UNIVERSIDAD TECNOLÓGICA NACIONAL
REGIONAL LA PLATA
DESARROLLO DE SOFTWARE
Trabajo Práctico Integrador - Compendio

Comisión: S31

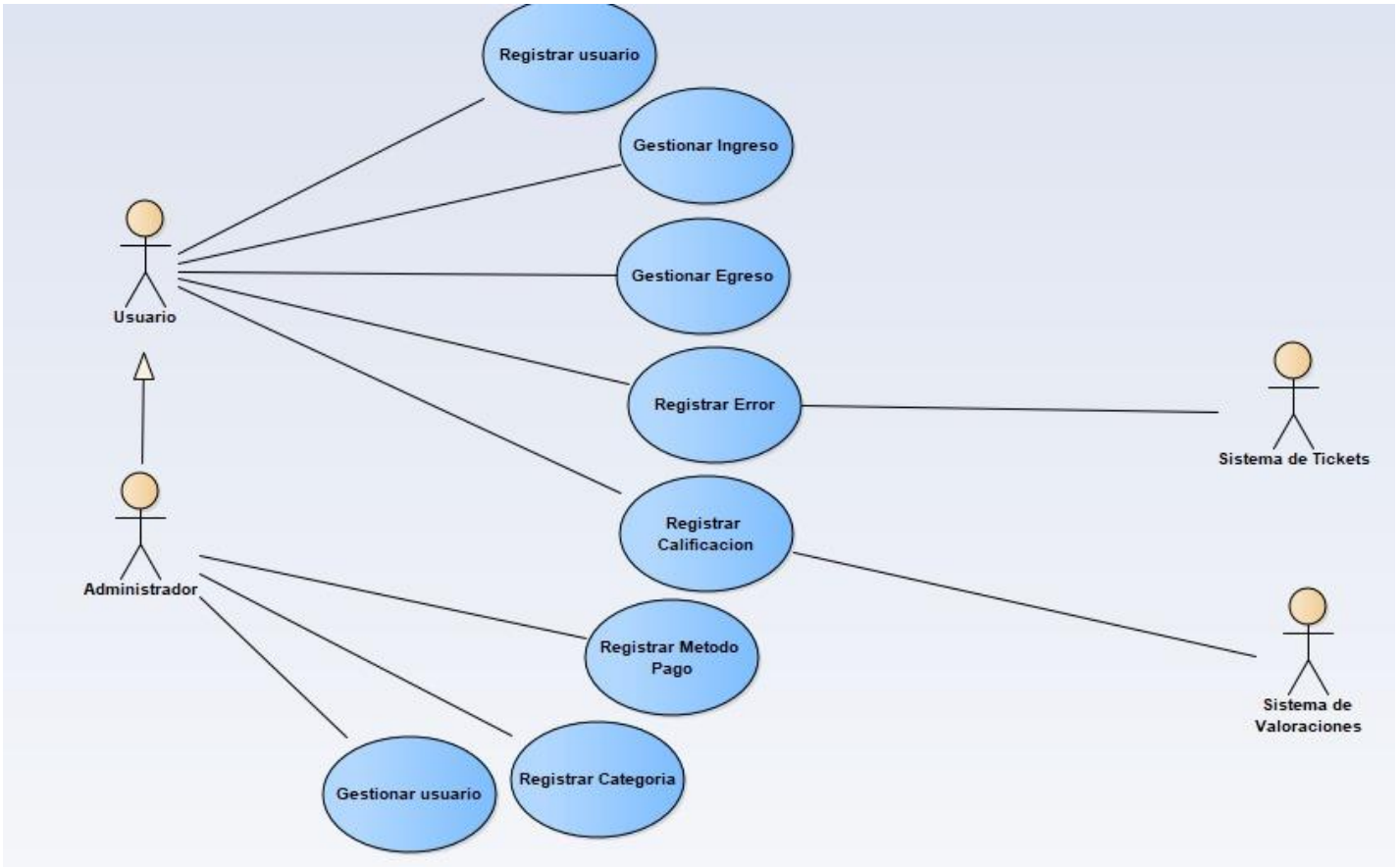
Año: 2024

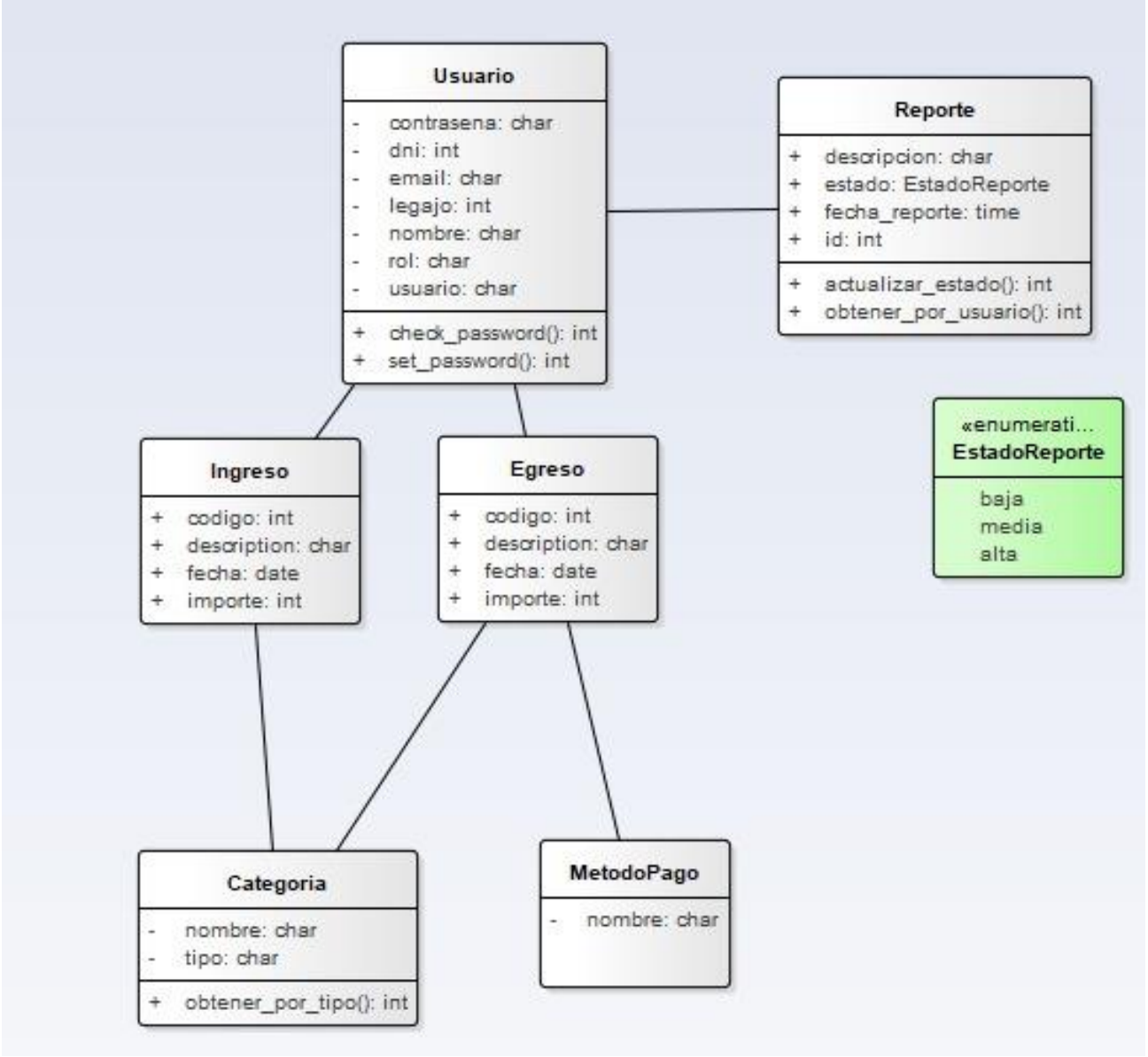
TEMA

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.




Usuario
- contrasena: char
- dni: int
- email: char
- legajo: int
- nombre: char
- rol: char
- usuario: char
+ check_password(): int
+ set_password(): int
Reporte
+ descripcion: char
+ estado: EstadoReporte
+ fecha_reporte: time
+ id: int
+ actualizar_estado(): int
+ obtener_por_usuario(): int
«enumerati...
EstadoReporte
baja
media
alta
Ingreso
+ codigo: int
+ description: char
+ fecha: date
+ importe: int
Egreso
+ codigo: int
+ description: char
+ fecha: date
+ importe: int
Categoria
- nombre: char
- tipo: char
+ obtener_por_tipo(): int
MetodoPago
- nombre: char

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

```jsx
const App = () => {
  const [auth, setAuth] = useState(false);

  return (
    <Router>
      <Routes>
        {/* 🌐 Rutas públicas con Navbar y Footer */}
        <Route
          path="/"
          element={
            <>
              <Navbar />
              <Landing />
              <Footer />
            </>
          }
        />
        <Route
          path="/login"
          element={
            <>
              <Navbar />
              <Login setAuth={setAuth} />
              <Footer />
            </>
          }
        />
        <Route
          path="/register"
          element={
            <>
              <Navbar />
              <Register />
              <Footer />
            </>
          }
        />
```

```jsx
        {/* 🔒 Rutas protegidas para usuarios comunes */}
        <Route
          path="/"
          element={
            <ProtectedRoute>
              <Layout setAuth={setAuth} />
            </ProtectedRoute>
          }
        >
          <Route path="dashboard" element={<Dashboard />} />
          <Route path="ingreso" element={<Ingreso />} />
          <Route path="egreso" element={<Egreso />} />
          <Route path="historial" element={<Historial />} />
          <Route path="report" element={<ReportIssue />} />
          <Route path="calificar" element={<RateSystem />} />
        </Route>

        {/* 🔒 Rutas protegidas SOLO PARA ADMINISTRADORES */}
        <Route
          path="/admin"
          element={
            <ProtectedRoute adminOnly={true}>
              <AdminDashboard /> {/* 📌 Renderiza directamente el panel de Admin */}
            </ProtectedRoute>
          }
        />
      </Routes>
    </Router>
  );
};
```

