

DOCUMENTACION API RESTFUL

1. INTRODUCCION

La presente documentación está destinada a los programadores que quieran hacer uso de la API de facturación online y quieran ponerlo en funcionamiento, entender la arquitectura de código, su base de datos y los fundamentos de su elección, el manejo de las configuraciones correspondientes y pasos para su puesta en marcha, como también conocer las mejoras planteadas a futuro.

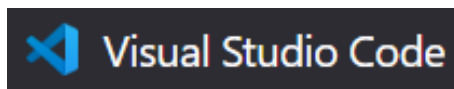
2. PROCEDIMIENTOS INICIALES Y PUESTA EN FUNCIONAMIENTO DE LA API

Para la puesta en funcionamiento de la API se recomienda la utilización de diferentes programas para lograr un funcionamiento más eficiente.

2.1 VISUAL STUDIO CODE

Como editor de texto se recomienda el uso de VISUAL STUDIO CODE.

Link de Descarga: <https://code.visualstudio.com/download>



2.2 XAMPP

Como motor de base de datos se recomienda el uso de XAMPP, el cual permite con APACHE crear un servidor local "Local Host" para hacer las correspondientes consultas desde la API: FrontEnd al BackEnd, y con MYSQL permitirá gestionar la base de datos y su correspondiente estructura.

Link de Descarga: <https://www.apachefriends.org/es/index.html>



Nota 1: Se encuentra configurado por defecto al puerto 3306, y si otro programa se encuentra utilizándolo no dejara que el XAMPP se ejecute correctamente. Ante esta situación ingresar como administrador al CMD, realizar una consulta sobre el estado de disponibilidad del puerto 3306 y de existir esa situación proceder a suspenderlo para que se ponga a disposición. Comandos para ejecutar la acción descripta:

```
C:\WINDOWS\system32>netstat -nao|findstr 0.0.0.0:3306
TCP    0.0.0.0:3306        0.0.0.0:0          LISTENING        5400
TCP    0.0.0.0:33060       0.0.0.0:0          LISTENING        5400

C:\WINDOWS\system32>taskkill /pid 5400 /f
Correcto: se terminó el proceso con PID 5400.
```

Nota 2: Tener en cuenta la configuración de privilegios para crear diferentes usuarios.

2.3 SISTEMA DE CONTROL DE VERSIONES

Como sistema de control de versiones se utiliza GIT y el repositorio GITHUB donde se encontraran todos los archivos, documentación y código necesarios.

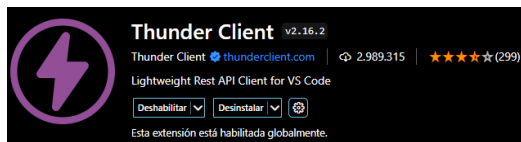
Link de Descarga: <https://git-scm.com/>

Link del Repositorio: <https://github.com/SaizMarcelo/PROYECTO-INFORMATICO>



2.4 TESTEOS DEL LADO DEL CLIENTE: THUNDER CLIENT

Los testeos realizados por el equipo de programación de la API: FACTURACION ONLINE son realizados utilizando la extensión Thunder Client, extensión del VISUAL STUDIO CODE.

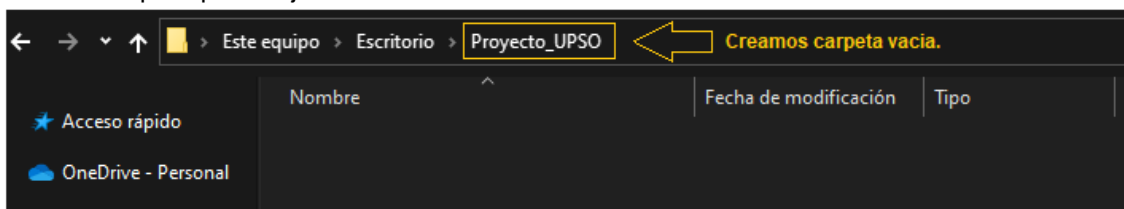


2.5 PROCEDIMIENTO PUESTA EN FUNCIONAMIENTO

Configuraciones necesarias y requeridas para la puesta en funcionamiento de la API: FACTURACION ONLINE.

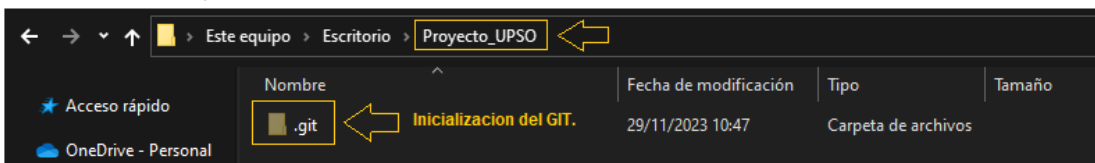
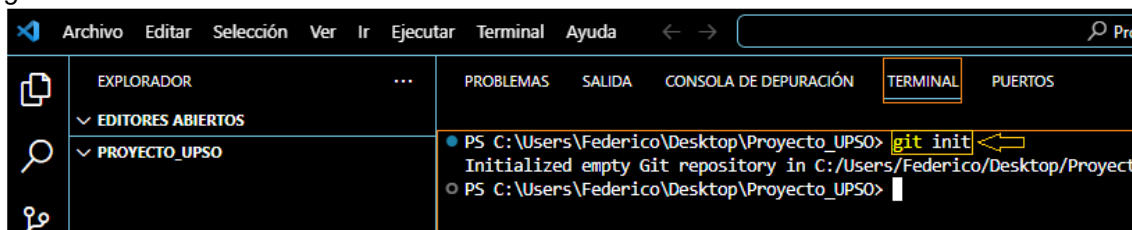
2.5.1. CREAR CARPETA

Crear en el escritorio local una carpeta vacía en la cual se vinculará git, se descargará toda la documentación y archivos necesarios y, se creará la zona de trabajo virtual y configuraciones necesarias para poder ejecutar de forma local la API: FACTURACION ONLINE.



2.5.2. INICIALIZACION GIT A LA CARPETA VACIA

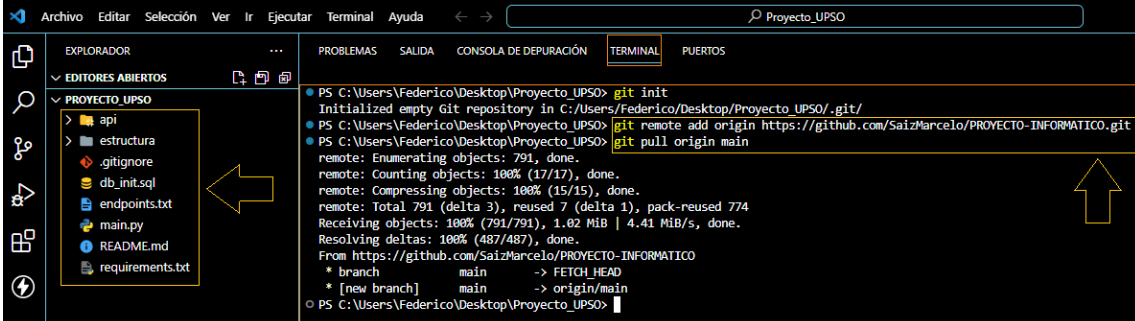
Vincular el sistema de control de versiones con la carpeta creada en el punto 2.5.2. En el editor de texto Visual Studio Code, posicionarse en la carpeta correspondiente e ingresar el comando `git init`.



2.5.3. AGREGAR AL REPOSITORIO LOCAL EL REPOSITORIO DE GITHUB: API: FACTURACION ONLINE.

En la terminal vincular el repositorio a la ruta. Luego realizar un *pull* a dicha ruta para traer al repositorio local todos los archivos del github.

```
git remote add origin https://github.com/SaizMarcelo/PROYECTO-INFORMATICO.git
git pull origin main
```

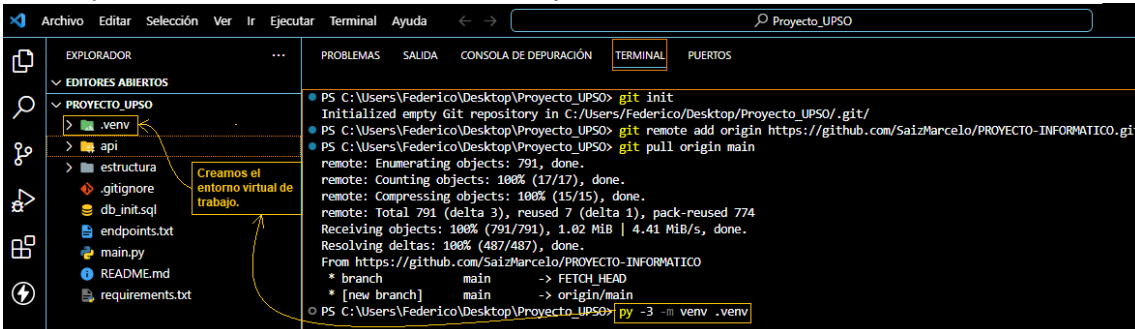


The screenshot shows the VS Code interface. The terminal window displays the execution of `git init`, `git remote add origin https://github.com/SaizMarcelo/PROYECTO-INFORMATICO.git`, and `git pull origin main`. The output shows that the repository was initialized, the remote was added, and the main branch was pulled from the origin. The file explorer on the left shows the project structure with a new `.git` directory created.

2.5.4. CREACIÓN DEL ENTORNO VIRTUAL DE TRABAJO.

Crear un entorno virtual de trabajo para que toda la API se encuentre contenida en el mismo. En la terminal indicar el comando: `py -3 -m venv .venv`. El 3 hace referencia a la versión de Python con la que se creará el entorno virtual de trabajo.

```
py -3 -m venv .venv
```

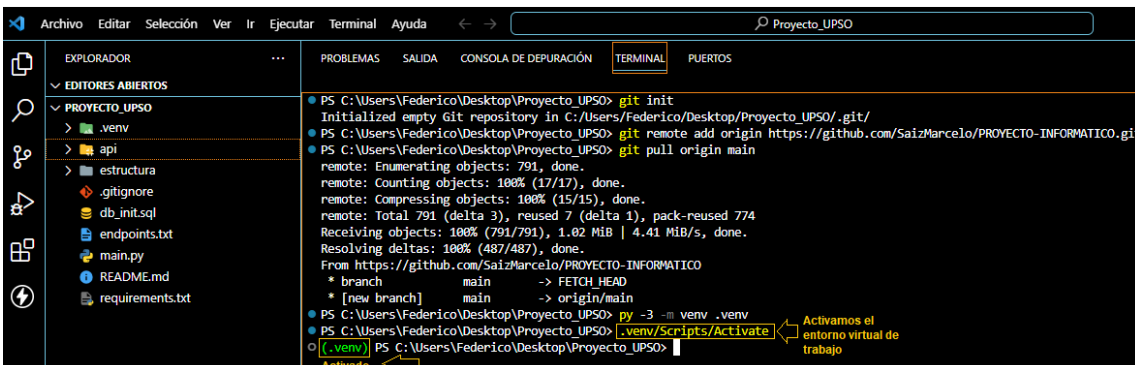


The screenshot shows the VS Code interface. The terminal window displays the execution of `py -3 -m venv .venv`. The output shows that the virtual environment was created successfully. The file explorer on the left shows the project structure with a new `.venv` directory created. A yellow arrow points to the `.venv` directory with the text "Creamos el entorno virtual de trabajo."

2.5.5. ACTIVACION DEL ENTORNO VIRTUAL:

Activar el entorno virtual con el comando `.venv/Scripts/Activate`

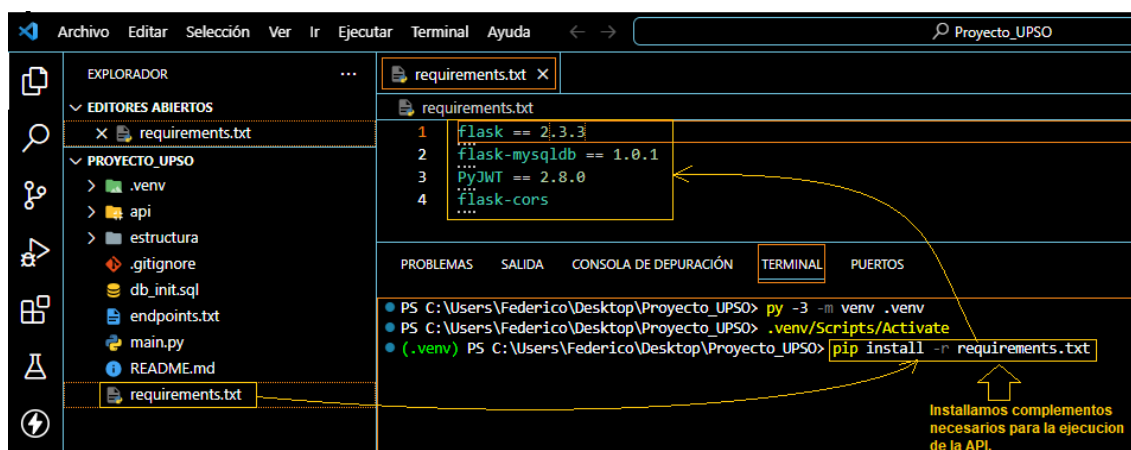
```
.venv/Scripts/Activate
```



The screenshot shows the VS Code interface. The terminal window displays the execution of `.venv/Scripts/Activate`. The output shows that the virtual environment was activated successfully. The file explorer on the left shows the project structure with the `.venv` directory activated. A yellow arrow points to the `.venv` directory with the text "Activamos el entorno virtual de trabajo".

2.5.6. DESCARGA DE LOS REQUERIMIENTOS MINIMOS PARA LA EJECUCION DE LA API: FACTURACION ONLINE:

El funcionamiento de una API requiere diferentes librerías con las cuales se trabaja, por ello dentro del archivo "requirements.txt" se encuentran todos los requerimientos mínimos y utilizados por el equipo de programadores para la API, por ello será necesario descargarlos en el entorno de trabajo virtual, ejecutando el comando `pip install -r requirements.txt`



Con el comando `pip list`, se podrá consultar la existencia de los complementes.



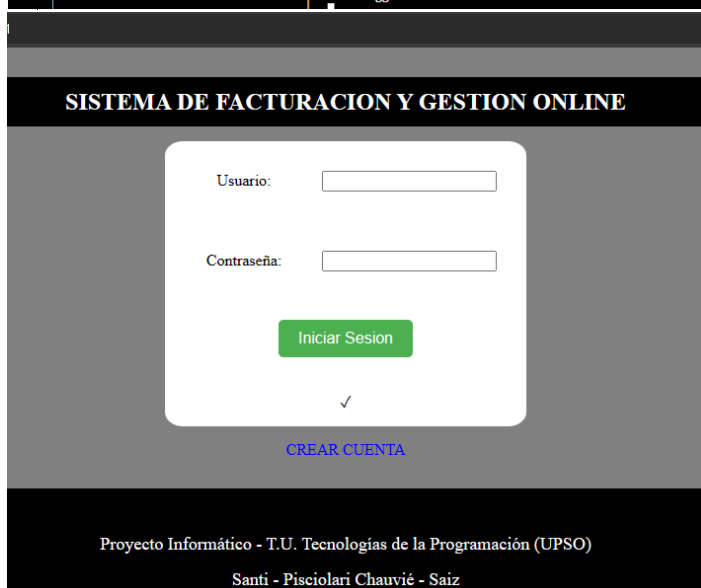
2.5.7 EJECUTAR LA API.

