

El backend de este proyecto está diseñado para gestionar las operaciones y la lógica de negocio necesarias para apoyar la gestión financiera de los usuarios en Mérida. Utiliza una arquitectura basada en el modelo MVC (Modelo-Vista-Controlador), que permite mantener el código organizado y modular. Este enfoque facilita la escalabilidad, el mantenimiento y la integración de futuras funcionalidades.

A continuación, se describe la estructura del backend y la función de cada carpeta y archivo en el proyecto:

Estructura del backend:

- config/: Esta carpeta contiene archivos de configuración del proyecto, como la conexión a la base de datos. Estos archivos permiten una configuración centralizada de aspectos clave, lo cual facilita el manejo de diferentes entornos.
- controllers/: Aquí se encuentran los controladores del proyecto. Los controladores se encargan de gestionar las peticiones del cliente y de interactuar con los modelos para procesar y devolver la información correspondiente. Cada controlador está organizado por funcionalidades o módulos específicos, lo cual mejora la organización del código.
- middleware/: Los middlewares son funciones que se ejecutan antes de que una solicitud llegue al controlador. Aquí se pueden definir middlewares para la autenticación, validación de datos y manejo de permisos. Los middlewares ayudan a mantener los controladores limpios ya reutilizar código común en varias rutas.
- models/: Esta carpeta contiene los modelos de la base de datos. Los modelos representan las entidades del sistema, y cada archivo de modelo define la estructura de una tabla o colección en la base de datos, junto con los métodos que permiten interactuar con ella. Estos modelos son esenciales para manejar la lógica de acceso y manipulación de datos en el sistema.
- node_modules/: Esta carpeta contiene todos los módulos y dependencias instaladas para el proyecto. Es administrada automáticamente por npm y permite acceder a las librerías necesarias para el funcionamiento del backend.
- routes/: Aquí se encuentran las rutas del proyecto. Cada archivo de rutas define los endpoints de la API y los métodos HTTP correspondientes (GET, POST, PUT, DELETE) que permiten al cliente interactuar con los controladores. Las rutas organizan la estructura de la API, facilitando la adición de nuevas funcionalidades.
- .env: Este archivo contiene las variables de entorno del proyecto, como claves secretas y configuraciones sensibles. Las variables de entorno se utilizan para proteger datos confidenciales y permiten la configuración de parámetros en distintos entornos sin modificar el código fuente.
- .gitignore: Archivo que indica a Git que archivos o carpetas no deben ser versionados. Esto incluye archivos sensibles como .env o la carpeta node_modules/, que se regeneran automáticamente.
- cronJobs.js: Archivo dedicado a la configuración de tareas programadas (cron jobs). Los cron jobs se utilizan para ejecutar tareas periódicas de forma automática, como el envío de recordatorios o la actualización de datos.
- package-lock.json: Este archivo asegura que las dependencias se instalen con versiones específicas, manteniendo la consistencia en las instalaciones del proyecto.
- package.json: Contiene la configuración del proyecto y la lista de dependencias y scripts. Este archivo permite definir los módulos requeridos, scripts de inicio y otra información relevante del proyecto.
- server.js: Archivo principal del servidor. En este archivo se inicializa el servidor y se configuran las rutas, middlewares y otros aspectos clave del backend. Es el punto de entrada del backend y suele estar encargado de escuchar las peticiones en un puerto específico.

```

> config
> controllers
> middleware
> models
> node_modules
> routes
  .env
  .gitignore
  cronJobs.js
  package-lock.json
  package.json
  server.js

```

Controllers

Cada controlador está diseñado para gestionar una parte particular del sistema financiero. Por ejemplo, el gastoController.js se encarga de las operaciones relacionadas con los gastos, mientras que el ingresoController.js maneja los ingresos. Existen controladores específicos para categorías de gastos, metas de ahorro, y recordatorios, cada uno facilitando el acceso a los datos y respondiendo a las peticiones del cliente. Además, se incluyen controladores para notificaciones automáticas y reportes, lo que permite ofrecer al usuario una experiencia completa y organizada.

categoriaGastosController

El controlador categoriaGastosController.js maneja las operaciones relacionadas con las categorías de gastos en el sistema. Su objetivo es permitir la gestión de las categorías que los usuarios pueden utilizar para clasificar y organizar sus gastos.

getCategoriasGastos: Este método obtiene todas las categorías de gastos de la base de datos. Si hay un error durante la consulta, devuelve un mensaje de error. Si la consulta es exitosa, devuelve un código de estado 200 junto con los datos.

```
const getCategoriasGastos = (req, res) => {
  CategoriaGasto.getAll((err, data) => {
    if (err) {
      console.error(`El error es ${err}`);
      return res.status(500).json({ error: "Error al obtener categorias" });
    }
    res.status(200).json(data);
  });
};
```

postCategoriasGastos: Este método crea una nueva categoría de gasto. Verifica que el campo nombre_categoria esté presente. Si falta, devuelve un error 400. Si está presente, intenta crear la categoría en la base de datos y devuelve un mensaje de éxito en caso de éxito.

```
if (!nombre_categoria) {
  return res
    .status(400)
    .json({ error: "El nombre de la categoria es requerido" });
}

const nuevaCategoriaGasto = {
  nombre_categoria,
};

CategoriaGasto.create(nuevaCategoriaGasto, (err, data) => {
  if (err) {
    res.status(500).json({ error: "Error al crear la categoria meta" });
    console.error(`El error es ${err}`);
  }

  res.status(201).json({
    message: "Categoria meta creada exitosamente",
  });
});
```

constputCategoriasGastos: Es un controlador que recibe dos parámetros, req y res, que representan la solicitud y la respuesta HTTP respectivamente. Este controlador se encarga de validar la entrada, procesar la solicitud de actualización, y enviar una respuesta adecuada al cliente.

```
.json({ error: "ID de categoria o nombre no proporcionado" });

let updateData = { nombre_categoria };

CategoriaGasto.update(categoria_gasto_id, updateData, (err, result) => {
  if (err) {
    res
      .status(500)
      .json({ error: "Error al actualizar la categoria", detalles: err });
    console.error(`El error es ${err}`);
  }
});
```

