**DOCUMENTACION AL PROGRAMADOR**

Versión - Commit: b11602e231f9eed6d09178791f5604e34992592c – (Nov 28, 2023)

1. **INTRODUCCION**

La presente documentación se encuentra destinada a todos aquellos programadores que quieran hacer uso de la API de FACTURACION ONLINE y quieran ponerlo en funcionamiento, entender su arquitectura de código, su base de datos 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.

De más está decir que la documentación está destinada a todos aquellos programadores que quieran colaborar con el proyecto y aportar su granito de arena para mejorar a futuro.

1. **PROCEDIMIENTOS INICIALES Y PUESTA EN FUNCIONAMIENTO DE LA API**

En esta sección iremos acompañando al programador a lo largo de todos los pasos necesarios que deberá realizar para poder poner en funcionamiento la API: FACTURACION ONLINE, para ello se recomienda la utilización de diferentes programas previos para lograr su mejor funcionamiento, los que serán comentados a continuación:

* 1. **VISUAL STUDIO CODE (recomendado)**

Como editor de texto vamos a utilizar VISUAL STUDIO CODE (recomendado) debido a que fue el utilizado al realizar y programar la API: FACTURACION ONLINE.

*Link de Descarga:* [*https://code.visualstudio.com/download*](https://code.visualstudio.com/download)



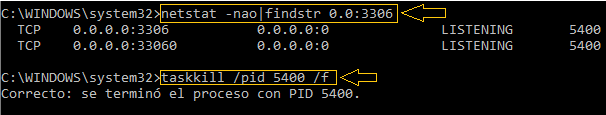
* 1. **XAMPP (recomendado)**

Como motor de base de datos vamos a utilizar (recomendado) XAMPP que nos permitirá con APACHE crear un servidor local “Local Host” para hacer las correspondientes consultas desde la API: FrontEnd al BackEnd, y con MYSQL nos permitirá gestionar nuestra base de datos y su correspondiente estructura.

*Link de Descarga:* [*https://www.apachefriends.org/es/index.html*](https://www.apachefriends.org/es/index.html)



Nota: tener en consideración que el presente programa 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 la posible solución recomendada es ingresar como administrador al CMD, realizar una consulta de si el puerto 3306 se encuentre siendo utilizado por otro programa, y de existir esa situación proceder a suspenderlo para que se ponga a disposición. Se puede realizar con los siguientes comandos:



* 1. **SISTEMA DE CONTROL DE VERSIONES**

Como sistema de control de versiones se utilizó GIT y el repositorio donde se encontrara todos los archivos, documentación y código necesarios para utilizar la API: FACTURACION ONLINE se encontraran en GITHUB.

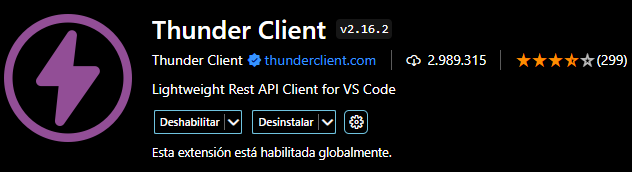
*Link de Descarga:* [*https://git-scm.com/*](https://git-scm.com/)

*Link del Repositorio:* [*https://github.com/SaizMarcelo/PROYECTO-INFORMATICO*](https://github.com/SaizMarcelo/PROYECTO-INFORMATICO)



* 1. **TESTEOS DEL LADO DEL CLIENTE: THUNDER CLIENT**

Los testeos realizados por el equipo de programación de la API: FACTURACION ONLINE fueron realizados en todo momento utilizando la extensión THUNDER CLIENT (recomendada), extensión del VISUAL STUDIO CODE.



* 1. **PROCEDIMIENTO PUESTA EN FUNCIONAMIENTO**

En el presente apartado se dar procedimiento paso a paso para realizar todas las configuraciones necesarias y requeridas para la puesta en funcionamiento de la API: FACTURACION ONLINE.

* + 1. **CREAR CARPETA**

Para dar comienzo a este procedimiento vamos a necesitar crear una carpeta vacía en donde vamos a vincular a git, descargar toda la documentación y archivos necesarios y crear la zona de trabajo virtual y configuraciones necesarias para poder ejecutar de forma local la API: FACTURACION ONLINE para que el programador pueda trabajar con la misma.



* + 1. **INICIALIZACION GIT A LA CARPETA VACIA**

Luego de creada la carpeta vacía donde se ira cargando toda la información vamos a vincular sistema de control de versiones con la carpeta creada previamente, para ello vamos a ingresar al editor de texto VISUAL ESTUDIO CODE y vamos a posicionarnos en la carpeta PROYECTO\_UPSO para luego ingresar el siguiente comando:



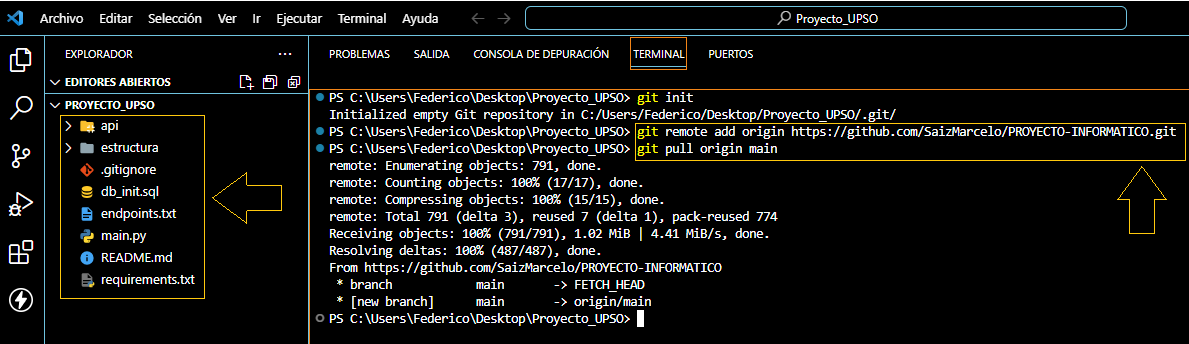


* + 1. **AGREGAMOS AL REPOSITORIO LOCAL EL REPOSITORIO DE GITHUB: API: FACTURACION ONLINE.**

Para realizar este paso vamos a ingresar en la terminar el siguiente comando para incorporar el repositorio de GitHub en el repositorio local: Primero se vincula el repositorio a la ruta y luego se realiza un “pull” a dicha ruta para traer del repositorio git todos los archivos al repositorio local.



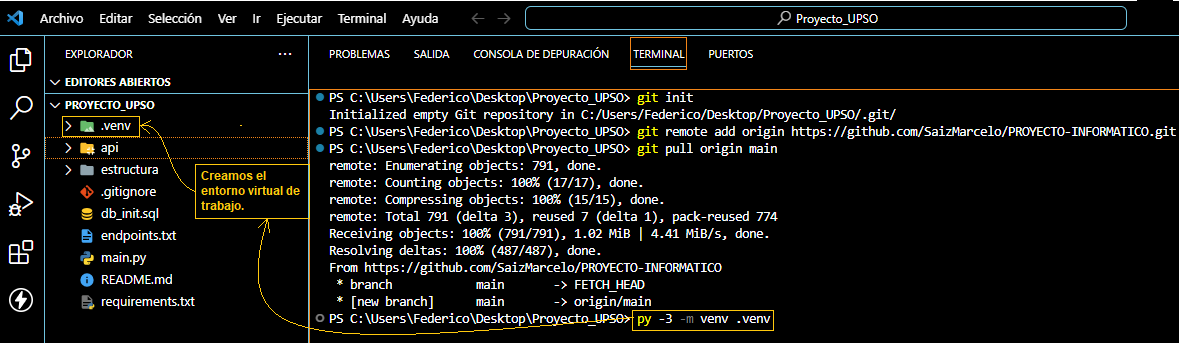




* + 1. **CREAMOS UN ENTORNO VIRTUAL DE TRABAJO:**

Luego de haber vinculado y descargado todos los archivos relacionados a la API: FACTURACION ONLINE vamos a crear un entorno virtual de trabajo para que toda la API se encuentre contenida en el mismo que sea consistente con la configuración planteada. Para lograr esto en la terminal vamos a indicar el siguiente comando:

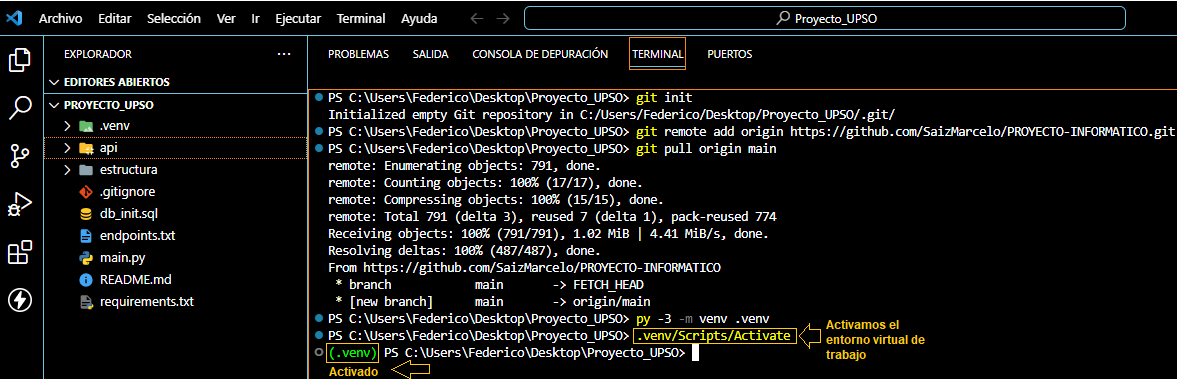
 El 3 hace referencia a la versión de Python con la que vamos a crear nuestro entorno virtual de trabajo.