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)

```jsx
import React, { useEffect, useState } from "react";
import { NavLink, Outlet, useNavigate } from "react-router-dom";
import { jwtDecode } from "jwt-decode";
import "../styles/Layout.css";

const Sidebar = ({ setAuth }) => {
  const navigate = useNavigate();
  const [isAdmin, setIsAdmin] = useState(false);

  useEffect(() => {
    const token = localStorage.getItem("token");
    if (token) {
      const decoded = jwtDecode(token);
      setIsAdmin(decoded.rol === "admin");
    }
  }, []);

  const handleLogout = () => {
    localStorage.removeItem("token");
    setAuth(false);
    navigate("/");
  };
```

```jsx
  return (
    <aside className="sidebar">
      <h1><NavLink to="/">STAT</NavLink></h1>
      <nav>
        <ul>
          <li><NavLink to="/dashboard">Dashboard</NavLink></li>
          <li><NavLink to="/ingreso">Ingresar Ingreso</NavLink></li>
          <li><NavLink to="/egreso">Ingresar Egreso</NavLink></li>
          <li><NavLink to="/historial">Historial</NavLink></li>
          {isAdmin && <li><NavLink to="/admin">Administración</NavLink></li>}
        </ul>
      </nav>

      <div className="sidebar-bottom">
        <button className="btn-rate" onClick={() => navigate("/calificar")}>
          Calificar
        </button>
        <button className="btn-report" onClick={() => navigate("/report")}>
          Reportar Problema
        </button>
        <button className="logout-btn" onClick={handleLogout}>
          Cerrar Sesión
        </button>
      </div>
    </aside>
  );
};

const Layout = ({ setAuth }) => {
  return (
    <div className="layout-container">
      <Sidebar setAuth={setAuth} />
      <div className="main-content">
        <Outlet />
      </div>
    </div>
  );
};

export default Layout;
```

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.

```jsx
import React from "react";
import { Navigate } from "react-router-dom";
import { jwtDecode } from "jwt-decode";

const ProtectedRoute = ({ children, adminOnly = false }) => {
  const token = localStorage.getItem("token");

  if (!token) {
    return <Navigate to="/login" replace />;
  }

  try {
    const decoded = jwtDecode(token);
    console.log("Datos del token:", decoded);  // ✅ Verifica el rol del usuario

    if (adminOnly && decoded.rol !== "admin") {
      console.warn("⚠️ No tienes permisos de administrador.");
      return <Navigate to="/dashboard" replace />;
    }
  } catch (error) {
    console.error("❌ Error al decodificar el token:", error);
    return <Navigate to="/login" replace />;
  }

  return children;
};

export default ProtectedRoute;
```