Backend

Module.exports: getcategoriasGastos, postcategoriasGastos, putcategoriasGastos y deletecategoriasGastos. Estas funciones representan operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre categorías de gastos.

```
module.exports = {
  getcategoriasGastos,
  postcategoriasGastos,
  putcategoriasGastos,
  deletecategoriasGastos
};
```

categoriaMetasController

El controlador categoriaMetasController.js maneja las operaciones relacionadas con las categorías de metas en el sistema. Su objetivo es permitir la gestión de las categorías que los usuarios pueden utilizar para clasificar y organizar sus metas.

getCategoriesMetas: Este método obtiene todas las categorías de metas de la base de datos. Si hay un error durante la consulta, devuelve un mensaje de error. Si la consulta es exitosa, devuelve un código de estado 200 junto con los datos.

```
const getCategoriesMetas = (req, res) => {
  categoriaMeta.getAll((err, data) => {
    if (err) {
      console.error(`El error es ${err}`);
      return res.status(500).json({ error: "Error al obtener categorías" });
    }
    res.status(200).json(data);
  });
};
```

postCategoriesMetas: Este método crea una nueva categoría de meta. Verifica que el campo nombre_categoria esté presente. Si falta, devuelve un error 400. Si está presente, intenta crear la categoría en la base de datos y devuelve un mensaje de éxito en caso de éxito.

```
if (!nombre_categoria) {
  return res
    .status(400)
    .json({ error: "El nombre de la categoría es requerido" });
}

const nuevaCategoriaMeta = {
  nombre_categoria,
};

categoriaMeta.create(nuevaCategoriaMeta, (err, data) => {
  if (err) {
    res.status(500).json({ error: "Error al crear la categoría meta" });
    console.error(`El error es ${err}`);
  }

  res.status(201).json({
    message: "Categoría meta creada exitosamente",
  });
});
```

putCategoriesMetas: Es un controlador que recibe dos parámetros, req y res, que representan la solicitud y la respuesta HTTP respectivamente. Este controlador se encarga de validar la entrada, procesar la solicitud de actualización y enviar una respuesta adecuada al cliente. Verifica que tanto el categoria_meta_id como el nombre_categoria estén presentes antes de intentar actualizar la categoría. Si ocurre un error, devuelve un mensaje de error; si la actualización es exitosa, devuelve un mensaje de confirmación.

```
return res
  .status(400)
  .json({ error: "ID de categoría o nombre no proporcionado" });
}

let updateData = { nombre_categoria };

categoriaMeta.update(categoria_meta_id, updateData, (err, result) => {
  if (err) {
    res
      .status(500)
      .json({ error: "Error al actualizar la categoría", detalles: err });
    console.error(`El error es ${err}`);
  }

  res.status(200).json({
    message: "Categoría meta actualizada exitosamente",
  });
});
```

Backend

`deleteCategoriasMetas`: Es un controlador que recibe dos parámetros, `req` y `res`, que representan la solicitud y la respuesta HTTP respectivamente. Este controlador se encarga de validar la entrada, realizar la operación de eliminación y responder de forma adecuada al cliente. Verifica que el `categoria_meta_id` esté presente antes de intentar eliminar la categoría. Si ocurre un error o si la categoría no se encuentra, devuelve mensajes de error; si la eliminación es exitosa, devuelve un mensaje de confirmación.

```
if (!categoria_meta_id) {
  return res.status(400).json({ error: "ID de categoría no proporcionado" });
}

categoriaMeta.delete(categoria_meta_id, (err, data) => {
  if (err) {
    console.error(`Error al eliminar la categoría: ${err}`);
    return res.status(500).json({ error: "Error al eliminar la categoría" });
  }

  if (data.affectedRows === 0) {
    return res.status(404).json({ message: "Categoria no encontrada" });
  }

  res.status(200).json({ message: "Categoria eliminada exitosamente" });
});
```

`Module.exports`: `getCategoriasMetas`, `postCategoriasMetas`, `putCategoriasMetas` y `deleteCategoriasMetas`. Estas funciones representan operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre categorías de metas.

```
module.exports = {
  getCategoriasMetas,
  postCategoriasMetas,
  putCategoriasMetas,
  deleteCategoriasMetas
};
```

gastoController

El controlador `gastoController.js` maneja las operaciones relacionadas con los gastos en el sistema. Su objetivo es permitir la gestión de los gastos que los usuarios registran, facilitando así su seguimiento y análisis.