* + 1. **ACTIVAMOS EL ENTORNO VIRTUAL:**

Para poder trabajar dentro del entorno virtual debemos realizar la activación del mismo, lo cual se podrá lograr con los siguientes comandos:

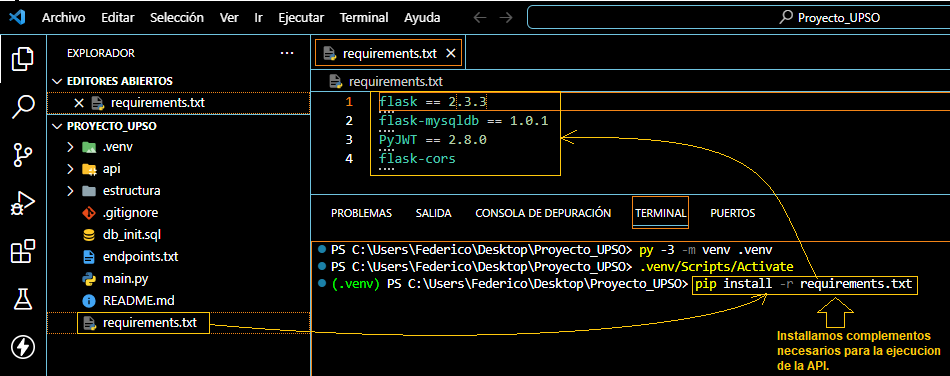




* + 1. **DESCARGAMOS REQUERIMIENTOS MINIMOS PARA LA EJECUCION DE LA API: FACTURACION ONLINE**:

El funcionamiento de una API requiere diferentes librerías con las cuales se trabaja, por ello dentro del archivo  se encuentran todos los requerimientos mínimos y utilizados por el equipo de programadores para la API, por ello será necesario descargar en el entorno de trabajo virtual estos requerimientos mínimos, y para ello debemos ejecutar el siguiente comando:



****

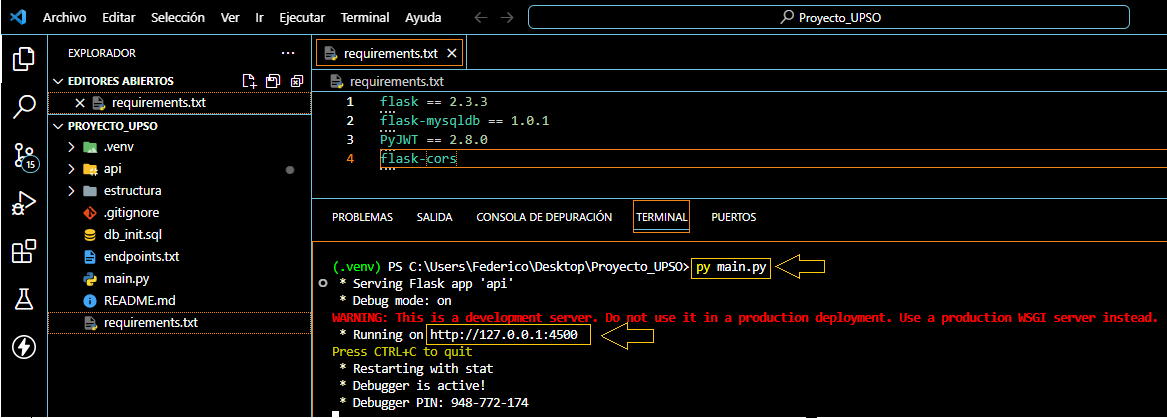
Luego podremos consultar la existencia de los complementos con el siguiente comando: 

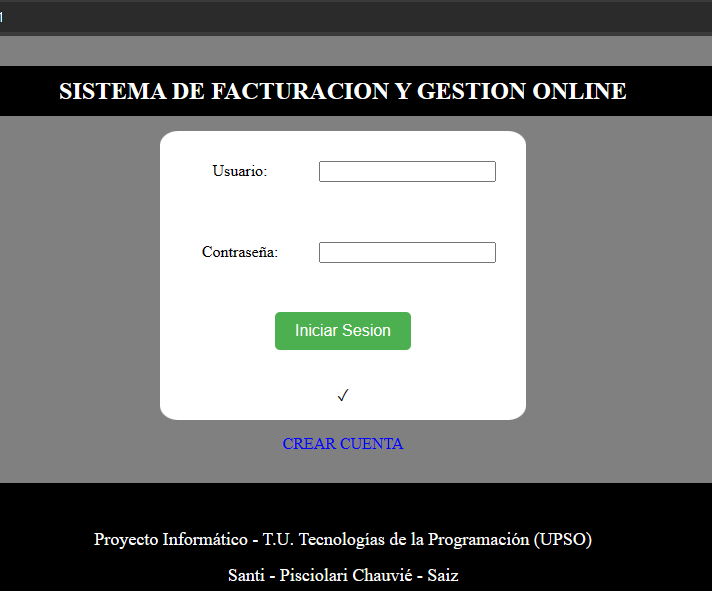


* + 1. **EJECUTAMOS LA API**.

Una vez realizados todos los pasos anteriores y posicionados en este punto ya estaríamos en condiciones de ejecutar la API para ver que se ejecuta correctamente, ello lo vamos a realizar con el siguiente comando:

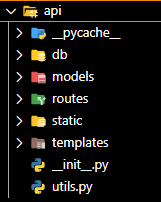
, luego ejecutamos el vínculo, link: .





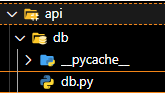
* 1. **ESTRUCTURA – ARQUITECTURA - API**

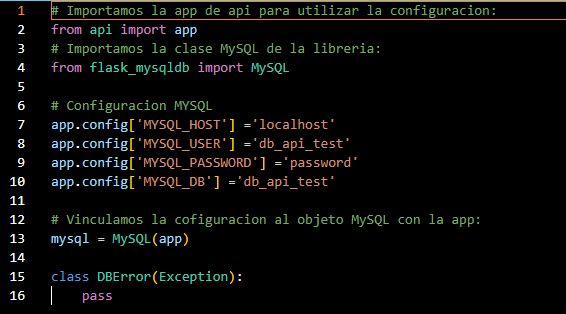
Una vez que ya logramos ejecutar correctamente la API, vamos a exponer como se encuentra estructurada la API, como es su arquitectura para que el programador que lo requiera tenga el conocimiento necesario para interpretar y adaptarse a dicha arquitectura de requerirlo.



* + 1. **api --> db --> db.py:**

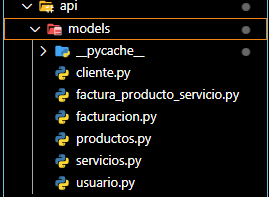
En esta ruta se encuentra alojada el db.py que contendrá las configuraciones requeridas por la base de datos relacionadas a MySQL, gestor de base de datos que utilizara la aplicación, a través de XAMPP, también se encuentra definido el DBError ante la existencia de errores ante la consulta a la base de datos.

****



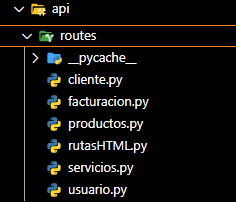
* + 1. **api --> models:**

En esta ruta se encontraran todos los archivos.py en los cuales se alojaran las CLASES relacionadas a sus correspondientes tablas SQL dentro de la base de datos, en dichas clases se encontrara 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, sus correspondientes CRUD (Create, Read, Update, Delete). Su código será analizado posteriormente.

****

* + 1. **api --> routes:**

En esta ruta se encontraran todos los archivos.py que contendrán el código de las URL que permiten vincular al FrontEndo con el BackEnd, siendo estas URL el intervalo entre los dos para mantener el nivel de seguridad y permitiendo crear las funcionalidades que podrá ejecutarse desde el FrontEnd, estas 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).

****

El código de estas rutas se analizara posteriormente pero a continuación se expondrán cuáles son las rutas: URL con las que se hacen las consultas individualizadas anteriormente y como se encuentran estructuradas las mismas:

* + - 1. **api --> routes --> cliente:** 
         1. Crear un cliente nuevo en un determinado usuario:



* + - * 1. Consultar de un determinado usuario un cliente por ID.



* + - * 1. Consultar todos los clientes de un determinado usuario.



* + - * 1. Modificar la información de un determinado cliente por ID de un determinado Usuario.



* + - * 1. Eliminar la información de un cliente por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).



* + - 1. **api --> routes --> facturacion:** 
         1. Crear una factura a un determinado usuario.



* + - * 1. Consultar de un determinado usuario una factura por ID.



* + - * 1. Consultar todas las facturas de un determinado usuario.



* + - * 1. Consultar ranking de servicios facturados de un determinado usuario.



* + - * 1. Consultar ranking de productos facturados de un determinado usuario.



* + - * 1. Consultar ranking de clientes facturados de un determinado usuario.



* + - * 1. Consulta movimiento de productos de un determinado usuario.



* + - * 1. Eliminar la información de una factura por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).



* + - * 1. **Nota**: se observa que no existe una ruta para “PUT” esto se debe a que una factura técnicamente es un instrumento que respalda una determinada operación, al emitirse la misma legalmente ya no se puede modificar en nada de sus datos, únicamente se podrá anular a través de otro documento denominado NOTA DE CREDITO (no implementado).
      1. **api --> routes --> productos:** 
         1. Crear un producto para un determinado usuario.



* + - * 1. Consultar un producto por ID de un determinado Usuario.



* + - * 1. Consultar todos los productos de un determinado Usuario.



* + - * 1. Modificar los datos de un producto por ID de un determinado Usuario.



* + - * 1. Eliminar la información de una producto por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).



* + - 1. **api --> routes --> servicios:** 
         1. Crear un servicio a un determinado usuario.



* + - * 1. Consultar servicio por ID de un determinado Usuario.



* + - * 1. Consultar todos los servicios de un determinado Usuario.



* + - * 1. Modificar los datos de un determinado servicios por ID de un determinado Usuario.



* + - * 1. Eliminar la información de una servicio por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).



* + - 1. **api --> routes --> usuario:** 
         1. Crear un token de ingreso para un usuario existente.



* + - * 1. Crear un nuevo usuario.



* + - * 1. Consultar usuario por ID.