Ejecutar con el comando `python main.py` y luego ingresar a visualizar el vínculo que se obtiene como respuesta `http://127.0.0.1::4500`.

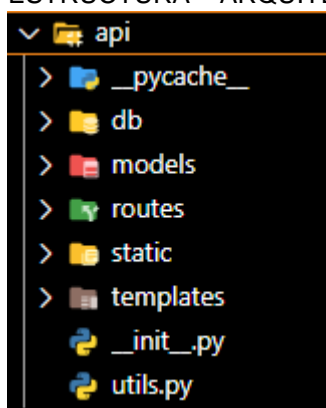
The screenshot shows a code editor with a file explorer on the left. The file explorer shows a project named 'PROYECTO_UPSO' with subfolders like '.venv', 'api', 'estructura', and files like 'db_init.sql', 'endpoints.txt', 'main.py', 'README.md', and 'requirements.txt'. The main editor window shows the 'requirements.txt' file with the following content:

```
1 flask == 2.3.3
2 flask-mysqldb == 1.0.1
3 PyJWT == 2.8.0
4 flask-cors
```

Below the editor, the terminal window is open, showing the command prompt and the output of running 'py main.py'. The output includes a warning about using a development server and the URL 'http://127.0.0.1:4500'.

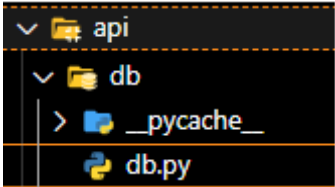


2.6 ESTRUCTURA – ARQUITECTURA - API



2.6.1. api --> db --> db.py:

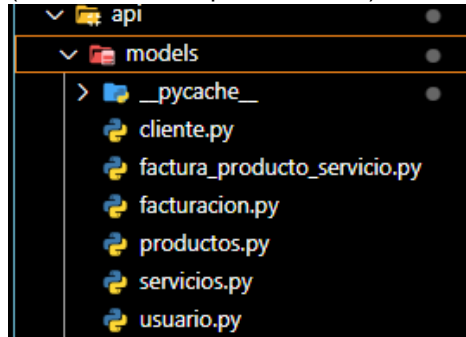
En esta ruta esá alojado el archivo “db.py” que contiene las configuraciones requeridas por la base de datos relacionadas a MySQL. También se encuentra definido el DBError ante la existencia de errores ante la consulta a la base de datos.



```
1 # Importamos la app de api para utilizar la configuracion:
2 from api import app
3 # Importamos la clase MySQL de la libreria:
4 from flask_mysql import MySQL
5
6 # Configuración MySQL
7 app.config['MYSQL_HOST'] = 'localhost'
8 app.config['MYSQL_USER'] = 'db_api_test'
9 app.config['MYSQL_PASSWORD'] = 'password'
10 app.config['MYSQL_DB'] = 'db_api_test'
11
12 # Vinculamos la configuración al objeto MySQL con la app:
13 mysql = MySQL(app)
14
15 class DBError(Exception):
16     pass
```

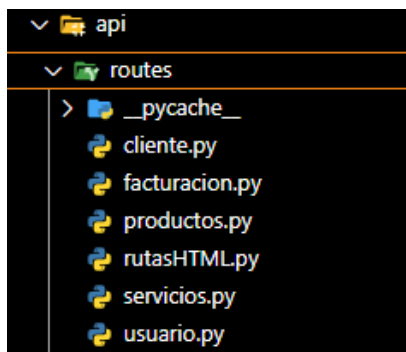
2.6.2. api --> models:

Contiene todos los archivos “.py” en los cuales se alojan las clases relacionadas a sus correspondientes tablas SQL dentro de la base de datos, en dichas clases se encuentra el código en el cual se define la estructura que se deberá respetar, tipos de datos, su correspondiente json a retornar, funciones de comprobación de existencia, ranking y sus correspondientes CRUD (Create, Read, Update, Delete).



2.6.3. api --> routes:

Contiene todos los archivos .py que contienen el código de las URL que permiten la vinculación entre el FrontEnd y el BackEnd, siendo estas el nexo entre los dos para mantener el nivel de seguridad y permitiendo crear las funcionalidades que podrá ejecutarse desde el FrontEnd. Las rutas tendrán asociadas cada funcionalidad relacionada a cada una de las clases para que impacten en la correspondiente base de datos a través de las consultas. En otras palabras, serán las funcionalidades de la api, permitiendo que el FrontEnd se conecte con el BackEnd para ejecutar funciones del CRUD: (Create: POST, Read: GET, Update: PUT, Delete: DELETE).



2.6.3.1. api --> routes --> cliente:

2.6.3.1.1. Crear un cliente nuevo en un determinado usuario:

```
/users/<int:user_id>/client', methods = ['POST']
```

2.6.3.1.2. Consultar de un determinado usuario un cliente por ID.

```
'/users/<int:user_id>/client/<int:client_id>', methods = ['GET']
```

2.6.3.1.3. Consultar todos los clientes de un determinado usuario.

```
/users/<int:user_id>/client', methods = ['GET']
```

2.6.3.1.4. Modificar la información de un determinado cliente por ID de un determinado Usuario.

```
/users/<int:user_id>/client/<int:client_id>', methods = ['PUT']
```

2.6.3.1.5. Eliminar la información de un cliente por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).

```
/users/<int:user_id>/client/<int:client_id>', methods = ['DELETE']
```

2.6.3.2. api --> routes --> facturacion:

2.6.3.2.1. Crear una factura a un determinado usuario.

```
'/users/<int:user_id>/invoice', methods = ['POST']
```

2.6.3.2.2. Consultar de un determinado usuario una factura por ID.

```
'/users/<int:user_id>/invoice/<int:invoice_id>', methods = ['GET']
```

2.6.3.2.3. Consultar todas las facturas de un determinado usuario.

```
'/users/<int:user_id>/invoice', methods = ['GET']
```

2.6.3.2.4. Consultar ranking de servicios facturados de un determinado usuario.

```
'/users/<int:user_id>/invoice_ranking_service', methods = ['GET']
```

2.6.3.2.5. Consultar ranking de productos facturados de un determinado usuario.

```
'/users/<int:user_id>/invoice_ranking_product', methods = ['GET']
```

2.6.3.2.6. Consultar ranking de clientes facturados de un determinado usuario.


```
'/users/<int:user_id>/invoice_ranking_client', methods = ['GET']
```

2.6.3.2.7. Consulta movimiento de productos de un determinado usuario.

```
'/users/<int:user_id>/invoice_control_flow_product', methods = ['GET']
```

2.6.3.2.8. Eliminar la información de una factura por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).

```
'/users/<int:user_id>/invoice/<int:invoice_id>', methods = ['DELETE']
```

Nota: No existe una ruta con el comando "PUT" debido a que la Ley prohíbe la anulación de facturas ya emitidas, y jamás se debe romper su correlación. Si hay una factura electrónica mal emitida y se detecta posteriormente que corresponde anularla, el documento correcto para ello es la emisión de una Nota de Crédito electrónica. (No implementado)

2.6.3.3. api --> routes --> productos:

2.6.3.3.1. Crear un producto para un determinado usuario.

```
'/users/<int:user_id>/product', methods = ['POST']
```

2.6.3.3.2. Consultar un producto por ID de un determinado Usuario.

```
'/users/<int:user_id>/product/<int:product_id>', methods = ['GET']
```

2.6.3.3.3. Consultar todos los productos de un determinado Usuario.

```
'/users/<int:user_id>/product', methods = ['GET']
```

2.6.3.3.4. Modificar los datos de un producto por ID de un determinado Usuario.

```
'/users/<int:user_id>/product/<int:product_id>', methods = ['PUT']
```

2.6.3.3.5. Eliminar la información de un producto por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).

```
'/users/<int:user_id>/product/<int:product_id>', methods = ['DELETE']
```

2.6.3.4. api --> routes --> servicios:

2.6.3.4.1. Crear un servicio a un determinado usuario.

```
'/users/<int:user_id>/service', methods = ['POST']
```

2.6.3.4.2. Consultar servicio por ID de un determinado Usuario.

```
'/users/<int:user_id>/service/<int:service_id>', methods = ['GET']
```

2.6.3.4.3. Consultar todos los servicios de un determinado Usuario.

```
'/users/<int:user_id>/service', methods = ['GET']
```

2.6.3.4.4. Modificar los datos de un determinado servicios por ID de un determinado Usuario.

```
'/users/<int:user_id>/service/<int:service_id>', methods = ['PUT']
```

2.6.3.4.5. Eliminar la información de una servicio por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).


```
'/users/<int:user_id>/service/<int:service_id>', methods = ['DELETE']
```

2.6.3.5. api --> routes --> usuario:

2.6.3.5.1. Autenticar al usuario utilizando la información de autenticación proporcionada en la solicitud.

```
'/login', methods = ['POST']
```

2.6.3.5.2. Crear un nuevo usuario.

```
'/signup', methods = ['POST']
```

2.6.3.5.3. Consultar usuario por ID.

```
'/users/<int:id>', methods = ['GET']
```

2.6.3.5.4. Consultar todos los usuarios.

```
'/users', methods = ['GET']
```

2.6.3.5.5. Modificar los datos de un usuario por ID.

```
'/users/<int:user_id>', methods = ['PUT']
```

2.6.3.5.6. Eliminar la información de una usuario por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).

```
'/users/<int:id>', methods = ['DELETE']
```

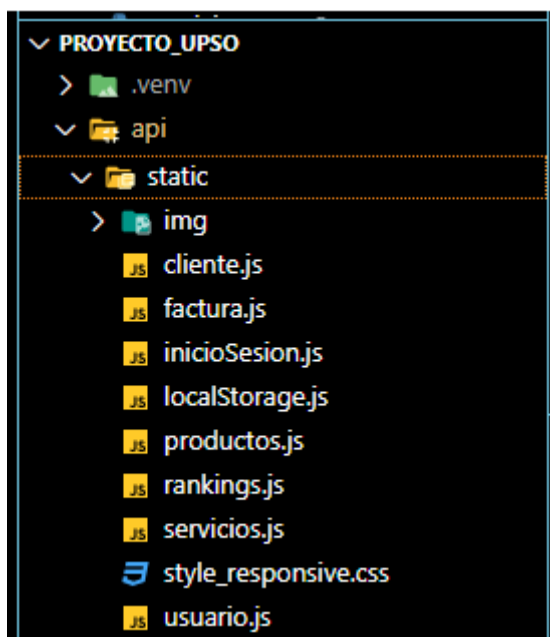
2.6.3.6. api --> routes --> rutasHTML:

Ruta que permite conectar con el archivo: 'public/dashboard.html'.

```
'/dashboard'
```

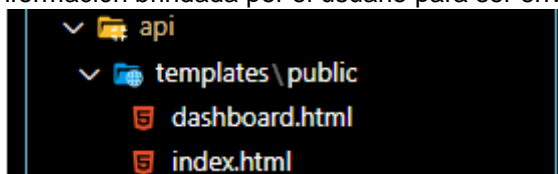
2.6.4. api --> static:

Contiene los archivos formato ".js" individualizados según las diferentes consultas que se pueden realizar desde el FrontEnd, son las funcionalidades que permiten tomar los datos e información del HTML prestada por el usuario y enviarlas en formato JSON apto y aceptado por los requerimientos establecidos en el BackEnd para así obtener las correspondientes respuestas del servidor. También se incluyen los estilos e imágenes a utilizar en el código css.



2.6.5. api --> templates\public:

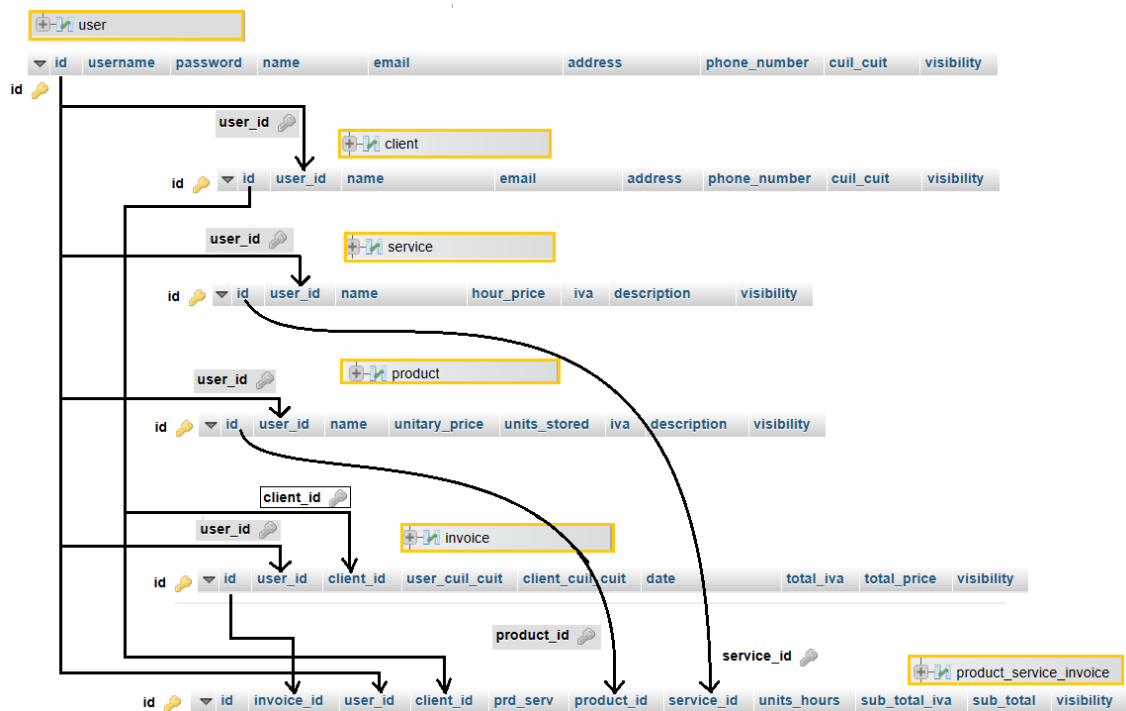
Contiene los archivos.html que se ejecutan en el navegador web y permiten recopilar la información brindada por el usuario para ser enviada al BackEnd.



2.7 ESTRUCTURA Y ARQUITECTURA DE LA BASE DE DATOS. FUNDAMENTOS

2.7.1. ESTRUCTURA DESCRIPTIVA.

La estructura y arquitectura de la base de datos ha sido definida en función de un análisis técnico dentro del equipo multidisciplinario, siendo en primera instancia acotado pero consistente con potencial de ampliación a futuro.



El usuario puede poseer ninguno o más de un cliente, ninguno o más de un servicio, ninguno o más de un producto o ninguna o más de una factura.

Una factura solo podrá tener un emisor (usuario) y un receptor (cliente). La misma puede no tener servicios o no tener productos.

Una FACTURA solo podrá tener (1) RECEPTOR: USUARIO, solo podrá tener (1) EMISOR: CLIENTE, podrá tener (0 o +) SERVICIOS, (0 o +) PRODUCTOS.