`getGastos`: Este método obtiene todos los gastos registrados en la base de datos. Si hay un error durante la consulta, devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, devuelve un código de estado 200 junto con los datos de los gastos.

```
const getGastos = (req, res) => {
  Gasto.getAll((err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener los Gastos" });
    }
    res.status(200).json(data);
  });
};
```

Backend

getGastoById: Este método obtiene los gastos asociados a un usuario específico, identificado por su usuario_id. Si ocurre un error durante la consulta, devuelve un mensaje de error con el código de estado 500. Si se encuentran gastos, se devuelve un código de estado 200 junto con los datos de los gastos del usuario.

```
const getGastoById = (req, res) => {
  const { usuario_id } = req.params;

  Gasto.getByUserId(usuario_id, (err, data) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener los gastos del usuario" });
    }
    res.status(200).json(data);
  });
};
```

getGastoByCategoria: Este método obtiene los gastos asociados a una categoría específica, identificada por categoria_gasto_id. En caso de error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los gastos de la categoría solicitada.

```
const getGastoByCategoria = (req, res) => {
  const { categoria_gasto_id } = req.params;

  Gasto.getByCategoriaId(categoria_gasto_id, (err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener los gastos" });
    }
    res.status(200).json(data);
  });
};
```

postGasto: Este método permite crear un nuevo gasto. Extrae monto, categoria_gasto_id y descripción del cuerpo de la solicitud. Se obtiene el usuario_id del token del usuario. Si alguno de estos campos falta, devuelve un error 400. Si todos los campos están presentes, intenta crear el gasto en la base de datos y devuelve un mensaje de éxito si se realiza correctamente.

```
const postGasto = (req, res) => {
  const { monto, categoria_gasto_id, descripcion } = req.body;
  const usuario_id = req.user.id; // Obtenemos usuario_id del token

  const nuevoGasto = {
    monto,
    categoria_gasto_id,
    descripcion,
  };

  Gasto.create(usuario_id, nuevoGasto, (err, data) => {
    if (err) {
      console.error(`Error al crear gasto: ${err}`);
      return res.status(500).json({ error: "Error al crear gasto" });
    }

    res.status(201).json({
      message: "Gasto añadido exitosamente",
      data: data,
    });
  });
};
```

Backend

putGasto: Este método actualiza un gasto existente. Recibe el id_gasto desde los parámetros de la solicitud y extrae monto, categoria_gasto_id y descripción del cuerpo. También obtiene el usuario_id del token en la solicitud. Si falta alguno de estos campos, se devuelve un error 400. intenta actualizar el gasto en la base de datos y maneja posibles errores, incluyendo la falta de permisos para editar el gasto o su inexistencia.

```
        return res.status(400).json({ error: "Todos los campos son obligatorios" });
    }

    const updateData = { monto, categoria_gasto_id, descripcion };

    Gasto.updateData(id_gasto, usuario_id, updateData, (err, result) => {
        if (err) {
            return res.status(500).json({ error: "Error al actualizar gasto", detalles: err });
        }

        if (result.affectedRows === 0) {
            return res.status(403).json({ message: "No tienes permiso para editar este gasto o no existe" });
        }
    });
}
```

deleteGasto: Este método elimina un gasto específico, identificado por id_gasto. Si ocurre un error durante la operación, se devuelve un mensaje de error con el código de estado 500. Si el gasto no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
Gasto.delete(id_gasto, (err, data) => {
    if (err) {
        console.error(`Error al eliminar el Gasto: ${err}`);
        return res.status(500).json({ error: "Error al eliminar el Gasto" });
    }

    if (data.affectedRows === 0) {
        return res.status(404).json({ message: "Gasto no encontrado" });
    }

    res.status(200).json({ message: "Gasto eliminado exitosamente" });
});
```

ingresoController

El controlador ingresoController.js maneja las operaciones relacionadas con los ingresos en el sistema. Su objetivo es permitir la gestión de los ingresos que los usuarios registran, facilitando así su seguimiento y análisis.

getIngresoByUserId: Este método obtiene los ingresos asociados a un usuario específico, identificado por su usuario_id. Si ocurre un error durante la consulta, se registra en la consola y se devuelve un mensaje de error con el código de estado 500. Si no se encuentran ingresos para el usuario, se devuelve un código de estado 404 con un mensaje informativo. Si se encuentran ingresos, se devuelve un código de estado 200 junto con los datos de los ingresos.

```
const getIngresoByUserId = (req, res) => {
    const { usuario_id } = req.params;
    console.log("Buscando ingreso para usuario_id:", usuario_id); // Debug

    Ingreso.getIngresoByUserId(usuario_id, (err, data) => {
        if (err) {
            console.error("Error en la consulta:", err); // Debug de errores
            return res.status(500).json({ error: "Error al obtener el ingreso" });
        }
        if (!data) {
            return res.status(404).json({ message: "Ingreso no encontrado para este usuario" });
        }
        res.status(200).json({ ingresos: data.ingresos });
    });
};
```

Backend

updateIngreso: Este método actualiza los ingresos de un usuario específico. Recibe el usuario_id desde los parámetros de la solicitud e ingresos desde el cuerpo. Si el campo ingresos es nulo o es nulo, se devuelve un error 400. Si se intenta actualizar y ocurre un error, se devuelve un mensaje de error con el código de estado 500. Si no se encuentra al usuario correspondiente, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la actualización.

```
const updateIngreso = (req, res) => {
  const { usuario_id } = req.params;
  const { ingresos } = req.body;

  if (ingresos == null) {
    return res.status(400).json({ error: "El ingreso es obligatorio" });
  }

  Ingreso.updateIngreso(usuario_id, ingresos, (err, result) => {
    if (err) {
      return res.status(500).json({ error: "Error al actualizar el ingreso" });
    }
    if (result.affectedRows === 0) {
      return res.status(404).json({ message: "Usuario no encontrado" });
    }
    res.status(200).json({ message: "Ingreso actualizado correctamente" });
  });
};
```

metaAhorroController

El controlador metaAhorroController.js maneja las operaciones relacionadas con las metas de ahorro en el sistema. Su objetivo es permitir la gestión de las metas que los usuarios establecen para su ahorro, facilitando así su seguimiento y análisis.

getMetasAhorro: Este método obtiene todas las metas de ahorro registradas en la base de datos. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de las metas.

