métodos y rutas

Método modificar_producto:

Este método en la clase **Catalogo** está diseñado para actualizar la información de un producto existente en la base de datos. Representa una operación de **Actualizar** en el contexto de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

```
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        sql = "UPDATE productos SET descripcion = %s, cantidad = %s,
precio = %s, imagen_url = %s, proveedor = %s WHERE codigo = %s"
        valores = (nueva_descripcion, nueva_cantidad, nuevo_precio,
nueva_imagen, nuevo_proveedor, codigo)
        self.cursor.execute(sql, valores)
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Vamos a desglosar su funcionamiento paso a paso:

1. **Parámetros del Método**
   Este método toma varios parámetros que representan la información a actualizar:
   - **codigo:** El identificador único del producto que se va a actualizar.
   - **nueva_descripcion:** La nueva descripción del producto.
   - **nueva_cantidad:** La nueva cantidad en stock del producto.
   - **nuevo_precio:** El nuevo precio del producto.
   - **nueva_imagen:** La nueva URL o nombre del archivo de imagen para el producto.

- **nuevo_proveedor:** El nuevo identificador del proveedor del producto.

2. **Construcción y Ejecución de la Consulta SQL de Actualización**
   Aquí, se construye una consulta SQL **UPDATE** para actualizar los detalles del producto en la base de datos.
   La consulta utiliza parámetros **%s** para evitar inyecciones SQL, proporcionando una manera segura de insertar valores en la consulta.
   valores es una tupla que contiene los nuevos valores para el producto, junto con su **codigo**.
   self.cursor.execute(sql, valores) ejecuta la consulta SQL con los valores proporcionados.

3. **Confirmación de Cambios y Retorno**
   self.conn.commit() asegura que los cambios se guarden en la base de datos.
   self.cursor.rowcount devuelve el número de filas afectadas por la última operación de ejecución. Si este número es mayor que 0, significa que la actualización tuvo éxito y, por lo tanto, se devuelve **True**.
   Si no se actualizó ningún registro (lo que podría suceder si no existe un producto con el **codigo** proporcionado), **rowcount** sería 0 y el método devolvería **False**.

En resumen: este método realiza las siguientes tareas:
1. Construye y ejecuta una consulta SQL para actualizar los detalles de un producto existente en la base de datos.
2. Confirma los cambios en la base de datos.
3. Devuelve **True** si la actualización afectó a alguna fila (es decir, si se realizó con éxito), y **False** en caso contrario.

Este método es un ejemplo clásico de cómo se manejan las actualizaciones de datos en una base de datos en aplicaciones web, asegurando la integridad de los datos y la seguridad contra inyecciones SQL.

Ruta Modificar Producto (`/productos/<int:codigo>` - PUT)

Este fragmento de código maneja solicitudes **PUT** que llegan a la ruta **"/productos/int:codigo"**, para modificar información de productos en una aplicación Flask. Recibe datos del formulario, procesa y guarda la nueva imagen, consulta el producto existente, y luego intenta modificarlo en el catálogo de productos. Si la modificación tiene éxito, se envía una respuesta de éxito; de lo contrario, se envía una respuesta de error.
La ruta Flask **/productos/<int:codigo>** con el método HTTP **PUT** está diseñada para actualizar la información de un producto existente en la base de datos, identificado por su código. Este **endpoint** de la API web permite modificar los datos de un producto específico.

```
@app.route("/productos/<int:codigo>", methods=["PUT"])
def modificar_producto(codigo):
    # Recojo los datos del form
    nueva_descripcion = request.form.get("descripcion")
    nueva_cantidad = request.form.get("cantidad")
    nuevo_precio = request.form.get("precio")
    nuevo_proveedor = request.form.get("proveedor")

    # Procesamiento de la imagen
    imagen = request.files['imagen']
    nombre_imagen = secure_filename(imagen.filename)
    nombre_base, extension = os.path.splitext(nombre_imagen)
```

```
    nombre_imagen = f"{nombre_base}_{int(time.time())}{extension}"
    imagen.save(os.path.join(ruta_destino, nombre_imagen))

    # Actualización del producto
    if catalogo.modificar_producto(codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nombre_imagen, nuevo_proveedor):
        return jsonify({"mensaje": "Producto modificado"}), 200
    else:
        return jsonify({"mensaje": "Producto no encontrado"}), 404
```

Veamos cómo funciona en detalle:

@app.route("/productos/<int:codigo>", methods=["PUT"]): Este decorador define una ruta en la aplicación Flask que responde a solicitudes HTTP **PUT**. La URL incluye una variable dinámica **<int:codigo>**, que representa el código del producto a modificar. Flask capturará automáticamente el valor de esta parte de la URL y lo pasará como un argumento entero al parámetro **codigo** de la función **modificar_producto**.
La función **modificar_producto** se asocia con esta URL y es invocada cuando se realiza una solicitud **PUT** a **/productos/** seguido de un número (el código del producto).

Se recuperan los nuevos datos del producto de **request.form**, que son los datos enviados en la solicitud PUT.
La función inicia procesando la imagen cargada como parte del formulario. Primero, se obtiene el nombre de la imagen de manera segura utilizando la función **secure_filename()**. Luego, se genera un nombre de archivo único combinando el nombre base de la imagen, un timestamp obtenido de **time.time()**, y la extensión original del archivo. Después de construir el nuevo nombre de la imagen, se guarda en el sistema de archivos en una ubicación específica definida por la variable ruta_destino. Recordemos que **secure_filename** nos permite asegurarnos que el nombre del archivo es seguro y que se añade una marca de tiempo para evitar colisiones de nombres. Luego la imagen se guarda en el servidor.

Se llama al método **modificar_producto** de la instancia **catalogo** de la clase **Catalogo**, pasando el **codigo** del producto y los nuevos datos.
Este método intenta actualizar el producto en la base de datos.
Si la actualización es exitosa, se devuelve una respuesta JSON con un mensaje de éxito y un código de estado **HTTP 200 (OK).**
Si el producto no se encuentra (por ejemplo, si no hay ningún producto con el **código** dado), se devuelve un mensaje de error con un código de estado **HTTP 404 (No Encontrado).**

En resumen: la ruta **/productos/<int:codigo>** con el método **PUT** en Flask funciona de la siguiente manera:

1. Recibe una solicitud PUT con los nuevos datos del producto y una imagen opcional.
2. Extrae y procesa estos datos, incluyendo la actualización de la imagen en el servidor.
3. Intenta actualizar el producto en la base de datos utilizando el código proporcionado.
4. Devuelve una respuesta JSON indicando el resultado de la operación, ya sea un éxito o un error.

Este endpoint es un ejemplo típico de cómo se manejan las solicitudes de actualización de registros en aplicaciones web a través de una API, permitiendo modificar datos existentes de manera segura y controlada.

Eliminar productos: Métodos y rutas

Método eliminar_producto:

Este método en la clase **Catalogo** está diseñado para eliminar un producto específico de la base de datos, utilizando su código como identificador. Este método representa una operación de **Eliminar** en el marco de las operaciones CRUD (Crear, Leer, Actualizar, Eliminar).