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.

```jsx
import React, { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom"; // 📌 Importar useNavigate
import { getAdminDashboard, addUser, deleteUser, addCategory, addPaymentMethod } from "../services/api";
import "../styles/admin.css";

const AdminDashboard = () => {
    const [usuarios, setUsuarios] = useState([]);
    const [error, setError] = useState("");
    const [newUser, setNewUser] = useState({ usuario: "", nombre: "", dni: "", email: "", contrasena: "", rol: "usuario" });
    const [newCategory, setNewCategory] = useState({ nombre: "", tipo: "ingreso" });
    const [newPaymentMethod, setNewPaymentMethod] = useState({ nombre: "" });

    const navigate = useNavigate(); // 📌 Hook para navegación

    useEffect(() => {
        fetchAdminData();
    }, []);

    const fetchAdminData = async () => {
        const token = localStorage.getItem("token");
        try {
            const response = await getAdminDashboard(token);
            setUsuarios(response.usuarios);
        } catch (err) {
            setError("Acceso denegado o error al cargar usuarios.");
        }
    };

    const handleBackToDashboard = () => {
        navigate("/dashboard"); // 📌 Redirigir al Dashboard
    };
```

```
const handleAddUser = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    try {
        await addUser(token, newUser);
        fetchAdminData();
        setNewUser({ usuario: "", nombre: "", dni: "", email: "", contrasena: "", rol: "usuario" });
    } catch (err) {
        setError("Error al agregar usuario.");
    }
};

const handleDeleteUser = async (legajo) => {
    const token = localStorage.getItem("token");
    try {
        await deleteUser(token, legajo);
        fetchAdminData();
    } catch (err) {
        setError("Error al eliminar usuario.");
    }
};

const handleAddCategory = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    try {
        await addCategory(token, newCategory);
        setNewCategory({ nombre: "", tipo: "ingreso" });
    } catch (err) {
        setError("Error al agregar categoria.");
    }
```

```
const handleAddPaymentMethod = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    try {
        await addPaymentMethod(token, newPaymentMethod);
        setNewPaymentMethod({ nombre: "" });
    } catch (err) {
        setError("Error al agregar método de pago.");
    }
};

return (
    <div className="admin-container">
        <h2>Panel de Administración</h2>
        {error && <p className="error">{error}</p>}

        {/* 📌 Botón para volver al Dashboard */}
        <button className="back-btn" onClick={handleBackToDashboard}>Volver al Dashboard</button>

        {/* Formulario para agregar usuario */}
        <h3>Agregar Usuario</h3>
        <form onSubmit={handleAddUser} className="admin-form">
            <input type="text" placeholder="Usuario" value={newUser.usuario} onChange={(e) => setNewUser({ ...newUser, usuario: e.target.value })} required />
            <input type="text" placeholder="Nombre" value={newUser.nombre} onChange={(e) => setNewUser({ ...newUser, nombre: e.target.value })} required />
            <input type="text" placeholder="DNI" value={newUser.dni} onChange={(e) => setNewUser({ ...newUser, dni: e.target.value })} required />
            <input type="email" placeholder="Email" value={newUser.email} onChange={(e) => setNewUser({ ...newUser, email: e.target.value })} required />
            <input type="password" placeholder="Contraseña" value={newUser.contrasena} onChange={(e) => setNewUser({ ...newUser, contrasena: e.target.value })} required />
            <select value={newUser.rol} onChange={(e) => setNewUser({ ...newUser, rol: e.target.value })}>
                <option value="usuario">Usuario</option>
                <option value="admin">Admin</option>
            </select>
            <button type="submit">Agregar Usuario</button>
```

```jsx
            {/* Formulario para agregar método de pago */}
            <h3>Agregar Método de Pago</h3>
            <form onSubmit={handleAddPaymentMethod} className="admin-form">
                <input
                    type="text"
                    placeholder="Nombre del Método de Pago"
                    value={newPaymentMethod.nombre}
                    onChange={(e) => setNewPaymentMethod({ ...newPaymentMethod, nombre: e.target.value })}
                    required
                />
                <button type="submit">Agregar Método de Pago</button>
            </form>

            {/* Formulario para agregar categoría */}
            <h3>Agregar Categoría</h3>
            <form onSubmit={handleAddCategory} className="admin-form">
                <input type="text" placeholder="Nombre de la Categoría" value={newCategory.nombre} onChange={(e) => setNewCategory({ ...newCategory, nombre: e.target.value })} required />
                <select value={newCategory.tipo} onChange={(e) => setNewCategory({ ...newCategory, tipo: e.target.value })}>
                    <option value="ingreso">Ingreso</option>
                    <option value="egreso">Egreso</option>
                </select>
                <button type="submit">Agregar Categoría</button>
            </form>
        </div>
    );
};

export default AdminDashboard;
```

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.

```jsx
import React, { useState } from "react";
import { sendRating } from "../services/api";
import "../styles/feedback.css"; // Importar el CSS unificado

const RateSystem = () => {
    const [rating, setRating] = useState(0);
    const [message, setMessage] = useState("");

    const handleSubmit = async () => {
        const token = localStorage.getItem("token");
        try {
            await sendRating(token, rating);
            setMessage("Gracias por tu calificación");
        } catch (error) {
            setMessage("Error al enviar calificación");
        }
    };

    return (
        <div className="feedback-container">
            <h2>Califica el sistema</h2>
            <div className="rating-container">
                {[1, 2, 3, 4, 5].map((star) => (
                    <span
                        key={star}
                        onClick={() => setRating(star)}
                        className={`star ${rating >= star ? "gold" : "gray"}`}
                    >
                        ★
                    </span>
                ))}
            </div>
            <button onClick={handleSubmit}>Enviar Calificación</button>
            {message && <p className={message.includes("Error") ? "error-message" : "feedback-message"}>{message}</p>}
        </div>
    );
};

export default RateSystem;
```