2.7.2. ESTRUCTURA TECNICA – MYSQL – CLASES: PYTHON - SQL:

USER

| # | Nombre | Tipo |
|---|--------------|--------------|
| 1 | id | int(10) |
| 2 | username | varchar(255) |
| 3 | password | varchar(255) |
| 4 | name | varchar(255) |
| 5 | email | varchar(255) |
| 6 | address | varchar(255) |
| 7 | phone_number | varchar(255) |
| 8 | cuil_cuit | varchar(15) |
| 9 | visibility | tinyint(4) |

```

class User():
    schema = {
        "username": str,
        "password": str,
        "name": str,
        "email": str,
        "address": str,
        "phone_number": str,
        "cuil_cuit": str
    }
        
```

```

CREATE TABLE IF NOT EXISTS user (
    id INT(10) NOT NULL AUTO_INCREMENT,
    username VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL,
    phone_number VARCHAR(255) NOT NULL,
    cuil_cuit VARCHAR(15) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id)
);
        
```

CLIENT

| # | Nombre | Tipo |
|---|--------------|--------------|
| 1 | id | int(10) |
| 2 | user_id | int(10) |
| 3 | name | varchar(255) |
| 4 | email | varchar(255) |
| 5 | address | varchar(255) |
| 6 | phone_number | varchar(255) |
| 7 | cuil_cuit | varchar(15) |
| 8 | visibility | tinyint(4) |

```
class Client():
    schema = {
        "user_id": int,
        "name": str,
        "email": str,
        "address": str,
        "phone_number": str,
        "cuil_cuit": str
    }
```

```
CREATE TABLE IF NOT EXISTS client (
    id INT(10) NOT NULL AUTO_INCREMENT,
    user_id INT(10) NOT NULL,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL,
    address VARCHAR(255) NOT NULL,
    phone_number VARCHAR(255) NOT NULL,
    cuil_cuit VARCHAR(15) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (user_id) REFERENCES user(id)
);
```

SERVICE

| # | Nombre | Tipo |
|---|-------------|---------------|
| 1 | id | int(10) |
| 2 | user_id | int(10) |
| 3 | name | varchar(255) |
| 4 | hour_price | int(10) |
| 5 | iva | int(10) |
| 6 | description | varchar(1000) |
| 7 | visibility | tinyint(4) |

```
class Service():
    schema = {
        "user_id": int,
        "name": str,
        "hour_price": int,
        "iva": int,
        "description": str
    }
```

```
CREATE TABLE IF NOT EXISTS service (
    id INT(10) NOT NULL AUTO_INCREMENT,
    user_id INT(10) NOT NULL,
    name VARCHAR(255) NOT NULL,
    hour_price INT(10) NOT NULL,
    iva INT(10) NOT NULL,
    description VARCHAR(1000) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (user_id) REFERENCES user(id)
);
```

PRODUCT

| # | Nombre | Tipo |
|---|---------------|---------------|
| 1 | id | int(10) |
| 2 | user_id | int(10) |
| 3 | name | varchar(255) |
| 4 | unitary_price | int(10) |
| 5 | units_stored | int(10) |
| 6 | iva | int(10) |
| 7 | description | varchar(1000) |
| 8 | visibility | tinyint(4) |

```
class Product():
    # ESQUEMA
    schema = {
        "user_id": int,
        "name": str,
        "unitary_price": int,
        "units_stored": int,
        "iva": int,
        "description": str
    }
```

```
CREATE TABLE IF NOT EXISTS product (
    id INT(10) NOT NULL AUTO_INCREMENT,
    user_id INT(10) NOT NULL,
    name VARCHAR(255) NOT NULL,
    unitary_price INT(10) NOT NULL,
    units_stored INT(10) NOT NULL,
    iva INT(10) NOT NULL,
    description VARCHAR(1000) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (user_id) REFERENCES user(id)
);
```

INVOICE

| # | Nombre | Tipo |
|---|------------------|-------------|
| 1 | id | int(10) |
| 2 | user_id | int(10) |
| 3 | client_id | int(10) |
| 4 | user_cuil_cuit | varchar(15) |
| 5 | client_cuil_cuit | varchar(15) |
| 6 | date | datetime |
| 7 | total_iva | int(10) |
| 8 | total_price | int(10) |
| 9 | visibility | tinyint(4) |

```
class Invoice():
    schema = {
        "user_id": int,
        "client_id": int,
        "user_cuil_cuit": str,
        "client_cuil_cuit": str,
        "products_services": list
    }
```

```
CREATE TABLE IF NOT EXISTS invoice (
    id INT(10) NOT NULL AUTO_INCREMENT,
    user_id INT(10) NOT NULL,
    client_id INT(10) NOT NULL,
    user_cuil_cuit VARCHAR(15) NOT NULL,
    client_cuil_cuit VARCHAR(15) NOT NULL,
    date DATETIME NOT NULL,
    total_iva INT(10) NOT NULL,
    total_price INT(10) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (user_id) REFERENCES user(id),
    FOREIGN KEY (client_id) REFERENCES client(id)
);
```

PRODUCT_SERVICE_INVOICE

| # | Nombre | Tipo |
|----|---------------|------------|
| 1 | id | int(10) |
| 2 | invoice_id | int(10) |
| 3 | user_id | int(10) |
| 4 | client_id | int(10) |
| 5 | prd_serv | char(1) |
| 6 | product_id | int(10) |
| 7 | service_id | int(10) |
| 8 | units_hours | int(10) |
| 9 | sub_total_iva | int(10) |
| 10 | sub_total | int(10) |
| 11 | visibility | tinyint(4) |


```

class Product_Service_Invoice():
    schema = {
        "invoice_id": int,
        "user_id": int,
        "client_id": int,
        "ps_id": int,
        "prd_serv": str,
        "units_hours": int,
        "iva_subtotal" : int,
        "sub_total": int
    }
    count_schema = {
        "ps_id": int,
        "prd_serv": str,
        "units_hours": int
    }
    
```



```

CREATE TABLE IF NOT EXISTS product_service_invoice (
    id INT(10) NOT NULL AUTO_INCREMENT,
    invoice_id INT(10) NOT NULL,
    user_id INT(10) NOT NULL,
    client_id INT(10) NOT NULL,
    prd_serv CHAR(1) NOT NULL,
    product_id INT(10),
    service_id INT(10),
    units_hours INT(10) NOT NULL,
    sub_total_iva INT(10) NOT NULL,
    sub_total INT(10) NOT NULL,
    visibility TINYINT NOT NULL,
    PRIMARY KEY(id),
    FOREIGN KEY (user_id) REFERENCES user(id),
    FOREIGN KEY (client_id) REFERENCES client(id),
    FOREIGN KEY (invoice_id) REFERENCES invoice(id),
    FOREIGN KEY (product_id) REFERENCES product(id),
    FOREIGN KEY (service_id) REFERENCES service(id)
);
    
```

2.7.3. POSIBLES MEJORAS EN LA ESTRUCTURA A FUTURO.

Cuestiones tributarias:

a) Tipo de contribuyente (Responsable monotributo, Responsable Inscripto, Exento en IVA, Consumidor Final, Sujeto no Categorizado, Proveedor del Exterior, IVA no alcanzado entre tantos otros) (Responsable monotributo, Responsable Inscripto, Exento en IVA, Consumidor Final, Sujeto no Categorizado, Proveedor del Exterior, IVA no alcanzado entre tantos otros) que es el usuario, a los efectos de determinar qué tipo de comprobante lo corresponde emitir.

b) Tipo de contribuyente (Responsable monotributo, Responsable Inscripto, Exento en IVA, Consumidor Final, Sujeto no Categorizado, Proveedor del Exterior, IVA no alcanzado entre tantos otros) que es el cliente a los efectos de determinar que tipo de comprobante le corresponde recibir.

Los diferentes comprobantes generan diferentes estructuras en cuanto al documento a emitir: Factura A – Factura B – Factura C – Factura de Crédito Electrónica, entre otras, y cada uno de ellos generan diferentes requerimientos a nivel código y restructuración de la base de datos.

Contemplar la importante necesidad de implementar las estructuras de las notas de crédito y notas de débito.

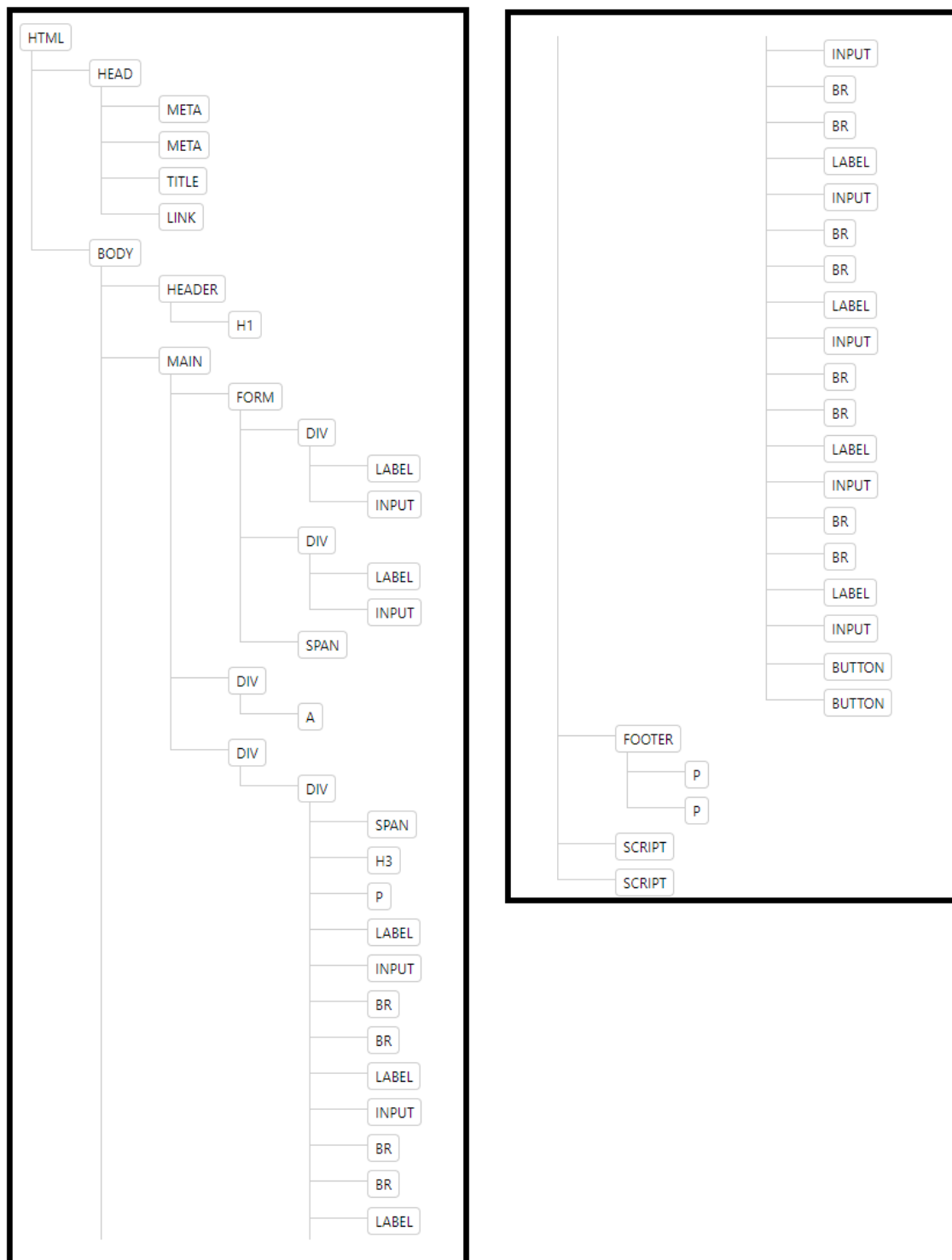
2.8. ESTRUCTURA HTML – INDEX – DASHBOARD

El archivo *index.html* está compuesto por un *body* con un *header*, *main*, *footer* y *scripts*. Contienen los elementos necesarios para obtener la información por parte del usuario para iniciar sesión, o en caso de no tener una cuenta, crear un nuevo usuario.

El archivo *dashboard.html* está compuesto por un *body* con un *header*, *nav*, *main*, *div*, *footer* y *scripts*. Contiene los elementos necesarios para obtener la información por parte del usuario para llevar a cabo las siguientes acciones dentro de su cuenta:

- Perfil – Editar Perfil
- Clientes – Nuevo Cliente / Lista / Ranking
- Productos – Nuevo Producto / Lista / Ranking
- Servicios – Nuevo Servicio / Lista / Ranking
- Facturación – Nueva Factura / Lista / Ranking
- Sesión – Salir

Se visualiza la estructura del html a partir del uso de la extensión de Google *HTML Tree Generator*, la cual genera un árbol de los elementos que componen la página.



The image displays three hierarchical diagrams, each representing a different level of abstraction in web development:

- HTML Document Structure:** This diagram shows the root of an HTML document. It branches into **HTML**, which further divides into **HEAD** and **BODY**. **HEAD** contains **META**, **TITLE**, and **LINK**. **BODY** contains **HEADER** (with **H1**), **NAV**, and five **SECTION** elements. Each **SECTION** contains **H2**, **A**, and **SPAN**.
- Web Page Structure:** This diagram illustrates the DOM tree of a web page. It starts with **MAIN**, which branches into **DIV** and **DIV**. The first **DIV** contains **SPAN**, **H3**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **BUTTON**, **BUTTON**, **BUTTON**, and **BUTTON**. The second **DIV** contains **DIV**, which further branches into **SPAN**, **H3**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **BUTTON**, and **BUTTON**.
- Form Structure:** This diagram shows the structure of a form. It starts with **DIV**, which branches into **DIV** and **DIV**. The first **DIV** contains **LABEL**, **INPUT**, **LABEL**, **INPUT**, **BUTTON**, and **BUTTON**. The second **DIV** contains **SPAN**, **H3**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **LABEL**, **INPUT**, **BUTTON**, and **BUTTON**.