```
const getMetasAhorro = (req, res) => {
  MetaAhorro.getAll((err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener las metas" });
    }
    res.status(200).json(data);
  });
};
```

getMetasByUserId: Este método obtiene las metas de ahorro asociadas a un usuario específico, identificado por su usuario_id. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de las metas del usuario.

```
const { usuario_id } = req.params;

MetaAhorro.getByUserId(usuario_id, (err, data) => {
  if (err) {
    return res
      .status(500)
      .json({ error: "Error al obtener las metas del usuario" });
  }
  res.status(200).json(data);
});
```

Backend

getMetasByCategoria: Este método obtiene las metas de ahorro asociadas a una categoría específica, identificada por categoria_meta_id. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de las metas de la categoría a solicitada.

```
const getMetasByCategoria = (req, res) => {
  const { categoria_meta_id } = req.params;

  MetaAhorro.getByCategoriaId(categoria_meta_id, (err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener las metas" });
    }
    res.status(200).json(data);
  });
};
```

postMetaAhorro: Este método permite crear una nueva meta de ahorro. Extrae los campos nombre_meta, monto_objetivo, fecha_limite, descripcion, y categoria_meta_id del cuerpo de la solicitud. También obtiene el usuario_id del token del usuario. Si falta alguno de los campos obligatorios, se devuelve un error 400. Si todos los campos están presentes, intenta crear la meta en la base de datos y devuelve un mensaje de éxito si se realiza correctamente

```
const postMetaAhorro = (req, res) => {
  const {
    nombre_meta,
    monto_objetivo,
    fecha_limite,
    descripcion,
    categoria_meta_id,
  } = req.body;
  const usuario_id = req.userId; // Obtenemos usuario_id del token

  // Validación de campos obligatorios
  if (!nombre_meta || !monto_objetivo || !fecha_limite || !categoria_meta_id) {
    return res.status(400).json({
      error: "Todos los campos obligatorios deben ser proporcionados",
    });
}
```

putMetaAhorro: Este método actualiza una meta de ahorro existente. Recibe el meta_id desde los parámetros de la solicitud y extrae nombre_meta, monto_objetivo, monto_actual, descripcion, y estado_de_meta del cuerpo de la solicitud. Verifica que el meta_id sea válido y que haya datos para actualizar. Si no hay datos, se devuelve un error 400. Llama a la función de actualización en el modelo y maneja posibles errores, incluyendo la falta de permisos para editar la meta o su inexistencia.

```
const {
  nombre_meta,
  monto_objetivo,
  monto_actual,
  descripcion,
  estado_de_meta,
} = req.body;

// Verificar que meta_id sea un número válido
if (!meta_id || isNaN(meta_id)) {
  return res.status(400).json({ error: "ID de la meta no es válido" });
}

// Filtrar solo los campos que están presentes en el cuerpo de la solicitud
let updateData = {};

if (nombre_meta) updateData.nombre_meta = nombre_meta;
if (monto_objetivo) updateData.monto_objetivo = monto_objetivo;
if (monto_actual) updateData.monto_actual = monto_actual;
if (descripcion) updateData.descripcion = descripcion;
if (estado_de_meta) updateData.estado_de_meta = estado_de_meta;

// Verificar si hay datos para actualizar
if (Object.keys(updateData).length === 0) {
  return res.status(400).json({ error: "No hay datos para actualizar" });
}
```

Backend

putMontoActual: Este método actualiza el monto actual de una meta de ahorro específica, identificada por meta_id. Extrae el montoAdicional del cuerpo de la solicitud. Si el monto adicional no es positivo, devuelve un error 400. Si la operación de actualización tiene éxito, devuelve un mensaje de confirmación.

```
const putMontoActual = (req, res) => {
  const { meta_id } = req.params;
  const { montoAdicional } = req.body;

  if (!montoAdicional || montoAdicional <= 0) {
    return res
      .status(400)
      .json({ error: "El monto adicional debe ser positivo" });
  }

  MetaAhorro.updateMontoActual(meta_id, montoAdicional, (err, result) => {
    if (err) {
      console.error(`Error al actualizar el monto de la meta: ${err}`);
      return res.status(500).json({ error: "Error al actualizar el monto" });
    }
    res.status(200).json({ message: "Monto actualizado correctamente" });
  });
};
```

deleteMetaAhorro: Este método elimina una meta de ahorro específica, identificada por meta_id. Si ocurre un error durante la operación, se devuelve un mensaje de error con el código de estado 500. Si la meta no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
const deleteMetaAhorro = (req, res) => {
  const { meta_id } = req.params;

  MetaAhorro.delete(meta_id, (err, data) => {
    if (err) {
      console.error(`Error al eliminar la meta: ${err}`);
      return res.status(500).json({ error: "Error al eliminar la meta" });
    }

    if (data.affectedRows === 0) {
      return res.status(404).json({ message: "Meta no encontrada" });
    }

    res.status(200).json({ message: "Meta eliminada exitosamente" });
  });
};
```

notificacionController

El controlador notificacionController.js maneja las operaciones relacionadas con las notificaciones en el sistema. Su objetivo es permitir la generación y gestión de notificaciones basadas en el seguimiento de gastos, metas y recordatorios de los usuarios.

generarNotificacionesDeGastos: Esta función genera notificaciones para cada usuario que ha alcanzado el 20% de sus ingresos mensuales en gastos. Obtiene la lista de usuarios y, para cada usuario con ingresos, calcula el total de gastos del mes. Si el porcentaje de gastos es igual o mayor al 20%, crea una notificación informando al usuario.

```
async function generarNotificacionesDeGastos() {
  const usuarios = await obtenerUsuarios();

  for (const usuario of usuarios) {
    if (!usuario.ingresos) continue;

    const totalGastos = await obtenerGastosDelMes(usuario.usuario_id);
    const ingresoMensual = usuario.ingresos;
    const porcentajeGasto = (totalGastos / ingresoMensual) * 100;

    if (porcentajeGasto >= 20) {
      await Notificacion.create({
        usuario_id: usuario.usuario_id,
        tipo: "gastos",
        mensaje: "Has alcanzado el 20% de tu ingreso mensual en gastos."
      });
    }
  }
}
```