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.

```
// Función para iniciar sesión
export const login = async (usuario, contrasena) => {
    const response = await fetch(`${API_URL}/login`, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ usuario, contrasena }),
    });

    if (!response.ok) {
        throw new Error("Credenciales incorrectas");
    }

    return response.json();
};

// Función para obtener datos del Dashboard
export const getDashboard = async (token, fecha) => {
    try {
        const url = fecha ? `${API_URL}/dashboard?fecha=${fecha}` : `${API_URL}/dashboard`;

        const response = await fetch(url, {
            method: "GET",
            headers: {
                "Authorization": `Bearer ${token}`,
                "Content-Type": "application/json"
            }
        });

        if (!response.ok) {
            throw new Error(`Error en la API: ${response.status} - ${response.statusText}`);
        }

        const data = await response.json();
        console.log("Datos del Dashboard obtenidos:", data);
        return data;
    } catch (error) {
        console.error("Error al obtener datos del Dashboard:", error);
        return { ingresos_totales: 0, egresos_totales: 0, ingresos_por_categoria: [], egresos_por_categoria: [] };
    }
};
```

```javascript
// Función para obtener datos del Dashboard
export const getDashboard = async (token, fecha) => {
    try {
        const url = fecha ? `${API_URL}/dashboard?fecha=${fecha}` : `${API_URL}/dashboard`;

        const response = await fetch(url, {
            method: "GET",
            headers: {
                "Authorization": `Bearer ${token}`,
                "Content-Type": "application/json"
            }
        });

        if (!response.ok) {
            throw new Error(`Error en la API: ${response.status} - ${response.statusText}`);
        }

        const data = await response.json();
        console.log("Datos del Dashboard obtenidos:", data);
        return data;
    } catch (error) {
        console.error("Error al obtener datos del Dashboard:", error);
        return { ingresos_totales: 0, egresos_totales: 0, ingresos_por_categoria: [], egresos_por_categoria: [] };
    }
};
```

```javascript
// Función para registrar un egreso
export const addExpense = async (token, descripcion, importe, idcategoria, idMetodoPago) => {
    const response = await fetch(`${API_URL}/add_expense`, {
        method: "POST",
        headers: {
            "Content-Type": "application/json",
            "Authorization": `Bearer ${token}`
        },
        body: JSON.stringify({ descripcion, importe, idcategoria, idMetodoPago }),
    });

    if (!response.ok) {
        const errorData = await response.json();
        throw new Error(errorData.error || "Error al registrar egreso");
    }

    return response.json();
};

// Función para obtener categorías por tipo (por ejemplo, "ingreso" o "egreso")
export const getCategories = async (token, tipo) => {
    const response = await fetch(`${API_URL}/categories?tipo=${tipo}`, {
        method: "GET",
        headers: {
            "Authorization": `Bearer ${token}`,
            "Content-Type": "application/json"
        }
    });
    if (!response.ok) {
        throw new Error("Error al obtener categorías");
    }
    return response.json();
};
```

```
export const getAdminDashboard = async (token) => {
  console.log("Token enviado:", token);  // 👀 Verifica si hay token

  const response = await fetch("http://127.0.0.1:5000/api/admin", {
      method: "GET",
      headers: {
          "Authorization": `Bearer ${token}`,
          "Content-Type": "application/json"
      }
  });

  const data = await response.json();
  console.log("Usuarios recibidos:", data); // 👀 Verifica los datos recibidos

  if (!response.ok) {
      throw new Error("Error al obtener datos del administrador");
  }

  return data;
};
```

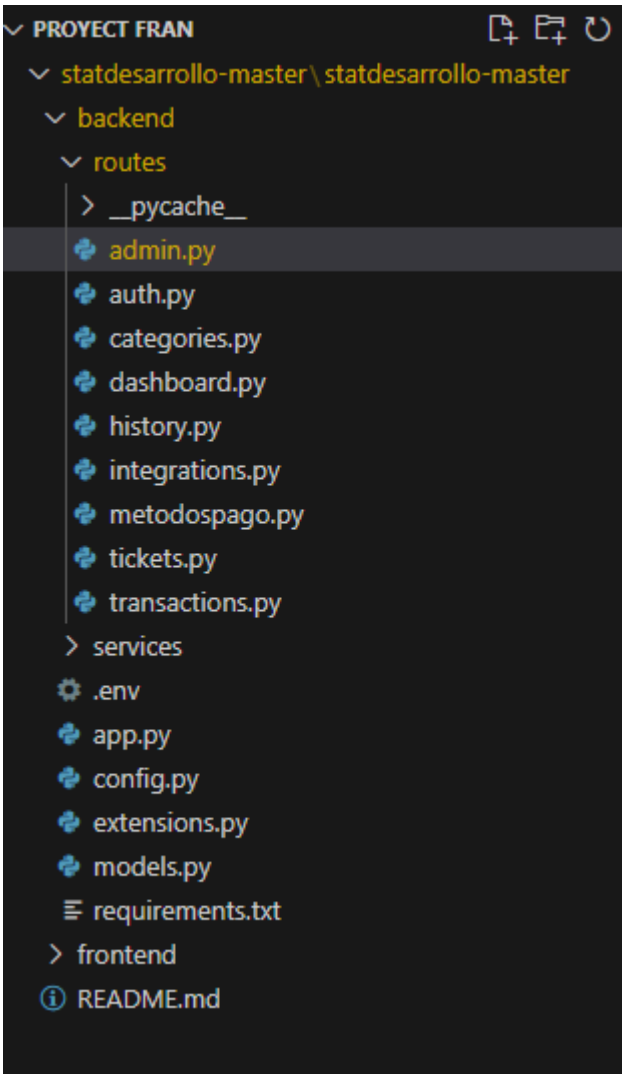
auth.js

`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 }`.

```
import { jwtDecode } from "jwt-decode";

export const getUser = (token) => {
  try {
    const decoded = jwtDecode(token); // Decodifica el token JWT
    return { usuario: decoded.usuario }; // Ahora devuelve el usuario en lugar del legajo
  } catch (error) {
    console.error("Error al decodificar el token:", error);
    return { usuario: "Desconocido" };
  }
};
```



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

- Importaciones varias

```python
from flask import Blueprint, jsonify, request
from flask_jwt_extended import jwt_required, get_jwt
from models import Usuario
from extensions import db
```

```python
@admin_bp.route('/api/admin', methods=['GET'])
@jwt_required()
def admin_dashboard():
    """Panel de Administración - Solo accesible por admins"""
    claims = get_jwt()
    if claims["rol"] != "admin":
        return jsonify({"error": "Acceso denegado"}), 403

    usuarios = Usuario.query.all()
    usuarios_lista = [{"legajo": u.legajo, "nombre": u.nombre, "usuario": u.usuario, "email": u.email, "rol": u.rol} for u in usuarios]

    return jsonify({"usuarios": usuarios_lista})
```

- 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.

```
claims = get_jwt()
if claims["rol"] != "admin":
    return jsonify({"error": "Acceso denegado"}), 403
```

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)**.

```
@admin_bp.route('/api/admin/delete_user/<int:legajo>', methods=['DELETE'])
@jwt_required()
def delete_user(legajo):
    claims = get_jwt()
    if claims["rol"] != "admin":
        return jsonify({"error": "Acceso denegado"}), 403

    usuario = Usuario.query.filter_by(legajo=legajo).first()
    if not usuario:
        return jsonify({"error": "Usuario no encontrado"}), 404

    db.session.delete(usuario)  # Elimina el usuario de la BD
    db.session.commit()  # Guarda los cambios

    return jsonify({"message": "Usuario eliminado con éxito"}), 200
```

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

---

```python
@admin_bp.route('/api/admin/add_user', methods=['POST'])
@jwt_required()
def add_user():
    claims = get_jwt()
    if claims["rol"] != "admin":
        return jsonify({"error": "Acceso denegado"}), 403

    data = request.get_json()
    usuario = data.get("usuario")
    nombre = data.get("nombre")
    dni = data.get("dni")
    email = data.get("email")
    contrasena = data.get("contrasena")
    rol = data.get("rol", "usuario")  # Valor por defecto: usuario

    if not usuario or not nombre or not dni or not email or not contrasena:
        return jsonify({"error": "Faltan datos"}), 400

    nuevo_usuario = Usuario(usuario=usuario, nombre=nombre, dni=dni, email=email, contrasena=contrasena, rol=rol)
    db.session.add(nuevo_usuario)
    db.session.commit()

    return jsonify({"message": "Usuario agregado con éxito"}), 201
```