2.9. NIVEL DE SEGURIDAD Y CONTROLES APLICADOS

```
def token_required(func):
    @wraps(func)
    def decorated(*args, **kwargs):
        token = None

        if 'x-access-token' in request.headers:
            token = request.headers['x-access-token']

        if not token:
            return jsonify({"message": "Falta Token Valido"}), 401

        user_id = None # inicializamos la cabecera en None.

        # Consultamos si la cabecera esta en la request.
        if 'user-id' in request.headers:

            # si esa cabecera existe a user_id le asignamos el contenido:
            user_id = request.headers['user-id']

        # Si no hay id del usuario retornamos un mensaje:
        if not user_id:
            return jsonify({"message": "Falta el usuario"}), 401

        try:
            data = jwt.decode(token, app.config['SECRET_KEY'], algorithms
= ['HS256'])
            token_id = data['id'] # obtenemos el 'id' que se encuentra
encapsulado dentro del Token.

            # Si el usuario es distinto al token_id retornamos un
mensaje:
            if int(user_id) != int(token_id):
                return jsonify({"message": "Error de id"})

        except Exception as e:
            return jsonify({"message": str(e)}), 401

        return func(*args, **kwargs)
    return decorated
```

token_required es la función que permite realizar los controles vinculadas a la existencia del token y que el mismo se encuentre vigente y no vencido, también realiza el control de que pertenezca al usuario correspondiente. El decorador **@wraps** permitirá aplicar este control a las diferentes rutas.

```
# Control de RECURSOS RELACIONADOS CON CLIENTES:
def client_resource(func):
    @wraps(func)
    def decorated(*args, **kwargs):
        client_id = kwargs['client_id']

        # Definimos un cursor:
        cur = mysql.connection.cursor()

        # Ejecutamos una consulta: para obtener el id_usuario
        cur.execute(f'SELECT user_id FROM client WHERE id = {client_id}')

        # De los datos esperamos recibir solo 1: si el cliente existe
        data tendra un valor.
        data = cur.fetchone()

        # Controlamos el valor de Data:
        if data:
            id_prop = data[0]

            # user_id que recibimos por cabecera:
            user_id = request.headers['user-id']

            # controlamos que sea el recurso del propietario quien
            consulta:
            if int(id_prop) != int(user_id):
                return jsonify({"message": "No tiene permiso para acceder
a este recurso"}), 401
            return func(*args, **kwargs)
        return decorated
```

client_resource es la función que permite llevar a cabo el control de propiedad de que el cliente consultado pertenezca al usuario solicitante.

```
# Control de RECURSOS RELACIONADOS CON PRODUCTOS:
def product_resource(func):
    @wraps(func)
    def decorated(*args, **kwargs):
        product_id = kwargs['product_id']

        # Definimos un cursor:
        cur = mysql.connection.cursor()

        # Ejecutamos una consulta: para obtener el id_usuario
        cur.execute(f'SELECT user_id FROM product WHERE id =
{product_id}')
```

```
# De los datos esperamos recibir solo 1: si el cliente existe
data tendra un valor.
data = cur.fetchone()

# Controlamos el valor de Data:
if data:
    id_prop = data[0]

    # user_id que recibimos por cabecera:
    user_id = request.headers['user-id']

    # controlamos que sea el recurso del propietario quien
consulta:
    if int(id_prop) != int(user_id):
        return jsonify({"message": "No tiene permiso para acceder
a este recurso"}), 401
    return func(*args, **kwargs)
return decorated
```

product_resource es la función que permite llevar a cabo el control de propiedad de que el producto consultado pertenezca al usuario solicitante.

```
# Control de RECURSOS RELACIONADOS CON FACTURAS:
def invoice_resource(func):
    @wraps(func)
    def decorated(*args, **kwargs):
        invoice_id = kwargs['invoice_id']

        # Definimos un cursor:
        cur = mysql.connection.cursor()

        # Ejecutamos una consulta: para obtener el id_usuario
        cur.execute(f'SELECT user_id FROM invoice WHERE id =
{invoice_id}')

        # De los datos esperamos recibir solo 1: si el cliente existe
data tendra un valor.
data = cur.fetchone()

        # Controlamos el valor de Data:
        if data:
            id_prop = data[0]

            # user_id que recibimos por cabecera:
            user_id = request.headers['user-id']
```

```
        # controlamos que sea el recurso del propietario quien
consulta:
        if int(id_prop) != int(user_id):
            return jsonify({"message": "No tiene permiso para acceder
a este recurso"}), 401
        return func(*args, **kwargs)
    return decorated
```

invoice_resource es la función que permite llevar a cabo el control de propiedad de que la factura consultada pertenezca al usuario solicitante.

```
# Control de RECURSOS RELACIONADOS CON SERVICIOS:
def service_resource(func):
    @wraps(func)
    def decorated(*args, **kwargs):
        service_id = kwargs['service_id']

        # Definimos un cursor:
        cur = mysql.connection.cursor()

        # Ejecutamos una consulta: para obtener el id_usuario
        cur.execute(f'SELECT user_id FROM service WHERE id =
{service_id}')

        # De los datos esperamos recibir solo 1: si el cliente existe
data tendra un valor.
        data = cur.fetchone()

        # Controlamos el valor de Data:
        if data:
            id_prop = data[0]

            # user_id que recibimos por cabecera:
            user_id = request.headers['user-id']

            # controlamos que sea el recurso del propietario quien
consulta:
            if int(id_prop) != int(user_id):
                return jsonify({"message": "No tiene permiso para acceder
a este recurso"}), 401
            return func(*args, **kwargs)
        return decorated
```

service_resource es la función que permite llevar a cabo el control de propiedad de que el servicio consultado pertenezca al usuario solicitante.

```
# Control de RECURSOS RELACIONADOS CON USUARIOS:
def user_resource(func):
```

```
@wraps(func)
def decorated(*args, **kwargs):

    # Id del usuario utilizado en la ruta:
    user_id_route = kwargs['user_id']

    # Id que viene por cabecera de la request o consulta:
    user_id = request.headers['user-id']

    # controlamos que sea el recurso del propietario quien consulta:
    if int(user_id_route) != int(user_id):
        return jsonify({"message": "No tiene permiso para acceder a
este recurso"}), 401
    return func(*args, **kwargs)
return decorated
```

user_resource es la función que permite llevar a cabo el control de consistencia de que el id de la ruta no sea diferente al de la cabecera de la consulta realizada por el frontEnd.

2.10. ESTRUCTURA DE USUARIO

2.10.1. models → usuario

```
class User():

    schema = {
        "username": str,
        "password": str,
        "name": str,
        "email": str,
        "address": str,
        "phone_number": str,
        "cuil_cuit": str
    }
```

Se define la clase llamada User, la cual tiene un atributo schema que especifica la estructura de datos esperada para un usuario. El atributo schema es un diccionario que define los campos esperados para un usuario, junto con el tipo de dato de cada campo.

```
def check_data_schema(data):
    if data == None or type(data) != dict:
        return False
    # check if data contains all keys of schema
    for key in User.schema:
        if key not in data:
            return False
        # check if data[key] has the same type as schema[key]
        if type(data[key]) != User.schema[key]:
            return False
```

```
return True
```

La función `check_data_schema(data)` se encarga de validar si la estructura de los datos cumple con el esquema predefinido. Primero, verifica si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase `User`. Si todas estas condiciones se cumplen, la función devuelve `True`; de lo contrario, devuelve `False`.

```
def __init__(self, row):
    self._id = row[0]
    self._username = row[1]
    self._password = row[2]
    self._name = row[3]
    self._email = row[4]
    self._address = row[5]
    self._phone_number = row[6]
    self._cuil_cuit = row[7]
    self._visibility = row[8]

    def to_json(self):
        return {
            "id": self._id,
            "username": self._username,
            "password": self._password,
            "name": self._name,
            "email": self._email,
            "address": self._address,
            "phone_number": self._phone_number,
            "cuil_cuit": self._cuil_cuit,
            "visibility": self._visibility
        }
```

Se define el constructor de la clase `User`. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función `to_json`, cuyo objetivo consiste en retornar los datos de la clase `User` en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

```
##### COMPROBACION DE EXISTENCIA
def user_exists(cuil_cuit):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM user WHERE cuil_cuit = %s AND
visibility = 1', (cuil_cuit,))
    cur.fetchall()
    return cur.rowcount > 0
```

El método `user_exist` consulta en la base de datos si el `cuil_cuit` existe en la misma, para verificar la existencia o no del usuario.

```
def login(auth):
```



```
""" Control: existen valores para la autenticacion?"""
if not auth or not auth.username or not auth.password:
    raise TypeError("Login No Autorizado")# existen librerias que
facilitan el manejo de errores

""" Control: existen y coincide el usuario en la BD?"""
cur = mysql.connection.cursor()
cur.execute("SELECT * FROM user WHERE username = %s AND password
= %s AND visibility =1", (auth.username, auth.password))
# Esperamos recibir una fila: devuelve 1ero en fila osea 1.
row = cur.fetchone()

# si la consulta no devuelve filas:
if not row:
    raise DBError("Login No Autorizado") # existen librerias que
facilitan el manejo de errores

""" El usuario existe en la BD y coincide su contraseña"""
# Creamos una variable codificada:
token = jwt.encode({
    'id': row[0],
    'exp': dt.datetime.utcnow() + dt.timedelta(minutes=100)    #
tiempo de exporar el password. Tiempo para que el usurio vuelva a iniciar
sesion.
}, app.config['SECRET_KEY'])

return {"Token": token, "Username": auth.username, "id": row[0]}
```

El método *login* implementa el inicio de sesión de la cuenta del usuario. Dentro del mismo se establecen controles: si posee autorización para ingresar, si posee la autorización si esta coincide con las credenciales del usuario en la base de datos. Si se superan las validaciones, se creará un token, con un tiempo de expiración determinado.

```
### CREATE
def create_user(data):

    if User.check_data_schema(data):
        # check if user already exists
        if User.user_exists(data["cuil_cuit"]):
            raise DBError("Error creating user - user already
exists")

        cur = mysql.connection.cursor()
        cur.execute('INSERT INTO user (username, password,
name, email, address, phone_number, cuil_cuit, visibility) VALUES (%s,
%s, %s, %s, %s, %s, %s, %s)',
```

```
        ( data["username"], data["password"],
data["name"], data["email"], data["address"], data["phone_number"],
data["cuil_cuit"], True))
        mysql.connection.commit()
        if cur.rowcount > 0:
            # get the id of the last inserted row
            cur.execute('SELECT LAST_INSERT_ID()')
            res = cur.fetchall()
            id = res[0][0]
            return User((id, data["username"], data["password"],
data["name"], data["email"], data["address"], data["phone_number"],
data["cuil_cuit"], True)).to_json()
        raise DBError("Error creating user - no row inserted")
        raise TypeError("Error creating user - wrong data schema")
```

La función `create_user(data)` se encarga de crear un nuevo usuario. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `User`. Luego, comprueba si el usuario ya existe en la base de datos. Si el usuario no existe, se ejecuta una consulta para insertar un nuevo usuario en la base de datos y se devuelve la información del usuario en formato JSON. Si ocurre algún error durante el proceso, se generará una excepción apropiada, como `DBError` o `TypeError`, junto con un mensaje descriptivo.

```
### READ
def get_user_by_id(id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM user WHERE id = %s AND visibility =
1', (id, ))
    data = cur.fetchall()
    if cur.rowcount > 0:
        return User(data[0]).to_json()
    raise DBError("Error getting user by id - no row found")

def get_all_users():
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM user')
    data = cur.fetchall()
    users = []
    if cur.rowcount > 0:
        for i in data:
            users.append( User(i).to_json())
        return users
    raise DBError("Error getting users - no row found")
```

La función `get_user_by_id(id)` recibe un ID de usuario y devuelve la información del usuario correspondiente en formato JSON si se encuentra en la base de datos. Si no se encuentra el usuario, se genera una excepción `DBError` con un mensaje descriptivo. La función `get_all_user(user_id)` recibe un ID de usuario y devuelve una lista de todos los usuarios

asociados a ese usuario en formato JSON. Si no se encuentra ningún usuario, se genera una excepción DBError con un mensaje descriptivo.

```
### UPDATE
def update_user(data):

    if User.check_data_schema(data):
        cur = mysql.connection.cursor()
        cur.execute('UPDATE user SET username = %s, password = %s,
name = %s, email= %s , address= %s , phone_number = %s, cuil_cuit = %s
WHERE id = %s',
                    (data["username"], data["password"],
data["name"], data["email"], data["address"], data["phone_number"],
data["cuil_cuit"], data["user_id"]))
        mysql.connection.commit()
        if cur.rowcount > 0:
            return User.get_user_by_id(data["user_id"])
            raise DBError("Error updating user - no row updated")
            raise TypeError("Error updating user - wrong data schema")
```

La función `update_user_by_id(id, data)` se encarga de actualizar la información de un usuario existente en la base de datos. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `User`. Luego, ejecuta una consulta para actualizar la información del usuario en la base de datos y devuelve la información actualizada del usuario en formato JSON si se actualiza correctamente. Si no se actualiza el usuario, se genera una excepción `DBError` con un mensaje descriptivo. Si los datos no cumplen con el esquema, se genera una excepción `TypeError` con un mensaje descriptivo.

```
### DELETE
def delete_user(id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM user WHERE id=%s AND visibility =1',
(id,))
    if cur.rowcount > 0:
        cur.execute('UPDATE user SET visibility=0 WHERE id=%s',
(id,))
        mysql.connection.commit()
        return {"message": "deleted", "id": id}
        raise DBError("Error getting client by id - no row found")
```

La función `remove_user_by_id(id)` realiza un borrado lógico de un usuario existente en la base de datos. Primero, se ejecuta una consulta para verificar si el usuario con el ID proporcionado existe y es visible. Si el usuario existe y es visible, se ejecuta otra consulta para actualizar la visibilidad del usuario a False. Luego, se devuelve un mensaje indicando que el usuario ha sido marcado como no visible junto con el ID del usuario. Si el usuario no se encuentra, se genera una excepción `DBError` con un mensaje descriptivo.

2.10.2. routes → usuarios

```
# Login
```

```
@app.route('/login', methods = ['POST'] )
def login():
    auth = request.authorization
    try:
        log = User.login(auth)
        return jsonify(log)
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El *endpoint* '/login' está configurado para aceptar solo solicitudes POST, y cuando se recibe una solicitud, se llama a la función login(). Dentro de la función, se intenta autenticar al usuario utilizando la información de autenticación proporcionada en la solicitud. Si la autenticación es exitosa, se devuelve la información del usuario en formato JSON. Si ocurre una excepción durante el proceso de autenticación, se devuelve un mensaje de error junto con el código de error 400.

```
# CREATE

@app.route('/signup', methods = ['POST'] )
def crate_user():
    data = request.get_json()
    try:
        new_user = User.create_user(data)
        return jsonify( new_user ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El *endpoint* '/signup' está configurado para aceptar solo solicitudes POST, y cuando se recibe una solicitud, se llama a la función create_user(). Dentro de la función, se obtiene la información del usuario en formato JSON desde la solicitud y se intenta crear un nuevo usuario utilizando esa información. Si la creación del usuario es exitosa, se devuelve la información del usuario en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de creación del usuario, se devuelve un mensaje de error junto con el código 400.

```
# READ

@app.route('/users/<int:user_id>', methods = ['GET'] )
@token_required
@user_resource
def get_user_by_id(user_id):
    try:
        user = User.get_user_by_id(user_id)
        return jsonify( user ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400

@app.route('/users', methods = ['GET'] )
def get_all_user():
    try:
        user = User.get_all_users()
```

```
        return jsonify( user ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El primer *endpoint* `/users/int:user_id`, está configurado para aceptar solicitudes GET y utiliza dos controles a partir de los decoradores `@token_required` y `@user_resource`. La función `'get_user_by_id'` intenta obtener la información de un usuario específico utilizando el ID proporcionado en la solicitud. Si la búsqueda es exitosa, se devuelve la información del usuario en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

El segundo *endpoint*, `/users`, también está configurado para aceptar solicitudes GET. La función `'get_all_user'` intenta obtener la información de todos los usuarios. Si la búsqueda es exitosa, se devuelve la información de todos los usuarios en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

```
# UPDATE
@app.route('/users/<int:user_id>', methods = ['PUT'] )
@token_required
@user_resource
def update_user(user_id):

    data = request.get_json()
    data["user_id"] = user_id
    try:
        user = User.update_user(data)
        return jsonify( user ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El *endpoint* `/users/int:user_id` está configurado para aceptar solicitudes PUT. La función `'update_user'` intenta actualizar la información de un usuario específico utilizando el ID proporcionado en la solicitud y la información proporcionada en el cuerpo de la solicitud. Si la actualización es exitosa, se devuelve la información actualizada del usuario en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de actualización, se devuelve un mensaje de error junto con el código de estado 400. Se ejecutan los controles de existencia y vigencia del token (`@token_required`) y control de existencia del usuario en la base de datos (`@user_resource`).

```
# DELETE
@app.route('/users/<int:user_id>', methods = ['DELETE'] )
@token_required
@user_resource
def remove_user(user_id):
    try:
        delete = User.delete_user(user_id)
        return delete
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Esta ruta permitirá ejecutar el método para remover un usuario según su Id, previo a ello se realizarán los controles tales como @token_required: existencia y vigencia del token, @user_resource: existencia del usuario en la base de datos.

2.11. ESTRUCTURA DE CLIENTE

2.11.1. model → client

```
class Client():  
  
    schema = {  
        "user_id": int,  
        "name": str,  
        "email": str,  
        "address": str,  
        "phone_number": str,  
        "cuil_cuit": str  
    }
```

Se define la clase llamada Client, la cual tiene un atributo schema que especifica la estructura de datos esperada para un cliente. El atributo schema es un diccionario que define los campos esperados para un cliente, junto con el tipo de dato de cada campo. Los campos especificados incluyen "user_id" (entero), "name" (cadena), "email" (cadena), "address" (cadena), "phone_number" (cadena) y "cuil_cuit" (cadena).