```
    def eliminar_producto(self, codigo):
        # Eliminamos un producto de la tabla a partir de su código
        self.cursor.execute(f"DELETE FROM productos WHERE codigo =
{codigo}")
        self.conn.commit()
        return self.cursor.rowcount > 0
```

Vamos a examinar en detalle cómo funciona este método:

1. **Parámetro del Método**
   Este método toma un único parámetro, **codigo**, que es el identificador único del producto que se desea eliminar.

2. **Ejecución de la Consulta SQL de Eliminación**
   El método ejecuta una consulta SQL **DELETE** para eliminar el producto de la tabla **productos**. La eliminación se basa en el **código** proporcionado.
   Esta consulta elimina el registro (o registros, aunque en la práctica debería ser único debido a la naturaleza del **codigo** como clave primaria) cuyo campo **codigo** coincida con el valor dado.

3. **Confirmación de Cambios y Retorno**
   self.conn.commit() asegura que la eliminación se refleje en la base de datos. Hasta que se ejecute **commit**, la eliminación es temporal y puede ser revertida.
   self.cursor.rowcount devuelve el número de filas afectadas por la última operación de ejecución. Si este número es mayor que 0, significa que al menos un registro fue eliminado, y por lo tanto, se devuelve **True**.
   Si **rowcount** es 0, esto indica que no se eliminó ningún registro (posiblemente porque no existía un producto con el **codigo** proporcionado), y se devuelve **False**.

En resumen: este método realiza las siguientes tareas:
1. Construye y ejecuta una consulta SQL para eliminar un producto específico de la base de datos.
2. Confirma los cambios en la base de datos.
3. Devuelve **True** si la eliminación afectó a alguna fila (indicando éxito), y **False** en caso contrario.

Ruta Eliminar Producto (`/productos/<int:codigo>` - DELETE)

Este código maneja solicitudes DELETE para eliminar productos. Consulta el producto existente, elimina su imagen si existe, y luego lo elimina del catálogo. Luego, devuelve una respuesta adecuada según el resultado de la eliminación.
La ruta Flask **/productos/<int:codigo>** con el método HTTP **DELETE** está diseñada para eliminar un producto específico de la base de datos, utilizando su código como identificador. Este endpoint de la API web facilita la eliminación segura de un registro de producto.

Expliquemos en detalle cómo funciona:

```
@app.route("/productos/<int:codigo>", methods=["DELETE"])
def eliminar_producto(codigo):
    # Primero, obtén la información del producto para encontrar la imagen
    producto = catalogo.consultar_producto(codigo)
    if producto:
        # Eliminar la imagen asociada si existe
        ruta_imagen = os.path.join(ruta_destino, producto['imagen_url'])
        if os.path.exists(ruta_imagen):
            os.remove(ruta_imagen)

        # Luego, elimina el producto del catálogo
        if catalogo.eliminar_producto(codigo):
            return jsonify({"mensaje": "Producto eliminado"}), 200
        else:
            return jsonify({"mensaje": "Error al eliminar el producto"}),
500
    else:
        return jsonify({"mensaje": "Producto no encontrado"}), 404
```

@app.route("/productos/<int:codigo>", methods=["DELETE"]): Este decorador define una ruta en la aplicación Flask que responde a solicitudes HTTP **DELETE**. La parte **<int:codigo>** en la ruta es una variable dinámica que representa el código del producto que se desea eliminar. Flask automáticamente manejará cualquier valor que se pase en esta parte de la URL como un entero y lo asignará al parámetro **codigo** de la función **eliminar_producto**.
La función **eliminar_producto** se asocia con esta URL y es llamada cuando se realiza una solicitud **DELETE** a **/productos/** seguido de un número (el código del producto).

La función comienza consultando el producto para obtener su información, especialmente la URL de la imagen, utilizando **catalogo.consultar_producto(codigo)**.
Si el producto existe, verifica si hay una imagen asociada en el servidor y, si es así, la elimina del sistema de archivos.
Posteriormente, intenta eliminar el producto de la base de datos llamando a **catalogo.eliminar_producto(codigo)**.
Si el producto se elimina correctamente, se devuelve una respuesta JSON con un mensaje de éxito y un código de estado **HTTP 200 (OK)**.
Si ocurre un error durante la eliminación (por ejemplo, si el producto no se puede eliminar de la base de datos por alguna razón), se devuelve un mensaje de error con un código de estado **HTTP 500 (Error Interno del Servidor).**
Si el producto no se encuentra (por ejemplo, si no existe un producto con el **codigo** proporcionado), se devuelve un mensaje de error con un código de estado **HTTP 404 (No Encontrado).**

En resumen: la ruta **/productos/<int:codigo>** con el método **DELETE** en Flask funciona de la siguiente manera:

1. Recibe una solicitud DELETE con el código de un producto específico.
2. Consulta la información del producto, incluida la imagen asociada, y la elimina del servidor si existe.
3. Intenta eliminar el producto de la base de datos.

4. Devuelve una respuesta JSON indicando el resultado de la operación, ya sea un éxito, un error en la eliminación o un error por no encontrar el producto.

Este endpoint es un ejemplo típico de cómo se manejan las solicitudes de eliminación de registros en aplicaciones web a través de una API, permitiendo eliminar datos de manera segura y controlada.

Observaciones Adicionales

- Este código está diseñado para ser un servidor backend para una aplicación web, manejando tanto la lógica de la base de datos como las solicitudes HTTP.
- Se hace uso de seguridad básica para nombres de archivos (con **secure_filename**) y se manejan imágenes como parte de la información del producto.
- La aplicación está estructurada para ser escalable y mantenible, con la lógica de la base de datos encapsulada en una clase y la lógica de la API manejada a través de rutas Flask.

Este código es un ejemplo típico de una aplicación web backend que utiliza Flask y MySQL, y es adecuado para usuarios con un nivel básico a intermedio de experiencia en programación.

Codigo fuente

Este es el código fuente analizado:

```
#------------------------------------------------------------------------
# Instalar con pip install Flask
from flask import Flask, request, jsonify
from flask import request

# Instalar con pip install flask-cors
from flask_cors import CORS

# Instalar con pip install mysql-connector-python
import mysql.connector

# Si es necesario, pip install Werkzeug
from werkzeug.utils import secure_filename

# No es necesario instalar, es parte del sistema standard de Python
import os
import time
#------------------------------------------------------------------------

app = Flask(__name__)
CORS(app)  # Esto habilitará CORS para todas las rutas

class Catalogo:
    # Constructor de la clase
    def __init__(self, host, user, password, database):
        # Primero, establecemos una conexión sin especificar la base de
datos
        self.conn = mysql.connector.connect(
```

```
            host=host,
            user=user,
            password=password
        )
        self.cursor = self.conn.cursor()

        # Intentamos seleccionar la base de datos
        try:
            self.cursor.execute(f"USE {database}")
        except mysql.connector.Error as err:
            # Si la base de datos no existe, la creamos
            if err.errno == mysql.connector.errorcode.ER_BAD_DB_ERROR:
                self.cursor.execute(f"CREATE DATABASE {database}")
                self.conn.database = database
            else:
                raise err

        # Una vez que la base de datos está establecida, creamos la tabla
si no existe
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS productos (
            codigo INT AUTO_INCREMENT,
            descripcion VARCHAR(255) NOT NULL,
            cantidad INT NOT NULL,
            precio DECIMAL(10, 2) NOT NULL,
            imagen_url VARCHAR(255),
            proveedor INT)''')
        self.conn.commit()

        # Cerrar el cursor inicial y abrir uno nuevo con el parámetro
dictionary=True
        self.cursor.close()
        self.cursor = self.conn.cursor(dictionary=True)

    #------------------------------------------------------------------
    def listar_productos(self):
        self.cursor.execute("SELECT * FROM productos")
        productos = self.cursor.fetchall()
        return productos

    #------------------------------------------------------------------
    def consultar_producto(self, codigo):
        # Consultamos un producto a partir de su código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        return self.cursor.fetchone()

    #------------------------------------------------------------------
    def mostrar_producto(self, codigo):
        # Mostramos los datos de un producto a partir de su código
```

```
        producto = self.consultar_producto(codigo)
        if producto:
            print("-" * 40)
            print(f"Código.....: {producto['codigo']}")
            print(f"Descripción: {producto['descripcion']}")
            print(f"Cantidad...: {producto['cantidad']}")
            print(f"Precio.....: {producto['precio']}")
            print(f"Imagen.....: {producto['imagen_url']}")
            print(f"Proveedor..: {producto['proveedor']}")
            print("-" * 40)
        else:
            print("Producto no encontrado.")