Backend

generarNotificacionesDeMetas: Esta función genera notificaciones para los usuarios que están cerca de completar sus metas de ahorro. Recorre todas las metas y calcula el porcentaje alcanzado. Si este porcentaje es igual o mayor al 90%, se crea una notificación que alerta al usuario sobre el progreso de su meta.

```
async function generarNotificacionesDeMetas() {
  const metas = await obtenerMetas();

  metas.forEach(async (meta) => {
    const porcentajeMeta = (meta.monto_actual / meta.monto_objetivo) * 100;

    if (porcentajeMeta >= 90) {
      await Notificacion.create({
        usuario_id: meta.usuario_id,
        tipo: "meta",
        mensaje: `Estás cerca de completar tu meta: ${meta.nombre_meta}.`,
      });
    }
  });
}
```

generarNotificacionesDeMetasVencidas: Esta función genera notificaciones para las metas de ahorro que han vencido. Obtiene las metas vencidas y crea una notificación para el usuario correspondiente, informando que no ha cumplido la meta y proporcionando un mensaje de aliento. Además, elimina la meta vencida de la base de datos.

```
const metasVencidas = await obtenerMetasVencidas();

metasVencidas.forEach(async (meta) => {
  const porcentajeAhorrado = (meta.monto_actual / meta.monto_objetivo) * 100;

  const mensaje = `Lo sentimos, no has podido cumplir la meta: ${meta.nombre_meta}. Solo lograste ahorrar ${meta.monto_actual}. No te desanimes, sigue adelante!`;

  // Enviar notificación al usuario
  await Notificacion.create({
    usuario_id: meta.usuario_id,
    tipo: "meta_vencida",
    mensaje,
  });

  // Eliminar la meta vencida de la base de datos
  await eliminarMeta(meta.meta_id);
});

}
```

generarNotificacionesDeRecordatorios: Esta función genera notificaciones de recordatorios próximos. Obtiene los recordatorios que están programados para el día actual o el siguiente. Dependiendo de la proximidad de la fecha, se entra un mensaje diferente al usuario (hoy, mañana o en el futuro cercano).

```
const fechaRecordatorio = new Date(recordatorio.fecha_recordatorio);

const diferenciaDias = Math.ceil(
  (fechaRecordatorio - fechaHoy) / (1000 * 60 * 60 * 24)
);

let mensaje = `Recordatorio próximo: ${recordatorio.descripcion} el ${recordatorio.fecha_recordatorio}.`;

if (diferenciaDias === 1) {
  mensaje = `Tu recordatorio está programado para mañana: ${recordatorio.descripcion}`;
} else if (diferenciaDias === 0) {
  mensaje = `¡Hoy es el día para: ${recordatorio.descripcion}!`;
}

if (diferenciaDias <= 1) {
  await Notificacion.create({
    usuario_id: recordatorio.usuario_id,
    tipo: "recordatorio",
    mensaje,
  });
}
```

Backend

eliminarNotificacionesViejas: Esta función elimina las notificaciones que han estado en la base de datos por más de un día. Utiliza una consulta SQL para eliminar las notificaciones que cumplen con este criterio y maneja posibles errores durante la operación.

```
async function eliminarNotificacionesViejas() {
  const query = `DELETE FROM notificaciones WHERE fecha < NOW() - INTERVAL 1 DAY`;
  db.query(query, (err, result) => {
    if (err) {
      console.error(`Error al eliminar notificaciones viejas: ${err}`);
    } else {
      console.log(`Notificaciones viejas eliminadas: ${result.affectedRows}`);
    }
  });
}
```

Funciones Auxiliares

obtenerMetasVencidas: Obtiene las metas de ahorro que han vencido, es decir, aquellas cuya fecha límite es anterior a la fecha actual y cuyo monto actual es menor que el objetivo. Devuelve una lista de metas vencidas.

```
return new Promise((resolve, reject) => {
  const query = `
    SELECT meta_id, usuario_id, nombre_meta, monto_objetivo, monto_actual, fecha_limite
    FROM metas_de_ahorro
    WHERE fecha_limite < CURDATE() AND monto_actual < monto_objetivo
  `;
  db.query(query, (err, results) => {
    if (err) return reject(err);
    resolve(results);
  });
});
```

eliminarMeta: Elimina una meta específica de la base de datos identificada por su ID. Devuelve el resultado de la operación de eliminación.

```
const eliminarMeta = async (metaId) => {
  return new Promise((resolve, reject) => {
    const query = `DELETE FROM metas_de_ahorro WHERE meta_id = ?`;
    db.query(query, [metaId], (err, result) => {
      if (err) return reject(err);
      resolve(result);
    });
  });
};
```