```
def check_data_schema(data):  
    if data == None or type(data) != dict:  
        return False  
    # check if data contains all keys of schema  
    for key in Client.schema:  
        if key not in data:  
            return False  
        # check if data[key] has the same type as schema[key]  
        if type(data[key]) != Client.schema[key]:  
            return False  
    return True
```

La función check_data_schema(data) se encarga de validar si la estructura de los datos cumple con el esquema predefinido. Primero, verifica si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase Client. Si todas estas condiciones se cumplen, la función devuelve True; de lo contrario, devuelve False.

```
def __init__(self, row):  
    self._id = row[0]  
    self._user_id = row[1]  
    self._name = row[2]  
    self._email = row[3]  
    self._address = row[4]  
    self._phone_number = row[5]
```

```
self._cuil_cuit = row[6]
self._visibility = row[7]

def to_json(self):
    return {
        "id": self._id,
        "user_id": self._user_id,
        "name": self._name,
        "email": self._email,
        "address": self._address,
        "phone_number": self._phone_number,
        "cuil_cuit": self._cuil_cuit,
        "visibility": self._visibility
    }
```

Se define el constructor de la clase *Client*. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función *to_json*, cuyo objetivo consiste en retornar los datos de la clase *User* en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

```
##### COMPROBACION DEL CLIENTE
def client_exists(cuil_cuit, user_id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM client WHERE cuil_cuit =%s AND user_id
= %s AND visibility = 1', (cuil_cuit, user_id))
    cur.fetchall()
    return cur.rowcount > 0
```

El método *client_exist* consulta en la base de datos si el *cuil_cuit* del cliente de ese usuario existe en la base de datos.

```
### CREATE
##### SE COMPRUEBA SU EXISTENCIA ANTES DE CREAR EL CLIENTE
def create_client(data):
    if Client.check_data_schema(data):
        # check if client already exists
        if Client.client_exists(data["cuil_cuit"], data["user_id"]):
            raise DBError("Error creating client - client already
exists")
        cur = mysql.connection.cursor()
        cur.execute('INSERT INTO client (user_id, name, email,
address, phone_number, cuil_cuit, visibility) VALUES ( %s, %s, %s, %s,
%s, %s, %s)',
                    ( data["user_id"], data["name"], data["email"],
data["address"], data["phone_number"], data["cuil_cuit"], True))
        mysql.connection.commit()
        if cur.rowcount > 0:
            # get the id of the last inserted row
```



```
cur.execute('SELECT LAST_INSERT_ID()')
res = cur.fetchall()
id = res[0][0]
return Client((id, data["user_id"], data["name"],
data["email"], data["address"], data["phone_number"], data["cuil_cuit"],
True)).to_json()
raise DBError("Error creating client - no row inserted")
raise TypeError("Error creating client - wrong data schema")
```

La función `create_client(data)` se encarga de crear un nuevo cliente. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `Client`. Luego, comprueba si el cliente ya existe en la base de datos. Si el cliente no existe, se ejecuta una consulta para insertar un nuevo cliente en la base de datos y se devuelve la información del cliente en formato JSON. Si ocurre algún error durante el proceso, se generará una excepción apropiada, como `DBError` o `TypeError`, junto con un mensaje descriptivo.

```
### READ
##### USO INTERNO DE CLIEN BY ID
def get_client_by_id(id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM client WHERE id = %s AND visibility =
1', (id,))
    data = cur.fetchall()
    if cur.rowcount > 0:
        return Client(data[0]).to_json()
    raise DBError("Error getting client by id - no row found")

def get_all_client(user_id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM client WHERE user_id =%s AND
visibility = 1', (user_id,))
    data = cur.fetchall()
    clientList = []
    if cur.rowcount > 0:
        for row in data:
            # Creamos un objeto Cliente:
            objClient = Client(row)
            # Agregamos un elemento json a la lista:
            clientList.append(objClient.to_json())
        return clientList
    raise DBError("Error getting client by id - no row found")
```

La función `get_client_by_id(id)` recibe un ID de cliente y devuelve la información del cliente correspondiente en formato JSON si se encuentra en la base de datos. Si no se encuentra el cliente, se genera una excepción `DBError` con un mensaje descriptivo. La función `get_all_client(user_id)` recibe un ID de usuario y devuelve una lista de todos los clientes asociados a ese usuario en formato JSON. Si no se encuentra ningún cliente, se genera una excepción `DBError` con un mensaje descriptivo.

```
### UPDATE
##### SE UTILIZA LA FUNCION DE CHECK DATA Y ACTUALIZA
def update_client_by_id(id, data):
    if Client.check_data_schema(data):
        cur = mysql.connection.cursor()
        cur.execute('UPDATE client SET name = %s, email= %s ,
address= %s , phone_number = %s, cuil_cuit = %s WHERE id = %s',
                    ( data["name"], data["email"], data["address"],
data["phone_number"], data["cuil_cuit"], id))
        mysql.connection.commit()
        if cur.rowcount > 0:
            return Client.get_client_by_id(id)
        raise DBError("Error updating client - no row updated")
        raise TypeError("Error updating client - wrong data schema")
```

La función `update_client_by_id(id, data)` se encarga de actualizar la información de un cliente existente en la base de datos. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `Client`. Luego, ejecuta una consulta para actualizar la información del cliente en la base de datos y devuelve la información actualizada del cliente en formato JSON si se actualiza correctamente. Si no se actualiza el cliente, se genera una excepción `DBError` con un mensaje descriptivo. Si los datos no cumplen con el esquema, se genera una excepción `TypeError` con un mensaje descriptivo.

```
### DELETE
def remove_client_by_id(id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM client WHERE id=%s AND visibility =
1', (id, ))
    if cur.rowcount > 0:
        cur.execute('UPDATE client SET visibility = %s WHERE id =
%s', (False, id))

        mysql.connection.commit()
        return {"message": "deleted", "id": id}

    raise DBError("Error removing client - client not found")
```

La función `remove_client_by_id(id)` realiza un borrado lógico de un cliente existente en la base de datos. Primero, se ejecuta una consulta para verificar si el cliente con el ID proporcionado existe y es visible. Si el cliente existe y es visible, se ejecuta otra consulta para actualizar la visibilidad del cliente a `False`. Luego, se devuelve un mensaje indicando que el cliente ha sido marcado como no visible junto con el ID del cliente. Si el cliente no se encuentra, se genera una excepción `DBError` con un mensaje descriptivo.

2.11.2. routes → client

```
# CREATE
@app.route('/users/<int:user_id>/client', methods = ['POST'] )
@token_required
```

```
@user_resource
def create_client(user_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        new_client = Client.create_client(data)
        return jsonify( new_client ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El *endpoint* `/users/<int:user_id>/client` acepta solicitudes POST y utiliza dos decoradores para realizar controles previos, `'@token_required'` y `'@user_resource'`. La función `create_client(user_id)` recibe los datos proporcionados en la solicitud, crea un nuevo cliente utilizando la clase `Client` y devuelve la información del cliente en formato JSON si se crea correctamente. Si ocurre una excepción durante el proceso de creación, se devuelve un mensaje de error junto con el código de estado 400.

```
# READ
# Ruta: Get de cliente a traves del ID:
@app.route('/users/<int:user_id>/client/<int:client_id>', methods =
['GET'] )
@token_required
@user_resource
@client_resource
def get_client_by_id(client_id, user_id):

    try:
        client = Client.get_client_by_id(client_id)
        return jsonify( client ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400

# RUTA: consulta todos los clientes de un usuario:
@app.route('/users/<int:user_id>/client', methods = ['GET'] )
@token_required
@user_resource
def get_all_clients_by_user_id(user_id):
    try:
        clients = Client.get_all_client(user_id)
        return jsonify( clients ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El primer *endpoint* está configurado para aceptar solicitudes GET y utiliza tres controles a partir de los decoradores `'@token_required'`, `'@user_resource'` y `'@client_resource'`. La función `get_client_by_id` intenta obtener la información de un cliente específico utilizando el ID proporcionado en la solicitud. Si la búsqueda es exitosa, se devuelve la información del cliente

en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

El segundo *endpoint* también está configurado para aceptar solicitudes GET. La función 'get_all_clients_by_user_id' intenta obtener la información de todos los clientes. Si la búsqueda es exitosa, se devuelve la información de todos los usuarios en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

```
# UPDATE
@app.route('/users/<int:user_id>/client/<int:client_id>', methods =
['PUT'] )
@token_required
@user_resource
@client_resource
def update_client(client_id, user_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        client = Client.update_client_by_id(client_id, data)
        return jsonify( client ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite actualizar la información de un cliente específico a través de una solicitud PUT. El *endpoint* está configurado para aceptar solicitudes PUT y toma dos parámetros en la URL, el ID del usuario y el ID del cliente. La función asociada a la ruta, `update_client(client_id, user_id)`, recibe los datos proporcionados en la solicitud, actualiza el cliente con el ID especificado y devuelve la información del cliente actualizada en formato JSON si la actualización se realiza correctamente. Si ocurre un error durante el proceso de actualización, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y cliente, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

```
# DELETE
@app.route('/users/<int:user_id>/client/<int:client_id>', methods =
['DELETE'] )
@token_required
@user_resource
@client_resource
def remove_client(client_id, user_id):
    try:
        client = Client.remove_client_by_id(client_id)
        return jsonify( client ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite eliminar un cliente específico a través de una solicitud DELETE. La ruta está configurada para aceptar solicitudes DELETE y toma dos parámetros en la URL, el ID del usuario y el ID del

cliente. La función asociada a la ruta, `remove_client(client_id, user_id)`, intenta hacer un borrado lógico del cliente con el ID especificado y devuelve un mensaje indicando que el cliente ha sido eliminado junto con el ID del cliente si la eliminación se realiza correctamente. Si ocurre un error durante el proceso de eliminación, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y cliente, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

2.12. ESTRUCTURA SERVICIOS

2.12.1. models → servicios

```
class Service():  
  
    schema = {  
        "user_id": int,  
        "name": str,  
        "hour_price": int,  
        "iva": int,  
        "description": str  
    }  
}
```

Se define la clase llamada `Service`, la cual tiene un atributo `schema` que especifica la estructura de datos esperada para un servicio. El atributo `schema` es un diccionario que define los campos esperados para un servicio, junto con el tipo de dato de cada campo.

```
## CHECK DATA SCHEMA ES UN COPY PASTE  
def check_data_schema(data):  
    if data == None or type(data) != dict:  
        return False  
    # check if data contains all keys of schema  
    for key in Service.schema:  
        if key not in data:  
            return False  
        # check if data[key] has the same type as schema[key]  
        if type(data[key]) != Service.schema[key]:  
            return False  
    return True
```

La función `check_data_schema(data)` se encarga de validar si la estructura de los datos cumple con el esquema predefinido. Primero, verifica si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase `Service`. Si todas estas condiciones se cumplen, la función devuelve `True`; de lo contrario, devuelve `False`.

```
def __init__(self, row):  
    self._id = row[0]  
    self._user_id = row[1]  
    self._name = row[2]  
    self._hour_price = row[3]  
    self._iva = row[4]  
    self._description = row[5]
```

```
self._visibility = row[6]

def to_json(self):
    return{
        "id": self._id,
        "user_id": self._user_id,
        "name": self._name,
        "hour_price": self._hour_price,
        "iva": self._iva,
        "description": self._description,
        "visibility": self._visibility
    }
```

Se define el constructor de la clase *Service*. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función *to_json*, cuyo objetivo consiste en retornar los datos de la clase *User* en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

```
def service_exists(name, user_id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM service WHERE name =%s AND user_id = %s AND visibility = 1', (name, user_id))
    cur.fetchall()
    return cur.rowcount > 0
```

El método *service_exists* consulta si el servicio existe en la base de datos.

```
### CREATE

def create_service(data):
    if not Service.check_data_schema(data):
        raise TypeError("Error creating service - wrong data schema")

    if Service.service_exists(data["name"], data["user_id"]):
        raise DBError("Error creating service - service already exists")

    cur = mysql.connection.cursor()
    cur.execute('INSERT INTO service (user_id, name, hour_price, iva, description, visibility) VALUES (%s, %s, %s, %s, %s, %s)',
                (data["user_id"], data["name"], data["hour_price"], data["iva"], data["description"], True))
    mysql.connection.commit()
    cur.execute('SELECT LAST_INSERT_ID()')
    row = cur.fetchone()

    id = row[0]
```

```
return Service((id, data["user_id"], data["name"],  
data["hour_price"], data["iva"], data["description"], True)).to_json()
```

La función `create_service(data)` se encarga de crear un nuevo servicio. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `Service`. Luego, comprueba si el servicio ya existe en la base de datos. Si el servicio no existe, se ejecuta una consulta para insertar un nuevo servicio en la base de datos y se devuelve la información del servicio en formato JSON. Si ocurre algún error durante el proceso, se generará una excepción apropiada, como `DBError` o `TypeError`, junto con un mensaje descriptivo.