    #------------------------------------------------------------------
    def agregar_producto(self, codigo, descripcion, cantidad, precio,
imagen, proveedor):
        # Verificamos si ya existe un producto con el mismo código
        self.cursor.execute(f"SELECT * FROM productos WHERE codigo =
{codigo}")
        producto_existe = self.cursor.fetchone()
        if producto_existe:
            return False

        # Si no existe, agregamos el nuevo producto a la tabla
        sql = "INSERT INTO productos (codigo, descripcion, cantidad,
precio, imagen_url, proveedor) VALUES (%s, %s, %s, %s, %s, %s)"
        valores = (codigo, descripcion, cantidad, precio, imagen,
proveedor)
        self.cursor.execute(sql, valores)
        self.conn.commit()
        return True

    #------------------------------------------------------------------
    def modificar_producto(self, codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nueva_imagen, nuevo_proveedor):
        sql = "UPDATE productos SET descripcion = %s, cantidad = %s,
precio = %s, imagen_url = %s, proveedor = %s WHERE codigo = %s"
        valores = (nueva_descripcion, nueva_cantidad, nuevo_precio,
nueva_imagen, nuevo_proveedor, codigo)
        self.cursor.execute(sql, valores)
        self.conn.commit()
        return self.cursor.rowcount > 0

    #------------------------------------------------------------------
    def eliminar_producto(self, codigo):
        # Eliminamos un producto de la tabla a partir de su código
        self.cursor.execute(f"DELETE FROM productos WHERE codigo =
{codigo}")
        self.conn.commit()
```

```
        return self.cursor.rowcount > 0

#-----------------------------------------------------------------------
# Cuerpo del programa
#-----------------------------------------------------------------------
# Crear una instancia de la clase Catalogo
catalogo = Catalogo(host='localhost', user='root', password='',
database='miapp')

# Carpeta para guardar las imagenes
ruta_destino = './static/imagenes/'

#-----------------------------------------------------------------------
@app.route("/productos", methods=["GET"])
def listar_productos():
    productos = catalogo.listar_productos()
    return jsonify(productos)

#-----------------------------------------------------------------------
@app.route("/productos/<int:codigo>", methods=["GET"])
def mostrar_producto(codigo):
    producto = catalogo.consultar_producto(codigo)
    if producto:
        return jsonify(producto)
    else:
        return "Producto no encontrado", 404

#-----------------------------------------------------------------------
@app.route("/productos", methods=["POST"])
def agregar_producto():
    # Recojo los datos del form
    codigo = request.form['codigo']
    descripcion = request.form['descripcion']
    cantidad = request.form['cantidad']
    precio = request.form['precio']
    proveedor = request.form['proveedor']
    imagen = request.files['imagen']
    nombre_imagen = secure_filename(imagen.filename)

    nombre_base, extension = os.path.splitext(nombre_imagen)
    nombre_imagen = f"{nombre_base}_{int(time.time())}{extension}"
    imagen.save(os.path.join(ruta_destino, nombre_imagen))

    if catalogo.agregar_producto(codigo, descripcion, cantidad, precio,
nombre_imagen, proveedor):
        return jsonify({"mensaje": "Producto agregado"}), 201
    else:
        return jsonify({"mensaje": "Producto ya existe"}), 400
```

```python
#-----------------------------------------------------------------------
@app.route("/productos/<int:codigo>", methods=["PUT"])
def modificar_producto(codigo):
    # Recojo los datos del form
    nueva_descripcion = request.form.get("descripcion")
    nueva_cantidad = request.form.get("cantidad")
    nuevo_precio = request.form.get("precio")
    nuevo_proveedor = request.form.get("proveedor")

    # Procesamiento de la imagen
    imagen = request.files['imagen']
    nombre_imagen = secure_filename(imagen.filename)
    nombre_base, extension = os.path.splitext(nombre_imagen)
    nombre_imagen = f"{nombre_base}_{int(time.time())}{extension}"
    imagen.save(os.path.join(ruta_destino, nombre_imagen))

    # Actualización del producto
    if catalogo.modificar_producto(codigo, nueva_descripcion,
nueva_cantidad, nuevo_precio, nombre_imagen, nuevo_proveedor):
        return jsonify({"mensaje": "Producto modificado"}), 200
    else:
        return jsonify({"mensaje": "Producto no encontrado"}), 404

#-----------------------------------------------------------------------
@app.route("/productos/<int:codigo>", methods=["DELETE"])
def eliminar_producto(codigo):
    # Primero, obtén la información del producto para encontrar la imagen
    producto = catalogo.consultar_producto(codigo)
    if producto:
        # Eliminar la imagen asociada si existe
        ruta_imagen = os.path.join(ruta_destino, producto['imagen_url'])
        if os.path.exists(ruta_imagen):
            os.remove(ruta_imagen)

        # Luego, elimina el producto del catálogo
        if catalogo.eliminar_producto(codigo):
            return jsonify({"mensaje": "Producto eliminado"}), 200
        else:
            return jsonify({"mensaje": "Error al eliminar el producto"}),
500
    else:
        return jsonify({"mensaje": "Producto no encontrado"}), 404

#-----------------------------------------------------------------------
if __name__ == "__main__":
    app.run(debug=True)
```

Con esto, finalizamos la implementación de la API.

ETAPA 5: Desarrollar un frontend para nuestro CRUD

index.html

Esta página actúa como un menú o punto de entrada para distintas funciones de la aplicación web, cada una relacionada con el manejo de productos a través de una API. El uso de una tabla para organizar los enlaces proporciona una estructura clara y sencilla para la navegación.

Código de index.html

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Ejemplo de uso de la API</title>
    <link rel="stylesheet" href="static/css/estilos.css">
</head>

<body>
    <div>
        <h1>Ejemplos de uso de la API</h1>
        <h3>Codo a Codo 2023</h3>
        <table>
            <tr>
                <td class="contenedor-centrado">
                    <a href="altas.html">Alta de productos</a>
                </td>
            </tr>
            <tr>
                <td class="contenedor-centrado">
                    <a href="listado.html">Listado de productos</a>
                </td>
            </tr>
```

```
            <tr>
                <td class="contenedor-centrado">
                    <a href="modificaciones.html">Modificar datos de
productos</a>
                </td>
            </tr>
            <tr>
                <td class="contenedor-centrado">
                    <a href="listadoEliminar.html">Eliminar productos</a>
                </td>
            </tr>
        </table>
</body>

</html>
```

Vamos a desglosar sus partes más relevantes:

Estructura Básica y Metadatos

<!DOCTYPE html>: Indica que el documento es un HTML5.
<html lang="es">: Define que el idioma principal del contenido del documento es español

Sección <head>: Contiene metadatos y enlaces a recursos externos.

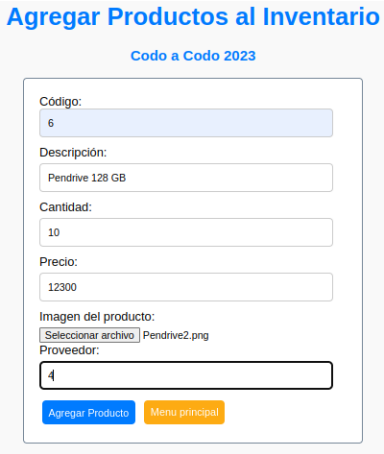
<meta charset="UTF-8">: Establece la codificación de caracteres para el documento en UTF-8, que es una codificación estándar para caracteres.
<meta name="viewport" content="width=device-width, initial-scale=1.0">: Configura la visualización para ser responsiva en dispositivos móviles. Ajusta la escala y el ancho del contenido según el dispositivo.
<title>Ejemplo de uso de la API</title>: Define el título de la página web, que aparecerá en la pestaña del navegador.
<link rel="stylesheet" href="static/css/estilos.css">: Vincula un archivo CSS externo llamado "estilos.css" para aplicar estilos a los elementos de la página.

Contenido del cuerpo

Sección <body>:
Aquí es donde se define el contenido que será visible para los usuarios en la página web.
** :** Un contenedor general para los elementos dentro de él.
<h1> y **<h3>:** Son encabezados que muestran textos "*Ejemplos de uso de la API*" y "*Codo a Codo 2023*", respectivamente.
<table>: Define una tabla que se utiliza para organizar y mostrar los enlaces en forma de lista.
Dentro de <table>, hay varias filas <tr> (table row), cada una conteniendo una celda <td> (table data).
Cada <td> tiene la clase **contenedor-centrado** (definida en "estilos.css" para alinear o estilizar el contenido) y contiene un enlace <a> que lleva a diferentes páginas HTML como "*altas.html*", "*listado.html*", "*modificaciones.html*", y "*listadoEliminar.html*". Estos enlaces sirven para navegar a diferentes funciones relacionadas con los productos, como añadir, listar, modificar y eliminar productos.