Backend

obtenerUsuarios: Obtiene todos los usuarios de la base de datos, incluyendo sus usuarios_id e ingresos. Devuelve una lista de usuarios.

```
const obtenerUsuarios = async () => {
  return new Promise((resolve, reject) => {
    const query = "SELECT usuario_id, ingresos FROM usuarios";
    db.query(query, (err, results) => {
      if (err) return reject(err);
      resolve(results);
    });
  });
};
```

obtenerGastosDelMes: Calcula el total de gastos de un usuario para el mes actual. Devuelve el monto total de los gastos, considerando únicamente aquellos registrados en el mes y año actuales.

```
const obtenerGastosDelMes = async (usuarioId) => {
  return new Promise((resolve, reject) => {
    const query = `
      SELECT COALESCE(SUM(monto), 0) AS total
      FROM gastos
      WHERE usuario_id = ? AND MONTH(fecha) = MONTH(CURDATE()) AND YEAR(fecha) = YEAR(CURDATE())
    `;
    db.query(query, [usuarioId], (err, results) => {
      if (err) return reject(err);
      resolve(results[0].total);
    });
  });
};
```

obtenerMetas: Obtiene todas las metas de ahorro registradas en la base de datos. Devuelve una lista de metas con detalles como el meta_id, usuario_id, nombre_meta, monto_objetivo, monto_actual y fecha_limite.

```
return new Promise((resolve, reject) => {
  const query = `
    SELECT meta_id, usuario_id, nombre_meta, monto_objetivo, monto_actual, fecha_limite
    FROM metas_de_ahorro
  `;
  db.query(query, (err, results) => {
    if (err) return reject(err);
    resolve(results);
  });
});
```

obtenerRecordatoriosProximos

Obtiene los recordatorios que están programados para el día actual o el día siguiente. Devuelve una lista de recordatorios próximos.

```
const obtenerRecordatoriosProximos = async () => {
  return new Promise((resolve, reject) => {
    const query = `
      SELECT recordatorio_id, usuario_id, descripcion, fecha_recordatorio
      FROM recordatorios
      WHERE fecha_recordatorio BETWEEN CURDATE() AND DATE_ADD(CURDATE(), INTERVAL 1 DAY)
    `;
    db.query(query, (err, results) => {
      if (err) return reject(err);
      resolve(results);
    });
  });
};
```

notificacionController

El controlador notificacionController.js maneja las operaciones relacionadas con las notificaciones en el sistema. Su objetivo es permitir la gestión de las notificaciones que se envían a los usuarios, facilitando así su seguimiento y manejo.

getNotificacionesByUserId: Este método obtiene todas las notificaciones asociadas a un usuario específico, identificado por su `usuario_id`. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de las notificaciones.

```
const getNotificacionesByUserId = (req, res) => {
  const { usuario_id } = req.params;

  Notificacion.getByUserId(usuario_id, (err, data) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener las notificaciones del usuario" });
    }
    res.status(200).json(data);
  });
};
```

postNotificacion: Este método permite crear una nueva notificación. Extrae los campos `usuario_id`, `tipo`, y `mensaje` del cuerpo de la solicitud. Si falta alguno de estos campos, se devuelve un error 400. Si todos los campos están presentes, intenta crear la notificación en la base de datos y devuelve un mensaje de éxito si se realiza correctamente, junto con el ID de la nueva notificación.

```
const postNotificacion = (req, res) => {
  const { usuario_id, tipo, mensaje } = req.body;

  // Validar que los campos obligatorios estén presentes
  if (!usuario_id || !tipo || !mensaje) {
    return res.status(400).json({
      error: "Todos los campos (usuario_id, tipo, mensaje) son obligatorios.",
    });
}
```

putNotificacion: Este método actualiza una notificación existente. Recibe el `notificacion_id` desde los parámetros de la solicitud y extrae `usuario_id`, `tipo`, `mensaje`, y `leída` del cuerpo de la solicitud. Si falta el `notificacion_id`, se devuelve un error 400. Se valida que todos los campos necesarios estén presentes; de no ser así, se devuelve un error 400. Llama a la función de actualización en el modelo y maneja posibles errores, incluyendo la falta de la notificación a actualizar.

```
const putNotificacion = (req, res) => {
  const { notificacion_id } = req.params;
  const { usuario_id, tipo, mensaje, leida } = req.body;

  if (!notificacion_id) {
    return res.status(400).json({
      error: "Se requiere el ID de la notificación para actualizar.",
    });
  }

  if (!usuario_id || !tipo || !mensaje || leida === undefined) {
    return res.status(400).json({
      error: "Todos los campos (usuario_id, tipo, mensaje, leida) son obligatorios.",
    });
  }

  let updateData = {
    usuario_id,
    tipo,
    mensaje,
    leida,
  };

  Notificacion.updateData(notificacion_id, updateData, (err, result) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al actualizar notificación", detalles: err });
    }
  });
};
```

deleteNotificacion: Este método elimina una notificación específica, identificada por `notificacion_id`. Si no se proporciona el ID, se devuelve un error 400. Si ocurre un error durante la operación, se devuelve un mensaje de error con el código de estado 500. Si la notificación no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
const deleteNotificacion = (req, res) => {
  const { notificacion_id } = req.params;

  if (!notificacion_id) [
    return res.status(400).json({
      error: "Se requiere el ID de la notificación para eliminar.",
    });
  ]

  Notificacion.delete(notificacion_id, (err, data) => {
    if (err) {
      console.error(`Error al eliminar la notificación: ${err}`);
      return res
        .status(500)
        .json({ error: "Error al eliminar la notificación" });
    }

    if (data.affectedRows === 0) {
      return res.status(404).json({ message: "Notificación no encontrada" });
    }

    res.status(200).json({ message: "Notificación eliminada exitosamente" });
  });
};
```

recordatorioController

El controlador `recordatorioController.js` maneja las operaciones relacionadas con los recordatorios en el sistema. Su objetivo es permitir la gestión de los recordatorios que los usuarios establecen, facilitando así su seguimiento y cumplimiento.

getRecordatorios: Este método obtiene todos los recordatorios registrados en la base de datos. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de los recordatorios.