* + - * 1. Consultar todos los usuarios.



* + - * 1. Modificar los datos de un usuario por ID.



* + - * 1. Eliminar la información de una usuario por ID de un determinado Usuario. (Se aplica el concepto de borrado lógico).

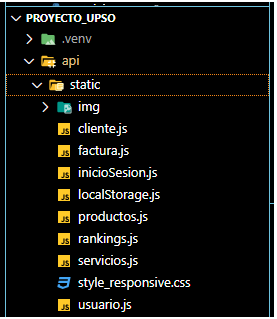


* + - 1. **api --> routes --> rutasHTML:** 
         1. Ruta que permite conectar con el archivo: ‘public/dashboard.html’**.**



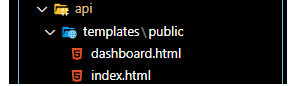
* + 1. **api --> static:**

En esta ruta se encontraran todos los archivos.js individualizados según las diferentes consultas que se pueden realizar desde el FrontEnd, son las funcionalidades que nos permitirán 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 asi obtener las respuestas del servidor a dichas consultas emitidas. El código se analizara posteriormente. También en esta ruta se encuentran los ESTILOS.CSS y IMÁGENES.

****

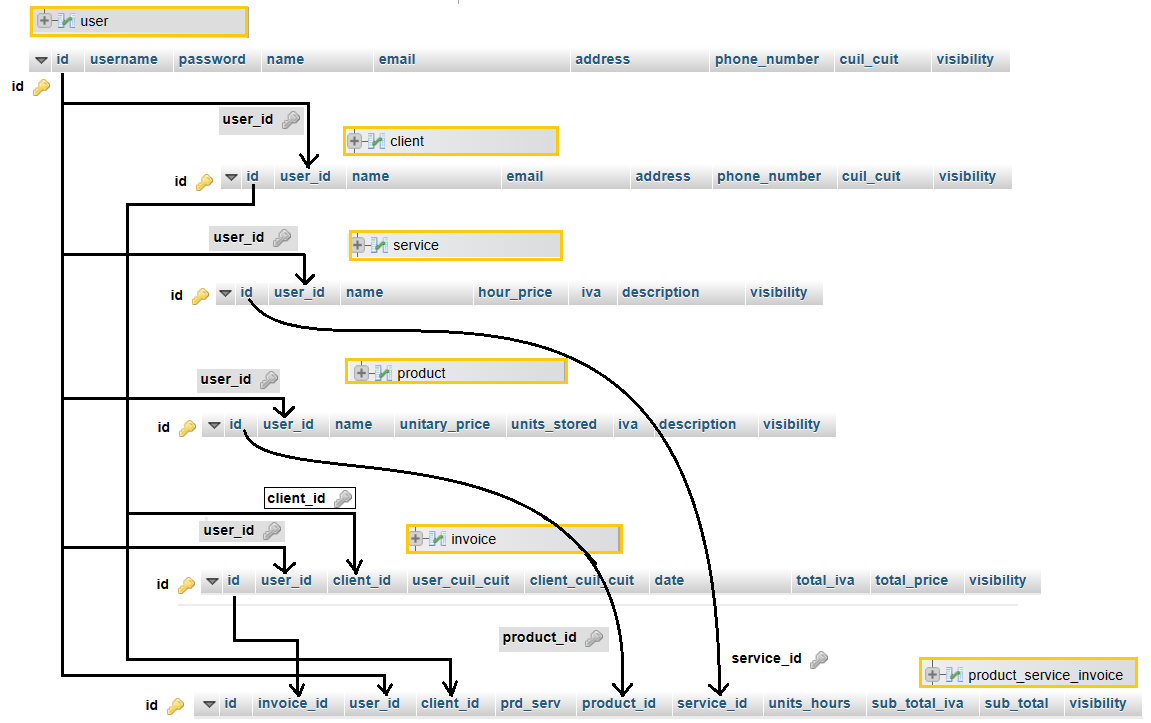
* + 1. **api --> templates\public:**

En esta ruta se encontraran los archivos.html que serán los que se ejecutaran por el navegador y permitirán recopilar la información brindada por el usuario para ser enviada al backEnd. Posteriormente se expondrá su estructura.



* 1. **ESTRUCTURA – ARQUITECTURA DE LA BASE DE DATOS Y SUS FUNDAMENTOS**
     1. **ESTRUCTURA ACTUAL.**

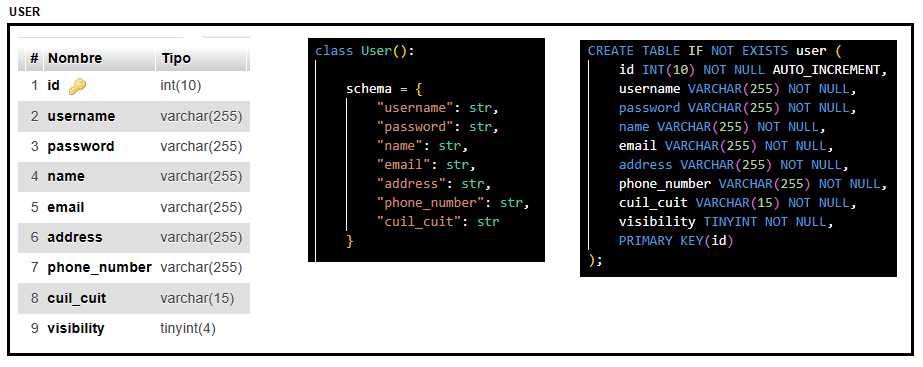
La estructura – arquitectura de la base de datos se definió en función de un análisis técnico dentro del equipo multidisciplinario, siendo en primera instancia acotado pero consistente con potencial de ampliación a futuro.

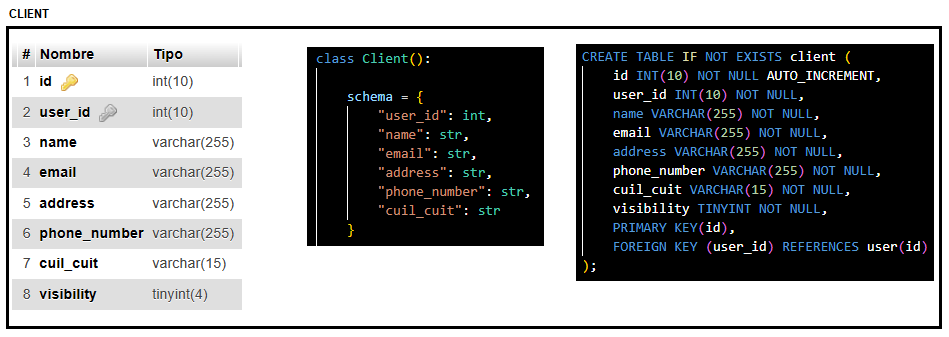


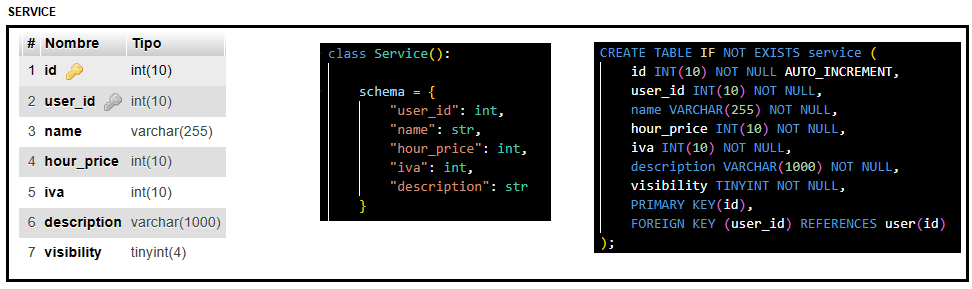
En esta estructura se entiende que el USUARIO puede tener (0 o +) CLIENTES, puede tener (0 o +) SERVICIOS, podre tener (0 o +) PRODUCTOS, y podrá tener (0 o +) FACTURAS.

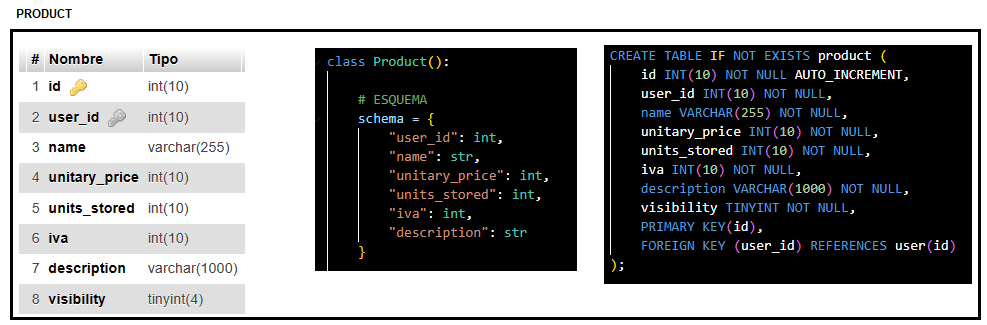
Una FACTURA solo podrá tener (1) RECEPTOR: USUARIO, solo podrá tener (1) EMISOR: CLIENTE, podrá tener (0 o +) SERVICIOS, (0 o +) PRODUCTOS.

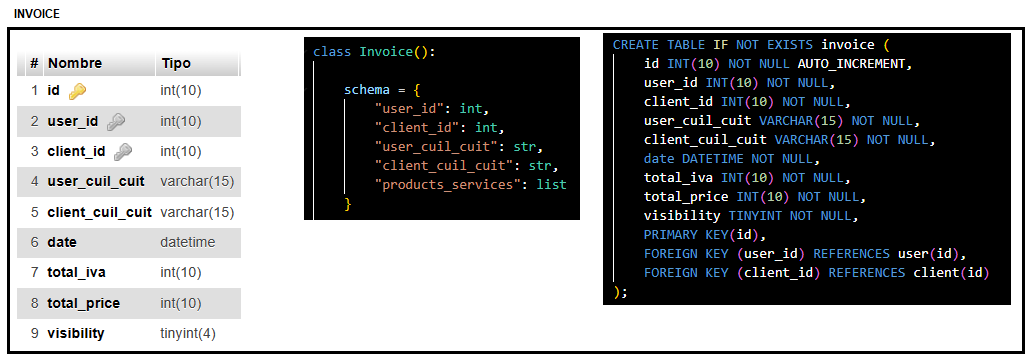
* + 1. **ESTRUCTURA TECNICA – MYSQL – CLASES: PYTHON - SQL:**

****

****

****

****

****

****

**Nota:** Al momento de utilizar XAMPP como motor de gestión de base de datos es necesario tener en cuenta la configuración de privilegios para crear diferentes usuarios, cuestión que genero complicaciones en su momento al grupo de programación.