Alta de productos (altas.html)

Página web para agregar productos a un inventario. Incluye un formulario para introducir los detalles del producto y un script de JavaScript para manejar el envío de estos datos al servidor de manera asíncrona, sin recargar la página.



Código de altas.html

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Agregar producto</title>
    <link rel="stylesheet" href="static/css/estilos.css">
</head>

<body>
    <h1>Agregar Productos al Inventario</h1>
    <h3>Codo a Codo 2023</h3>
    <!--enctype="multipart/form-data" es necesario para enviar archivos
al back.-->
    <form id="formulario" enctype="multipart/form-data">     >
        <label for=" codigo">Código:</label>
        <input type="text" id="codigo" name="codigo" required><br>
        <label for="descripcion">Descripción:</label>
        <input type="text" id="descripcion" name="descripcion"
required><br>
        <label for="cantidad">Cantidad:</label>
        <input type="number" id="cantidad" name="cantidad" required><br>
        <label for="precio">Precio:</label>
        <input type="number" step="0.01" id="precio" name="precio"
required><br>
        <label for="imagenProducto">Imagen del producto:</label>
```

```
        <input type="file" id="imagenProducto" name="imagen">
        <label for="proveedorProducto">Proveedor:</label>
        <input type="text" id="proveedorProducto" name="proveedor">
        <button type="submit">Agregar Producto</button>
        <a href="index.html">Menu principal</a>
    </form>
    <script src="altas.js"></script>
</body>

</html>
```

Analicemos sus partes más relevantes:

Estructura Básica y Metadatos

Esta parte es la misma que en index.html, solo cambia el contenido de la etiqueta <title>, que ahora es “Agregar producto”.

Contenido del cuerpo

Sección <body>:
Aquí es donde se define el contenido que será visible para los usuarios en la página web.
<h1> y **<h3>:** Son encabezados que indican el propósito de la página y el año de edición del proyecto.
<form id="formulario" enctype="multipart/form-data">: Un formulario para ingresar los detalles del producto. El atributo ***enctype*** se utiliza en un formulario HTML para especificar cómo se deben codificar los datos antes de enviarlos al servidor cuando se utiliza el método POST.
Cuando trabajamos con formularios que incluyen la carga de archivos (por ejemplo, campos de tipo file que permiten a los usuarios seleccionar archivos para ser enviados al servidor), necesitamos utilizar el valor **multipart/form-data**, que indica que los datos del formulario se codificarán como una serie de bloques MIME, permitiendo así que se envíen archivos binarios.
En este caso tenemos un campo de tipo file en el formulario, que permite a los usuarios seleccionar un archivo local y enviarlo al servidor. La codificación "multipart/form-data" es necesaria para manejar adecuadamente la transmisión de archivos binarios.
Etiquetas (<label>) y campos de entrada (<input>): Permiten orientar al usuario sobre los datos que deben ingresarse y cargar el código, descripción, cantidad, precio, seleccionar una imagen y escribir el proveedor del producto. Todos los campos tienen un id y name asociados, que son importantes para la recopilación de datos.
<button type="submit">: Botón para enviar el formulario.
<a href="index.html">: Enlace para volver al menú principal.
Finalmente, por fuera del formulario, tenemos **<script src="altas.js">** que nos vincula el archivo .html con un archivo de JavaScript que controlará el comportamiento del formulario.

Código de altas.js

El script vinculado con **altas.html** se encarga de tomar los datos ingresados en un formulario, enviarlos al servidor para agregar un nuevo producto al inventario y manejar la respuesta del servidor, ya sea un éxito o un error, actualizando la interfaz del usuario en consecuencia.

```
const URL = "http://127.0.0.1:5000/"
// Capturamos el evento de envío del formulario
```

```
document.getElementById('formulario').addEventListener('submit', function
(event) {
    event.preventDefault(); // Evitamos que se envie el form
    var formData = new FormData();
    formData.append('codigo', document.getElementById('codigo').value);
    formData.append('descripcion',
document.getElementById('descripcion').value);
    formData.append('cantidad',
document.getElementById('cantidad').value);
    formData.append('precio', document.getElementById('precio').value);
    formData.append('proveedor',
document.getElementById('proveedorProducto').value);
    formData.append('imagen',
document.getElementById('imagenProducto').files[0]);

    fetch(URL + 'productos', {
        method: 'POST',
        body: formData // Aquí enviamos formData en lugar de JSON
    })
        .then(function (response) {
            if (response.ok) {
                return response.json();
            }
        })
        .then(function (data) {
            alert('Producto agregado correctamente.');
            // Limpiar el formulario para el proximo producto
            document.getElementById('codigo').value = "";
            document.getElementById('descripcion').value = "";
            document.getElementById('cantidad').value = "";
            document.getElementById('precio').value = "";
            document.getElementById('imagen').value = "";
            document.getElementById('proveedor').value = "";
        })
        .catch(function (error) {
            // Mostramos el error, y no limpiamos el form.
            alert('Error al agregar el producto.');
        });
})
```

Análisis detallado de su funcionamiento:

1. **Definición de la URL del Servidor:**
   const URL = "http://127.0.0.1:5000/": Esta constante contiene la dirección del servidor local donde se realizarán las solicitudes. En este caso, es una URL de desarrollo local.

2. **Evento de envío del formulario:**

document.getElementById('formulario').addEventListener('submit', function (event) { ... }): Aquí se agrega un oyente de eventos al formulario con **id='formulario'**. Este oyente reacciona al evento de "submit" (envío) del formulario.
event.preventDefault(): Evita el comportamiento predeterminado del navegador al enviar un formulario, que normalmente recargaría la página.

3. **Creación del objeto FormData:**
var formData = new FormData(): Crea un objeto `FormData`, útil para enviar datos de formulario, incluidos archivos, al servidor.
formData.append(...): Estas instrucciones añaden los valores de los diferentes campos del formulario al objeto `formData`. Los datos incluyen el código, la descripción, la cantidad, el precio, el proveedor y la imagen del producto.
Estos datos se agregan como pares **clave/valor** al objeto para los campos del formulario. Estos valores se obtienen de los elementos del formulario por sus identificadores, por ejemplo:

```
formData.append('codigo', document.getElementById('codigo').value);
```

- **formData.append**: Es el método de la interfaz FormData que se utiliza para agregar un nuevo par clave/valor al objeto FormData.

- **'codigo':** Es la clave que se asigna al valor, en este contexto es el nombre del campo que se está agregando.
- **document.getElementById('codigo').value:** Aquí obtenemos la referencia al elemento HTML del formulario que tiene el id 'codigo', que es un campo de entrada de texto. Para obtener el valor actual del campo de entrada usamos **.value**. De esta manera obtenemos el valor que el usuario ha ingresado en el campo de entrada con el id 'codigo'.

Resultado: El valor obtenido se agrega al objeto FormData con la clave 'codigo'. Esta línea está tomando el valor que un usuario ha ingresado en un campo de formulario con el id 'codigo' y lo está asociando con la clave 'codigo' en el objeto FormData.

En resumen: con la interfaz FormData construimos y gestionamos pares clave/valor representando campos del formulario. Estos pares clave/valor pueden ser campos de texto y con el **método** append agregamos estos pares al objeto FormData, donde la clave es el nombre del campo y el valor es el valor asociado a ese campo.
En el caso de 'imagen' añadimos el primer archivo seleccionado en el campo de entrada de archivo (<input type="file">).

Estos datos de FormData podemos verlos como una construcción de una estructura de datos que representa los datos del formulario. Cuando envías este objeto FormData a través de una solicitud HTTP, generalmente se interpreta en el servidor como una colección de campos y valores que puedes procesar según tus necesidades.
Los pares clave/valor que se utilizan en el objeto FormData se comportan de manera similar a un diccionario en otros lenguajes de programación. En JavaScript, se podría decir que forman parte de una estructura de datos similar a un diccionario.

4. **Envío de Datos al Servidor con Fetch API:**
fetch(URL + 'productos', { ... }): Esta función realiza una solicitud HTTP **POST** al servidor. La URL completa es la combinación de la URL definida y el endpoint **'productos'**.
Segundo parámetro de **fetch**: es una configuración de opciones donde especificamos el método y el cuerpo de la solicitud.

- **method: 'POST':** Define el método HTTP como POST, que se usa para enviar y crear nuevos datos en el servidor.
- **body: formData:** Establece el cuerpo de la solicitud HTTP como el objeto **formData** que contiene los datos del formulario. Dado que formData puede contener archivos, no se utiliza JSON aquí.

5. **Manejo de la Respuesta del Servidor:**
   .then(function (response) { ... }): Esta parte del código maneja la respuesta del servidor. Si la respuesta es exitosa (**response.ok**), convierte los datos de la respuesta a formato JSON.
   .then(function (data) { ... }): En caso de éxito, muestra una alerta informando que el producto se agregó correctamente y limpia los campos del formulario para que puedan ser utilizados para un nuevo producto.

6. **Manejo de Errores:**
   .catch(function (error) { ... }): Captura y maneja cualquier error que pueda ocurrir durante la solicitud fetch. Si hay un error, muestra una alerta de error y no limpia los campos del formulario para que el usuario pueda corregir la entrada.

Listado de productos (listado.html)

Esta página HTML está diseñada para mostrar un listado de productos. Crea una tabla de productos con sus detalles, utilizando JavaScript para realizar una solicitud al servidor y recuperar los datos de los productos, que luego agrega dinámicamente en la tabla. El script maneja tanto la recuperación exitosa de los datos como los posibles errores que puedan surgir durante el proceso.

Código de listado.html

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Listado de Productos</title>
    <link rel="stylesheet" href="static/css/estilos.css">
</head>
```

```
<body>
    <h1>Listado de Productos</h1>
    <h3>Codo a Codo 2023</h3>
    <table>
        <thead>
            <tr>
                <th>Código</th>
                <th>Descripción</th>
                <th>Cantidad</th>
                <th align="right">Precio</th>
                <th>Imagen</th>
                <th align="right">Proveedor</th>
            </tr>
        </thead>
        <tbody id="tablaProductos"> </tbody>
    </table>
    <div class="contenedor-centrado">
        <a href="index.html">Menu principal</a>
    </div>
    <script src="listado.js"></script>
</body>

</html>
```

Analicemos sus partes más relevantes:

Estructura Básica y Metadatos

Esta parte es la misma que en index.html, solo cambia el contenido de la etiqueta <title>, que ahora es “Listado de Productos”.

Contenido del cuerpo

Sección <body>:
<h1> y **<h3>**: Encabezados que muestran el título y el subtítulo de la página.
<table>: Una tabla para mostrar los productos. Dentro de la tabla:
- **<thead>** y **<tr>**: Encabezado de la tabla con columnas para código, descripción, cantidad, precio, imagen y proveedor.
- **<tbody id="tablaProductos">**: Cuerpo de la tabla donde se insertarán los datos de los productos a través de JavaScript.

Código de listado.js

Este código utiliza la API Fetch en JavaScript para realizar una solicitud GET al servidor y obtener información sobre todos los productos. Luego, manipula el DOM para mostrar estos productos en una tabla HTML.

```
const URL = "http://127.0.0.1:5000/"
// Realizamos la solicitud GET al servidor para obtener todos los
productos
```

```
fetch(URL + 'productos')
    .then(function (response) {
        if (response.ok) {
            return response.json();
        }
    })
    .then(function (data) {
        let tablaProductos = document.getElementById('tablaProductos');
        // Iteramos sobre los productos y agregamos filas a la tabla
        for (let producto of data) {
            let fila = document.createElement('tr');
            fila.innerHTML = '<td>' + producto.codigo + '</td>' +
                '<td>' + producto.descripcion + '</td>' +
                '<td align="right">' + producto.cantidad + '</td>' +
                '<td align="right">' + producto.precio + '</td>' +
                // Mostrar miniatura de la imagen
                '<td><img src=./static/imagenes/' + producto.imagen_url +
' alt="Imagen del producto" style="width: 100px;"></td>' +
                '<td align="right">' + producto.proveedor + '</td>';
            tablaProductos.appendChild(fila);
        }
    })
    .catch(function (error) {
        // Código para manejar errores
        alert('Error al obtener los productos.');
    });
```

1. **Definición de la URL del Servidor:**
   const URL = "http://127.0.0.1:5000/": Esta constante contiene la dirección del servidor local donde se recuperarán los datos de los productos. En este caso, es una URL de desarrollo local.

2. **Solicitud GET para Obtener Productos:**
   fetch(URL + 'productos'): Utiliza la API **fetch** para realizar una solicitud HTTP **GET** al servidor. La URL completa es la combinación de **URL** y 'productos', apuntando al endpoint que devuelve los datos de los productos.

3. **Procesamiento de la Respuesta del Servidor:**
   .then(function (response) { ... }): Esta parte del código maneja la respuesta del servidor. Si la respuesta es exitosa (**response.ok**), convierte el cuerpo de la respuesta de formato JSON a un objeto JavaScript y pasa estos datos a la siguiente promesa .then.

4. **Mostrar Datos en la página:**
   .then(function (data) { ... }): Esta función maneja los datos convertidos del JSON.
   let tablaProductos = document.getElementById('tablaProductos'): Selecciona el elemento del DOM donde se mostrarán los productos.
   El bucle **for (let producto of data) { ... }:** Itera sobre cada producto en el array **data**:
   - **let fila = document.createElement('tr'):** Crea una nueva fila de tabla (**<tr>**) para cada producto.

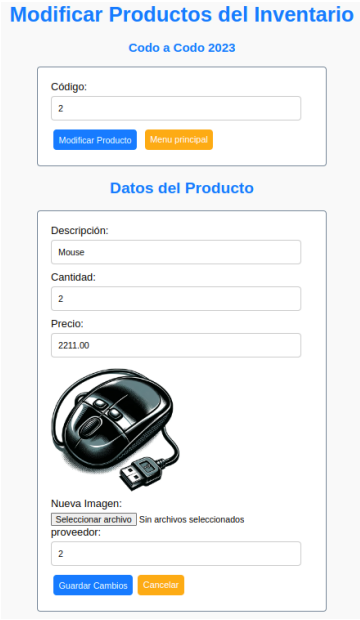
- **fila.innerHTML = ...:** Establece el contenido de la fila, incluyendo las celdas (**<td>**) con el código, la descripción, la cantidad, el precio, la imagen (como una miniatura) y el proveedor del producto.
- **tablaProductos.appendChild(fila)**: Añade la fila al cuerpo de la tabla en el HTML.

5. **Manejo de Errores:**
   .catch(function (error) { ... }): Captura y maneja cualquier error que pueda surgir durante la solicitud **fetch** o el procesamiento de datos. Si ocurre un error, muestra una alerta indicando que hubo un problema al obtener los productos.

En resumen: Este script realiza una solicitud al servidor para obtener una lista de productos y luego muestra esos productos en una tabla en la página web. Gestiona tanto la respuesta exitosa como los posibles errores que puedan surgir en el proceso.

Modificación de productos (modificaciones.html)

Esta página web le permite al usuario obtener los detalles de un producto específico, modificar esos detalles y enviar los cambios al servidor utilizando Vue.js para una experiencia de usuario dinámica y reactiva.



