**UNIVERSIDAD TECNOLÓGICA NACIONAL REGIONAL LA PLATA**

**DESARROLLO DE SOFTWARE**

**Trabajo Práctico Integrador - Compendio**

**Comisión:** S31

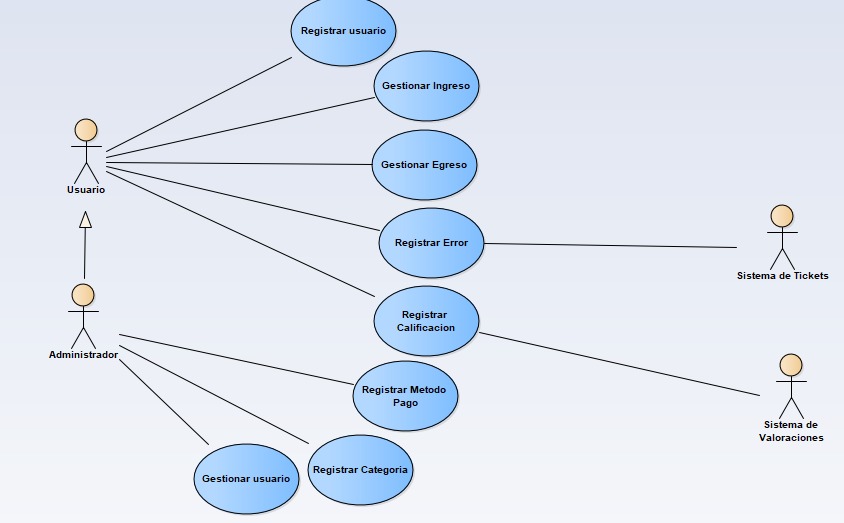
**Año**: 2024

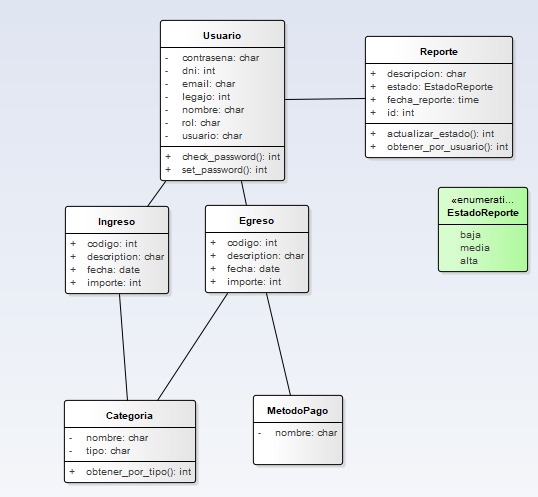
**TEMA**

Sistema de Gestión Financiera (STATS)

**OBJETIVO DEL TRABAJO**

Este proyecto es un sistema de gestión financiera que permite a los usuarios registrar y gestionar ingresos, egresos, métodos de pago y categorías. Además, cuenta con integración con un sistema externo de tickets para reportar y gestionar problemas.





**FRONTEND**

**Directory structure:**

**└── frontend/** Contiene todo el código fuente del frontend del proyecto.

**├── README.md**

**├── package-lock.json**

**├── package.json**

**├── .gitignore**

**├── public/** Archivos estáticos y assets (imágenes, íconos).

**│ ├── index.html**

**│ ├── manifest.json**

**│ ├── robots.txt**

**│ └── assets/**

**│ └── images/**

**└── src/** Código principal: components, páginas, estilos y servicios.

**├── App.css**

**├── App.js**

**├── App.test.js**

**├── index.css**

**├── index.js**

**├── reportWebVitals.js**

**├── setupTests.js**

**├── components/** Componentes reutilizables

**│ ├── Footer.js**

**│ ├── Layout.js**

**│ ├── Navbar.js**

**│ └── proteccion.js**

**├── pages/** Páginas del sitio

**│ ├── Admin.js**

**│ ├── Calificacion.js**

**│ ├── Dashboard.js**

**│ ├── Egreso.js**

**│ ├── Historial.js**

**│ ├── Ingreso.js**

**│ ├── Landing.js**

**│ ├── Login.js**

**│ ├── Register.js**

**│ └── Report.js**

**├── services/** Archivos para gestionar llamadas a la API y autenticación

**│ ├── api.js**

**│ └── auth.js**

**└── styles/** Hojas de estilo CSS organizadas por funcionalidad/sección.

**├── Layout.css**

**├── admin.css**

**├── auth.css**

**├── dashboard.css**

**├── feedback.css**

**├── footer.css**

**├── global.css**

**├── historial.css**

**├── landing.css**

**├── navbar.css**

**└── transactions.css**

**App.js**

useState se usa para gestionar el estado de autenticación (auth), que indica si un usuario está logueado o no.

Este estado se actualiza en el componente Login a través de setAuth, lo que permite cambiar su valor cuando el usuario se autentica correctamente.

Luego, tenemos las diferentes rutas para navegar por los componentes de la app. Las rutas públicas incluyen Landing, Login y Register, que muestran Navbar y

Footer.

Las rutas protegidas están envueltas en ProtectedRoute, asegurando que solo los usuarios autenticados puedan acceder a Dashboard, Ingreso, Egreso, Historial, Report y Calificacion, con Layout como contenedor.

Por último, la ruta de administración (/admin) usa ProtectedRoute con

adminOnly={true}, permitiendo el acceso solo a administradores



**components/**

**Layout.js**

useEffect obtiene el token almacenado en localStorage y lo decodifica con jwtDecode para determinar si el usuario tiene el rol de administrador, actualizando isAdmin en consecuencia.

Luego, tenemos las diferentes rutas dentro de la barra lateral para navegar por los componentes de la app.

* **Rutas generales**: Dashboard, Ingreso, Egreso e Historial están disponibles para todos los usuarios autenticados.
* **Ruta de administración:** Se muestra solo si isAdmin es true, permitiendo el acceso a la sección de administración.
* **Acciones adicionales:** Los botones permiten calificar el servicio (/calificar), reportar un problema (/report) y cerrar sesión (handleLogout), que borra el token, actualiza auth a false y redirige al usuario a la página principal.

El Layout organiza la estructura de la aplicación, integrando la Sidebar y el contenido dinámico de cada ruta dentro de Outlet(marcador de posición que React Router usa para renderizar el componente)



**proteccion.js**

ProtectedRoute verifica si el usuario tiene un token almacenado en localStorage.

* Si no hay token, redirige automáticamente a la página de login (/login).
* Si el token es válido, lo decodifica con jwtDecode para obtener la información del usuario, como su rol.
* Si adminOnly es true, verifica si el usuario es administrador. Si no lo es, lo redirige a dashboard, evitando que acceda a rutas restringidas.
* Si ocurre un error al decodificar el token, se muestra un mensaje en consola y se redirige al usuario al login.

Si todas las validaciones son correctas, el componente renderiza children, permitiendo el acceso a la ruta protegida.



**pages/**

**Admin.js** Gestionar usuarios, categorías y métodos de pago.

useEffect se encarga de **cargar los datos iniciales de los usuarios** para mostrar la lista en el panel de administración, con fetchAdminData() obtiene el token de autenticación desde localStorage y recupera la lista de usuarios. Si hay un error, se muestra un mensaje al usuario.