```
const getRecordatorios = (req, res) => {
  Recordatorio.getAll((err, data) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener los recordatorios" });
    }
    res.status(200).json(data);
  });
};
```

getRecordatoriosByUserId: Este método obtiene los recordatorios asociados a un usuario específico, identificado por su `usuario_id`. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de los recordatorios del usuario.

```
const getRecordatoriosByUserId = (req, res) => {
  const { usuario_id } = req.params;

  Recordatorio.getByUserId(usuario_id, (err, data) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener los recordatorios del usuario" });
    }
    res.status(200).json(data);
  });
};
```

`postRecordatorio`: Este método permite crear un nuevo recordatorio. Extrae los campos `usuario_id`, `descripcion` y `fecha_recordatorio` del cuerpo de la solicitud. Si falta alguno de estos campos obligatorios, se devuelve un error 400. Si todos los campos están presentes, intenta crear el recordatorio en la base de datos y devuelve un mensaje de éxito si se realiza correctamente, junta con el ID del nuevo recordatorio.

```
const postRecordatorio = (req, res) => {
  const { usuario_id, descripcion, fecha_recordatorio } = req.body;

  // Validar que los campos obligatorios estén presentes
  if (!usuario_id || !descripcion || !fecha_recordatorio) {
    return res.status(400).json({
      error: "Todos los campos (usuario_id, descripcion, fecha_recordatorio) son obligatorios."
    });
}
```

`putRecordatorio`: Este método actualiza un recordatorio existente. Recibe el `recordatorio_id` desde los parámetros de la solicitud y extrae `usuario_id`, `descripcion`, y `fecha_recordatorio` del cuerpo. Si falta el `recordatorio_id`, se devuelve un error 400. Se valida que todos los campos necesarios estén presentes; de no ser así, se devuelve un error 400. Llama a la función de actualización en el modelo y maneja posibles errores, incluyendo la falta del recordatorio a actualizar.

```
const putRecordatorio = (req, res) => {
  const { recordatorio_id } = req.params;
  const { usuario_id, descripcion, fecha_recordatorio } = req.body;

  if (!recordatorio_id) {
    return res.status(400).json({
      error: "Se requiere el ID del recordatorio para actualizar."
    });
  }

  if (!usuario_id || !descripcion || !fecha_recordatorio) {
    return res.status(400).json({
      error: "Todos los campos (usuario_id, descripcion, fecha_recordatorio) son obligatorios."
    });
  }

  let updateData = {
    usuario_id,
    descripcion,
    fecha_recordatorio,
  };

  Recordatorio.updateData(recordatorio_id, updateData, (err, result) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al actualizar recordatorio", detalles: err });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ message: "Recordatorio no encontrado" });
    }
  });
}
```

deleteRecordatorio: Este método elimina un recordatorio específico, identificado por recordatorio_id. Si no se proporciona el ID, se devuelve un error 400. Si ocurre un error durante la operación, se devuelve un mensaje de error con el código de estado 500. Si el recordatorio no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
const deleteRecordatorio = (req, res) => {
  const { recordatorio_id } = req.params;

  if (!recordatorio_id) {
    return res.status(400).json({
      error: "Se requiere el ID del recordatorio para eliminar.",
    });
  }

  Recordatorio.delete(recordatorio_id, (err, data) => {
    if (err) {
      console.error(`Error al eliminar el recordatorio: ${err}`);
      return res
        .status(500)
        .json({ error: "Error al eliminar el recordatorio" });
    }

    if (data.affectedRows === 0) {
      return res.status(404).json({ message: "Recordatorio no encontrado" });
    }

    res.status(200).json({ message: "Recordatorio eliminado exitosamente" });
  });
};
```

reporteController

El controlador reporteController.js maneja las operaciones relacionadas con los reportes en el sistema. Su objetivo es permitir la gestión de los reportes que los usuarios generan, facilitando así su seguimiento y análisis.

getReportes: Este método obtiene todos los reportes registrados en la base de datos. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de los reportes

```
const getReportes = (req, res) => {
  Reporte.getAll((err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener los reportes" });
    }
    res.status(200).json(data);
  });
};
```

getReportesByUserId: Este método obtiene los reportes asociados a un usuario específico, identificado por su usuario_id. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de los reportes del usuario.

```
const getReportesByUserId = (req, res) => {
  const { usuario_id } = req.params;

  Reporte.getByUserId(usuario_id, (err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener los reportes del usuario" });
    }
    res.status(200).json(data);
  });
};
```

postReporte: Este método permite crear un nuevo reporte. Extrae los campos título y descripción del cuerpo de la solicitud, y obtiene el usuario_id del token del usuario. Si falta alguno de los campos obligatorios, se devuelve un error 400. Si todos los campos están presentes, intenta crear el reporte en la base de datos y devuelve un mensaje de éxito si se realiza correctamente, junto con los datos del nuevo reporte.

```
const postReporte = (req, res) => {
  const { titulo, descripcion } = req.body;
  const usuario_id = req.userId; // Obtener el usuario_id del token

  if (!titulo || !descripcion) {
    return res.status(400).json({ error: "Título y descripción son obligatorios" });
  }

  const nuevoReporte = { titulo, descripcion };

  Reporte.create(usuario_id, nuevoReporte, (err, data) => {
    if (err) {
      console.error(`Error al crear reporte: ${err}`);
      return res.status(500).json({ error: "Error al crear el reporte" });
    }
    res.status(201).json({ message: "Reporte añadido exitosamente", data });
  });
};
```

