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.

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.

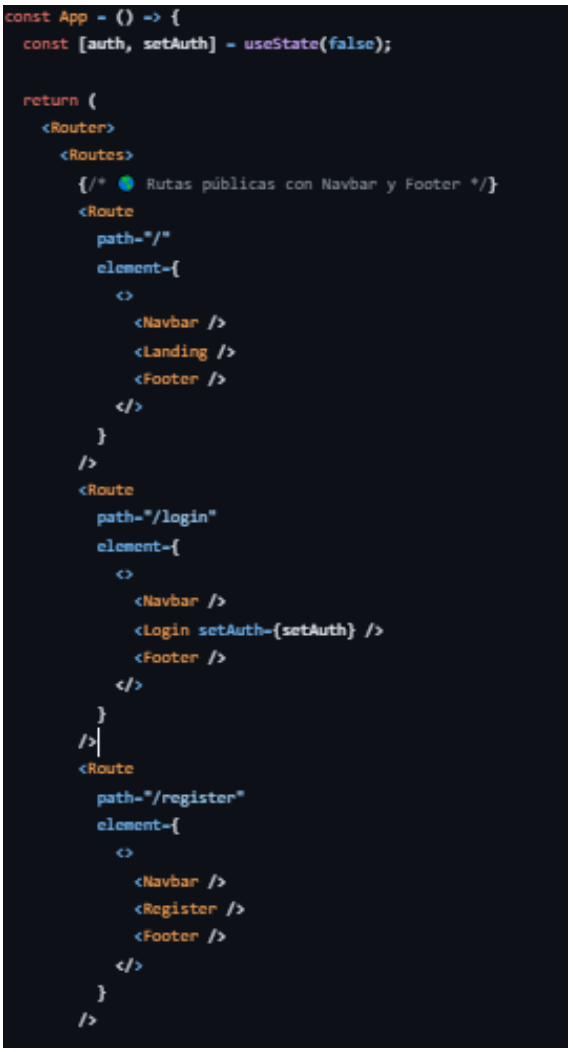
Este estado se actualiza en el componente `Login` a través de `setAuth`, lo que permite cambiar su valor cuando el usuario se autentica correctamente.

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

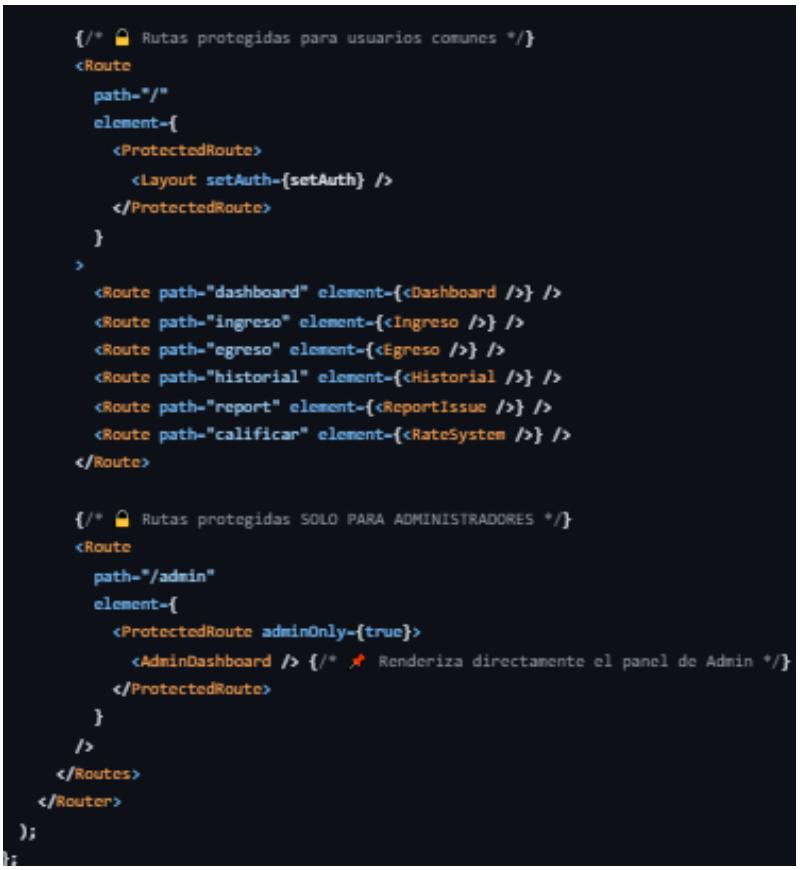
Las rutas públicas incluyen `Landing`, `Login` y `Register`, que muestran `Navbar` y `Footer`.