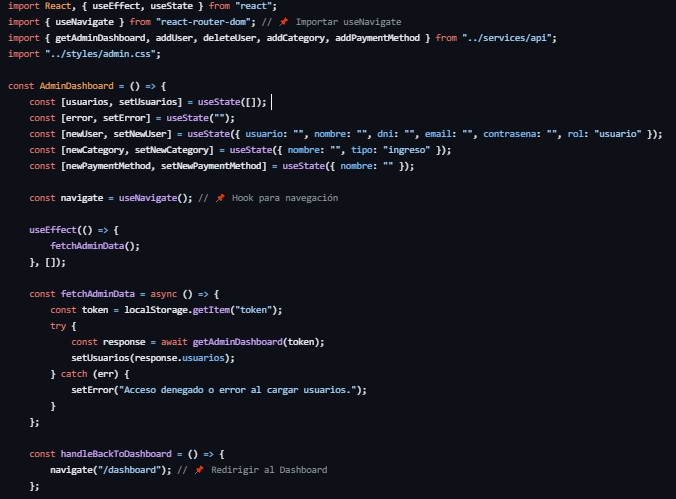
Para la **gestión de usuarios**, se pueden agregar nuevos mediante un formulario con usuario, nombre, DNI, email, contraseña y rol.

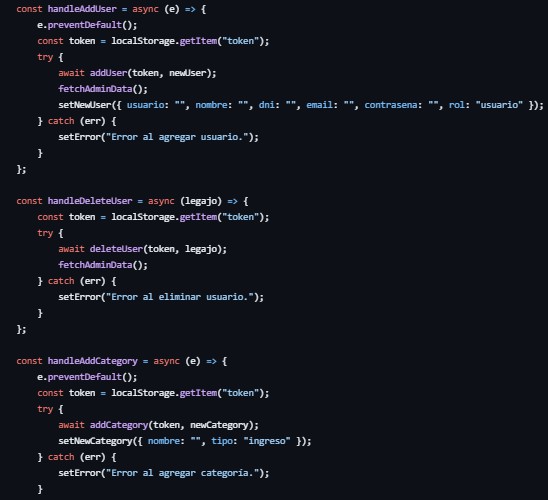
handleAddUser() envía los datos y actualiza la lista, mientras que

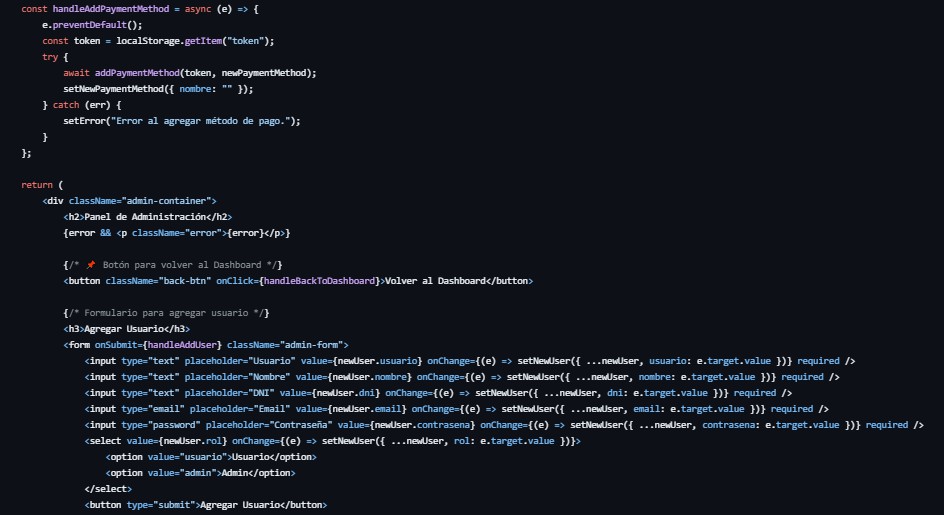
handleDeleteUser() elimina un usuario y recarga la información.

Las funciones addCategory(token, newCategory) y addPaymentMethod(token, newPaymentMethod) envían datos al backend para agregar una nueva **categoría** o un nuevo **método de pago**, respectivamente. Ambas funciones realizan una **petición HTTP (POST)** a una API, incluyendo el token de autenticación en los encabezados y los datos correspondientes (newCategory o newPaymentMethod) en el cuerpo de la solicitud. Si la petición es exitosa, la nueva categoría o método de pago se agrega a la base de datos. Si falla, se captura el error y se muestra un mensaje en la interfaz.

La **navegación** se maneja con useNavigate(), permitiendo volver al Dashboard con un botón.

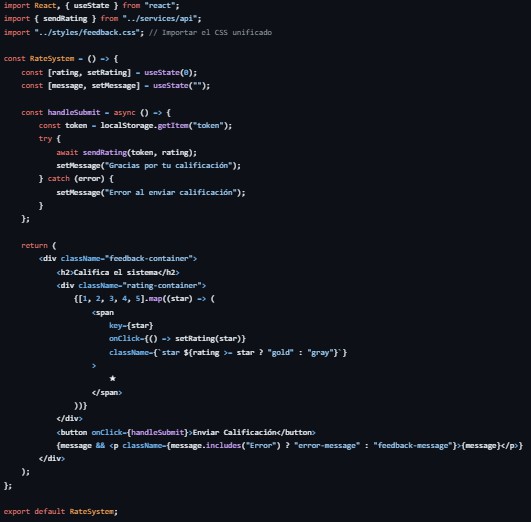








**Calificacion.js**

Este componente permite a los usuarios calificar el sistema con una puntuación de 1 a 5 estrellas. Al hacer click en una estrella, se actualiza el estado rating. Al enviar la calificación, se realiza una solicitud a la API utilizando el token de autenticación guardado en el localStorage. Dependiendo de la respuesta, se muestra un mensaje de éxito o error. El componente también maneja el estado visual de las estrellas, cambiando su color según la calificación seleccionada, y muestra el mensaje correspondiente debajo del botón de envío.

**Dashboard.js**

Dashboard se encarga de mostrar estadísticas financieras mediante gráficos interactivos.

dashboardData: Almacena los datos sobre ingresos y egresos obtenidos del servidor.

error: Muestra un mensaje de error en caso de que falle la carga de los datos.

user: Contiene el nombre del usuario autenticado.

fechaSeleccionada: Permite al usuario filtrar los datos de acuerdo a la fecha seleccionada.

**useEffect**: Este hook se ejecuta cada vez que cambia fechaSeleccionada, lo que provoca que se ejecute la función fetchData. En esta función:

* Se obtiene el token de autenticación desde localStorage.
* Luego, se consulta la API para obtener la información del usuario y los datos del dashboard, que incluyen los ingresos y egresos por categoría.

Si se reciben los datos correctamente, se muestran a través de gráficos de barras y

pastel (Bar y Pie de chart.js) representando los ingresos y egresos por categoría, además de una comparación general entre los ingresos y egresos totales.

Incluye un selector de fecha para que el usuario pueda filtrar los datos. Cuando se selecciona una nueva fecha, los gráficos se actualizan automáticamente.