```
def get_all_services_by_user_id(user_id):  
    # Creamos un cursor:  
    cur = mysql.connection.cursor()  
  
    # Realizamos una consulta SQL:  
    cur.execute('SELECT * FROM service WHERE user_id = %s AND  
visibility = %s', (user_id, True))  
  
    # Esperamos mas de un registro:  
    data = cur.fetchall()  
  
    # Creamos una lista vacia:  
    servicelist = []  
    if cur.rowcount > 0:  
        # Creamos un bucle for: por cada fila:  
        for row in data:  
            # Creamos un objeto servicee:  
            objservice = Service(row)  
            # Agregamos un elemento json a la lista:  
            servicelist.append(objservice.to_json())  
        # Retornamos la lista de elementos json:  
        return (servicelist)  
    raise DBError("Error getting service by id - no row found")
```

La función `get_service_by_id(id)` recibe un ID de servicio y devuelve la información del servicio correspondiente en formato JSON si se encuentra en la base de datos. Si no se encuentra el servicio, se genera una excepción `DBError` con un mensaje descriptivo.

La función `get_all_services_by_user_id(user_id)` recibe un ID de usuario y devuelve una lista de todos los servicios asociados a ese usuario en formato JSON. Si no se encuentra ningún servicio, se genera una excepción `DBError` con un mensaje descriptivo.

UPDATE

```
def update_service(data, service_id):  
  
    if not Service.check_data_schema(data):  
        raise TypeError("Error updating service - wrong data schema")
```



```
cur = mysql.connection.cursor()
cur.execute('SELECT * FROM service WHERE id=%s AND visibility =
1', (service_id,))

if cur.rowcount > 0:
    cur.execute('UPDATE service SET name = %s, hour_price = %s,
iva = %s, description = %s WHERE id=%s',
                (data["name"], data["hour_price"], data["iva"],
data["description"], service_id))

    mysql.connection.commit()
    if cur.rowcount > 0:
        return Service.get_service_by_id(service_id)
    raise DBError("Error updating service - no row updated")
raise DBError("Error updating service - service not found")
```

La función `update_service(id, service_id)` se encarga de actualizar la información de un servicio existente en la base de datos. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `Service`. Luego, ejecuta una consulta para actualizar la información del servicio en la base de datos y devuelve la información actualizada del servicio en formato JSON si se actualiza correctamente. Si no se actualiza el servicio, se genera una excepción `DBError` con un mensaje descriptivo. Si los datos no cumplen con el esquema, se genera una excepción `TypeError` con un mensaje descriptivo.

```
### DELETE

def remove_service(service_id):
    cur = mysql.connection.cursor()
    # Consultamos si esta visible el servicio y existe
    cur.execute('SELECT * FROM service WHERE id=%s AND visibility =
1', (service_id,))

    # Si existe el servicio lo "ELIMINAMOS" (cambio de visibilidad)
    if cur.rowcount > 0:
        cur.execute('UPDATE service SET visibility=%s WHERE id=%s',
(False, service_id))
        mysql.connection.commit()
        return ({ "message": "deleted", "id": service_id })

    raise DBError("Error removing service - service not found")
```

La función `remove_service(service_id)` realiza un borrado lógico de un servicio existente en la base de datos. Primero, se ejecuta una consulta para verificar si el servicio con el ID proporcionado existe y es visible. Si el servicio existe y es visible, se ejecuta otra consulta para actualizar la visibilidad del servicio a `False`. Luego, se devuelve un mensaje indicando que el servicio ha sido eliminado junto con el ID del servicio. Si el servicio no se encuentra, se genera una excepción `DBError` con un mensaje descriptivo.

2.12.2. routes → servicios

```
# CREATE
@app.route('/users/<int:user_id>/service', methods = ['POST'] )
@token_required
@user_resource
def create_service(user_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        new_service = Service.create_service(data)
        return jsonify( new_service ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El *endpoint* `/users/<int:user_id>/service` acepta solicitudes POST y utiliza dos decoradores para realizar controles previos, `'@token_required'` y `'@user_resource'`. La función `create_service(user_id)` recibe los datos proporcionados en la solicitud, crea un nuevo servicio utilizando la clase `Service` y devuelve la información del servicio en formato JSON si se crea correctamente. Si ocurre una excepción durante el proceso de creación, se devuelve un mensaje de error junto con el código de estado 400.

```
# READ

# Ruta: Get de service a traves del ID:
@app.route('/users/<int:user_id>/service/<int:service_id>', methods = ['GET'] )
@token_required
@user_resource
@service_resource
def get_service_by_id(service_id, user_id):

    try:
        service = Service.get_service_by_id(service_id)
        return jsonify( service ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400

# Consultar todos los servicios
@app.route('/users/<int:user_id>/service', methods = ['GET'] )
@token_required
@user_resource
def get_all_services_by_user_id(user_id):

    try:
        services = Service.get_all_services_by_user_id(user_id)
        return jsonify( services ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El primer *endpoint* está configurado para aceptar solicitudes GET y utiliza tres controles a partir de los decoradores '@token_required', '@user_resource' y '@service_resource'. La función 'get_service_by_id' intenta obtener la información de un servicio específico utilizando el ID proporcionado en la solicitud. Si la búsqueda es exitosa, se devuelve la información del servicio en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

El segundo *endpoint* también está configurado para aceptar solicitudes GET. La función 'get_all_services_by_user_id' intenta obtener la información de todos los servicios. Si la búsqueda es exitosa, se devuelve la información de todos los servicios en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

```
# UPDATE
@app.route('/users/<int:user_id>/service/<int:service_id>', methods =
['PUT'] )
@token_required
@user_resource
@service_resource
def update_service(user_id, service_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        service = Service.update_service(data, service_id)
        return jsonify( service ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite actualizar la información de un servicio específico a través de una solicitud PUT. El *endpoint* está configurado para aceptar solicitudes PUT y toma dos parámetros en la URL, el ID del usuario y el ID del servicio. La función asociada a la ruta, update_service(user_id, service_id), recibe los datos proporcionados en la solicitud, actualiza el servicio con el ID especificado y devuelve la información del servicio actualizada en formato JSON si la actualización se realiza correctamente. Si ocurre un error durante el proceso de actualización, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y servicio, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

```
# DELETE
@app.route('/users/<int:user_id>/service/<int:service_id>', methods =
['DELETE'] )
@token_required
@user_resource
@service_resource
def remove_service(service_id, user_id):

    try:
        remove = Service.remove_service(service_id)
        return jsonify( remove ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite eliminar un servicio específico a través de una solicitud DELETE. La ruta está configurada para aceptar solicitudes DELETE y toma dos parámetros en la URL, el ID del servicio y el ID del usuario. La función asociada a la ruta, `remove_service(service_id, user_id)`, intenta hacer un borrado lógico del servicio con el ID especificado y devuelve un mensaje indicando que el servicio ha sido eliminado junto con el ID del servicio si la eliminación se realiza correctamente. Si ocurre un error durante el proceso de eliminación, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y servicio, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

2.13. ESTRUCTURA PRODUCTOS

2.13.1. models → productos

```
class Product():  
  
    # ESQUEMA  
    schema = {  
        "user_id": int,  
        "name": str,  
        "unitary_price": int,  
        "units_stored": int,  
        "iva": int,  
        "description": str  
    }
```

Se define la clase llamada Product, la cual tiene un atributo schema que especifica la estructura de datos esperada para un producto. El atributo schema es un diccionario que define los campos esperados para un producto, junto con el tipo de dato de cada campo.

```
# CHEK DATA SCHEMA  
def check_data_schema(data):  
    if data == None or type(data) != dict:  
        return False  
    # check if data contains all keys of schema  
    for key in Product.schema:  
        if key not in data:  
            return False  
        # check if data[key] has the same type as schema[key]  
        if type(data[key]) != Product.schema[key]:  
            return False  
    return True
```

La función `check_data_schema(data)` se encarga de validar si la estructura de los datos cumple con el esquema predefinido. Primero, verifica si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase Product. Si todas estas condiciones se cumplen, la función devuelve True; de lo contrario, devuelve False.

```
def __init__(self, row):  
    self._id = row[0]  
    self._user_id = row[1]
```

```
self._name = row[2]
self._unitary_price = row[3]
self._units_stored = row[4]
self._iva = row[5]
self._description = row[6]
self._visibility = row[7]

def to_json(self):
    return{
        "id": self._id,
        "user_id": self._user_id,
        "name": self._name,
        "unitary_price": self._unitary_price,
        "units_stored": self._units_stored,
        "iva": self._iva,
        "description": self._description,
        "visibility": self._visibility
    }
```

Se define el constructor de la clase Product. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función *to_json*, cuyo objetivo consiste en retornar los datos de la clase *User* en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

```
##### COMPROBACION DE LA EXISTENCIA DEL PRODUCTO
def product_exists(name, user_id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM product WHERE name =%s AND user_id =
%s AND visibility = 1', (name, user_id))
    cur.fetchall()
    return cur.rowcount > 0
```

El método *Product_exists* consulta si el producto existe en la base de datos.

```
### CREATE
def create_product(data):
    if Product.check_data_schema(data):
        # check if product already exists
        if Product.product_exists(data["name"], data["user_id"]):
            raise DBError("Error creating product - product already
exists")

        cur = mysql.connection.cursor()
        cur.execute('INSERT INTO product (user_id, name,
unitary_price, units_stored, iva, description, visibility) VALUES (%s,
%s, %s, %s, %s, %s)', (data["user_id"], data["name"],
data["unitary_price"], data["units_stored"], data["iva"],
data["description"], True))
```

```
mysql.connection.commit()
if cur.rowcount > 0:
    # get the id of the last inserted row
    cur.execute('SELECT LAST_INSERT_ID()')
    res = cur.fetchall()
    id = res[0][0]
    return Product((id, data["user_id"], data["name"] ,
data["unitary_price"], data["units_stored"], data["iva"],
data["description"], True)).to_json()
    raise DBError("Error creating product - no row inserted")
    raise TypeError("Error creating product - wrong data schema")
```

La función create_Product(data) se encarga de crear un nuevo producto. Primero, verifica si los datos cumplen con el esquema utilizando la función check_data_schema(data) de la clase Product. Luego, comprueba si el producto ya existe en la base de datos. Si el producto no existe, se ejecuta una consulta para insertar un nuevo producto en la base de datos y se devuelve la información del producto en formato JSON. Si ocurre algún error durante el proceso, se generará una excepción apropiada, como DBError o TypeError, junto con un mensaje descriptivo.

```
### READ
def get_product_by_id(id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM product WHERE id=%s AND visibility =
1', (id,))
    data = cur.fetchall()

    if cur.rowcount > 0:
        objProduct = Product(data[0])
        return objProduct.to_json()
    raise DBError("Error getting product - product not found")

def get_all_products_by_user_id(user_id):
    # Creamos un cursor:
    cur = mysql.connection.cursor()

    # Realizamos una consulta SQL:
    cur.execute('SELECT * FROM product WHERE user_id = %s AND
visibility = 1', (user_id, ))

    # Esperamos mas de un registro:
    data = cur.fetchall()

    # Creamos una lista vacia:
    productList = []

    # Creamos un bucle for: por cada fila:
    for row in data:
        # Creamos un objeto producto:
```

```
objProduct = Product(row)

# Agregamos un elemento json a la lista:
productList.append(objProduct.to_json())
# Retornamos la lista de elementos json:
return productList
```

La función `get_product_by_id(id)` recibe un ID de producto y devuelve la información del producto correspondiente en formato JSON si se encuentra en la base de datos. Si no se encuentra el producto, se genera una excepción `DBError` con un mensaje descriptivo.

La función `get_all_products_by_user_id(user_id)` recibe un ID de usuario y devuelve una lista de todos los productos asociados a ese usuario en formato JSON. Si no se encuentra ningún producto, se genera una excepción `DBError` con un mensaje descriptivo.

```
### UPDATE

def update_product(product_id, data):
    if not Product.check_data_schema(data):
        raise TypeError("Error data type not allowed")

    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM product WHERE id=%s AND visibility = 1', (product_id,))

    if cur.rowcount > 0:
        cur.execute('UPDATE product SET name = %s, unitary_price = %s, units_stored = %s, iva = %s, description = %s WHERE id=%s',
                    (data["name"], data["unitary_price"], data["units_stored"], data["iva"], data["description"], product_id))

        mysql.connection.commit()
        return Product.get_product_by_id(product_id)

    raise DBError("Error updating product - product not found")
```

La función `update_product(product_id, data)` se encarga de actualizar la información de un producto existente en la base de datos. Primero, verifica si los datos cumplen con el esquema utilizando la función `check_data_schema(data)` de la clase `Product`. Luego, ejecuta una consulta para actualizar la información del producto en la base de datos y devuelve la información actualizada del producto en formato JSON si se actualiza correctamente. Si no se actualiza el producto, se genera una excepción `DBError` con un mensaje descriptivo. Si los datos no cumplen con el esquema, se genera una excepción `TypeError` con un mensaje descriptivo.

```
### DELETE

def remove_product_by_id(product_id):
    # Conectamos el cursor
    cur = mysql.connection.cursor()
```



```
# Consultamos si esta visible el producto
cur.execute('SELECT * FROM product WHERE id=%s AND visibility =
1', (product_id, ))

# Si existe el producto lo "ELIMINAMOS" (cambio de visibilidad)
if cur.rowcount > 0:
    cur.execute('UPDATE product SET visibility=%s WHERE id=%s',
(False, product_id))
    mysql.connection.commit()
    return {"message": "deleted", "id": product_id}

raise DBError("Error removing product - product not found")
```

La función `remove_product(Product_id)` realiza un borrado lógico de un producto existente en la base de datos. Primero, se ejecuta una consulta para verificar si el producto con el ID proporcionado existe y es visible. Si el producto existe y es visible, se ejecuta otra consulta para actualizar la visibilidad del producto a `False`. Luego, se devuelve un mensaje indicando que el producto ha sido eliminado junto con el ID del producto. Si el producto no se encuentra, se genera una excepción `DBError` con un mensaje descriptivo.

2.13.2. routes → products

```
# CREATE
@app.route('/users/<int:user_id>/product', methods = ['POST'] )
@token_required
@user_resource
def create_product(user_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        product = Product.create_product(data)
        return jsonify (product)
    except Exception as e:
        return jsonify( {"message": e.args[0]}), 404
```

El *endpoint* `/users/<int:user_id>/product` acepta solicitudes POST y utiliza dos decoradores para realizar controles previos, `'@token_required'` y `'@user_resource'`. La función `create_product(user_id)` recibe los datos proporcionados en la solicitud, crea un nuevo producto utilizando la clase `Product` y devuelve la información del producto en formato JSON si se crea correctamente. Si ocurre una excepción durante el proceso de creación, se devuelve un mensaje de error junto con el código de estado 400.