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

```python
if not usuario or not nombre or not dni or not email or not contrasena:
    return jsonify({"error": "Faltan datos"}), 400
```

- Verifica que **todos** los campos obligatorios estén presentes.
- Si falta alguno, retorna un **error 400 (Bad Request)**.

```python
nuevo_usuario = Usuario(usuario=usuario, nombre=nombre, dni=dni, email=email, contrasena=contrasena, rol=rol)
db.session.add(nuevo_usuario)
db.session.commit()
```

- 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:

```python
from flask import Blueprint, request, jsonify
from flask_jwt_extended import create_access_token, jwt_required, get_jwt_identity
from models import Usuario, db
from werkzeug.security import check_password_hash
from sqlalchemy.exc import IntegrityError
from services.email_services import enviar_email
```

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

```python
auth_bp = Blueprint('auth', __name__)
```

- 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.

```
@auth_bp.route('/api/register', methods=['POST'])
def register():
    """Registro de usuarios con email"""
    data = request.get_json()

    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

    nuevo_usuario = Usuario(
        usuario=data['usuario'],
        nombre=data['nombre'],
        dni=data['dni'],
        email=data['email'],  # 📌 Nuevo campo
        rol=data.get('rol','usuario')
    )
    nuevo_usuario.set_password(data['contrasena'])

    try:
        db.session.add(nuevo_usuario)
        db.session.commit()

        # 📌 Enviar correo de bienvenida
        asunto = "Registro exitoso en Finanzas-Stat"
        mensaje = f"Hola {data['nombre']}, tu registro fue exitoso en Finanzas-Stat."
        enviar_email(data['email'], asunto, mensaje)

        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
```

- 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.

```
nuevo_usuario = Usuario(
    usuario=data['usuario'],
    nombre=data['nombre'],
    dni=data['dni'],
    email=data['email'],  # 📌 Nuevo campo
    rol=data.get('rol','usuario')
)
nuevo_usuario.set_password(data['contrasena'])
```

- 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 correo de bienvenida
asunto = "Registro exitoso en Finanzas-Stat"
mensaje = f"Hola {data['nombre']}, tu registro fue exitoso en Finanzas-Stat."
enviar_email(data['email'], asunto, mensaje)
```

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

```
except IntegrityError:
    db.session.rollback()
    return jsonify({"error": "Usuario, DNI o Email ya registrado"}), 400
```

- 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.

```
@auth_bp.route('/api/login', methods=['POST'])
def login():
    """Inicio de sesión y generación de token JWT"""
    data = request.get_json()
    usuario = Usuario.query.filter_by(usuario=data.get('usuario')).first()

    if not usuario or not usuario.check_password(data.get('contrasena')):
        return jsonify({"error": "Credenciales incorrectas"}), 401

    # 📌 Ahora el token incluye el rol
    access_token = create_access_token(identity=str(usuario.legajo), additional_claims={"rol": usuario.rol})

    return jsonify({"access_token": access_token, "usuario": usuario.usuario, "rol": usuario.rol})
```

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.

```
# 📌 Ahora el token incluye el rol
access_token = create_access_token(identity=str(usuario.legajo), additional_claims={"rol": usuario.rol})

return jsonify({"access_token": access_token, "usuario": usuario.usuario, "rol": usuario.rol})
```

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**:

```python
@categories_bp.route('/api/categories', methods=['GET'])
def get_categories():
    tipo = request.args.get('tipo')
    if not tipo or tipo not in ['ingreso', 'egreso']:
        return jsonify({"error": "Tipo de categoría inválido"}), 400

    categorias = Categoria.obtener_por_tipo(tipo)  # Método en el modelo
    return jsonify({
        "categories": [{"id": c.id, "nombre": c.nombre} for c in categorias]
    })
```

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

```python
if not tipo or tipo not in ['ingreso', 'egreso']:
    return jsonify({"error": "Tipo de categoría inválido"}), 400
```

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.

```python
categorias = Categoria.obtener_por_tipo(tipo)  # Método en el modelo
return jsonify({
    "categories": [{"id": c.id, "nombre": c.nombre} for c in categorias]
})
```

- 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.

```python
@categories_bp.route('/api/categories', methods=['OPTIONS'])
def options_category():
    return '', 204  # Respuesta vacía con código 204 (No Content)
```

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.

```python
@categories_bp.route('/api/categories', methods=['POST'])
@jwt_required()
def add_category():
    claims = get_jwt()
    if claims["rol"] != "admin":
        return jsonify({"error": "Acceso denegado"}), 403

    data = request.get_json()
    print("Datos recibidos:", data)  # <-- Agregar esta línea para ver qué datos llegan

    nombre = data.get("nombre")
    tipo = data.get("tipo")

    if not nombre or tipo not in ["ingreso", "egreso"]:
        return jsonify({"error": "Datos inválidos"}), 400

    nueva_categoria = Categoria.crear(nombre, tipo)
    if nueva_categoria:
        return jsonify({"message": "Categoría agregada con éxito"}), 201
    return jsonify({"error": "Error al agregar categoría"}), 500
```

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

```python
claims = get_jwt()
if claims["rol"] != "admin":
    return jsonify({"error": "Acceso denegado"}), 403
```

- 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

```python
nombre = data.get("nombre")
tipo = data.get("tipo")

if not nombre or tipo not in ["ingreso", "egreso"]:
    return jsonify({"error": "Datos inválidos"}), 400
```

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

En routes/dashboard.py:

```python
def dashboard():
    """Obtener estadísticas del usuario autenticado con filtro opcional por fecha"""
    user_id = get_jwt_identity()
    fecha = request.args.get("fecha")  # 📌 Obtener la fecha desde la URL

    print(f"Usuario autenticado en backend: {user_id}, Fecha seleccionada: {fecha}")

    # Filtrar ingresos y egresos por fecha si se proporciona
    ingresos_query = db.session.query(Ingreso).filter_by(legajousuario=user_id)
    egresos_query = db.session.query(Egreso).filter_by(legajousuario=user_id)

    if fecha:
        ingresos_query = ingresos_query.filter(Ingreso.fecha == fecha)
        egresos_query = egresos_query.filter(Egreso.fecha == fecha)

    # Obtener totales
    ingresos_totales = db.session.query(func.sum(Ingreso.importe)).filter(Ingreso.legajousuario == user_id)
    egresos_totales = db.session.query(func.sum(Egreso.importe)).filter(Egreso.legajousuario == user_id)

    if fecha:
        ingresos_totales = ingresos_totales.filter(Ingreso.fecha == fecha)
        egresos_totales = egresos_totales.filter(Egreso.fecha == fecha)

    ingresos_totales = ingresos_totales.scalar() or 0
    egresos_totales = egresos_totales.scalar() or 0