**services/** Archivos para gestionar llamadas a la API y autenticación

**api.js**

interacción con una API backend

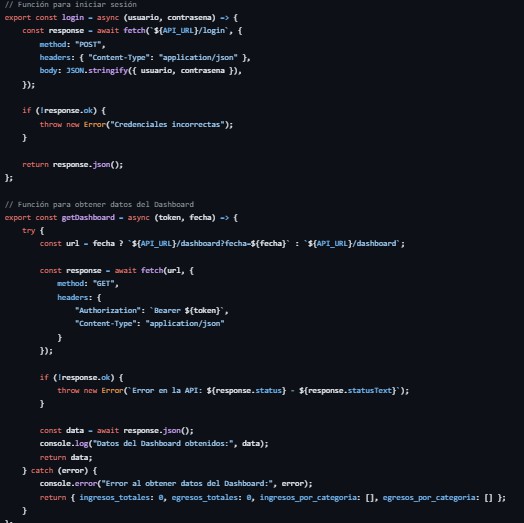
Las funciones **register** y **login** envían solicitudes POST a la API para registrar un nuevo usuario y autenticar a un usuario existente. El **login** devuelve un token de autorización que se utiliza para validar solicitudes.

**getDashboard** y **getHistory** envían solicitudes GET a la API, solicitando datos relacionados con el panel de control del usuario y su historial. Estas funciones pasan el token de autorización en los encabezados de la solicitud para acceder a los datos protegidos.

**addIncome** y **addExpense** permiten al usuario registrar ingresos y egresos en la API, enviando los detalles correspondientes.

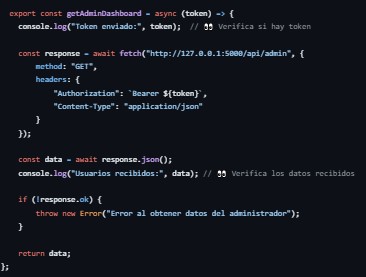
**addUser** y **deleteUser** permiten a los administradores gestionar usuarios, realizando solicitudes POST y DELETE a la API. El token de autorización es necesario para usuarios con permisos.

**getPaymentMethods** obtiene una lista de métodos de pago disponibles a través de la API, mientras que **sendRating** permite al usuario enviar calificaciones para servicios, enviando los datos correspondientes en el cuerpo de la solicitud.









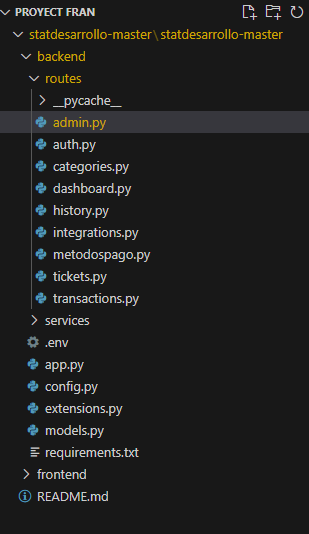
**auth.js**

getUser toma un token JWT como argumento, con información sobre el usuario. Utiliza la librería jwt-decode para decodificar el token y extraer sus datos. Conviertiendo el token en un objeto JavaScript.

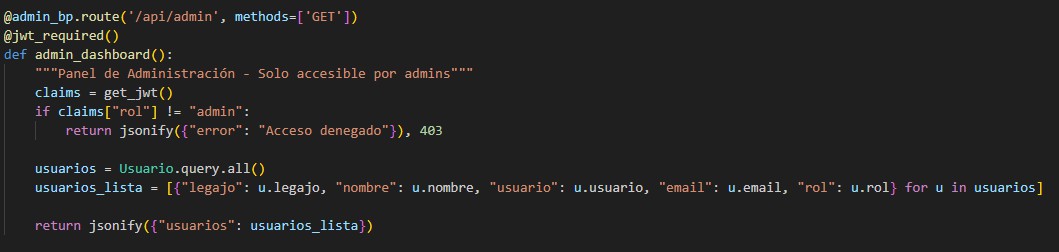
Se extrae el valor de usuario del objeto decodificado y se devuelve en un nuevo objeto

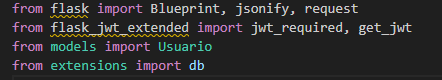
{ usuario: decoded.usuario }.



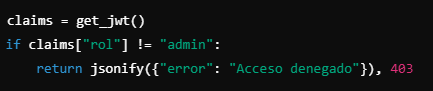


Dentro de routes/**admin.py** tenemos lo siguiente:

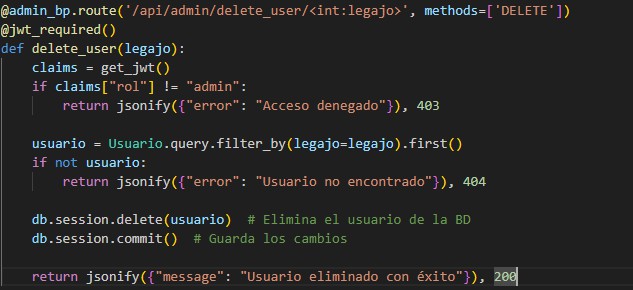
* Importaciones varias



* Este fragmento de código define una ruta en Flask para un **Panel de Administración** que solo pueden acceder usuarios con rol de administrado
* Declara una ruta para **/api/admin** que solo responde a solicitudes **GET**.
* **@jwt\_required()** Esta línea asegura que solo usuarios autenticados pueden acceder a esta ruta. Requiere que el cliente envíe un JWT (JSON Web Token) válido.

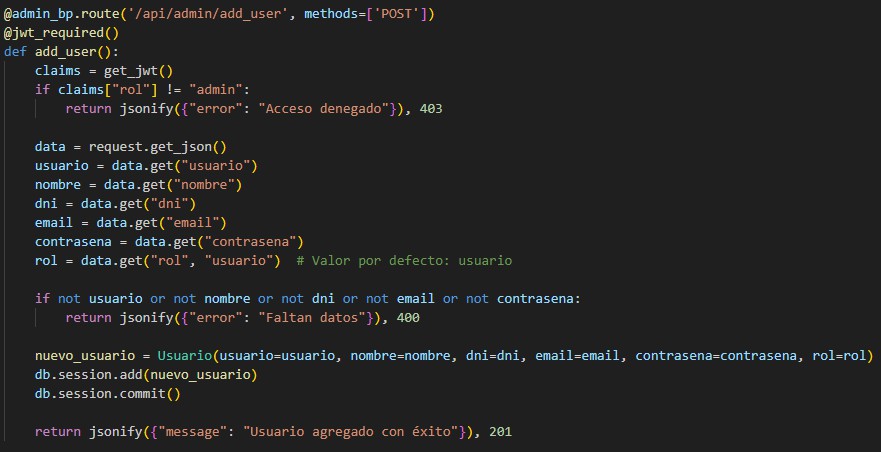


* 1. Obtiene las **reclamaciones** (claims) del JWT, que contienen información sobre el usuario, como su rol.
  2. Verifica si el rol es diferente de "admin".
  3. Si el rol **no** es "admin", retorna un **error 403 (Acceso denegado)**.



Este fragmento de código define una ruta en Flask para **eliminar un usuario** mediante su legajo, accesible solo para administradores.

* Utiliza SQLAlchemy para obtener **todos los registros** de la tabla Usuario.
* Retorna un **JSON** con la lista de usuarios.
* Elimina el usuario de la base de datos y luego guarda los cambios

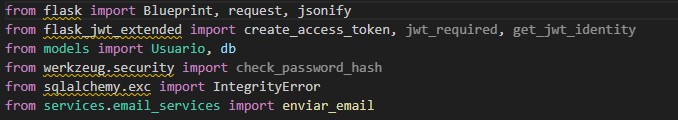
Este fragmento de código define una ruta en Flask para **agregar un nuevo usuario** en el sistema, accesible solo para administradores

* Obtiene los datos del **JSON** enviado en la solicitud.
* Extrae los campos necesarios



* Verifica que **todos** los campos obligatorios estén presentes.
* Si falta alguno, retorna un **error 400 (Bad Request)**.
* Crea una **instancia** de Usuario con los datos proporcionados.
* **Agrega** el nuevo usuario a la sesión de la base de datos.
* **Confirma** los cambios con commit() para guardar el usuario.