```
# READ

# Ruta: Get de producte a traves del ID:
@app.route('/users/<int:user_id>/product/<int:product_id>', methods =
['GET'] )
@token_required
```



```
@user_resource
@product_resource
def get_product_by_id(product_id, user_id):
    try:
        product = Product.get_product_by_id(product_id)
        return jsonify(product)
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 404

# Consultar todos los productos
@app.route('/users/<int:user_id>/product', methods = ['GET'] )
@token_required
@user_resource
def get_all_products_by_user_id(user_id):
    try:
        products = Product.get_all_products_by_user_id(user_id)
        return jsonify (products)
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

El primer *endpoint* está configurado para aceptar solicitudes GET y utiliza tres controles a partir de los decoradores '@token_required', '@user_resource' y '@product_resource'. La función 'get_product_by_id' intenta obtener la información de un producto específico utilizando el ID proporcionado en la solicitud. Si la búsqueda es exitosa, se devuelve la información del producto en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

El segundo *endpoint* también está configurado para aceptar solicitudes GET. La función 'get_all_products_by_user_id' intenta obtener la información de todos los productos. Si la búsqueda es exitosa, se devuelve la información de todos los productos en formato JSON junto con el código de estado 201. Si ocurre una excepción durante el proceso de búsqueda, se devuelve un mensaje de error junto con el código de estado 400.

```
# UPDATE
@app.route('/users/<int:user_id>/product/<int:product_id>', methods =
['PUT'] )
@token_required
@user_resource
@product_resource
def update_product(user_id, product_id):
    data = request.get_json()
    data["user_id"] = user_id
    try:
        update = Product.update_product(product_id, data)
        return jsonify(update)
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite actualizar la información de un producto específico a través de una solicitud PUT. El *endpoint* está configurado para aceptar solicitudes PUT y toma dos parámetros en la URL, el ID del usuario y el ID del producto. La función asociada a la ruta, `update_product(user_id, product_id)`, recibe los datos proporcionados en la solicitud, actualiza el producto con el ID especificado y devuelve la información del producto actualizada en formato JSON si la actualización se realiza correctamente. Si ocurre un error durante el proceso de actualización, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y producto, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

```
# DELETE
@app.route('/users/<int:user_id>/product/<int:product_id>', methods =
['DELETE'] )
@token_required
@user_resource
@product_resource
def remove_product(product_id, user_id):
    try:
        remove = Product.remove_product_by_id(product_id)
        return jsonify( remove ), 201
    except Exception as e:
        return jsonify( {"message": e.args[0]} ), 400
```

Permite eliminar un producto específico a través de una solicitud DELETE. La ruta está configurada para aceptar solicitudes DELETE y toma dos parámetros en la URL, el ID del producto y el ID del usuario. La función asociada a la ruta, `remove_product(product_id, user_id)`, intenta hacer un borrado lógico del producto con el ID especificado y devuelve un mensaje indicando que el producto ha sido eliminado junto con el ID del producto si la eliminación se realiza correctamente. Si ocurre un error durante el proceso de eliminación, se devuelve un mensaje de error junto con el código de estado 400. La ruta está protegida por autenticación de token y recursos de usuario y producto, lo que garantiza que solo los usuarios autorizados puedan acceder a ella.

2.14. ESTRUCTURA FACTURACION

2.14.1. models → facturacion

```
class Invoice():

    schema = {
        "user_id": int,
        "client_id": int,
        "user_cuil_cuit": str,
        "client_cuil_cuit": str,
        "products_services": list
    }
```

Se define la clase llamada Invoice, la cual tiene un atributo schema que especifica la estructura de datos esperada para una factura. El atributo schema es un diccionario que define los campos esperados para una factura, junto con el tipo de dato de cada campo.

```
def check_data_schema(data):  
    if data == None or type(data) != dict:  
        return False  
    # check if data contains all keys of schema  
    for key in Invoice.schema:  
        if key not in data:  
            return False  
        # check if data[key] has the same type as schema[key]  
  
        if type(data[key]) != Invoice.schema[key]:  
            return False  
    return True
```

La función check_data_schema(data) se encarga de validar si la estructura de los datos cumple con el esquema predefinido. Primero, verifica si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase Invoice. Si todas estas condiciones se cumplen, la función devuelve True; de lo contrario, devuelve False.

```
def __init__(self, row):  
    self._id = row[0] # Nro De Factura - primary key  
    self._user_id = row[1]  
    self._client_id = row[2]  
    self._user_cuil_cuit = row[3]  
    self._client_cuil_cuit = row[4]  
    self._date = row[5]  
    self._total_iva = row[6]  
    self._total_price = row[7]  
    self._visibility = row[8]  
  
    def to_json(self):  
        return{  
            "id": self._id,  
            "user_id": self._user_id,  
            "client_id": self._client_id,  
            "user_cuil_cuit": self._user_cuil_cuit,  
            "client_cuil_cuit": self._client_cuil_cuit,  
            "date": self._date,  
            "total_iva": self._total_iva,  
            "total_price": self._total_price,  
            "visibility": self._visibility  
        }
```

Se define el constructor de la clase Invoice. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función *to_json*, cuyo objetivo consiste en retornar los datos de la clase *Invoice* en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

```
def create_invoice(data):  
  
    if not Invoice.check_data_schema(data):  
        raise TypeError("Error creating invoice - wrong data schema")  
  
    if len(data["products_services"]) < 1:  
        raise TypeError("Error creating invoice - product/service not  
found")  
  
    for product_service in data["products_services"]:  
        if not  
Product_Service_Invoice.check_data_count_schema(product_service):  
            raise TypeError("Error creating invoice - wrong data  
schema")  
  
    date = datetime.now()  
    total_iva = 0  
    total_price = 0  
    for product_service in data["products_services"]:  
        partial =  
Product_Service_Invoice.count_price(product_service)  
        total_iva += partial["iva_sub_total"]  
        total_price += partial["sub_total"]
```

La función *create_invoice(data)* crea una factura a partir de los datos proporcionados:
a) Verifica la estructura de datos proporcionados utilizando la función *Client.check_data_schema(data)*. Si los datos no cumplen con la estructura de datos, se levanta una excepción *TypeError*.

b) Verifica si hay al menos un producto o servicio en la lista *data["products_services"]*. Si no hay productos o servicios, se levanta una excepción *TypeError*.

c) Verifica si cada producto o servicio en la lista *data["products_services"]* cumple con la estructura de datos *Product_Service_Invoice.check_data_count_schema(product_service)*. Si no cumplen, se levanta una excepción *TypeError*.

d) Calcula la fecha actual utilizando *datetime.now()*.

c) Calcula el total de IVA y el total de precio para la factura. Para cada producto o servicio en la lista *data["products_services"]*, se calcula el precio parcial utilizando *Product_Service_Invoice.count_price(product_service)*. Luego, se suman los totales de IVA y precio parciales.

Si todo va bien durante la ejecución de la función, se devuelve la información de la factura en formato JSON. Si ocurre un error durante el proceso, se devuelve un mensaje de error junto con el código de estado 400.

```
##### Creacion de la factura
cur = mysql.connection.cursor()
cur.execute('INSERT INTO invoice (user_id, client_id,
user_cuil_cuit, client_cuil_cuit, date, total_iva, total_price,
visibility) VALUES ( %s, %s, %s, %s, %s, %s, %s, %s)',
            ( data["user_id"], data["client_id"],
data["user_cuil_cuit"], data["client_cuil_cuit"], date, total_iva,
total_price, True))
mysql.connection.commit()

if cur.rowcount < 1:
    raise DBError("Error creating invoice - no row inserted")
```

Después, ejecuta una consulta para insertar la factura en la base de datos y verifica si se ha insertado correctamente. Si no se ha insertado correctamente, se levanta una excepción.

```
##### Creacion de los registros de productos y servicios
cur.execute('SELECT LAST_INSERT_ID()')
res = cur.fetchall()
id = res[0][0]

invoice = (Invoice((id, data["user_id"], data["client_id"],
data["user_cuil_cuit"], data["client_cuil_cuit"], date, total_iva,
total_price, True)).to_json())
invoice_list = []

for product_service in data["products_services"]:
    partial =
Product_Service_Invoice.count_price(product_service)

    iva_sub_total = partial["iva_sub_total"]
    sub_total = partial["sub_total"]

    if product_service["prd_serv"] == "s":
        cur.execute('INSERT INTO product_service_invoice
(invoice_id, user_id, client_id, prd_serv, product_id, service_id,
units_hours, sub_total_iva, sub_total, visibility) VALUES ( %s, %s, %s,
%s, NULL, %s, %s, %s, %s, %s)',
                    (id, data["user_id"], data["client_id"],
product_service["prd_serv"], product_service["ps_id"],
product_service["units_hours"], iva_sub_total, sub_total, True))
        mysql.connection.commit()
        cur.execute('SELECT LAST_INSERT_ID()')
        ps_id = cur.fetchone()
        product_service_invoice =
Product_Service_Invoice((ps_id[0], id, data["user_id"],
data["client_id"], product_service["prd_serv"],None,
```

```
product_service["ps_id"], product_service["units_hours"], iva_sub_total,
sub_total, True)).to_json()

        elif product_service["prd_serv"] == "p":
            cur.execute('INSERT INTO product_service_invoice
(invoice_id, user_id, client_id, prd_serv, product_id, service_id,
units_hours, sub_total_iva, sub_total, visibility) VALUES ( %s, %s, %s,
%s, %s, NULL, %s, %s, %s, %s)',
                        ( id, data["user_id"], data["client_id"],
product_service["prd_serv"], product_service["ps_id"],
product_service["units_hours"], iva_sub_total, sub_total, True))
            mysql.connection.commit()
            cur.execute('SELECT LAST_INSERT_ID()')
            ps_id = cur.fetchone()

            # ACTUALIZAMOS EL STOCK
            cur.execute('SELECT units_stored FROM product WHERE id =
%s', (product_service["ps_id"],))
            row = cur.fetchone()
            stock = row[0]
            reduce_stock = stock - product_service["units_hours"]
            cur.execute('UPDATE product SET units_stored = %s WHERE
id = %s', (reduce_stock, product_service["ps_id"]))
            mysql.connection.commit()

            product_service_invoice =
Product_Service_Invoice((ps_id[0], id, data["user_id"],
data["client_id"], product_service["ps_id"], None,
product_service["prd_serv"], product_service["units_hours"],
iva_sub_total, sub_total, True)).to_json()

        else:
            raise DBError("Error creating invoice - no row inserted")

    if cur.rowcount < 1:
        raise DBError("Error creating invoice - no row inserted")

    invoice_list.append(product_service_invoice)

output = {
    "invoice": invoice,
    "service_products": invoice_list
}
```

```
return output
```

Después de insertar la factura principal en la tabla "invoice", el código recupera el ID de la factura recién creada utilizando `SELECT LAST_INSERT_ID()`. Luego, itera a través de los productos o servicios en la factura, calcula los totales parciales y realiza las inserciones correspondientes en la tabla "product_service_invoice" para cada producto o servicio.

Resumen:

- Después de insertar la factura principal, recupera el ID de la factura recién creada utilizando `SELECT LAST_INSERT_ID()`.
- Itera a través de los productos o servicios en la factura y calcula los totales parciales para cada uno.
- Para cada producto o servicio, realiza una inserción en la tabla "product_service_invoice" con los detalles correspondientes, como el ID de la factura, el ID del usuario, el ID del cliente, etc.
- Si ocurre un error durante el proceso de inserción, se levanta una excepción `DBError`.
- Después de completar todas las inserciones, devuelve un objeto JSON que contiene la información de la factura principal y una lista de los productos o servicios incluidos en la factura.

Aquí se completa el proceso de creación de una factura en la base de datos, incluyendo la inserción de la factura principal y de los productos o servicios asociados a la factura.

Operaciones vinculadas al CRUD.

```
##### READ
def get_all_invoice_by_user_id(user_id):
    cur = mysql.connection.cursor()
    cur.execute('SELECT * FROM invoice WHERE user_id = %s AND
visibility = 1', (user_id, ))
    data = cur.fetchall()

    if not data:
        raise DBError("Error - row not found")
    invoiceList = []

    # Creamos un bucle for: por cada fila:
    for row in data:
        # Creamos un objeto invoiceo:

        objInvoice = Invoice(row).to_json()
        invoiceList.append(objInvoice)
    invoiceList.reverse()

    return invoiceList
```

La función `get_all_invoice_by_user_id(user_id)` recibe un ID de usuario y devuelve una lista de todas las facturas asociadas a ese usuario en formato JSON. Si no se encuentra ninguna factura, se genera una excepción `DBError` con un mensaje descriptivo.

```
def get_invoice_by_id(id):  
  
    cur = mysql.connection.cursor()  
    cur.execute('SELECT * FROM invoice WHERE id = %s AND visibility =  
1', (id, ))  
    data = cur.fetchall()  
  
    if not data:  
        raise DBError("Error")  
  
    objInvoice = Invoice(data[0]).to_json()  
  
    output = {  
        "invoice": objInvoice,  
        "service_products": []}  
  
    cur.execute('SELECT * FROM product_service_invoice WHERE  
invoice_id = %s AND visibility = 1', (id, ))  
    service = cur.fetchall()  
    for serv in service:  
        objProdServ = Product_Service_Invoice(serv)  
        output["service_products"].append(objProdServ.to_json())  
  
    return output
```

La función `get_invoice_by_id` tiene como objetivo obtener una factura a través de su ID. Acciones:

- Utiliza un cursor para ejecutar una consulta SQL y obtener los datos de la tabla `invoice` donde `id` coincida con el ID proporcionado.
- Si no se encontraron datos, se levanta una excepción `DBError`.
- Si se encontraron datos, se crea un objeto `Invoice` a partir de los datos obtenidos y se convierte a JSON.
- Se crea una lista vacía `service_products` para almacenar los productos o servicios asociados a la factura.
- Ejecuta una consulta SQL para obtener los datos de la tabla `product_service_invoice` donde `invoice_id` coincida con el ID de la factura.
- Para cada registro obtenido, se crea un objeto `Product_Service_Invoice` a partir de los datos y se lo agrega a la lista `service_products`.
- Devuelve un diccionario que contiene la factura (en formato JSON) y la lista de productos o servicios asociados (en formato JSON) como valores.

```
##### DELETE  
  
def remove_invoice(invoice_id):  
    cur = mysql.connection.cursor()
```



```
# Consultamos si esta visible el producto
cur.execute('SELECT * FROM invoice WHERE id=%s AND visibility =
1', (invoice_id, ))

# Si existe el producto lo "ELIMINAMOS" (cambio de visibilidad)
if cur.rowcount > 0:
    cur.execute('UPDATE invoice SET visibility=%s WHERE id=%s',
(False, invoice_id))
    mysql.connection.commit()

    cur.execute('SELECT * FROM product_service_invoice WHERE
invoice_id=%s AND visibility =%s', (invoice_id, True))

    # Si existe el producto lo "ELIMINAMOS" (cambio de
visibilidad)
    if cur.rowcount > 0:
        cur.execute('UPDATE product_service_invoice SET
visibility=%s WHERE invoice_id=%s', (False, invoice_id))
        mysql.connection.commit()

    return {"message": "deleted", "id": invoice_id}

raise DBError("Error deleting invoice - id not found")
```

La función `remove_invoicet(invoice_id)` realiza un borrado lógico de una factura existente en la base de datos. Primero, se ejecuta una consulta para verificar si la factura con el ID proporcionado existe y es visible. Si la factura existe y es visible, se ejecuta otra consulta para actualizar la visibilidad de la factura a False. Luego, se devuelve un mensaje indicando que la factura ha sido eliminado junto con el ID de la factura. Si la factura no se encuentra, se genera una excepción `DBError` con un mensaje descriptivo.