Código de modificaciones.html

```
<!DOCTYPE html>
<html lang="es">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Modificar Producto</title>
    <link rel="stylesheet" href="static/css/estilos.css">
</head>
```

```
<body>
    <h1>Modificar Productos del Inventario</h1>
    <h3>Codo a Codo 2023</h3>
    <div id="app">
        <form @submit.prevent="obtenerProducto">
            <label for="codigo">Código:</label>
            <input type="text" v-model="codigo" required><br>
            <button type="submit">Modificar Producto</button>
            <a href="index.html">Menu principal</a>
        </form>

        <div v-if="mostrarDatosProducto">
            <h2>Datos del Producto</h2>
            <form @submit.prevent="guardarCambios">
                <label for="descripcionModificar">Descripción:</label>
                <input type="text" id="descripcionModificar" v-
model="descripcion" required><br>

                <label for="cantidadModificar">Cantidad:</label>
                <input type="number" id="cantidadModificar" v-
model="cantidad" required><br>

                <label for="precioModificar">Precio:</label>
                <input type="number" step="0.01" id="precioModificar" v-
model="precio" required><br>

                <!-- Imagen actual del producto -->
                <img v-if="imagen_url && !imagenSeleccionada"
:src="'./static/imagenes/' + imagen_url"
                    alt="Imagen del producto" style="max-width: 200px;">

                <!-- Vista previa de la nueva imagen seleccionada -->
                <img v-if="imagenSeleccionada" :src="imagenUrlTemp"
alt="Vista previa de la nueva imagen"
                    style="max-width: 200px;">

                <!-- Input para nueva imagen -->
                <label for="nuevaImagen">Nueva Imagen:</label>
                <input type="file" id="nuevaImagen"
@change="seleccionarImagen"><br>

                <label for="proveModificar">proveedor:</label>
                <input type="number" id="proveModificar" v-
model="proveedor" required><br>

                <button type="submit">Guardar Cambios</button>
                <a href="modificaciones.html">Cancelar</a>
            </form>
```

```
        </div>
    </div>

    <script src="https://unpkg.com/vue@next"></script>
    <script src="modificaciones.js"></script>

</body>
</html>
```

Analicemos sus partes más relevantes:

Estructura Básica y Metadatos

Esta parte es la misma que en index.html, solo cambia el contenido de la etiqueta <title>, que ahora es “Modificar Producto”.

Contenido del cuerpo

Sección <body>:
<h1> y **<h3>**: Encabezados para el título y subtítulo de la página.
** :** Contenedor principal que será controlado por Vue.js. Este contenedor tendrá dos formularios.

Primer formulario: Selector de producto
<form @submit.prevent="obtenerProducto">: Formulario con un manejador de eventos de Vue.js para evitar el envío por defecto y llamar a la función **obtenerProducto** cuando el formulario se envíe.

- **<label for="codigo">Código:</label> y <input type="text" v-model="codigo" required>**: Es un campo de entrada para el código del producto. Además establece una conexión bidireccional entre el valor del campo de entrada y la propiedad **codigo** en los datos de Vue.js. Cualquier cambio en el campo de entrada se reflejará en la propiedad **codigo**, y viceversa.
- **<button type="submit">Modificar Producto</button>:** Un botón que envía el formulario cuando se hace clic. El tipo "submit" indica que es un botón de envío.
- **Menu principal:** Un enlace que redirige a "index.html" cuando se hace clic. Este enlace se presenta como una opción para volver al menú principal sin enviar el formulario.

Contenedor y formulario secundario:
** :** Esta sección se muestra solo si mostrarDatosProducto es verdadero.
<h2>Datos del Producto</h2>: Encabezado de nivel 2 que indica que se presentarán los datos del producto.
<form @submit.prevent="guardarCambios">: Formulario con manejador de eventos para evitar el envío por defecto y llamar a la función guardarCambios desarrollada en JavaScript

- **<label for="descripcionModificar">Descripción:</label>:** Etiqueta para describir el campo de entrada.
- **<input type="text" id="descripcionModificar" v-model="descripcion" required>**: Campo de entrada para la descripción del producto. A través de v-model enlazamos el valor del campo con la propiedad **descripcion** en los datos de Vue.js. Con **required** indicamos que este campo es obligatorio.

- **<label for="cantidadModificar">Cantidad:</label>:** Etiqueta para describir el campo de entrada.
- **<input type="number" id="cantidadModificar" v-model="cantidad" required>:** Campo de entrada para la cantidad del producto. A través de v-model enlazamos el valor del campo con la propiedad **cantidad** en los datos de Vue.js. Con **required** indicamos que este campo es obligatorio.
- **<label for="precioModificar">Precio:</label>:** Etiqueta para describir el campo de entrada
- **<input type="number" step="0.01" id="precioModificar" v-model="precio" required>**: Campo de entrada para el precio del producto. A través de v-model enlazamos el valor del campo con la propiedad **precio** en los datos de Vue.js. La propiedad **step="0.01"** establece el paso para admitir valores decimales. Con **required** indicamos que este campo es obligatorio.
- **<img v-if="imagen_url && !imagenSeleccionada" :src="'./static/imagenes/' + imagen_url" alt="Imagen del producto" style="max-width: 200px;">:** Representa la imagen actual del producto. Se muestra solo si hay una imagen actual y no se ha seleccionado una nueva. Se utiliza **:src** para enlazar el atributo src con la ruta de la imagen actual, con **style="max-width: 200px;"** establece el ancho máximo de la imagen.
- **<img v-if="imagenSeleccionada" :src="imagenUrlTemp" alt="Vista previa de la nueva imagen" style="max-width: 200px;">:** Representa la vVista previa de la nueva imagen seleccionada. Se muestra solo si se ha seleccionado una nueva imagen. Se utiliza **:src** para enlazar el atributo src con la URL temporal de la nueva imagen, con **style="max-width: 200px;"** establece el ancho máximo de la imagen.
- **<label for="nuevaImagen">Nueva Imagen:</label>:** Etiqueta para describir el campo de entrada.
- **<input type="file" id="nuevaImagen" @change="seleccionarImagen">:** Es el input para nueva imagen, es un campo de entrada de tipo archivo. Con **@change="seleccionarImagen"** llama a la función **seleccionarImagen** cuando se selecciona una nueva imagen.
- **<label for="proveModificar">Proveedor:</label>:** Etiqueta para describir el campo de entrada.
- **<input type="number" id="proveModificar" v-model="proveedor" required>**: Campo de entrada para el proveedor del producto. A través de v-model enlazamos el valor del campo con la propiedad **proveedor** en los datos de Vue.js. Con **required** indicamos que este campo es obligatorio.
- **<button type="submit">Guardar Cambios</button>:** Botón para enviar el formulario y llamar a la función **guardarCambios**.
- **Cancelar**: Enlace para cancelar la modificación y redirigir a **"modificaciones.html"**.

Código de modificaciones.js

Esta parte del código establece la base para una aplicación Vue.js, definiendo una URL de conexión al servidor y estableciendo un conjunto de variables que se usarán para manejar datos como el código, la descripción, la cantidad, el precio y la imagen de un producto, así como para controlar algunos aspectos de la interfaz de usuario:

```
const URL = "http://127.0.0.1:5000/"

const app = Vue.createApp({
```

```
    data() {
        return {
            codigo: '',
            descripcion: '',
            cantidad: '',
            precio: '',
            proveedor: '',
            imagen_url: '',
            imagenUrlTemp: null,
            mostrarDatosProducto: false,
        };
    },
    methods: {
        obtenerProducto() {
            fetch(URL + 'productos/' + this.codigo)
                .then(response => response.json())
                .then(data => {
                    this.descripcion = data.descripcion;
                    this.cantidad = data.cantidad;
                    this.precio = data.precio;
                    this.proveedor = data.proveedor;
                    this.imagen_url =  data.imagen_url;
                    this.mostrarDatosProducto = true;
                })
                .catch(error => console.error('Error:', error));
        },
        seleccionarImagen(event) {
            const file = event.target.files[0];
            this.imagenSeleccionada = file;
            this.imagenUrlTemp = URL.createObjectURL(file); // Crea una
URL temporal para la vista previa
        },
        guardarCambios() {
            const formData = new FormData();
            formData.append('codigo', this.codigo);
            formData.append('descripcion', this.descripcion);
            formData.append('cantidad', this.cantidad);
            formData.append('proveedor', this.proveedor);
            formData.append('precio', this.precio);

            if (this.imagenSeleccionada) {
                formData.append('imagen', this.imagenSeleccionada,
this.imagenSeleccionada.name);
            }

            fetch(URL + 'productos/' + this.codigo, {
                method: 'PUT',
                body: formData,
            })
```

```
                .then(response => response.json())
                .then(data => {
                    alert('Producto actualizado correctamente');
                    this.limpiarFormulario();
                })
                .catch(error => {
                    console.error('Error:', error);
                    alert('Error al actualizar el producto');
                });
        },
        limpiarFormulario() {
            this.codigo = '';
            this.descripcion = '';
            this.cantidad = '';
            this.precio = '';
            this.imagen_url = '';
            this.imagenSeleccionada = null;
            this.imagenUrlTemp = null;
            this.mostrarDatosProducto = false;
        }
    }
});

app.mount('#app');
```

Veamos que hace cada parte del código:

1. **Definición de la URL del Servidor:**
   const URL = "http://127.0.0.1:5000/": Esta constante contiene la dirección del servidor local. En este caso, es una URL de desarrollo local, es decir que el servidor estaría corriendo en tu propia máquina.