* + 1. **POSIBLES MEJORAS EN LA ESTRUCTURA A FUTURO.**

Con relacion a las mejoras, esta estructura desde el punto de vista técnico tiene mucho para agregarse, ya que técnica y tributariamente requiere aplicar diferentes puntos de vista, como por ejemplo que tipo de contribuyente es el USUARIO y que tipo de contribuyente es el CLIENTE los cuales pueden ser (Responsable monotributo, Responsable Inscripto, Exento en IVA, Consumidor Final, Sujeto no Categorizado, Proveedor del Exterior, IVA no alcanzado entre tantos otros) esta combinación de situaciones entre el USUARIO – CLIENTE genera diferentes estructuras en cuanto al documento a emitir que podrían enumerarse algunos tales como Factura A – Factura B – Factura C – Factura de Crédito Electrónica entra otras, y cada una de estos comprobantes generan diferentes requerimientos a nivel código y restructuración de la base de datos.

Otra cuestión que se podría tener en consideración es que una operación al ser emitida no se puede eliminar debido a que el documento se generó, ante esta situación para anular dicha operación se debería incluir operaciones tales como NOTAS DE CREDITOS, NOTAS DE DEBITO entre otro orden de ideas.

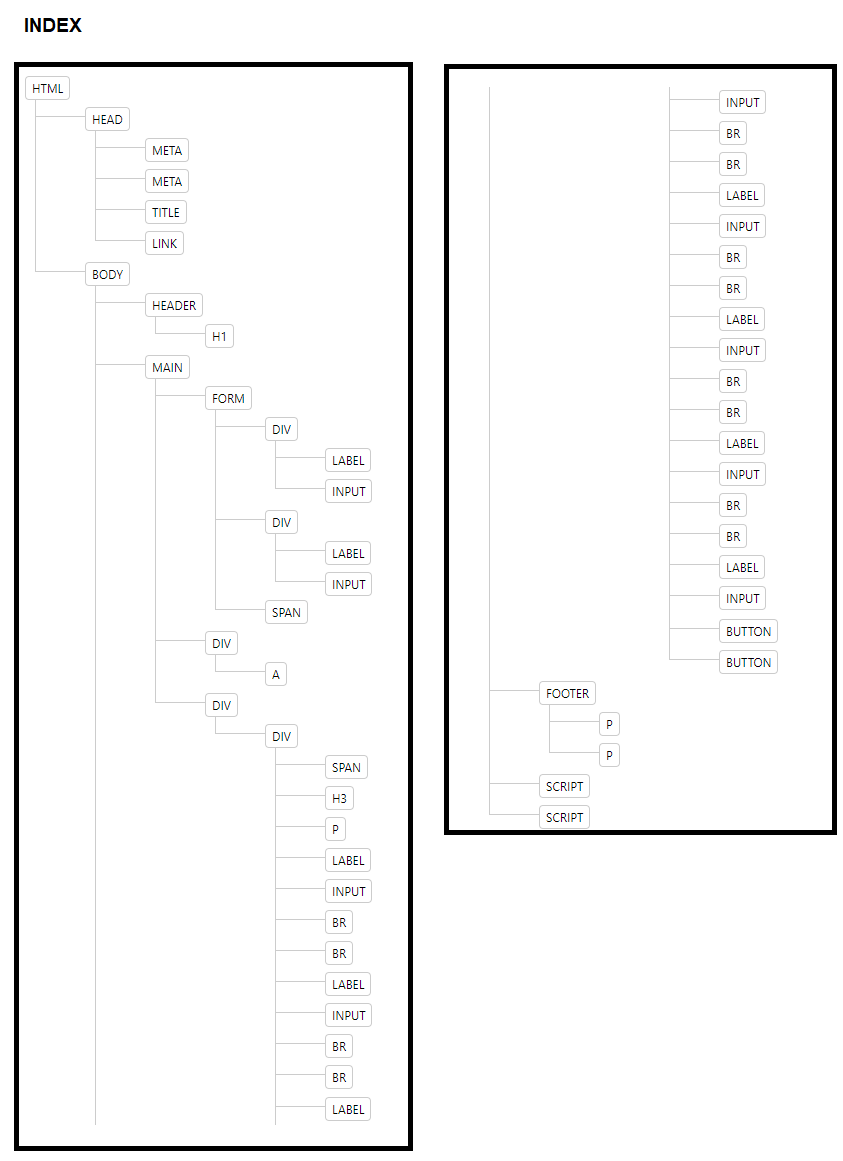
* 1. **ESTRUCTURA HTML – INDEX – DASHBOARD**

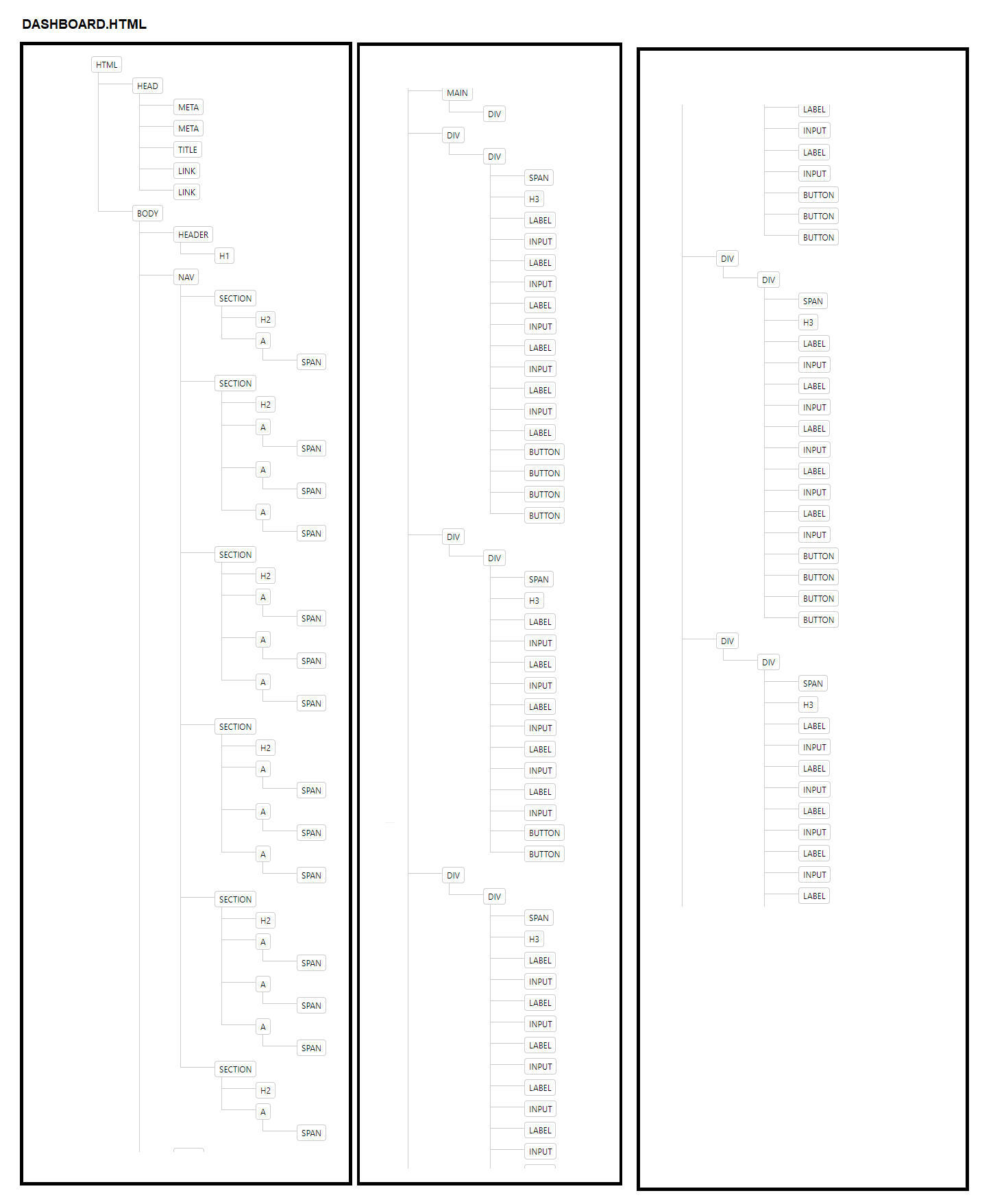
A continuación se presenta el esquema del HTML tanto del archivo INDEX.html como el DASHBOARD.HTML.