    # Obtener ingresos y egresos por categoría
    ingresos_por_categoria = db.session.query(
        Categoria.nombre, func.sum(Ingreso.importe)
    ).join(Ingreso).filter(Ingreso.legajousuario == user_id)

    egresos_por_categoria = db.session.query(
        Categoria.nombre, func.sum(Egreso.importe)
    ).join(Egreso).filter(Egreso.legajousuario == user_id)
```

```
    if fecha:
        ingresos_por_categoria = ingresos_por_categoria.filter(Ingreso.fecha == fecha)
        egresos_por_categoria = egresos_por_categoria.filter(Egreso.fecha == fecha)

    ingresos_por_categoria = ingresos_por_categoria.group_by(Categoria.nombre).all()
    egresos_por_categoria = egresos_por_categoria.group_by(Categoria.nombre).all()

    # Transformar en JSON
    ingresos_json = [{"categoria": i[0], "total": i[1]} for i in ingresos_por_categoria]
    egresos_json = [{"categoria": e[0], "total": e[1]} for e in egresos_por_categoria]

    return jsonify({
        "ingresos_totales": ingresos_totales,
        "egresos_totales": egresos_totales,
        "ingresos_por_categoria": ingresos_json,
        "egresos_por_categoria": egresos_json,
        "mensaje": "Acceso autorizado",
        "usuario_id": user_id
    }), 200
```

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**.

```
# Obtener totales
ingresos_totales = db.session.query(func.sum(Ingreso.importe)).filter(Ingreso.legajousuario == user_id)
egresos_totales = db.session.query(func.sum(Egreso.importe)).filter(Egreso.legajousuario == user_id)
```

Calculo de totales.

```
if fecha:
    ingresos_totales = ingresos_totales.filter(Ingreso.fecha == fecha)
    egresos_totales = egresos_totales.filter(Egreso.fecha == fecha)
```

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

```python
# Obtener ingresos y egresos por categoría
ingresos_por_categoria = db.session.query(
    Categoria.nombre, func.sum(Ingreso.importe)
).join(Ingreso).filter(Ingreso.legajousuario == user_id)

egresos_por_categoria = db.session.query(
    Categoria.nombre, func.sum(Egreso.importe)
).join(Egreso).filter(Egreso.legajousuario == user_id)

if fecha:
    ingresos_por_categoria = ingresos_por_categoria.filter(Ingreso.fecha == fecha)
    egresos_por_categoria = egresos_por_categoria.filter(Egreso.fecha == fecha)

ingresos_por_categoria = ingresos_por_categoria.group_by(Categoria.nombre).all()
egresos_por_categoria = egresos_por_categoria.group_by(Categoria.nombre).all()
```

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.

```python
# Transformar en JSON
ingresos_json = [{"categoria": i[0], "total": i[1]} for i in ingresos_por_categoria]
egresos_json = [{"categoria": e[0], "total": e[1]} for e in egresos_por_categoria]

return jsonify({
    "ingresos_totales": ingresos_totales,
    "egresos_totales": egresos_totales,
    "ingresos_por_categoria": ingresos_json,
    "egresos_por_categoria": egresos_json,
    "mensaje": "Acceso autorizado",
    "usuario_id": user_id
}), 200
```

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:**

```python
@history_bp.route('/api/history', methods=['GET'])
@jwt_required()
def get_history():
    """Obtener el historial completo del usuario autenticado"""
    user_id = get_jwt_identity()

    ingresos = Ingreso.query.filter_by(legajousuario=user_id).all()
    egresos = Egreso.query.filter_by(legajousuario=user_id).all()

    history_data = {
        "ingresos": [{
            "codigo": i.codigo,
            "descripcion": i.descripcion,
            "importe": i.importe,
            "fecha": i.fecha.strftime("%Y-%m-%d"),
            "categoria": i.categoria.nombre
        } for i in ingresos],
        "egresos": [{
            "codigo": e.codigo,
            "descripcion": e.descripcion,
            "importe": e.importe,
            "fecha": e.fecha.strftime("%Y-%m-%d"),
            "categoria": e.categoria.nombre,
            "metodo_pago": e.metodo_pago.nombre
        } for e in egresos]
    }

    return jsonify(history_data), 200
```

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.

```python
@history_bp.route('/api/history/delete_income/<int:codigo>', methods=['DELETE'])
@jwt_required()
def delete_income(codigo):
    """Eliminar un ingreso"""
    user_id = get_jwt_identity()
    ingreso = Ingreso.query.filter_by(codigo=codigo, legajousuario=user_id).first()

    if not ingreso:
        return jsonify({"error": "Ingreso no encontrado"}), 404

    db.session.delete(ingreso)
    db.session.commit()
    return jsonify({"message": "Ingreso eliminado"}), 200
```

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.

```python
@history_bp.route('/api/history/delete_expense/<int:codigo>', methods=['DELETE'])
@jwt_required()
def delete_expense(codigo):
    """Eliminar un egreso"""
    user_id = get_jwt_identity()
    egreso = Egreso.query.filter_by(codigo=codigo, legajousuario=user_id).first()

    if not egreso:
        return jsonify({"error": "Egreso no encontrado"}), 404

    db.session.delete(egreso)
    db.session.commit()
    return jsonify({"message": "Egreso eliminado"}), 200
```

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

En routes/**integrations.py**:

```python
@integrations_bp.route('/api/external/tickets', methods=['GET'])
@jwt_required()
def fetch_tickets():
    """Endpoint para obtener tickets desde la API externa (o simulación)"""
    user_id = get_jwt_identity()
    tickets = get_tickets(user_id)
    return jsonify(tickets), 200

@integrations_bp.route('/api/external/reviews', methods=['GET'])
def fetch_reviews():
    """Endpoint para obtener reseñas desde la API externa (o simulación)"""
    reviews = get_reviews()
    return jsonify(reviews), 200
```

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

En routes/**metodospago.py**:

```python
# 🔹 Obtener métodos de pago
@payment_bp.route('/api/payment_methods', methods=['GET'])
def get_payment_methods():
    metodos = MetodoPago.query.all()
    return jsonify({
        "metodos": [{"id": m.id, "nombre": m.nombre} for m in metodos]
    })
```

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.