Dentro de routes/**auth.py** tenemos lo siguiente:



* Importaciones de librerías varias con adicional de servicios de email y hashes de contraseña



* Blueprint es una forma de **modularizar** y organizar rutas en Flask, permitiendo dividir la aplicación en **componentes reutilizables** y **mantenibles**.
* En este caso, se llama auth\_bp, lo que dice que se utiliza para gestionar **funcionalidades de autenticación**, como iniciar sesión, registrarse o cerrar sesión.



return jsonify({"message": "Usuario registrado exitosamente y correo

enviado"}), 201

except IntegrityError: db.session.rollback()

return jsonify({"error": "Usuario, DNI o Email ya registrado"}), 400









enviar\_email(data['email'], asunto, mensaje)







Stat."

asunto = "Registro exitoso en Finanzas-Stat"

mensaje = f"Hola {data['nombre']}, tu registro fue exitoso en Finanzas-

Enviar correo de bienvenida

#

try:

db.session.add(nuevo\_usuario) db.session.commit()

)

nuevo\_usuario.set\_password(data['contrasena'])

rol=data.get('rol','usuario')

Nuevo campo

nuevo\_usuario = Usuario(

usuario=data['usuario'], nombre=data['nombre'], dni=data['dni'],

email=data['email'], #

if not all([data.get('usuario'), data.get('nombre'), data.get('dni'),

data.get('contrasena'), data.get('email')]):

return jsonify({"error": "Todos los campos son obligatorios"}), 400





























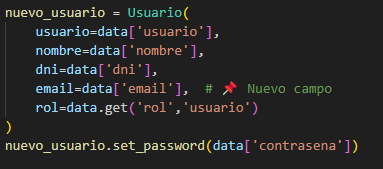




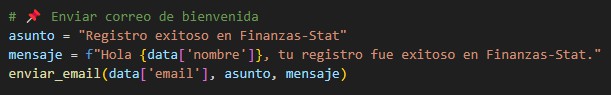




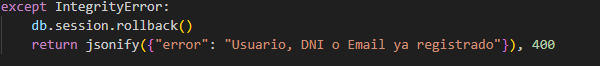
* @auth\_bp.route('/api/register', methods=['POST'])
* def register():
* """Registro de usuarios con email"""
* data = request.get\_json()
* Este fragmento de código define una ruta en Flask para **registrar nuevos usuarios** utilizando su correo electrónico, con funcionalidad adicional para enviar un correo de bienvenida.

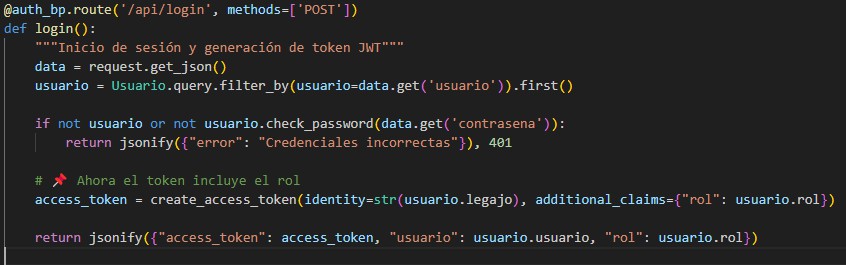


* Crea una **instancia** de Usuario con los datos proporcionados.
* **Hashea** la contraseña utilizando set\_password() antes de guardarla en la base de datos (esto es una buena práctica de seguridad).



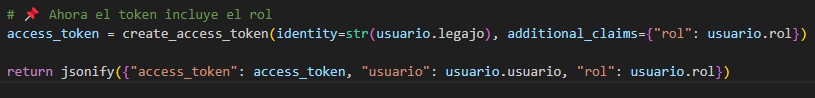
* enviar\_email() para enviar un **correo de bienvenida** al usuario recién registrado.
* Incluye un **asunto** y un **mensaje personalizado** con el nombre del usuario.



* Captura excepciones de **IntegrityError**, que pueden ocurrir si el nombre de usuario, DNI o email ya existen en la base de datos (probablemente debido a **restricciones de unicidad** en el modelo Usuario).
* Realiza un **rollback** para deshacer cualquier cambio en la base de datos.
* Retorna un **error 400 (Bad Request)** con un mensaje de error.

Este fragmento de código define una ruta en Flask para el **inicio de sesión** de usuarios, generando un

**token JWT** que incluye el rol del usuario.

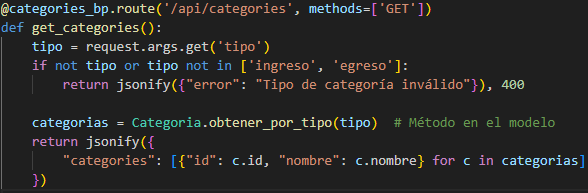
Solo responde a solicitudes **POST**, ya que se están enviando credenciales para autenticación.

Genera un **token de acceso JWT** utilizando la función create\_access\_token() de Flask-JWT-Extended. Incluye información adicional en el token:

* identity: El **legajo** del usuario, convertido a str para que sea serializable en el token.
* additional\_claims: Un **diccionario** con el **rol** del usuario. Esto permite definir permisos en otras rutas según el rol
* Retorna un **JSON** con:
* access\_token: El **token JWT** generado.
* usuario: El **nombre de usuario**.
* rol: El **rol** del usuario (por ejemplo, "admin" o "usuario").

Esto permite que el frontend guarde el token y el rol para gestionar la **autorización** en la interfaz.

En routes/**categories**.**py**:



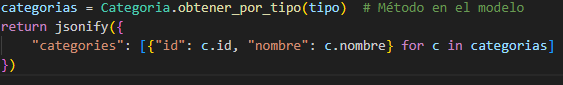
Este fragmento de código define una ruta en Flask para **obtener categorías** según su tipo (ingreso o egreso)



Verifica dos cosas:

* Que el parámetro tipo **exista**.
* Que el valor de tipo sea **válido**, es decir, solo puede ser ingreso o egreso.

Si no se cumple alguna de estas condiciones, retorna un **error 400 (Bad Request)** con un mensaje de error.



* Llama al método obtener\_por\_tipo() del modelo Categoria para obtener **todas las categorías**

que correspondan al tipo especificado.

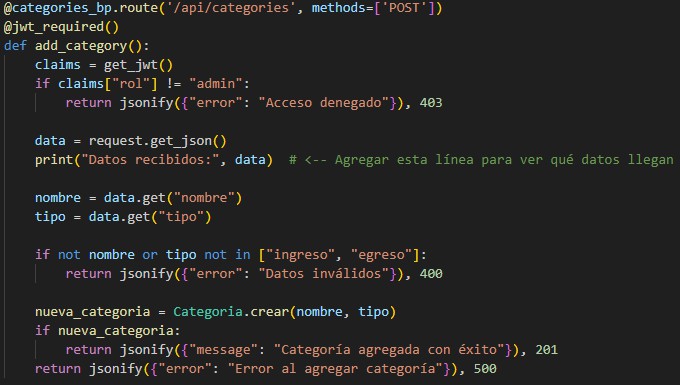
* Este método se espera que haga una **consulta a la base de datos** para obtener las categorías filtradas por tipo.
* Crea un **JSON** con una lista de categorías, donde cada categoría incluye
* id: El identificador de la categoría.
* nombre: El nombre de la categoría.