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



```
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 />
            </>
          }
        />
```



```
        {/* 🔒 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("/");
  };
```

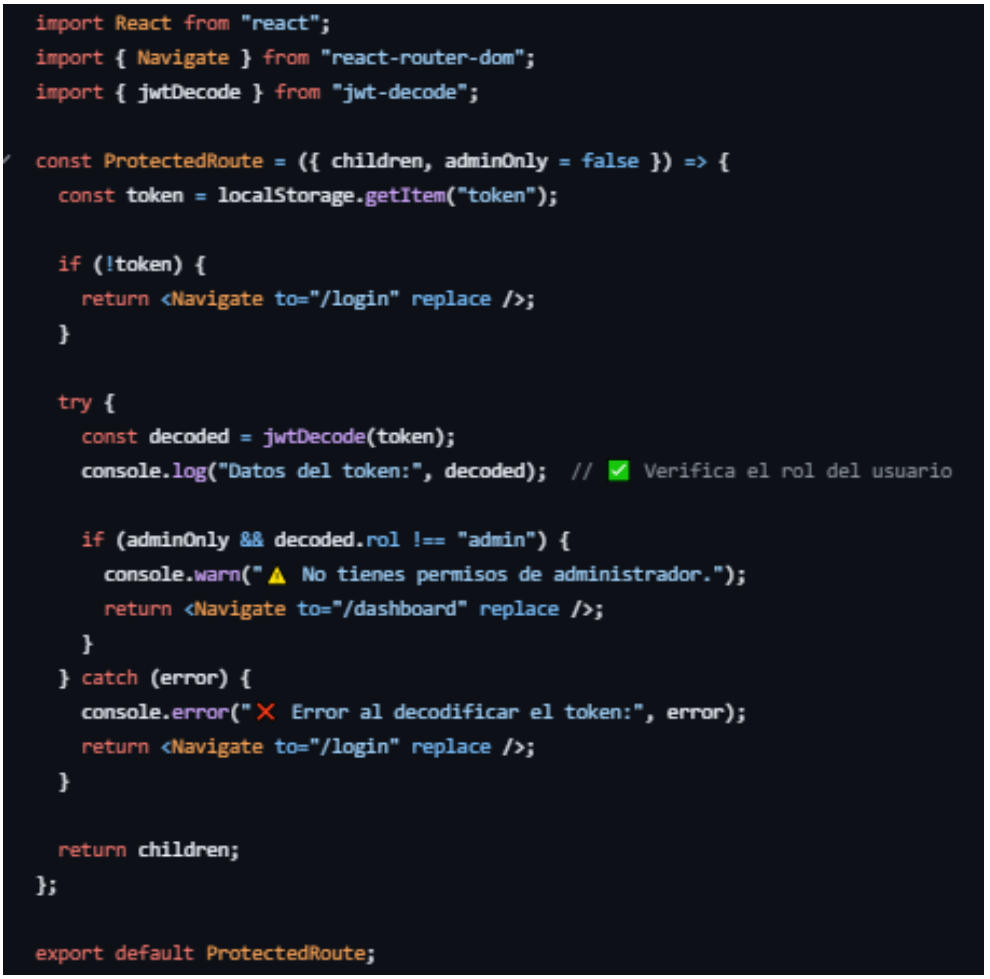
```jsx
  return (
    <aside className="sidebar">
      <h1><NavLink to="/">STAT</NavLink></h1>
      <nav>
        <ul>
          <li><NavLink to="/dashboard">Dashboard</NavLink></li>
          <li><NavLink to="/ingreso">Ingresar Ingreso</NavLink></li>
          <li><NavLink to="/egreso">Ingresar Egreso</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>
```

```
            {/* 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.