```python
# 🔹 Agregar método de pago (solo admin)
@payment_bp.route('/api/payment_methods', methods=['POST'])
@jwt_required()
def add_payment_method():
    claims = get_jwt()
    if claims["rol"] != "admin":
        return jsonify({"error": "Acceso denegado"}), 403

    data = request.get_json()
    nombre = data.get("nombre")

    if not nombre:
        return jsonify({"error": "Nombre del método de pago requerido"}), 400

    nuevo_metodo = MetodoPago(nombre=nombre)
    db.session.add(nuevo_metodo)
    db.session.commit()

    return jsonify({"message": "Método de pago agregado con éxito"}), 201
```

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:**

```python
@tickets_bp.route('/api/tickets', methods=['GET'])
@jwt_required()
def obtener_tickets():
    """ Obtener los tickets de un usuario autenticado """
    user_id = get_jwt_identity()
    tickets = get_tickets(user_id)
    return jsonify({"tickets": tickets}), 200
```

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

```python
@tickets_bp.route('/api/tickets', methods=['POST'])
@jwt_required()
def crear_ticket():
    """ Reportar un problema """
    user_id = get_jwt_identity()
    data = request.get_json()
    print("Datos recibidos en backend:", data)  # ✅ Verificar qué está recibiendo

    descripcion = data.get("description")

    if not descripcion:
        return jsonify({"error": "La descripción es obligatoria"}), 400

    resultado = report_issue(user_id, descripcion)
    print("Respuesta de report_issue:", resultado)  # ✅ Verificar qué devuelve la API externa
    return jsonify(resultado), 201 if "error" not in resultado else 400
```

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**

```python
@transactions_bp.route('/api/add_income', methods=['POST'])
@jwt_required()
def add_income():
    """Agregar un ingreso"""
    user_id = get_jwt_identity()
    data = request.get_json()

    if not all([data.get('descripcion'), data.get('importe'), data.get('idcategoria')]):
        return jsonify({"error": "Todos los campos son obligatorios"}), 400

    nuevo_ingreso = Ingreso(
        descripcion=data['descripcion'],
        importe=data['importe'],
        idcategoria=data['idcategoria'],
        legajousuario=user_id
    )

    db.session.add(nuevo_ingreso)
    db.session.commit()
    return jsonify({"message": "Ingreso registrado correctamente"}), 201
```

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

```python
@transactions_bp.route('/api/add_expense', methods=['POST'])
@jwt_required()
def add_expense():
    """Agregar un egreso"""
    user_id = get_jwt_identity()
    data = request.get_json()

    if not all([data.get('descripcion'), data.get('importe'), data.get('idcategoria'), data.get('idMetodoPago')]):
        return jsonify({"error": "Todos los campos son obligatorios"}), 400

    nuevo_egreso = Egreso(
        descripcion=data['descripcion'],
        importe=data['importe'],
        idcategoria=data['idcategoria'],
        idMetodoPago=data['idMetodoPago'],
        legajousuario=user_id
    )

    db.session.add(nuevo_egreso)
    db.session.commit()
    return jsonify({"message": "Egreso registrado correctamente"}), 201
```

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

```python
@transactions_bp.route('/api/incomes', methods=['GET'])
@jwt_required()
def get_incomes():
    """Obtener ingresos del usuario autenticado"""
    user_id = get_jwt_identity()
    ingresos = Ingreso.query.filter_by(legajousuario=user_id).all()

    return jsonify([{
        "codigo": i.codigo,
        "descripcion": i.descripcion,
        "importe": i.importe,
        "fecha": i.fecha.strftime("%Y-%m-%d"),
        "categoria": i.categoria.nombre
    } for i in ingresos])
```

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

```python
@transactions_bp.route('/api/expenses', methods=['GET'])
@jwt_required()
def get_expenses():
    """Obtener egresos del usuario autenticado"""
    user_id = get_jwt_identity()
    egresos = Egreso.query.filter_by(legajousuario=user_id).all()

    return jsonify([{
        "codigo": e.codigo,
        "descripcion": e.descripcion,
        "importe": e.importe,
        "fecha": e.fecha.strftime("%Y-%m-%d"),
        "categoria": e.categoria.nombre,
        "metodo_pago": e.metodo_pago.nombre
    } for e in egresos])
```

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

Ahora en services/**email_services.py**:

```python
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from config import Config

def enviar_email(destinatario, asunto, mensaje):
    msg = MIMEMultipart()
    msg['From'] = Config.MAIL_USERNAME
    msg['To'] = destinatario
    msg['Subject'] = asunto

    # 📌 Asegurar codificación UTF-8 para evitar el error de ASCII
    msg.attach(MIMEText(mensaje.encode('utf-8').decode('utf-8'), 'plain', 'utf-8'))

    try:
        server = smtplib.SMTP(Config.MAIL_SERVER, Config.MAIL_PORT)
        server.starttls()
        server.login(Config.MAIL_USERNAME, Config.MAIL_PASSWORD)
        server.sendmail(Config.MAIL_USERNAME, destinatario, msg.as_string())
        server.quit()
        print(f"Correo enviado a {destinatario}")
    except Exception as e:
        print(f"Error al enviar el correo: {str(e)}")
```

- 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**:

```python
def get_tickets(user_id):
    """
    Simula la obtención de tickets de un usuario desde la base de datos local.
    """
    return [
        {
            "id": ticket.id,
            "description": ticket.descripcion,
            "status": ticket.estado,
            "fecha_reporte": ticket.fecha_reporte.strftime("%Y-%m-%d %H:%M:%S"),
        }
        for ticket in Reporte.query.filter_by(legajousuario=user_id).all()
    ]
```

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.

```python
def report_issue(user_id, descripcion):
    """
    Simula el reporte de un problema creando un ticket en la base de datos.
    """
    nuevo_ticket = Reporte(legajousuario=user_id, descripcion=descripcion)
    db.session.add(nuevo_ticket)
    db.session.commit()

    return {
        "message": "Ticket registrado con éxito (modo offline)",
        "ticket_id": nuevo_ticket.id,
        "status": nuevo_ticket.estado,
    }
```

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

```python
def get_reviews():
    """Simula la obtención de reseñas desde la API externa"""
    if Config.REVIEWS_API_URL:
        url = f"{Config.REVIEWS_API_URL}/reviews"
        response = requests.get(url)
        return response.json() if response.status_code == 200 else None
    else:
        # Simulación hasta que la API esté lista
        return [{"id": 1, "user_id": 2, "rating": 5, "comment": "Sistema excelente"}]
```

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

```
MAIL_SERVER=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=financierastat@gmail.com
MAIL_PASSWORD=TheLegends123
```

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

```python
from flask import Flask, send_from_directory
from extensions import db, migrate, jwt, cors
from config import Config
from dotenv import load_dotenv
import os

app = Flask(__name__)

app.config.from_object(Config)
load_dotenv()

db.init_app(app)
migrate.init_app(app, db)
jwt.init_app(app)
cors.init_app(app)

# Registra Blueprints con prefijo /STAT/api
from routes.auth import auth_bp
from routes.dashboard import dashboard_bp
from routes.categories import categories_bp
from routes.metodospago import payment_bp
from routes.transactions import transactions_bp
from routes.history import history_bp
from routes.integrations import integrations_bp
from routes.admin import admin_bp
from routes.tickets import tickets_bp

app.register_blueprint(auth_bp)
app.register_blueprint(dashboard_bp)
app.register_blueprint(payment_bp)
app.register_blueprint(categories_bp)
app.register_blueprint(transactions_bp)
app.register_blueprint(history_bp)
app.register_blueprint(integrations_bp)
app.register_blueprint(admin_bp)
app.register_blueprint(tickets_bp)