2. **Creación de la Aplicación Vue:**
   const app = Vue.createApp({ ... }): Esta línea crea una nueva aplicación Vue. Vue.createApp es un método para iniciar una nueva aplicación Vue. Dentro de **Vue.createApp**, se define un objeto con varias propiedades y métodos. Veamos cada parte:

   a. **Definición de la Función data:**
      data() { ... }: Esta es una función que Vue utiliza para declarar las variables reactivas. Estas variables son parte del estado de la aplicación y Vue las rastrea para actualizar la interfaz de usuario cuando cambian sus valores.
      return { ... }: La función **data** devuelve un objeto. Este objeto contiene las siguientes propiedades:
      - **codigo**: Cadena vacía, usada para almacenar el código identificativo del producto.
      - **descripcion:** Cadena vacía, usada para almacenar la descripción del producto.
      - **cantidad:** Cadena vacía, usada para almacenar la cantidad del producto.
      - **precio:** Cadena vacía, usada para almacenar el precio del producto.
      - **proveedor:** Cadena vacía, usada para almacenar el código del proveedor.
      - **imagen_url:** Cadena vacía, usada para almacenar la URL de la imagen del producto.

- **imagenUrlTemp:** Inicialmente null, usada para almacenar una URL temporal de una imagen seleccionada para previsualización.
- **mostrarDatosProducto:** Valor booleano (false inicialmente) que se usa para controlar la visualización de los datos del producto en la interfaz.

b. Definición de Métodos en Vue.js:

methods: { ... }: Esta sección del código define los métodos (funciones) que pueden ser utilizados en la aplicación Vue.

Método obtenerProducto: Se usa para obtener los detalles de un producto desde un servidor.

- **fetch(URL + 'productos/' + this.codigo):** Realiza una solicitud de red al servidor para obtener los datos del producto. Usa la URL definida anteriormente y añade **'productos/'** seguido del código del producto.
- **.then(response => response.json()):** Una vez que la respuesta llega del servidor, se convierte de formato JSON a un objeto JavaScript.
- **.then(data => { ... }):** En este bloque, se asignan los datos obtenidos a las variables correspondientes de la aplicación Vue.
- **.catch(error => console.error('Error:', error)):** Si ocurre un error durante la solicitud, se captura y se imprime en la consola.

Método seleccionarImagen: Se activa cuando el usuario selecciona una imagen para cargar.

- **const file = event.target.files[0]:** Obtiene el primer archivo seleccionado por el usuario.
- **this.imagenSeleccionada = file:** Guarda el archivo en una variable.
- **this.imagenUrlTemp = URL.createObjectURL(file):** Crea y almacena una URL temporal para la imagen seleccionada, que puede ser utilizada para mostrar una vista previa de la imagen.

Método guardarCambios: Se usa para enviar los datos modificados del producto al servidor.

- **const formData = new FormData():** Crea un nuevo objeto FormData, que se utiliza para enviar datos de formulario a través de una solicitud HTTP.
- **formData.append(...):** Añade los datos del producto al objeto formData.
- **if (this.imagenSeleccionada) { ... }:** Si se ha seleccionado una imagen nueva, la añade al formData. En este caso, append no tiene dos argumentos (name, value) sino que tiene tres: name, value, filename. Esto es así porque **filename** es el nombre del archivo asociado al campo. Esto se utiliza cuando se está trabajando con campos de tipo archivo (<input type="file">). El tercer argumento, filename, se utiliza para especificar el nombre del archivo que se enviará al servidor.
- **fetch(URL + 'productos/' + this.codigo, { ... }):** Envía los datos modificados al servidor mediante una solicitud HTTP PUT. En el segundo parámetro de **fetch**: especificamos el método (PUT) y el cuerpo de la solicitud con el objeto **formData**. Dado que formData puede contener archivos, no se utiliza JSON aquí.
- **.then(response => response.json()):** Convierte la respuesta del servidor a un objeto JavaScript.
- **.then(data => { ... }):** Muestra una alerta indicando que el producto ha sido actualizado y luego llama al método **limpiarFormulario**.
- **.catch(error => { ... }):** Captura y muestra errores si la solicitud falla.

Método limpiarFormulario: Restablece todas las variables relacionadas con el formulario a sus valores iniciales, lo que efectivamente "limpia" el formulario.

3. **Montar la Aplicación Vue:**
   app.mount('#app'); es la instrucción que activa la aplicación Vue, conectándola con un elemento específico del HTML. Esto permite que Vue maneje dinámicamente la parte de la página web dentro de ** ... **, haciéndola interactiva y reactiva a los cambios de datos.
   La línea **app.mount('#app');** en el script de Vue.js tiene una función muy específica y crucial en el contexto de una aplicación Vue:
   - **app:** Es una referencia a la instancia de la aplicación Vue que se creó previamente en el script con **Vue.createApp({...})**.
   - **.mount('#app'):** Este es un método que se llama sobre la instancia de la aplicación Vue (app). Su propósito es "montar" o activar la aplicación Vue en el DOM del navegador.
   - **'#app':** Es un selector CSS que apunta a un elemento en el HTML con el id igual a app. En el HTML, hay un ** **, que es el punto de anclaje para toda la aplicación Vue.

Funcionamiento:
Cuando se llama a **app.mount('#app')**, Vue busca en el documento HTML un elemento con el id app. Vue entonces toma el control de este elemento y de todo su contenido.
Esto significa que Vue puede reaccionar a los cambios en sus datos y actualizar automáticamente el HTML en este elemento. También maneja eventos, como clicks o entradas de formulario, y actualiza los datos según las interacciones del usuario.
Todo el código de Vue definido en la instancia app (como datos, métodos, etc.) solo afectará y será aplicable dentro de este elemento.
En resumen: este script permite a los usuarios obtener los detalles de un producto, seleccionar una nueva imagen para el producto, enviar los cambios al servidor y limpiar el formulario una vez que los cambios han sido enviados y procesados. Todo esto forma parte de la funcionalidad de una aplicación Vue.js para modificar información de productos en un sistema de gestión.

Baja de productos (listadoEliminar.html)

Esta página web le permite al usuario obtener un listado de los productos, con la posibilidad de eliminarlos.

Código de listadoEliminar.html

```
<!DOCTYPE html>
<html lang="es">

<head>
```

```
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Listado de Productos</title>
    <link rel="stylesheet" href="static/css/estilos.css">
</head>

<body>
    <h1>Baja de Productos</h1>
    <h3>Codo a Codo 2023</h3>
    <table>
        <thead>
            <tr>
                <th>Código</th>
                <th>Descripción</th>
                <th>Cantidad</th>
                <th align="right">Precio</th>
                <th>Acciones</th>
            </tr>
        </thead>
        <tbody>
            <tr v-for="producto in productos">
                <td>{{ producto.codigo }}</td>
                <td>{{ producto.descripcion }}</td>
                <td>{{ producto.cantidad }}</td>
                <td align="right">{{ producto.precio }}</td>
                <td><button
@click="eliminarProducto(producto.codigo)">Eliminar</button></td>
            </tr>
        </tbody>
    </table>
    <div class="contenedor-centrado">
        <a href="index.html">Menu principal</a>
    </div>
    <script src="https://unpkg.com/vue@next"></script>
    <script src="listadoEliminar.js"></script>
</body>

</html>
```

Analicemos sus partes más relevantes:

Estructura Básica y Metadatos

Esta parte es la misma que en index.html, solo cambia el contenido de la etiqueta <title>, que ahora es “Listado de Productos”.