Este fragmento de código define una ruta en Flask para manejar solicitudes **OPTIONS** en la ruta de categorías.

# ¿Qué es una Solicitud OPTIONS?

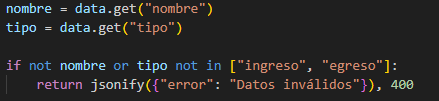
* Una solicitud **OPTIONS** es enviada automáticamente por los navegadores **antes** de realizar una solicitud HTTP (como GET, POST, DELETE, etc.) a un dominio diferente.



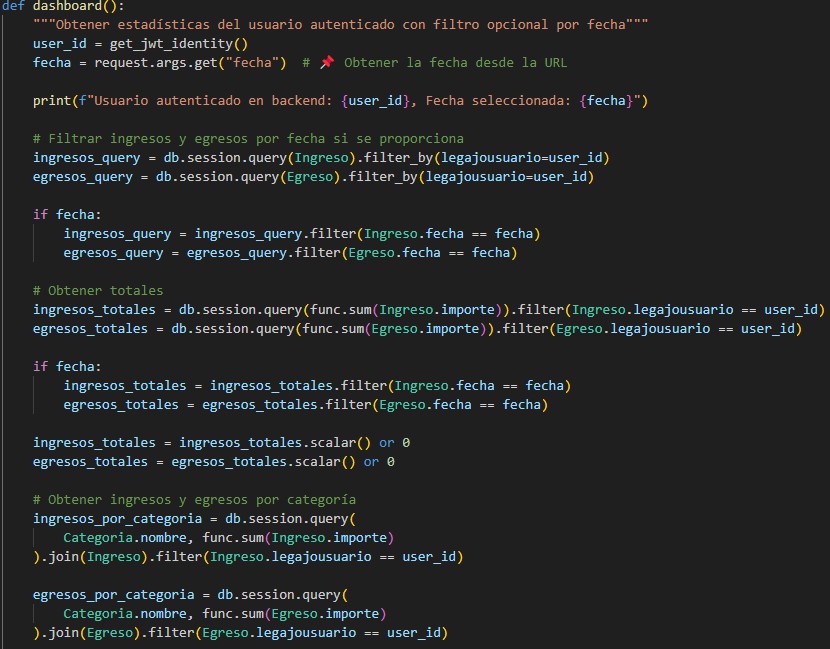
Este fragmento de código define una ruta en Flask para **agregar una nueva categoría**.

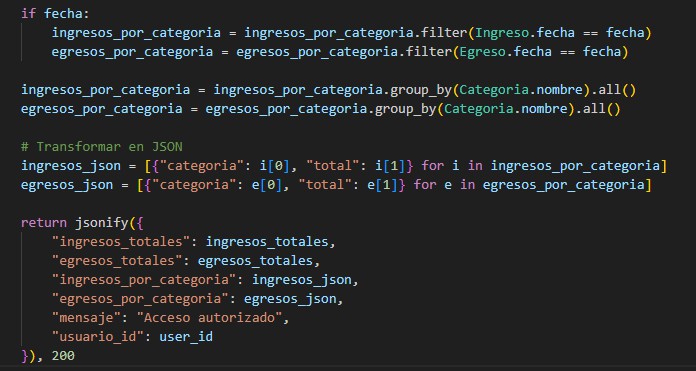


* Utiliza el decorador @jwt\_required() para **requerir un token JWT**.
* Extrae los **claims** (información adicional) del JWT usando get\_jwt().
* **Verifica** si el usuario tiene el rol de admin. Si no lo es, retorna un 403



Extrae campos de nombre y tipo, luego valida que no esten vacíos y que **tipo** sea **ingreso** o **egreso**

****En routes/**dashboard.py:**



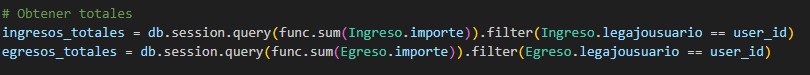
Este fragmento de código proporciona una ruta para obtener estadísticas completas del usuario autenticado, incluyendo ingresos y egresos totales, así como el desglose de esos ingresos y egresos por categoría, con la opción de filtrar por una fecha específica.

Define la ruta **/api/dashboard** que utiliza el método **GET**. Esta ruta está destinada a obtener las

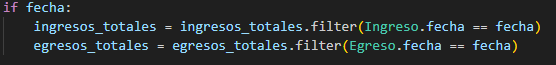
**estadísticas del usuario** autenticado.

* get\_jwt\_identity() obtiene el **ID del usuario autenticado** del JWT, y se asigna a la variable

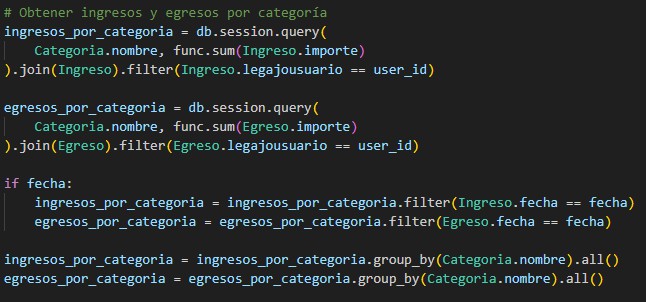
# user\_id.

* Se intenta obtener el parámetro **fecha** de la consulta, Si no se pasa ninguna fecha, la consulta no estará filtrada por fecha, lo que mostrará datos para todos los registros del usuario.
* Crea consultas para obtener todos los ingresos y egresos **relacionados con el usuario autenticado**. Si se proporciona una **fecha**, las consultas se **filtran por esa fecha**.

Calculo de totales.



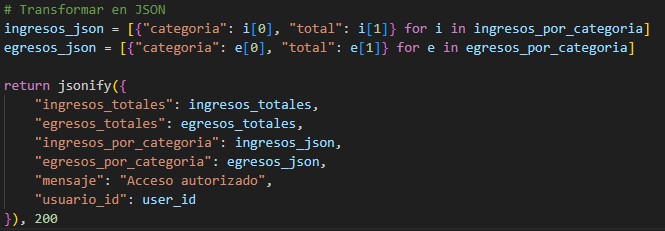
Si se proporciona una fecha en la solicitud, los totales se **filtran** para que solo incluyan ingresos y egresos de esa fecha.



Se obtiene la **suma de ingresos** y **suma de egresos** por cada **categoría** asociada al usuario.

Para cada categoría, la suma de los importes de los ingresos y egresos se agrupa por el nombre de la categoría.

* Agrupa los ingresos y egresos por la categoría group\_by(Categoria.nombre).
* all() devuelve todos los resultados agrupados.



Transforma los resultados de **ingresos por categoría** y **egresos por categoría** a un formato adecuado para JSON.

Cada categoría tiene un **nombre** y un **total** de los ingresos o egresos asociados. La respuesta incluye:

# Totales de ingresos y egresos.

* **Desglose por categoría** de ingresos y egresos.
* Un mensaje de confirmación: **"Acceso autorizado"**.