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

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

---

En **config.py**

```python
import os

class Config:
    SECRET_KEY = os.getenv('SECRET_KEY', 'clave_secreta')
    SQLALCHEMY_DATABASE_URI = 'postgresql://postgres:TheLegends123@localhost/GestionFina'
    SQLALCHEMY_TRACK_MODIFICATIONS = False
    JWT_SECRET_KEY = os.getenv('JWT_SECRET_KEY', 'supersecreto')


    # URLs de APIs externas (vacías por ahora)
    TICKETS_API_URL = None  # Se llenará cuando esté disponible
    REVIEWS_API_URL = None  # Se llenará cuando esté disponible

    MAIL_SERVER = os.getenv("MAIL_SERVER", "smtp.gmail.com")
    MAIL_PORT = int(os.getenv("MAIL_PORT", 587))
    MAIL_USERNAME = os.getenv("MAIL_USERNAME")  # No pongas aquí directamente el correo
    MAIL_PASSWORD = os.getenv("MAIL_PASSWORD")  # No pongas aquí directamente la contraseña
    MAIL_DEFAULT_SENDER = os.getenv("MAIL_DEFAULT_SENDER", MAIL_USERNAME)
```

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:**

```python
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_jwt_extended import JWTManager
from flask_cors import CORS

db = SQLAlchemy()
migrate = Migrate()
jwt = JWTManager()
cors = CORS()
```

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:

```python
class Usuario(db.Model):
    __tablename__ = 'usuarios'
    legajo = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(255), nullable=False)
    dni = db.Column(db.String(20), unique=True, nullable=False)
    usuario = db.Column(db.String(255), unique=True, nullable=False)
    email = db.Column(db.String(255), unique=True, nullable=False)
    contrasena = db.Column(db.String(255), nullable=False)
    rol = db.Column(db.String(50), nullable=False, default='usuario')

    def set_password(self, password):
        self.contrasena = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.contrasena, password)
```

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.

```python
class Categoria(db.Model):
    __tablename__ = 'categorias'
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(255), nullable=False)
    tipo = db.Column(db.String(50), nullable=False)  # "Ingreso" o "Egreso"
    @classmethod
    def obtener_por_tipo(cls, tipo):
        """Devuelve todas las categorías de un tipo específico."""
        return cls.query.filter_by(tipo=tipo).all()

    @classmethod
    def crear(cls, nombre, tipo):
        """Crea una nueva categoría."""
        nueva_categoria = cls(nombre=nombre, tipo=tipo)
        db.session.add(nueva_categoria)
        db.session.commit()
        return nueva_categoria
```

- **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().**

```python
class MetodoPago(db.Model):
    __tablename__ = 'metodopago'
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(255), nullable=False)
```

```python
class Ingreso(db.Model):
    __tablename__ = 'ingresos'
    codigo = db.Column(db.Integer, primary_key=True)
    idcategoria = db.Column(db.Integer, db.ForeignKey('categorias.id'), nullable=False)
    legajousuario = db.Column(db.Integer, db.ForeignKey('usuarios.legajo'), nullable=False)
    descripcion = db.Column(db.String(255))
    fecha = db.Column(db.Date, default=datetime.utcnow)
    importe = db.Column(db.Float, nullable=False)

    categoria = db.relationship('Categoria', backref='ingresos')
    usuario = db.relationship('Usuario', backref='ingresos')
    @property
    def categoria_nombre(self):
        return self.categoria.nombre if self.categoria else "Sin categoría"

class Egreso(db.Model):
    __tablename__ = 'egresos'
    codigo = db.Column(db.Integer, primary_key=True)
    idcategoria = db.Column(db.Integer, db.ForeignKey('categorias.id'), nullable=False)
    legajousuario = db.Column(db.Integer, db.ForeignKey('usuarios.legajo'), nullable=False)
    descripcion = db.Column(db.String(255))
    fecha = db.Column(db.Date, default=datetime.utcnow)
    importe = db.Column(db.Float, nullable=False)
    idMetodoPago = db.Column(db.Integer, db.ForeignKey('metodopago.id'), nullable=False)

    categoria = db.relationship('Categoria', backref='egresos')
    usuario = db.relationship('Usuario', backref='egresos')
    metodo_pago = db.relationship('MetodoPago', backref='egresos')

    @property
    def categoria_nombre(self):
        return self.categoria.nombre if self.categoria else "Sin categoría"
```

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".**

```python
class Reporte(db.Model):
    __tablename__ = 'reportes'

    id = db.Column(db.Integer, primary_key=True)
    legajousuario = db.Column(db.Integer, db.ForeignKey('usuarios.legajo'), nullable=False)  # Usuario que reporta
    descripcion = db.Column(db.Text, nullable=False)  # Descripción del problema
    fecha_reporte = db.Column(db.DateTime, default=datetime.utcnow)  # Fecha del reporte
    estado = db.Column(db.String(50), default="pendiente")  # Estado del reporte: "pendiente", "en proceso", "resuelto"

    usuario = db.relationship('Usuario', backref='reportes')  # Relación con el usuario que reporta

    @classmethod
    def crear(cls, legajousuario, descripcion):
        """Crea un nuevo reporte y lo guarda en la base de datos."""
        nuevo_reporte = cls(legajousuario=legajousuario, descripcion=descripcion)
        db.session.add(nuevo_reporte)
        db.session.commit()
        return nuevo_reporte

    @classmethod
    def obtener_por_usuario(cls, legajousuario):
        """Obtiene todos los reportes de un usuario."""
        return cls.query.filter_by(legajousuario=legajousuario).all()

    @classmethod
    def actualizar_estado(cls, reporte_id, nuevo_estado):
        """Actualiza el estado de un reporte."""
        reporte = cls.query.get(reporte_id)
        if reporte:
            reporte.estado = nuevo_estado
            db.session.commit()
            return reporte
        return None
```

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

---

En **requirements.txt**

```
alembic==1.13.2
asgiref==3.8.1
blinker==1.8.2
certifi==2025.1.31
charset-normalizer==3.4.1
click==8.1.7
colorama==0.4.6
contourpy==1.3.1
cycler==0.12.1
Django==5.1.2
et_xmlfile==2.0.0
Flask==3.0.3
Flask-Admin==1.6.1
Flask-Cors==5.0.0
Flask-JWT-Extended==4.7.1
Flask-Login==0.6.3
Flask-Migrate==4.0.7
Flask-SQLAlchemy==3.1.1
Flask-WTF==1.2.1
fonttools==4.55.0
greenlet==3.0.3
```

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.