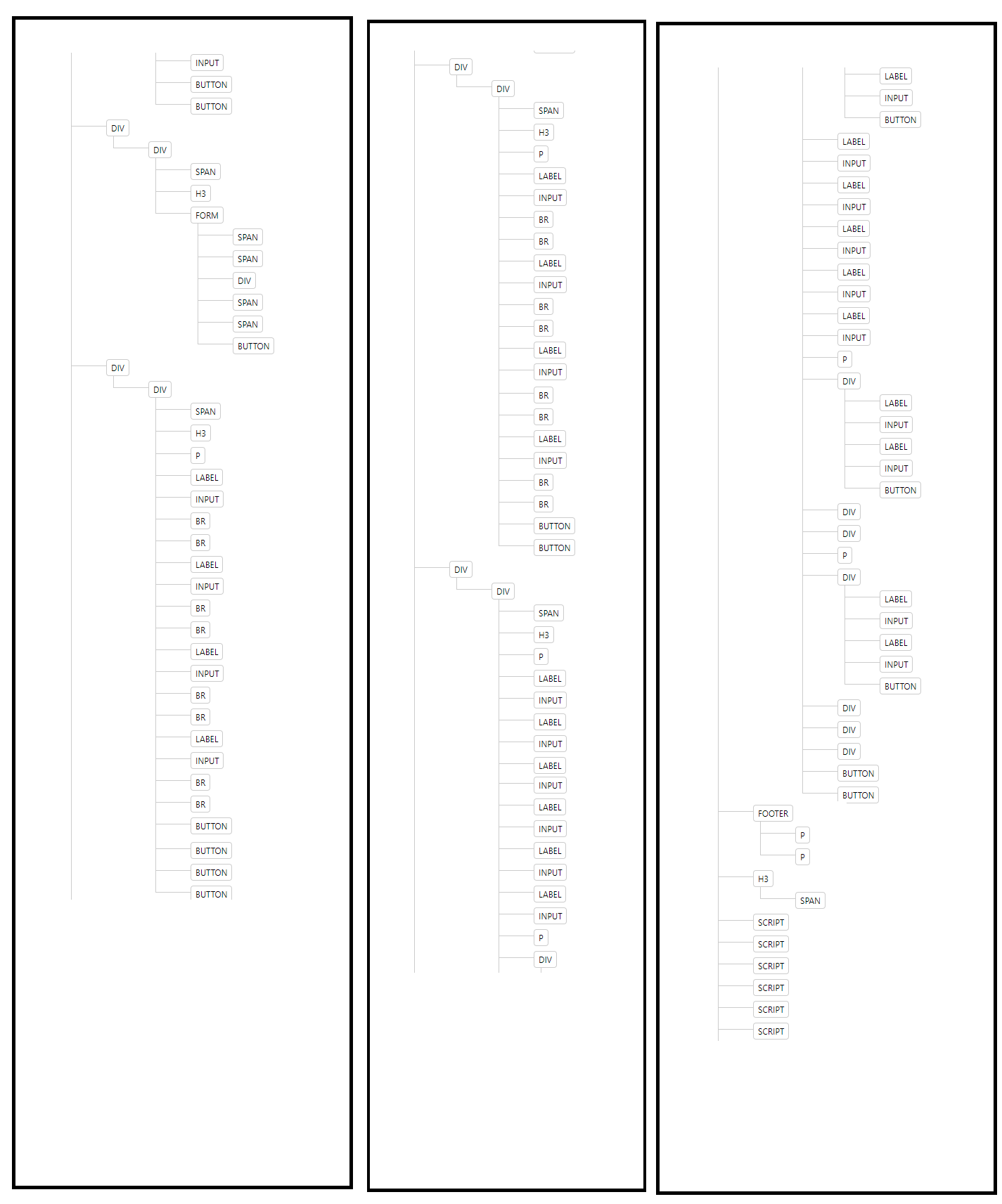
En el **index.html** tenemos dentro del BODY el HEADER, MAIN, FOOTER, SCREPT los que contendrán los elementos necesarios para obtener la información por parte del usuario para INGRESAR y para LOGEARSE con una nueva cuenta.

En el **dashboard.html** tenemos dentro del BODY el HEADER, NAV, MAIN, DIV, FOOTER, SCRIPT en dicha estructura se encontraran todos los elementos necesarios para obtener la información por parte del usuario para:

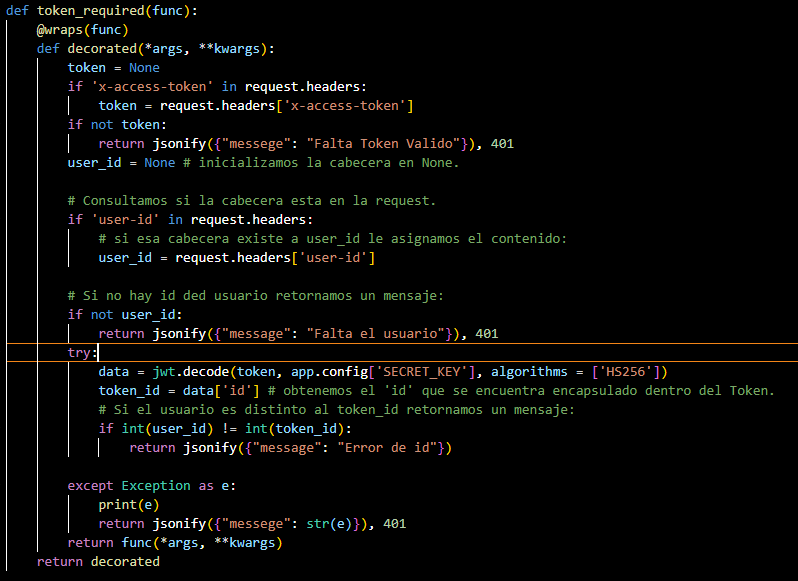
* Perfil – Editar Perfil
* Clientes – Nuevo Cliente / Lista / Ranking
* Productos – Nuevo Producto / Lista / Ranking
* Servicios – Nuevo Servicio / Lista / Ranking
* Facturación – Nueva Factura / Lista / Ranking
* Sesión - Salir

****

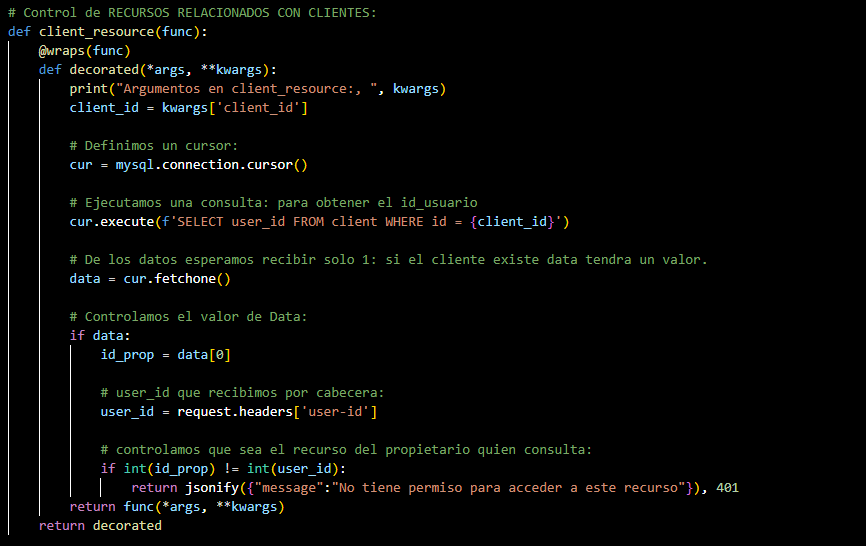
****

****

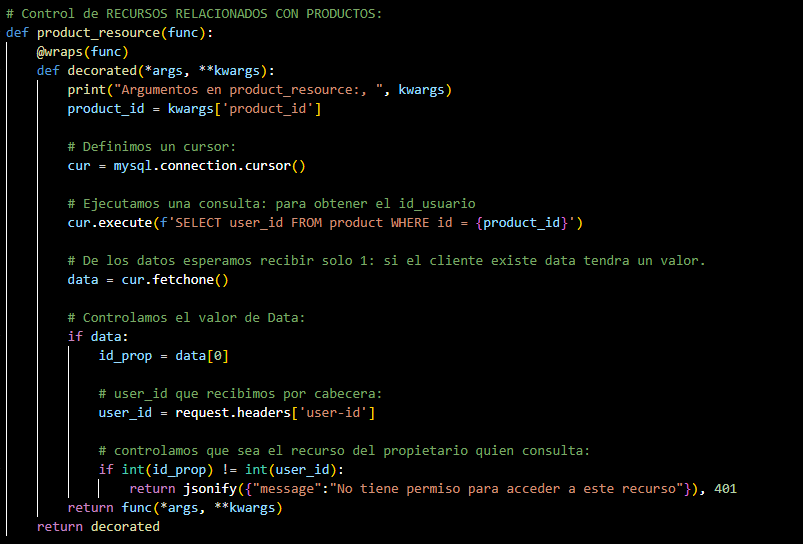
* 1. **NIVEL DE SEGURIDAD Y CONTROLES APLICADOS**



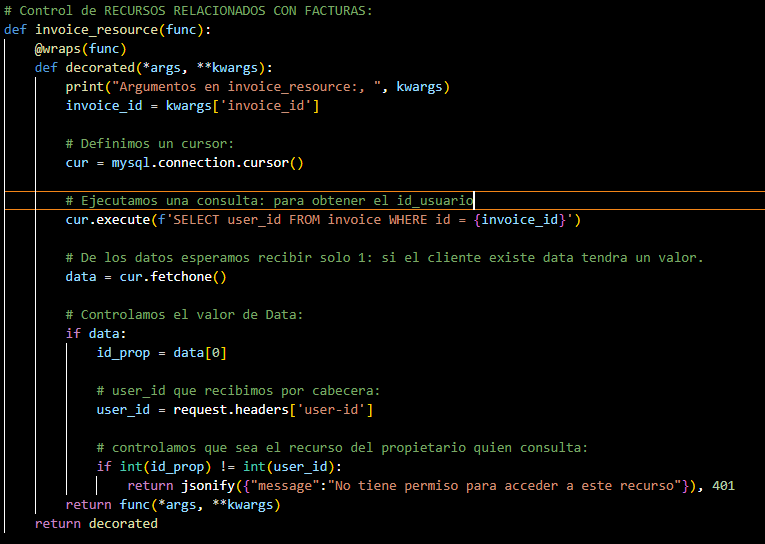
**TOKEN\_REQIRED**: es la función que permitirá realizar los controles de existencia de TOKEN y que el mismo se encuentre vigente y no vencido, también se realiza el control de que pertenezca a un usuario siendo un segundo control de seguridad. Controlar existencia de TOKEN y existencia de USUARIO. El decorador @wraps nos permite luego hacer la llamada desde las diferentes rutas para que haga un control previo y continuamente para cada ruta/consulta que se quiere realizar por parte del FrontEnd.