# El ID del usuario autenticado.

En routes/**history.py:**



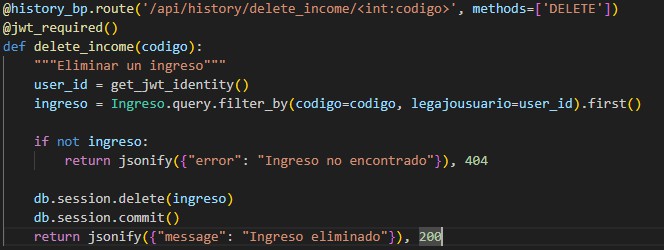
Este fragmento de código define una ruta **/api/history** que permite a los usuarios autenticados obtener su historial completo de ingresos y egresos

Se recuperan todos los **ingresos** (Ingreso.query.filter\_by(legajousuario=user\_id)) y **egresos** (Egreso.query.filter\_by(legajousuario=user\_id)) del usuario autenticado. Esto se hace buscando todos los registros cuyo **legajousuario** coincida con el **ID del usuario**.

Los **ingresos** y **egresos** se convierten a un formato adecuado para JSON.

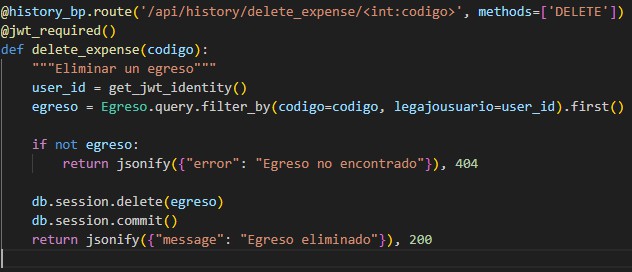
La respuesta es un **JSON** que contiene el historial de **ingresos** y **egresos** del usuario autenticado.

Se devuelve con el código de estado **200 (OK)**, indicando que la solicitud fue procesada correctamente.

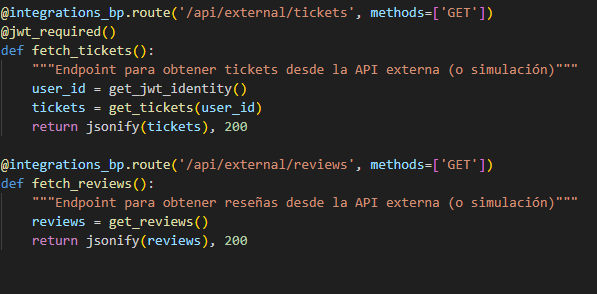


Este fragmento de código define una ruta **/api/history/delete\_income/<int:codigo>** que permite a los usuarios eliminar un ingreso específico.

El parámetro **codigo** es un identificador único para el ingreso que se va a eliminar.



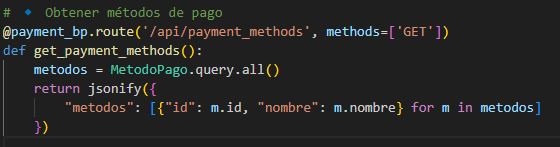
Este es el mismo para el tipo **Egreso**

****En routes/**integrations.py:**

Este código define dos rutas para interactuar con una API externa y obtener información de **tickets** y

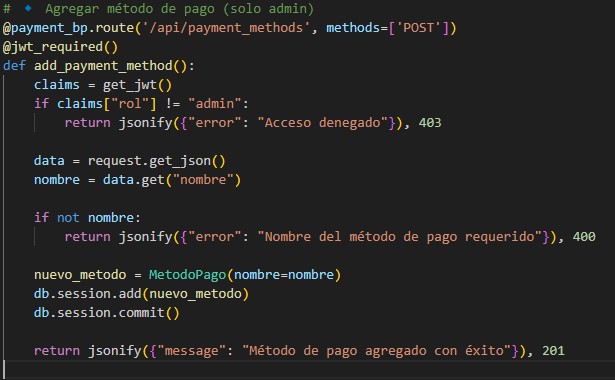
# reseñas.

En routes/**metodospago.py:**

****

Este fragmento de código define un endpoint para obtener los **métodos de pago** disponibles desde la base de datos.

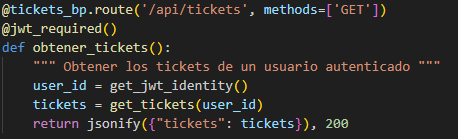
Se crea una lista de diccionarios donde cada diccionario contiene el **id** y **nombre** de cada método de pago. Esto se utiliza para estructurar los datos en formato JSON.



Este fragmento de código define un endpoint para **agregar un método de pago** al sistema, con restricciones de acceso para que **solo los administradores** puedan realizar esta acción.

* Verifica que el **rol** del usuario sea **"admin"**. Si no es un administrador, se retorna un error con el mensaje "Acceso denegado" y un código de estado **403 (Forbidden)**.
* Obtiene los datos en formato JSON de la solicitud **POST**.
* Extrae el nombre del nuevo método de pago de los datos proporcionados. Si no se proporciona un nombre, se retorna un error con el mensaje "Nombre del método de pago requerido" y un código de estado **400 (Bad Request)**.
* Se crea una nueva instancia de la clase MetodoPago con el nombre proporcionado.
* Agrega el nuevo método de pago a la sesión de base de datos.
* Realiza el **commit** a la base de datos, guardando los cambios

En routes/**tickets.py:**

****

Este fragmento de código define un endpoint para **obtener los tickets** de un **usuario autenticado**.

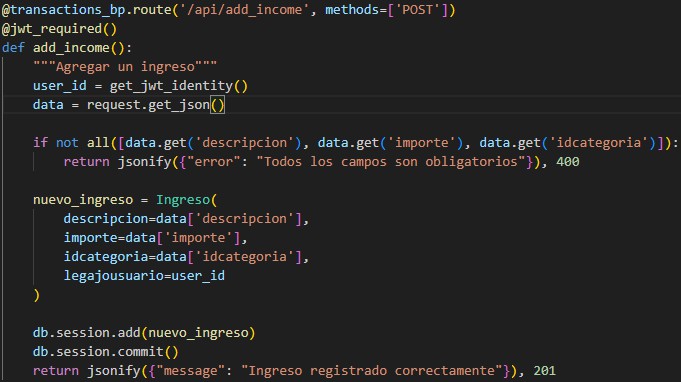
Este fragmento de código define un **endpoint para crear un ticket** en el sistema, permitiendo a los usuarios reportar problemas.

Define un endpoint para la ruta **/api/tickets** con el **método POST**.

* **@jwt\_required()**: Asegura que el usuario esté autenticado mediante un token JWT.
* **user\_id = get\_jwt\_identity()**: Extrae el **ID del usuario** desde el token JWT, lo que garantiza que el ticket será asociado con el usuario autenticado.