```
##### RANKING FACTURA CLIENTE
def ranking_clients(user_id):
    try:
        # BUSCAMOS TODOS LAS FACTURAS POR CLIENTE DEL USUARIO
        invoices = Invoice.get_all_invoice_by_user_id(user_id)
    except Exception as e:
        raise e

    # CREAMOS VARIABLES DE APOYO Y SALIDA
    output = []
    invoiceObj = {}

    for invoice in invoices:
```

```
# Actualiza las compras por cliente
if invoiceObj.get(invoice["client_id"]):
    invoiceObj[invoice["client_id"]]["total_buys"]+= 1
    invoiceObj[invoice["client_id"]]["total_mount"]+=
invoice["total_price"]

# Ingresa compra de cliente
else:
    invoiceObj[invoice["client_id"]] = {"total_buys": 1,
                                         "total_mount":
invoice["total_price"]}]

# CREAMOS OBJETOS CON LOS DATOS ENTEROS POR CLIENTE
for key in invoiceObj.keys():
    objAux = {"ps_id": key,
              "count": invoiceObj[key]["total_buys"],
              "total_mount": invoiceObj[key]["total_mount"]}

# Ingresamos en la salida
output.append(objAux)

# Ordenamos los clientes por cantidad de compras
for i in range(len(output) - 1):

    for j in range(len(output) - i):
        if output[j + i]["count"] > output[i]["count"]:
            aux = output[i]
            output[i] = output[j + i]
            output[j + i] = aux

return output
```

La función realiza un ranking de clientes basado en sus compras. La función toma el ID de un usuario como entrada y devuelve una lista de clientes ordenados por la cantidad de compras realizadas y el monto total de las compras:

- La función obtiene todas las facturas asociadas al usuario utilizando la función `Invoice.get_all_invoice_by_user_id(user_id)`.
- Luego, itera a través de las facturas y actualiza un diccionario `invoiceObj` que almacena la cantidad total de compras y el monto total de las compras para cada cliente.
- Luego, crea una lista de objetos JSON que contienen el ID del cliente, la cantidad total de compras y el monto total de las compras para cada cliente.
- Finalmente, ordena la lista de clientes por la cantidad total de compras.

El resultado final es una lista de clientes ordenados por la cantidad total de compras y el monto total de las compras. Este ranking puede ser útil para identificar a los clientes más importantes o para analizar el comportamiento de compra de los clientes.

2.15. ESTRUCTURA FACTURA PRODUCTO SERVICIO

```
class Product_Service_Invoice():

    schema = {
        "invoice_id": int,
        "user_id": int,
        "client_id": int,
        "ps_id": int,
        "prd_serv": str,
        "units_hours": int,
        "iva_subtotal" : int,
        "sub_total": int
    }

    count_schema = {
        "ps_id": int,
        "prd_serv": str,
        "units_hours": int
    }
```

La clase Product_Service_Invoice es parte del sistema de facturación. Define dos atributos estáticos, schema y count_schema, que especifican la estructura de datos esperada para un elemento de factura y para el conteo de elementos respectivamente.

El atributo schema define la estructura de datos esperada para un elemento de factura, incluyendo el ID de la factura, ID de usuario, ID de cliente, ID de producto/servicio, tipo de producto/servicio, unidades/horas, subtotal de IVA y subtotal. Mientras que el atributo count_schema define la estructura de datos esperada para el conteo de elementos, incluyendo el ID de producto/servicio, tipo de producto/servicio y unidades/horas.

```
def check_data_schema(data):
    if data == None or type(data) != dict:
        return False
    # check if data contains all keys of schema
    for key in Product_Service_Invoice.schema:
        if key not in data:
            return False
        # check if data[key] has the same type as schema[key]

        if type(data[key]) != Product_Service_Invoice.schema[key]:

            return False
    return True
```

```
def check_data_count_schema(data):
```

```
if data == None or type(data) != dict:
    return False
# check if data contains all keys of schema
for key in Product_Service_Invoice.count_schema:
    if key not in data:
        return False
    # check if data[key] has the same type as schema[key]
    if type(data[key]) !=
Product_Service_Invoice.count_schema[key]:
        return False
return True
```

Las funciones `check_data_schema(data)` y `check_data_count_schema(data)` se encargan de validar si las estructuras de los datos cumple con los esquemas predefinidos. Primero, verifican si los datos son nulos y si son un diccionario. Luego, compara si los datos contienen todas las claves definidas en el esquema y si los tipos de datos de las claves coinciden con los tipos de datos definidos en el esquema de la clase `Product_Service_Invoice`. Si todas estas condiciones se cumplen, la función devuelve `True`; de lo contrario, devuelve `False`.

```
def __init__(self, row):
    self._id = row[0]
    self._invoice_id = row[1]
    self._user_id = row[2]
    self._client_id = row[3]
    self._prd_serv = row[4]
    self._product_id = row[5]
    self._service_id = row[6]
    self._units_hours = row[7]
    self._iva_subtotal = row[8]
    self._sub_total = row[9]
    self._visibility = row[10]

def to_json(self):
    json = {
        "id": self._id,
        "invoice_id": self._invoice_id,
        "user_id": self._user_id,
        "client_id": self._client_id,
        "prd_serv": self._prd_serv,
        "units_hours": self._units_hours,
        "iva_subtotal" : self._iva_subtotal,
        "sub_total": self._sub_total,
        "visibility": self._visibility
    }
    if self._product_id:
        json["ps_id"] = self._product_id
    elif self._service_id:
        json["ps_id"] = self._service_id
```

```
return json
```

Se define el constructor de la clase `Product_Service_Invoice`. El constructor se recibirá como parámetro en formato de fila, esta fila ingresa por la ruta correspondiente a partir de una consulta recibida desde el FrontEnd. Luego, se define la función `to_json`, cuyo objetivo consiste en retornar los datos de la clase `Invoice` en formato JSON, para poder enviar la respuesta correspondiente al BackEnd.

El último fragmento del código es parte del método `to_json` de la clase `Product_Service_Invoice`. Su función es añadir el ID del producto o del servicio al objeto JSON que se está creando, dependiendo de cuál de los dos esté presente en la instancia de la clase:

`if self._product_id::` Verifica si el ID del producto (`_product_id`) está presente en la instancia actual.

`json["ps_id"] = self._product_id:` Si el ID del producto está presente, se añade al objeto JSON con la clave `"ps_id"`.

`elif self._service_id::` Si el ID del producto no está presente, se verifica si el ID del servicio (`_service_id`) está presente en la instancia actual.

`json["ps_id"] = self._service_id:` Si el ID del servicio está presente, se añade al objeto JSON con la clave `"ps_id"`.

`return json:` Finalmente, se devuelve el objeto JSON resultante.

En resumen, este fragmento de código añade al objeto JSON el ID del producto o del servicio, dependiendo de cuál esté presente en la instancia de la clase `Product_Service_Invoice`. Esto permite incluir de forma dinámica el ID del producto o del servicio en el objeto JSON, dependiendo de los datos con los que se esté trabajando.

```
def count_price(data):
    if not Product_Service_Invoice.check_data_count_schema(data):
        raise TypeError("Error data schema - data schema not
allowed")

    if data["prd_serv"] == 'p':

        product = Product.get_product_by_id(data["ps_id"])

        if data["units_hours"] > product["units_stored"]:
            raise DBError("Error stock - product out of stock")

        value = ((product["unitary_price"] * data["units_hours"]) *
(((100 + product["iva"]))) / 100))
        iva_value = value - (product["unitary_price"] *
data["units_hours"])
        value = int(value)
        iva_value = int(iva_value)
```

```
        output = {
            "iva_sub_total": iva_value,
            "sub_total": value
        }

        return output

    elif data["prd_serv"] == 's':

        service = Service.get_service_by_id(data["ps_id"])

        value = ((service["hour_price"] * data["units_hours"]) *
            (((100 + service["iva"]))) / 100))
        iva_value = value - (service["hour_price"] *
            data["units_hours"])
        value = int(value)
        iva_value = int(iva_value)
        output = {
            "iva_sub_total": iva_value,
            "sub_total": value
        }

        return output

    else:
        raise DBError("Error - attribute not found")
```

La función `count_price` es una función que calcula el precio total de un producto o servicio. La función toma el objeto como entrada y devuelve el precio total correspondiente. La función verifica si el objeto es un objeto válido y si no lo es, devuelve un error.

Se divide en dos partes principales, una para productos y otra para servicios. Para productos, la función obtiene el producto correspondiente al ID del producto y verifica si la cantidad especificada es mayor que el stock del producto. Si es así, devuelve un error. Luego, calcula el precio total y el precio sin IVA, y devuelve un diccionario con estos valores.

Para servicios, la función obtiene el servicio correspondiente al ID del servicio calcula el precio total y el precio sin IVA, y devuelve un diccionario con estos valores.

```
##### GET ALL SERVICE BY ID
def get_invoice_service_list(user_id):

    cur = mysql.connection.cursor()
    output = []
```

```
cur.execute('SELECT * FROM product_service_invoice WHERE  
prd_serv = "s" AND user_id = %s AND visibility = 1', (user_id, ))  
service = cur.fetchall()  
if cur.rowcount < 1:  
    raise DBError("Error service invoice - row not found")  
for serv in service:  
    objProdServ = Product_Service_Invoice(serv)  
    output.append(objProdServ.to_json())  
  
return output
```

Esta función se utiliza para obtener una lista de facturas de servicio relacionadas con un usuario específico y visibilidad 1 (no eliminada). Primero, se ejecuta una consulta SQL para seleccionar todas las facturas de servicio que cumplan con estos criterios. Luego, para cada factura seleccionada, se crea un objeto Product_Service_Invoice utilizando el método to_json() y se lo añade a la lista de salida output. Finalmente, se devuelve la lista de salida output.

```
##### GET ALL PRODUCTS BY ID  
def get_invoice_product_list(user_id):  
  
    cur = mysql.connection.cursor()  
    output = []  
  
    cur.execute('SELECT * FROM product_service_invoice WHERE  
prd_serv = "p" AND user_id = %s AND visibility = 1', (user_id, ))  
    product = cur.fetchall()  
    if cur.rowcount < 1:  
        raise DBError("Error product invoice - row not found")  
    for prod in product:  
        objProdServ = Product_Service_Invoice(serv)  
        output.append(objProdServ.to_json())  
  
    return output
```

Esta función se utiliza para obtener una lista de facturas de producto relacionadas con un usuario específico y visibilidad 1 (no eliminada). Primero, se ejecuta una consulta SQL para seleccionar todas las facturas de producto que cumplan con estos criterios. Luego, para cada factura seleccionada, se crea un objeto Product_Service_Invoice utilizando el método to_json() y se lo añade a la lista de salida output. Finalmente, se devuelve la lista de salida output.

```
def count_product_selled(user_id):  
    try:  
        product_list =  
Product_Service_Invoice.get_invoice_product_list(user_id)  
    except Exception as e:  
        raise e  
  
    output = []
```

```
productsObj = {}
for product in product_list:
    if productsObj.get(product["ps_id"]):
        productsObj[product["ps_id"]]["total_units_hours"]+=
product["units_hours"]
        productsObj[product["ps_id"]]["total_mount"]+=
product["sub_total"]
    else:
        productsObj[product["ps_id"]] = {"total_units_hours":
product["units_hours"],
                                           "total_mount":
product["sub_total"]}

for key in productsObj.keys():
    objAux = {"ps_id": key,
              "count": productsObj[key]["total_units_hours"],
              "total_mount": productsObj[key]["total_mount"]}
    output.append(objAux)

for i in range(len(output) - 1):

    for j in range(len(output) - i):
        if output[j + i]["count"] > output[i]["count"]:
            aux = output[i]
            output[i] = output[j + i]
            output[j + i] = aux

return output

def count_service_ofered(user_id):
    try:
        service_list =
Product_Service_Invoice.get_invoice_service_list(user_id)
    except Exception as e:
        raise e

    output = []
    servicesObj = {}

    for service in service_list:
        if servicesObj.get(service["ps_id"]):
            servicesObj[service["ps_id"]]["total_units_hours"]+=
service["units_hours"]
            servicesObj[service["ps_id"]]["total_mount"]+=
service["sub_total"]
        else:
```



```
        servicesObj[service["ps_id"]] = {"total_units_hours":  
service["units_hours"],  
                                          "total_mount":  
service["sub_total"]}  
  
    for key in servicesObj.keys():  
        objAux = {"ps_id": key,  
                  "count": servicesObj[key]["total_units_hours"],  
                  "total_mount": servicesObj[key]["total_mount"]}  
        output.append(objAux)  
  
    for i in range(len(output) - 1):  
  
        for j in range(len(output) - i):  
            if output[j + i]["count"] > output[i]["count"]:  
                aux = output[i]  
                output[i] = output[j + i]  
                output[j + i] = aux  
  
    return output
```

Ambas funciones toman como parámetro user_id y realizan operaciones en la lista de productos o servicios obtenidos a través de la clase Product_Service_Invoice.

La función count_product_selled obtiene una lista de productos vendidos por el usuario, calcula el total de unidades y el monto total vendido por cada producto, y luego ordena la lista de productos por el total de unidades en orden descendente antes de devolverla. La función count_service_ofered hace lo mismo que la anterior, pero para los servicios ofrecidos en lugar de los productos vendidos.

Ambas funciones manejan excepciones al obtener las listas de productos o servicios, y devuelven una lista de objetos que contienen el ID del producto o servicio, el total de unidades/horas y el monto total. El código utiliza diccionarios para calcular los totales y ordenar la lista de productos o servicios.

```
# Movimiento de stock  
def control_flow_product(user_id):  
    cur = mysql.connection.cursor()  
    output = []  
  
    cur.execute('SELECT * FROM product_service_invoice WHERE prd_serv  
= "p" AND user_id = %s AND visibility = 1', (user_id, ))  
    product = cur.fetchall()  
  
    if cur.rowcount < 1:  
        raise DBError("Error product invoice - row not found")  
  
    for prod in product:  
        objProduct = Product_Service_Invoice(prod)
```

```
        json = objProduct.to_json()
        cur.execute('SELECT date FROM invoice WHERE id = %s AND
visibility = 1', (json["invoice_id"] ,))
        row = cur.fetchone()
        date = row[0]

        report = {
            "id_product": json["ps_id"],
            "id_invoice": json["invoice_id"],
            "flow": (json["units_hours"] * -1),
            "date": date,
            "total_price": json["sub_total"]
        }

        output.append(report)
    output.reverse()

    return output
```

La función `control_flow_product` se encarga de manejar el movimiento de stock de producto. Recupera los registros de facturas de productos para un usuario específico, procesa la información y devuelve una lista de diccionarios que representan el movimiento de stock de los productos.

- a) Se establece una conexión a la base de datos y se inicializa una lista vacía llamada `output`.
- b) Se ejecuta una consulta SQL para seleccionar los registros de la tabla `product_service_invoice` donde el tipo es "p", el ID de usuario es igual a `user_id` y la visibilidad es 1 (no eliminado). Los resultados de la consulta se almacenan en la variable `product`.
- c) Se verifica si se encontraron filas en el resultado de la consulta. Si no se encontró ninguna fila, se lanza una excepción `DBError` con el mensaje "Error product invoice - row not found".
- d) Se itera a través de los registros de `product` y se realiza lo siguiente para cada uno:
 - d.1) Se crea un objeto `Product_Service_Invoice` a partir del registro.
 - d.2) Se convierte el objeto a formato JSON.
 - d.3) Se ejecuta una segunda consulta SQL para obtener la fecha de la factura a partir del ID de la factura en el registro actual. La fecha se almacena en la variable `date`.

Se crea un diccionario llamado `report` que contiene la información del producto, incluyendo el ID del producto, el ID de la factura, el flujo de unidades, la fecha y el precio total. Este diccionario se agrega a la lista `output`.

Después de iterar a través de todos los registros, se invierte el orden de la lista `output` y se retorna.