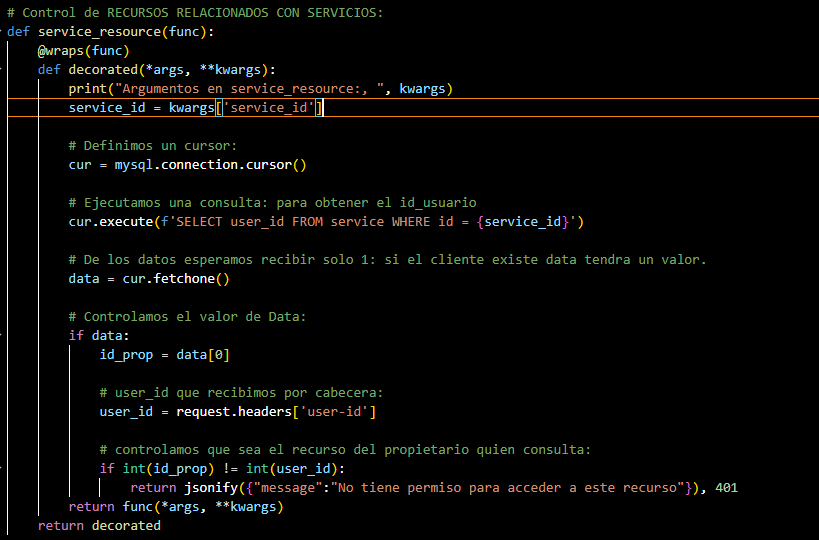
**CLIENT\_RESOURCE**: esta función nos permitirá controla que el cliente consultado por el frontEnd sea un cliente propiedad del usuario que lo solicita por la ruta enviada desde el frontEnd. Sería un control de propiedad.



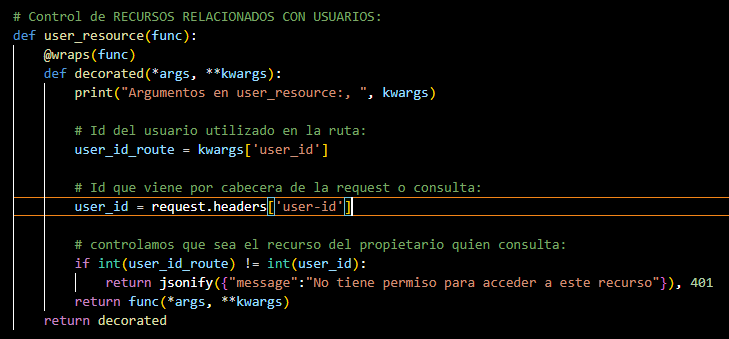
**PRODUCT\_RESOURCE**: esta función nos permitirá controlar que un determinado producto pertenece o es propiedad del usuario que consulta a través de la ruta, por parte del frontEnd. Control de propiedad.



**INVOICE\_RESOURCE**: esta función nos permitirá consulta que una factura es propiedad o fue emitida por un determinado usuario. Control de propiedad.

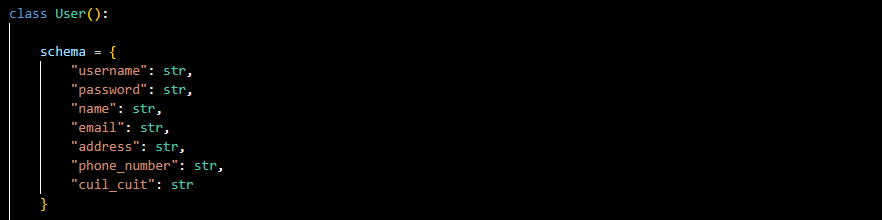


**SERVICE\_RESOURCE**: esta función permitirá controlar que el servicio que se consulta es propiedad del usuario que se encuentra realizando la consulta a través de la ruta desde el frontEnd. Control de propiedad.

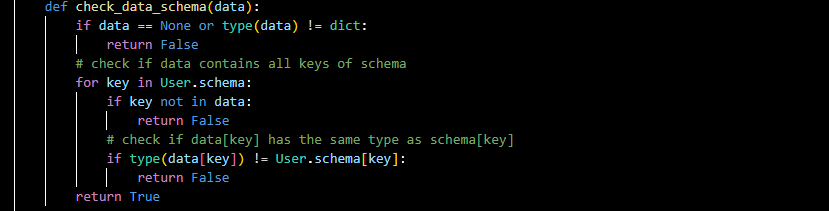


**USER\_RESOURCE**: esta función nos permite controlar que el id de la ruta no sea diferente al de la cabecera de la consulta realizada por el frontEnd. Control de consistencia.

* 1. **ESTRUCTURA DE USUARIO**
     1. **MODELS --> USUARIO**



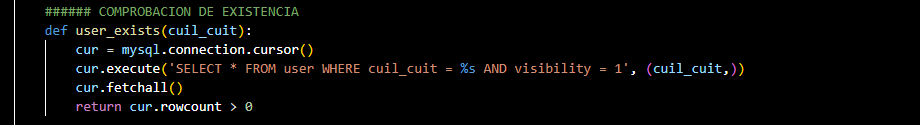
Se define la estructura de la tabla que deberá tener una analogía con la estructura de la base de datos, y definición del tipo de datos aceptados por la misma.



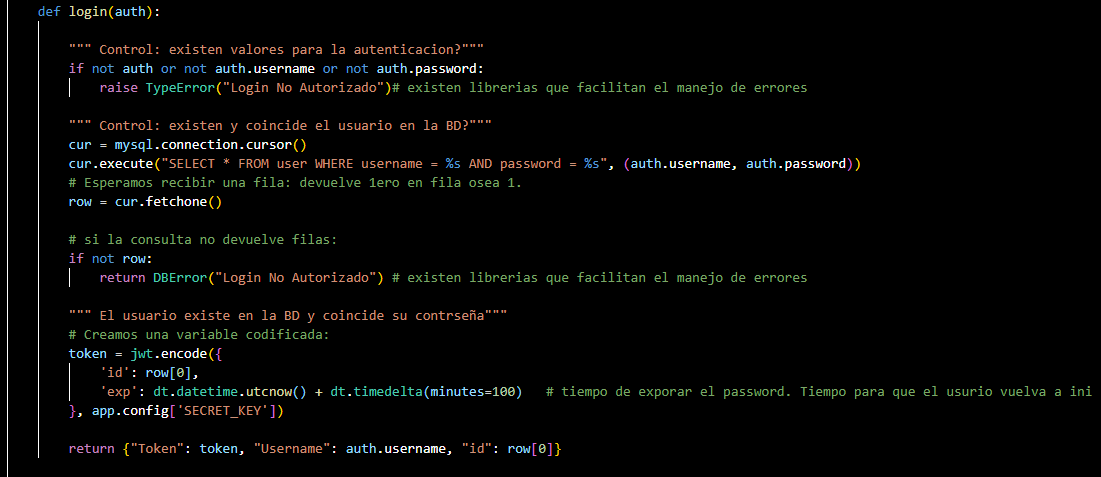
Este método permitirá controlar que la estructura recibida como parámetros corresponde a una estructura en la cual los tipos de datos coinciden correctamente con los requeridos por la clase User.



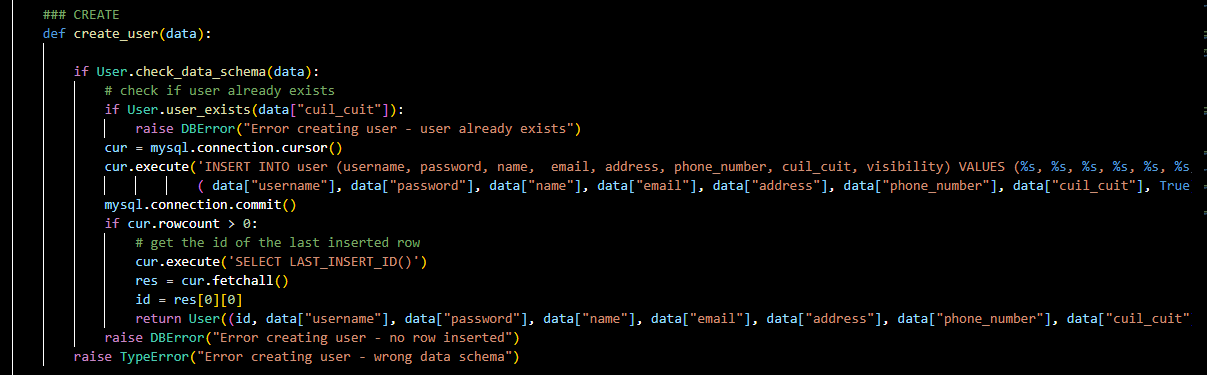
El constructor será una fila recibida de una determinada consulta enviada la cual ingresara por la RUTA respectiva la cual es enviada por el frontEnd. Y también se expone en la imagen como será la estructura que retornara la clase User en formato JSON que será la respuesta a la consulta realizada por el usuario desde el frontEnd.



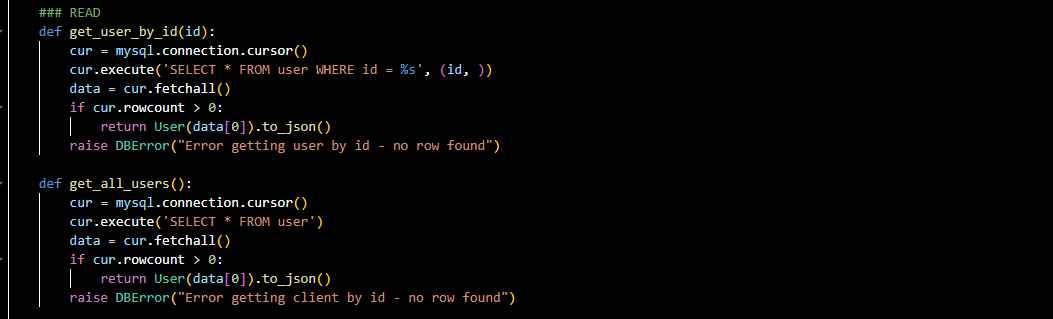
El método USER\_EXIST permitirá consultar en la base de datos si el cuil\_cuit exite en la misma.