```
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

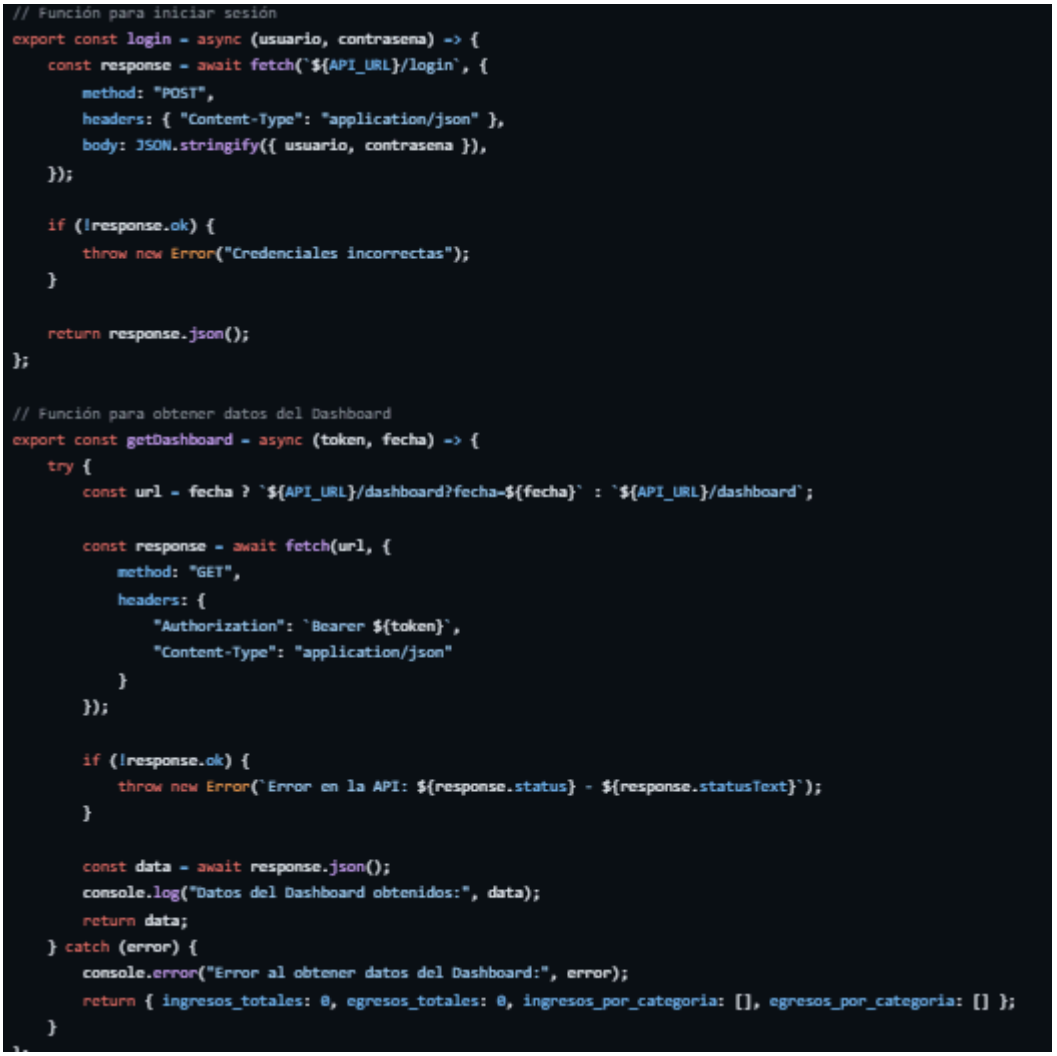
Las funciones **`register`** y **`login`** envían solicitudes POST a la API para registrar un nuevo usuario y autenticar a un usuario existente. El **`login`** devuelve un token de autorización que se utiliza para validar solicitudes.