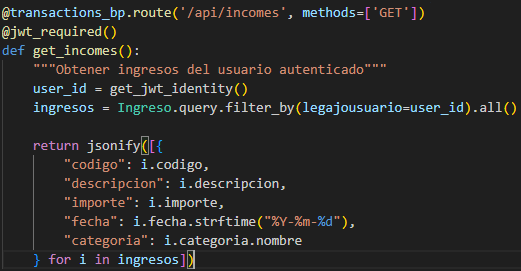
Llama a la función **report\_issue()** para reportar el problema. Esta función interactúa con un sistema de tickets o una API externa que procesa el reporte. El resultado de esta función se guarda en la variable **resultado**

En routes/**transaciones.py**

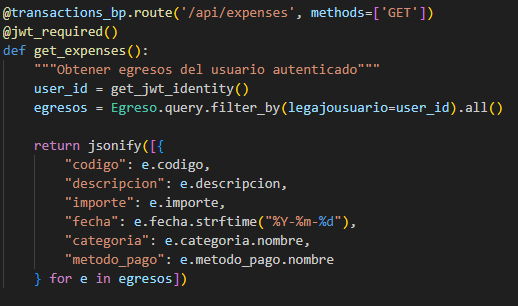
****

Este fragmento de código define un **endpoint para agregar un ingreso** al sistema.

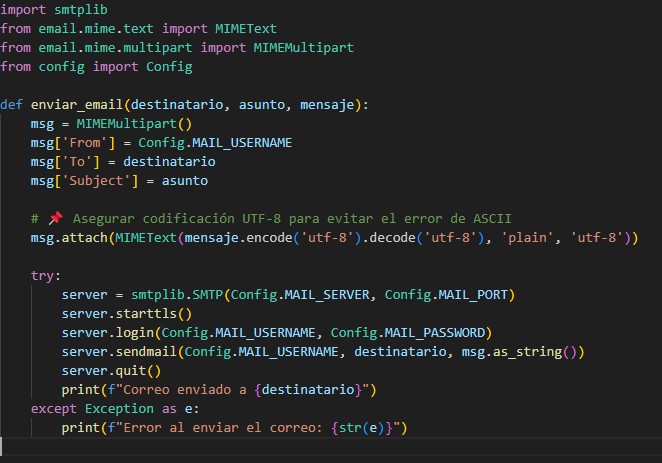
Este fragmento de código define un **endpoint para agregar un egreso** al sistema.



Este fragmento de código define un **endpoint para obtener los ingresos** del usuario autenticado.

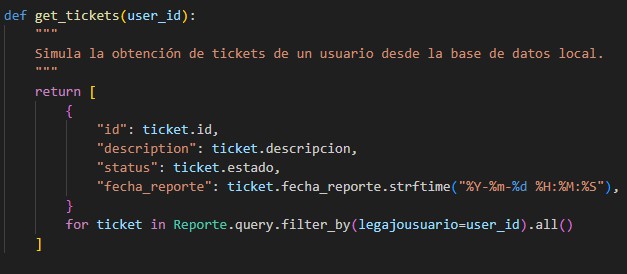


Este fragmento de código define un **endpoint para obtener los egresos** del usuario autenticado.

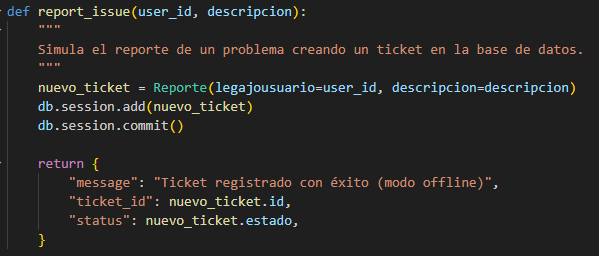
****Ahora en services/**email\_services.py:**

* Importamos librerias necesarias para utilizar un **servicio de emails**
* **MIMEMultipart():** Crea un objeto que puede contener múltiples partes (por ejemplo, texto, imágenes o archivos adjuntos).
* **Cabeceras**: Se definen las cabeceras del mensaje (**From, To, Subject**) con los valores de la configuración y los parámetros proporcionados.
* **smtplib.SMTP**: Crea una conexión con el servidor SMTP (con los parámetros de configuración proporcionados: servidor y puerto).
* **starttls()**: Inicia una capa de seguridad TLS para cifrar la comunicación
* **login()**: Se autentica con las credenciales proporcionadas.
* **sendmail()**: Envía el correo.
* **quit()**: Cierra la conexión con el servidor.

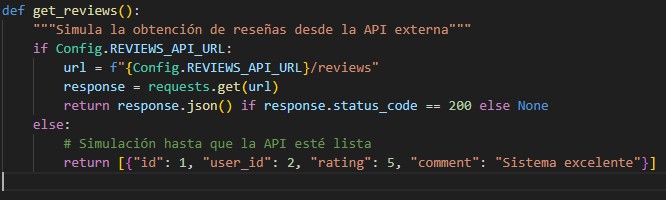
En services/**external\_services.py:**



La función **get\_tickets** simula la obtención de tickets de un usuario desde una base de datos, retornando una lista de diccionarios con la información relevante de cada ticket.



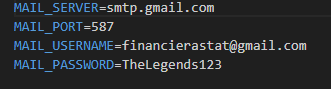
La función **report\_issue** simula la creación de un ticket en la base de datos para reportar un problema.



La función get\_reviews simula la obtención de reseñas desde una API externa, y tiene una estructura que maneja dos casos: uno donde la API externa está configurada y otro donde se simulan reseñas.

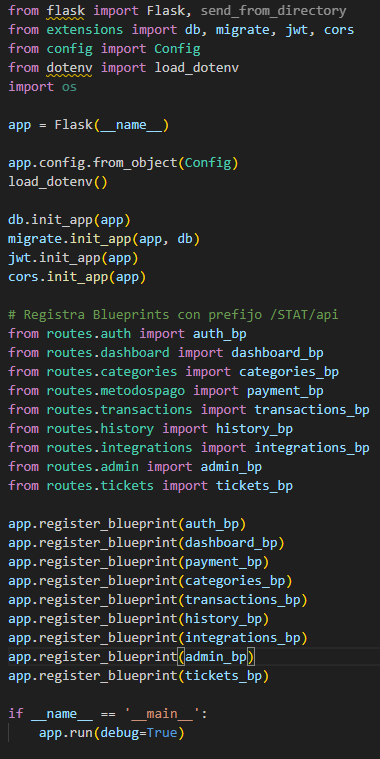
* Si se encuentra configurada la URL de la API externa en Config.REVIEWS\_API\_URL, realiza una solicitud GET a esa URL.
* Si la solicitud tiene éxito devuelve las reseñas obtenidas en formato JSON.
* Si la API no está configurada o la solicitud falla, la función devuelve un conjunto simulado de reseñas.

En **.env**

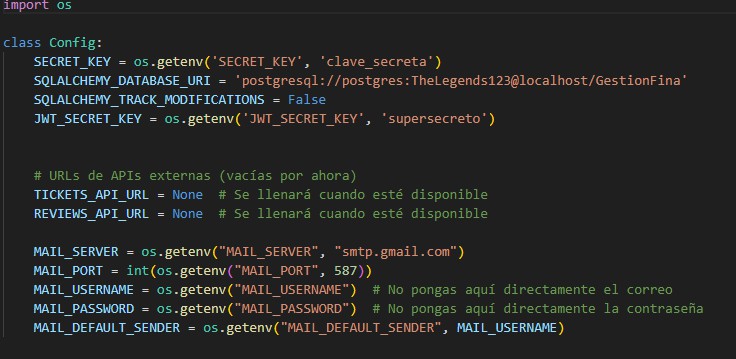
****

Para no compartir información sensible (credenciales de correo electrónico) creamos un archivo “.env” que se crea solo en donde se levanta el proyecto y tiene la configuración para el email utilizado, desde aca va a tomar los datos necesarios para mandar los emails.