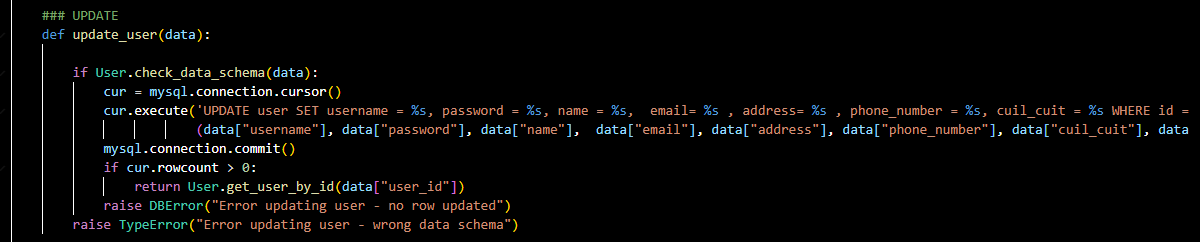
Este método permitirá crear un TOKEN para asignarle al usuario que se logea con numbre de usuario y password correctos según la base de datos, el token tendrá un tiempo de vida, vencido el mismo se deberá logearse nuevamente.



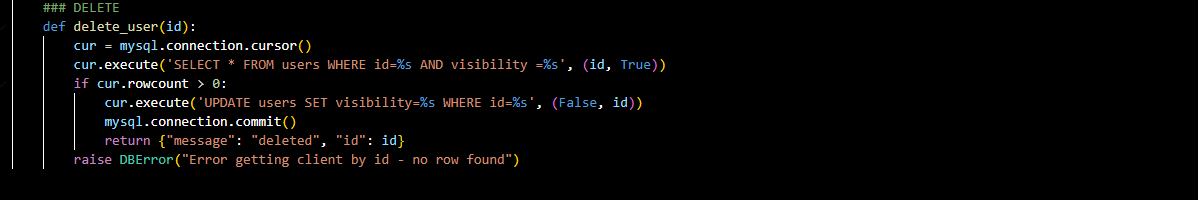
Este método permitirá crear un nuevo usuario, si la información recibida tiene la estructura y tipos de datos necesarios por la base de datos, y el usuario no existe según el cuil\_cuit en la base de datos se procede a enviar la información de creación del usuario a la base de datos.



Estos métodos permitirán obtener la información del usuario según su ID y también consultar todos los usuarios de la base de datos.



Este método permitirá modificar los datos de un determinado usuario, enviados por el frontEnd siempre que la estructura de la información y tipos de datos sea el correcto.



Este método permite identificar un usuario activo y si es superior a 0 por que se encuentra activo definirlo como desactivado para que el usuario ya no tenga visualización del mismo, concepto de borrado lógico.

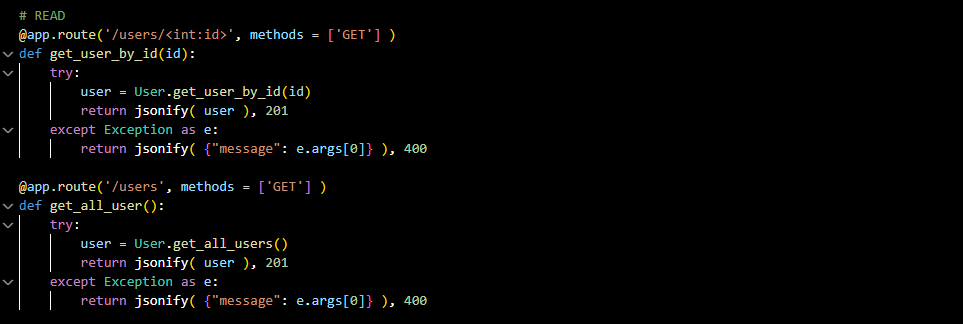
* + 1. **ROUTES --> USUARIOS**



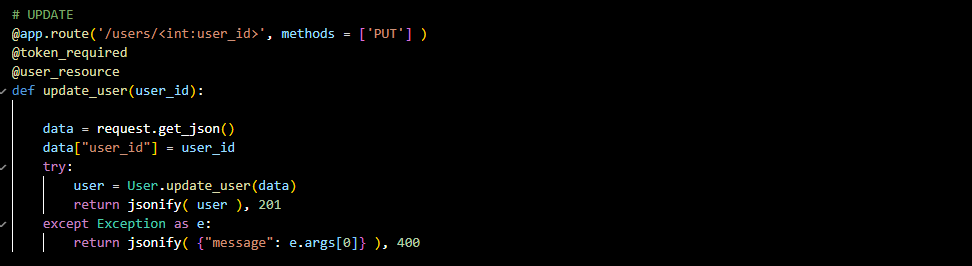
Esta ruta permite que el frontEnd envie un objeto auth, que contendrá la información de nombre de usuario y password los cuales serán validadas contra la base de datos, luego generara un token con un tiempo vida que permitirá al usuario mantenerse en el dashboard.



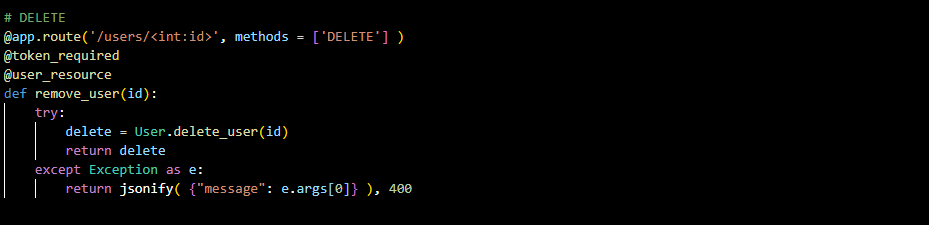
Esta ruta permitirá enviar los datos obtenidos del frontEnd al backend para crear un nuevo usuario en la base de datos.



Estos métodos permitirán consultar la información de un determinado usuario según su ID y realizar una consulta de todos los usuarios de la base de datos.

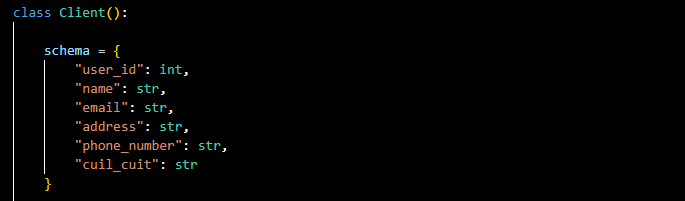


Esta ruta permitirá modificar los datos de un usuario según su ID, previo a su ejecución se realizaran controles como **@token\_required:** control de existencia y vigencia de token, **@user\_resource:** control de existencia de usuario en la base de datos.

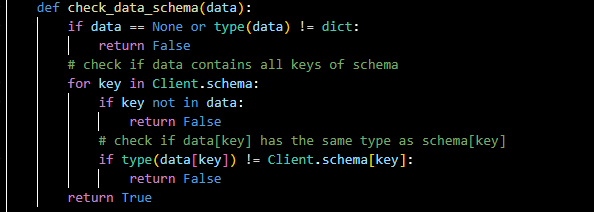


Esta ruta permitirá ejecutar el método para remover un usuario según su Id, previo a ello se realizaran los controles tales como **@token\_required**: existencia y vigencia del token, **@user\_resource:** existencia del usuario en la base de datos.

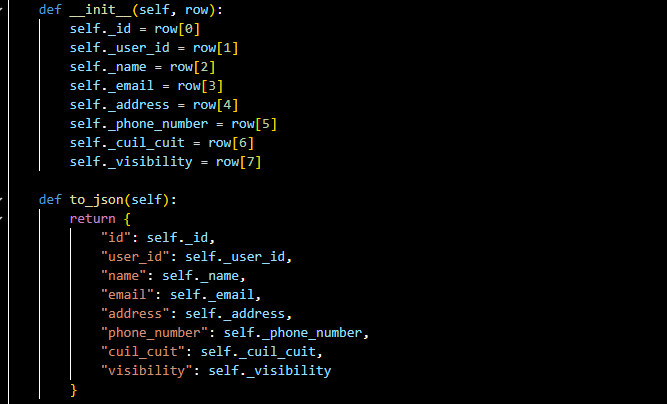
* 1. **ESTRUCTURA DE CLIENT**
     1. **MODEL --> CLIENT**



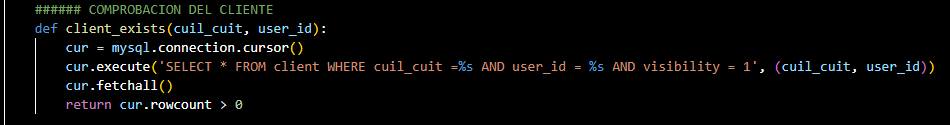
En la clase CLIENT definimos la estructura que va a tener la tabla, la cual deberá coincidir con las base de datos para que exista consistencia, y los datos deberán ser los soportados por la misma.