Contenido del cuerpo

Sección <body>:
<h1> y **<h3>**: Encabezados que muestran el título y el subtítulo de la página.
<table>: Una tabla para mostrar los productos. Dentro de la tabla:
<thead> y **<tr>**: Encabezado de la tabla con columnas para código, descripción, cantidad, precio y acciones.
<tbody>: Sección del cuerpo de la tabla donde se mostrarán los datos de los productos.
<tr v-for="producto in productos">: Utiliza la directiva **v-for** de Vue.js para iterar sobre el array de productos (productos) y crear una fila por cada producto.
<td>{{ producto.codigo }}</td>, <td>{{ producto.descripcion }}</td>, etc.: Celdas de la fila que muestran los detalles de cada producto, como el código, la descripción, la cantidad, el precio, y un botón de acción para eliminar el producto.
<button @click="eliminarProducto(producto.codigo)">Eliminar</button>: Un botón dentro de cada fila que, cuando se hace clic, activa la función **eliminarProducto** de Vue.js, pasando como argumento el código del producto.
** :** Un contenedor con una clase de estilo que centra su contenido.
Menu principal: Un enlace que redirige a index.html.
<script src="https://unpkg.com/vue@next"></script>: Enlaza la biblioteca Vue.js desde la CDN de unpkg.
<script src="listadoEliminar.js"></script>: Enlaza con el script "listadoEliminar.js" que contiene la lógica de Vue.js para esta página.

En resumen, este HTML crea una interfaz para listar productos con la opción de eliminarlos. La funcionalidad dinámica y la gestión de eventos están impulsadas por Vue.js, lo que permite una experiencia de usuario más interactiva.

Código de listadoEliminar.js

Este script, desarrollado con Vue.js, gestiona la visualización y eliminación de productos en una interfaz web. Al cargar la página, se obtiene la lista de productos desde un servidor mediante una solicitud HTTP GET. Luego, Vue.js dinámicamente actualiza la interfaz para mostrar estos productos en una tabla. La función **eliminarProducto(codigo)** permite al usuario eliminar un producto específico. Cuando se confirma la eliminación, se envía una solicitud HTTP DELETE al servidor, y si es exitosa, la tabla se actualiza para reflejar la eliminación. Este script combina la eficiencia de Vue.js para la manipulación de la interfaz de usuario con las solicitudes HTTP para interactuar con un servidor de productos.

```
const URL = "http://127.0.0.1:5000/"
const app = Vue.createApp({
    data() {
        return {
            productos: []
        }
    },
    methods: {
        obtenerProductos() {
            // Obtenemos el contenido del inventario
            fetch(URL + 'productos')
                .then(response => {
                    // Parseamos la respuesta JSON
                    if (response.ok) { return response.json(); }
```

```
                })
                .then(data => {
                    // El código Vue itera este elemento para generar la
tabla
                    this.productos = data;
                })
                .catch(error => {
                    console.log('Error:', error);
                    alert('Error al obtener los productos.');
                });
        },
        eliminarProducto(codigo) {
            if (confirm('¿Estás seguro de que quieres eliminar este
producto?')) {
                fetch(URL + `productos/${codigo}`, { method: 'DELETE' })
                    .then(response => {
                        if (response.ok) {
                            this.productos =
this.productos.filter(producto => producto.codigo !== codigo);
                            alert('Producto eliminado correctamente.');
                        }
                    })
                    .catch(error => {
                        alert(error.message);
                    });
            }
        }
    },
    mounted() {
        //Al cargar la página, obtenemos la lista de productos
        this.obtenerProductos();
    }
});
app.mount('body');
```

Veamos que hace cada parte del código:

1. **Definición de la URL del Servidor:**
   const URL = "http://127.0.0.1:5000/": Esta constante contiene la dirección del servidor local. En este caso, es una URL de desarrollo local, es decir que el servidor estaría corriendo en tu propia máquina.

2. **Creación de la Aplicación Vue:**
   const app = Vue.createApp({ ... }): Inicia una nueva aplicación Vue. Dentro de esta función se definen la estructura de datos y los métodos de la aplicación.

3. **Estructura de Datos (data):**
   data() { return { productos: [] } }: Define una propiedad **productos** en el estado de la aplicación Vue. Inicialmente, es un array vacío que almacenará los datos de los productos obtenidos del servidor.

4. **Métodos de Vue (methods):**

   Método obtenerProductos: Se utiliza para obtener los productos del servidor.
   - **fetch(URL + 'productos'):** Realiza una solicitud GET al servidor y obtener la lista de productos.
   - **.then(response => { ... }):** Procesa la respuesta. Si es exitosa (**response.ok**), convierte los datos de la respuesta de formato JSON a un objeto JavaScript.
   - **.then(data => { this.productos = data; }):** Asigna los datos de los productos obtenidos a la propiedad **productos** del estado de Vue.
   - **.catch(error => { ... }):** Captura y maneja errores, mostrando una alerta en caso de error al obtener los productos.

   Método eliminarProducto(codigo): Se utiliza para eliminar un producto.
   - Se muestra un diálogo de confirmación. Si el usuario confirma, se realiza una solicitud DELETE al servidor a través de **fetch(URL + 'productos/${codigo}', {method: 'DELETE' })**.
   - Actualiza la lista de productos (**this.productos = this.productos.filter(...)**) para reflejar la eliminación del producto.
   - Maneja la respuesta del servidor y posibles errores, mostrando alertas adecuadas.

5. **Ciclo de Vida de Vue (mounted):**
   mounted(): Es un gancho del ciclo de vida de Vue que se llama después de que la instancia de Vue ha sido montada. Aquí, se llama al método obtenerProductos para cargar la lista de productos cuando la página se carga por primera vez.

6. **Montaje de la Aplicación Vue:**
   app.mount('body'): Monta la aplicación Vue en el elemento **<body>** del DOM. Esto activa Vue en la página, haciendo que sea reactivo y maneje el contenido dinámicamente según los datos y las interacciones del usuario.
   Es decir, se crea una aplicación Vue que carga y muestra una lista de productos desde un servidor, permite la eliminación de productos con confirmación del usuario, y actualiza la interfaz de forma dinámica en respuesta a las interacciones del usuario y los cambios de datos.

Hoja de estilos (estilos.css)

```
/* Estilos para todo el proyecto */
body {
    font-family: Arial, sans-serif;
    background-color: #f9f9f9;
}

.contenedor-centrado {
    display: flex;
    width: 100%;
    justify-content: center;
}

p {
```

```
    font-family: 'Times New Roman', Times, serif;
    background-color: #f9f9f9;
    max-width: 400px;
    margin: 0 auto;
    padding: 20px;
}

h1, h2, h3, h4, h5, h6 {
    text-align: center;
    color: #007bff;
}

form {
    max-width: 400px;
    margin: 0 auto;
    padding: 20px;
    background-color: white;
    border: 1px solid lightslategray;
    border-radius: 5px;
}

table {
    max-width: 90%;
    margin: 0 auto;
    padding: 20px;
    background-color: white;
    border: 1px solid lightslategray;
    border-radius: 5px;
}

label {
    display: block;
    margin-bottom: 5px;
}

input[type="text"], input[type="number"], textarea {
    width: 90%;
    padding: 10px;
    margin-bottom: 10px;
    border: 1px solid #cccccc;
    border-radius: 5px;
}

input[type="submit"] {
    padding: 10px;
    background-color: #007bff;
    color: #ffffff;
    border: none;
    border-radius: 5px;
```

```
    cursor: pointer;
}

input[type="submit"]:hover {
    background-color: #0056b3;
}

button {
    padding: 8px;
    margin: 4px;
    background-color: #007bff;
    color: #ffffff;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

button:hover {
    background-color: #0056b3;
}

a {
    padding: 8px;
    margin: 4px;
    background-color: #fdab13;
    color: #ffffff;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    text-decoration: none;
    font-size: 85%;
}

a:hover {
    background-color: #cd7b00;
}
```