`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.

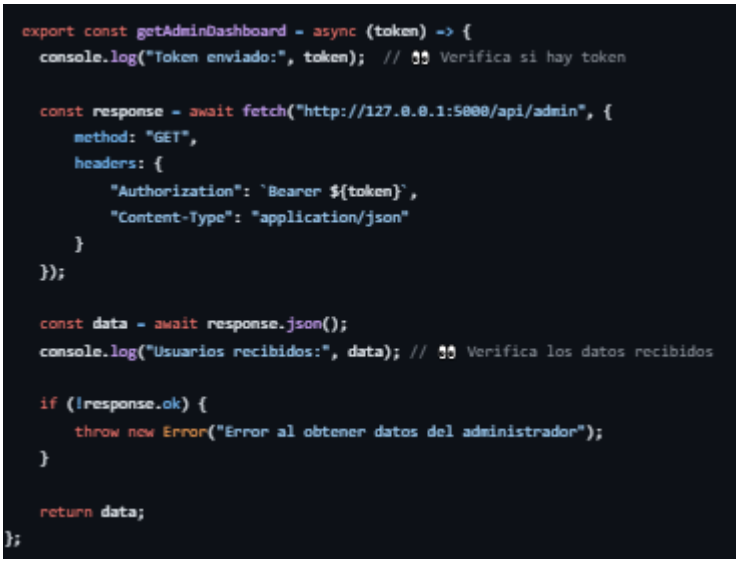
`getPaymentMethods` obtiene una lista de métodos de pago disponibles a través de la API, mientras que **`sendRating`** permite al usuario enviar calificaciones para servicios, enviando los datos correspondientes en el cuerpo de la solicitud.



```
// 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();
};
```

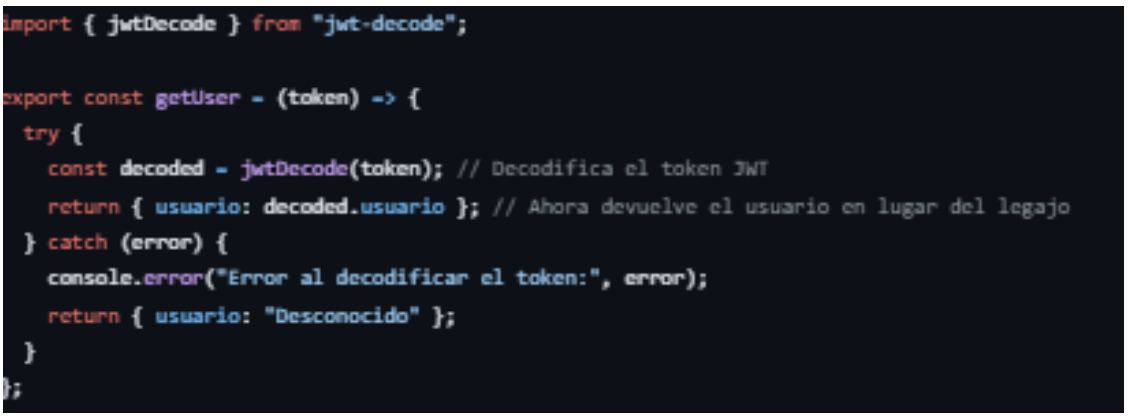


```
export const getAdminDashboard = async (token) => {
  console.log("Token enviado:", token);  // 👀 Verifica si hay token

  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" };
  }
};
```