En **app.py**

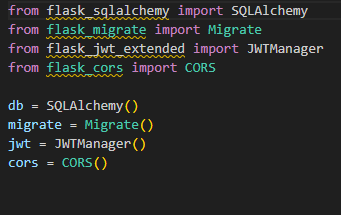
****

Importamos librerias, blueprints y establecemos las rutas necesarias para iniciar la aplicación

****En **config.py**

Establecemos las configuraciones necesarias como para la base de datos, para el servicios de email utilizado y las URL de APIs externas

# En extencios.py:

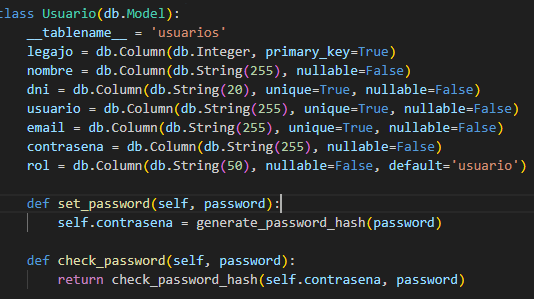
****

Este fragmento de código muestra las inicializaciones de las extensiones de Flask que se estan usando en la aplicación.

* **SQLAlchemy (db)**: Permite interactuar con la base de datos de manera eficiente mediante objetos Python.
* **Migrate (migrate)**: Esta extensión se encarga de las migraciones de la base de datos, permitiéndote hacer cambios en la estructura de la base de datos de forma segura
* **JWTManager (jwt)**: Administra los tokens JWT (JSON Web Tokens) necesarios para la autenticación.
* **CORS (cors)**: Permite que la API sea accesible desde otros orígenes (dominios). Esto es útil si el frontend está en un dominio diferente al backend y necesitas habilitar la comunicación entre ellos.

En **models.py:**

En esta sección se muestran todos los modelos de clases:



Se definen todos los atributos, la longitud y tipo de dato, si es único y si puede estar vacio

set\_password: Este método usa generate\_password\_hash de la librería werkzeug.security para encriptar la contraseña antes de almacenarla en la base de datos

check\_password: Este método compara la contraseña proporcionada con la contraseña almacenada (que está encriptada) usando check\_password\_hash de la misma librería.

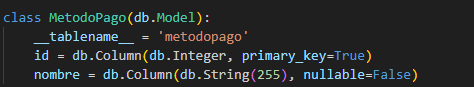


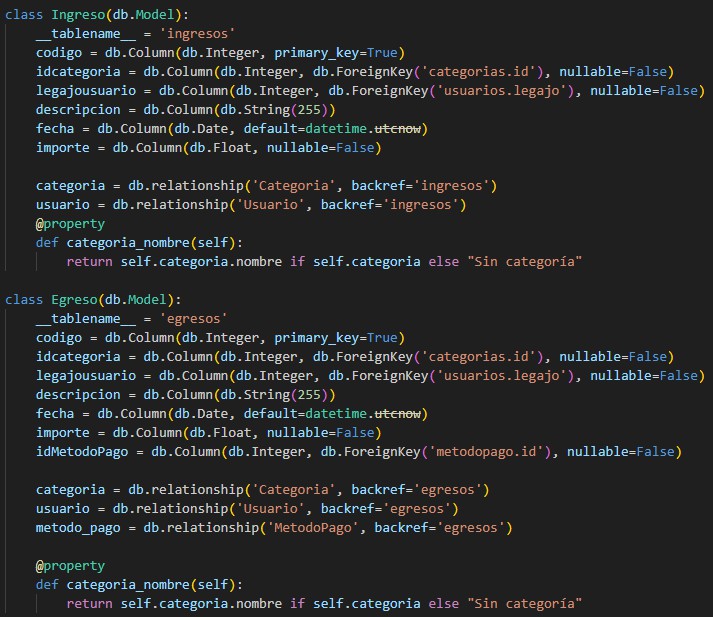
* **obtener\_por\_tipo:** Este método de clase recibe un parámetro tipo (que puede ser 'ingreso' o 'egreso') y retorna todas las categorías que coinciden con ese tipo. Utiliza

**cls.query.filter\_by(tipo=tipo)** para realizar la consulta.

* **crear**: Este método de clase permite crear una nueva categoría con el nombre y tipo

proporcionados. Después de crearla, se añade a la base de datos con **db.session.add()** y se confirma con **db.session.commit().**

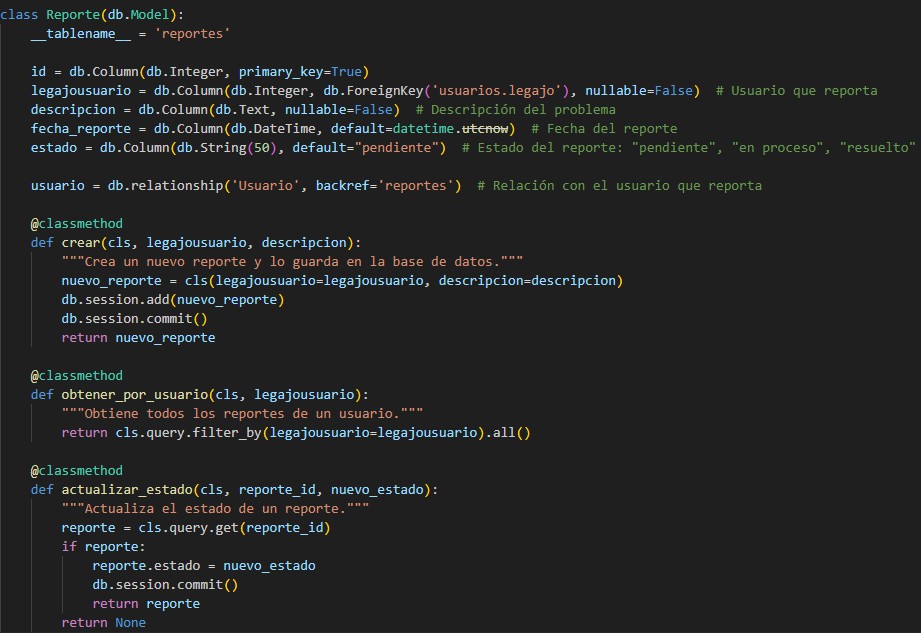


Relaciones:

* **categoria**: Relación con la tabla Categoria. Utiliza **db.relationship** para permitir el acceso directo a la categoría asociada a este ingreso. El parámetro **backref='ingresos'** establece una referencia inversa que permite acceder a todos los ingresos de una categoría.
* **usuario**: Relación con la tabla Usuario, que permite acceder al usuario que registró este ingreso. De nuevo, el parámetro **backref='ingresos**' permite acceder a todos los ingresos de un usuario.

Propiedad:

* **categoria\_nombre**: Una propiedad que devuelve el nombre de la categoría asociada al ingreso. Si no hay categoría asociada (es decir, si la relación es None), devuelve el valor **"Sin categoría".**

La clase reporte esta diseñada para manejar los reportes de problemas de el usuario

# En requirements.txt

El archivo requirements.txt es un archivo de texto que contiene una lista de todas las dependencias (paquetes de Python) necesarias para ejecutar un proyecto, facilita la instalación y configuración de todas las dependencias necesarias para un proyecto, asegurando que cualquier persona que trabaje en el proyecto o lo ejecute en otro entorno tenga todas las bibliotecas correctas instaladas.