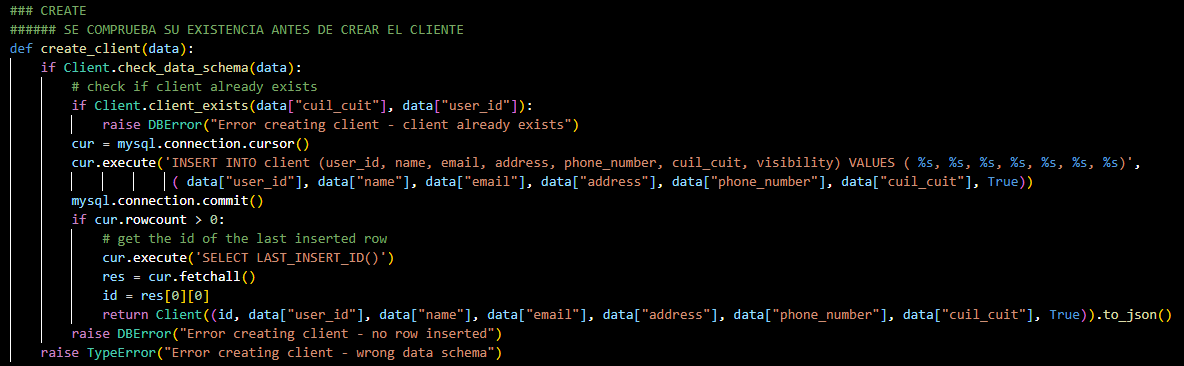
Este método permitirá controlar que la estructura recibida como parámetros corresponde a una estructura en la cual los tipos de datos coinciden correctamente con los requeridos por la clase Client.



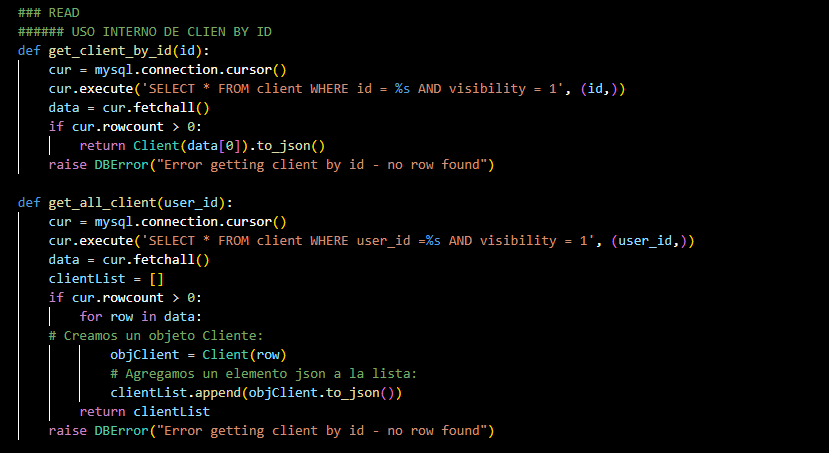
El constructor será una fila recibida de una determinada consulta enviada la cual ingresara por la RUTA respectiva la cual es enviada por el frontEnd. Y también se expone en la imagen como será la estructura que retornara la clase client en formato JSON que será la respuesta a la consulta realizada por el usuario desde el frontEnd.



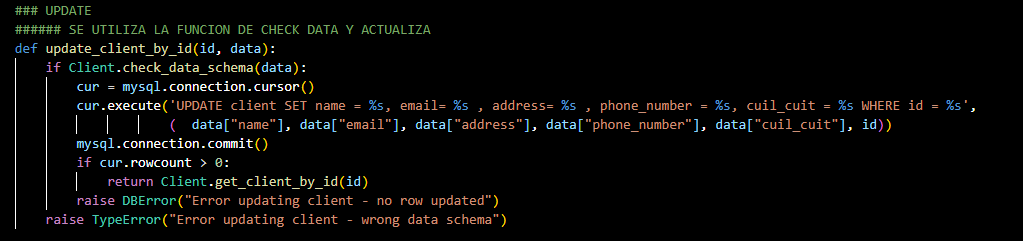
El método CLIENT\_EXIST permitirá consultar en la base de datos si existe el cuil\_cuit para el usuario particular.



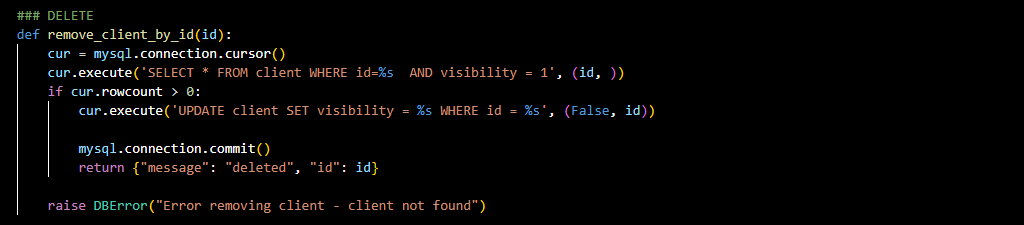
Este método permitirá junto con los anteriores controlar que el cliente no exista antes de realizar la creación y que los datos de la tabla sean válidos según su tipo, según los enviados por el usuario desde el frontEnd, de no existir inconvenientes se procederá a crear el cliente con toda la información obtenida desde el frontEnd enviado por el usuario.



Estos métodos se relacionan con la consulta en la base de datos de un determinado cliente en función de su ID y la consulta de todos los clientes en función de un usuario en particular, retornando el Json correspondiente a la consulta realizada.

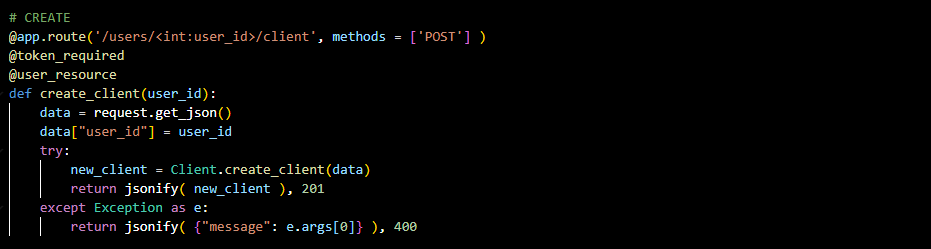


Este método permitirá modificar los datos de un determinado cliente en función de ID, controlada la estructura de la información se realiza la modificación en la base de dato pasándoles todos los parámetros relacionados a dicha actualización de la información.

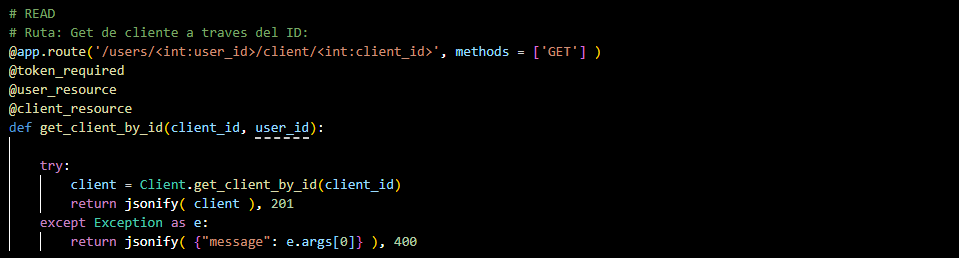


Este método es el que nos permitirá realizar un borrado lógico a un determinado cliente según el ID, posicionando su visibilidad en desactivada para el usuario pero persistiendo en la base de dato.

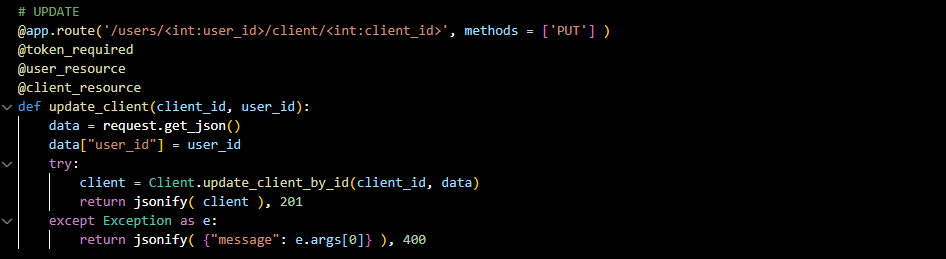
* + 1. **ROUTS --> CLIENT**



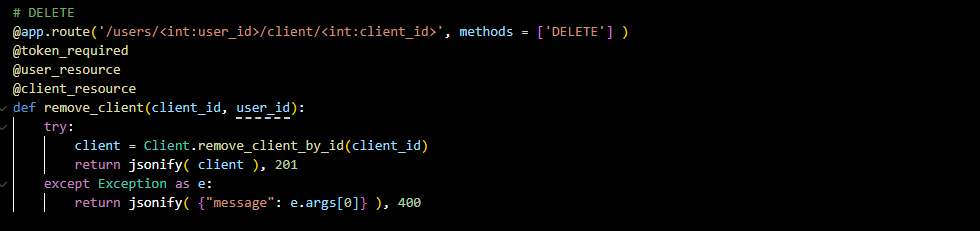
Esta ruta permite conectar la solicitud enviada por el frontEnd permitendo de esta manera que la capte el backend, previo a realizar ciertos controles: **@token\_required:** existencia de token vigente y valido. **@user\_resource:** existencia del usuario solicitante dentro de la base de datos como tal. Luego de los controles esta ruta/url, permitirá pasar la información captada **data** desde el frontEnd para procesarla y enviarla a la base de datos como **new=client**.



Esta ruta permite consultar desde el frontend la informacion de un determinado cliente, previo a su ejecución se realizan diferentes controles tales como **@token\_required**: existencia y validez del token, **@user\_resoruce**: existencia del usuario que consulta en la base de datos, **@cliente\_resource**: si el cliente consultado es propiedad del usuario que consulta. Va a retornar de cumplirse con todos los controles previos, la información del cliente consultado.



Este ruta permitirá ejecutar el método para modificar los datos de un determinado cliente según su ID de un determinado usuario, previo a su ejecución se realizaran los controles de **@token\_required:** existencia y validez de token, **@user\_resource**: existencia del usuario que realiza la solicitud, **@cliente\_resource**: si el cliente a modificar pertenece o es propiedad del usuario que solicita su modificaicon.



Este ruta permitirá eliminar, (aplicando el concepto de borrado lógico) un determinado cliente según su ID de un determinado usuario, previo a su ejecución se realizan ciertos controles tales como: **@token\_required**: existencia y validez de token, **@user\_resource:** existencia del usuario que solicita tal operación exista en la base de datos, **@client\_resource**: controla que el cliente que se solicita su eliminación sea propiedad del usuario que solicita dicha operación.