putReporte: Este método actualiza un reporte existente. Recibe el reporte_id desde los parámetros de la solicitud y extrae título, descripción y estado del cuerpo de la solicitud. Si falta alguno de estos campos obligatorios, se devuelve un error 400. Llama a la función de actualización en el modelo y maneja posibles errores, incluyendo la falta del reporte a actualizar.

```
const putReporte = (req, res) => {
  const { reporte_id } = req.params;
  const { titulo, descripcion, estado } = req.body;

  if (!titulo || !descripcion || !estado) {
    return res.status(400).json({ error: "Todos los campos son obligatorios" });
  }

  const updateData = { titulo, descripcion, estado };

  Reporte.update(reporte_id, updateData, (err, result) => {
    if (err) {
      console.error(`Error al actualizar el reporte: ${err}`);
      return res.status(500).json({ error: "Error al actualizar el reporte" });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ message: "Reporte no encontrado" });
    }

    res.status(200).json({ message: "Reporte actualizado correctamente", result });
  });
};
```

deleteReporte: Este método elimina un reporte específico, identificado por reporte_id. Si ocurre un error durante la operación, se devuelve un mensaje de error con el código de estado 500. Si el reporte no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
const deleteReporte = (req, res) => {
  const { reporte_id } = req.params;

  Reporte.delete(reporte_id, (err, result) => {
    if (err) {
      console.error(`Error al eliminar el reporte: ${err}`);
      return res.status(500).json({ error: "Error al eliminar el reporte" });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ message: "Reporte no encontrado" });
    }

    res.status(200).json({ message: "Reporte eliminado exitosamente" });
  });
};
```

usuarioController

El controlador usuarioController.js maneja las operaciones relacionadas con los usuarios en el sistema. Su objetivo es permitir la gestión de los usuarios, facilitando así su registro, autenticación, y administración.

getUsuarios: Este método obtiene todos los usuarios registrados en la base de datos. Si ocurre un error durante la consulta, se devuelve un mensaje de error con el código de estado 500. Si la consulta es exitosa, se devuelve un código de estado 200 junto con los datos de los usuarios.

```
const getUsuarios = (req, res) => {
  Usuario.getAll((err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener usuarios" });
    }
    res.status(200).json(data);
  });
};
```

getInfoUsuarios: Este método ejecuta múltiples consultas en paralelo para obtener información sobre los usuarios. Obtiene el total de usuarios, los usuarios del mes actual y los datos necesarios para generar una gráfica. Si ocurre un error en cualquiera de las consultas, se devuelve un mensaje de error con el código de estado 500. Si todas las consultas son exitosas, se devuelve un código de estado 200 junto con un objeto que contiene todos los datos solicitados.

```
const getInfoUsuarios = (req, res) => {
  // Ejecuta todas las consultas en paralelo
  Usuario.getTotalUsuarios((err, totalUsuarios) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener el total de usuarios" });
    }

    Usuario.getUsuariosMesActual((err, usuariosMes) => {
      if (err) {
        return res
          .status(500)
          .json({ error: "Error al obtener usuarios del mes actual" });
      }

      Usuario.getGraficaUsuarios((err, graficaUsuarios) => {
        if (err) {
          return res
            .status(500)
            .json({ error: "Error al obtener datos para la gráfica" });
        }

        // Devuelve todos los datos como un objeto
        res.status(200).json({
          totalUsuarios,
          usuariosMes,
          graficaUsuarios,
        });
      });
    });
  });
};
```

`createUsuarios`: Este método permite crear un nuevo usuario. Extrae los campos `nombre_usuario`, `email`, y `password_usuario` del cuerpo de la solicitud. Se utiliza bcrypt para encriptar la contraseña antes de almacenarla. Si falta alguno de los campos obligatorios, se devuelve un error 400. Si todos los campos están presentes, intenta crear el usuario en la base de datos y devuelve un mensaje de éxito si se realiza correctamente.

```
const createUsuarios = (req, res) => {
  const { nombre_usuario, email, password_usuario } = req.body;

  const hashedPassword = bcrypt.hashSync(password_usuario, 8);

  const nuevoUsuario = {
    nombre_usuario,
    email,
    password_usuario: hashedPassword,
  };

  Usuario.create(nuevoUsuario, (err, data) => {
    if (err) {
      res.status(500).json({ error: "Error al crear usuario" });
      console.error(`El error es ${err}`);
    }

    res.status(201).json({
      message: "Usuario creado exitosamente",
    });
  });
};
```

`loginUsuario`: Este método permite a un usuario iniciar sesión. Busca el usuario por su `email`. Si no se encuentra, devuelve un error 404. Si el usuario existe, compara la contraseña proporcionada con la almacenada utilizando bcrypt. Si la contraseña es correcta, genera un token JWT que incluye el ID y el rol del usuario. Devuelve un mensaje de éxito, junto con el token y el rol del usuario.

```
const loginUsuario = (req, res) => {
  const { email, password_usuario } = req.body;

  // Buscar usuario por email
  Usuario.getByEmail(email, (err, user) => {
    if (err || !user) {
      console.error(`El error es ${err}`);
      return res.status(404).json({ error: "Usuario no encontrado" });
    }

    // Verifica que el campo de contraseña no sea undefined
    if (!user.password_usuario) {
      return res
        .status(500)
        .json({ error: "Error en el servidor: contraseña no definida" });
    }

    // Comparar contraseñas
    const passwordIsValid = bcrypt.compareSync(
      password_usuario,
      user.password_usuario
    );
    if (!passwordIsValid) {
      return res.status(401).json({ error: "Contraseña incorrecta" });
    }

    // Generar token JWT incluyendo el rol del usuario
    const token = jwt.sign(
      { id: user.usuario_id, rol: user.rol }, // Incluye el rol en el token
      process.env.JWT_SECRET,
      {
        expiresIn: 86400, // 24 horas
      }
    );
  });
};
```

putUsuario: Este método actualiza la información de un usuario existente. Recibe el `usuario_id` desde los parámetros de la solicitud y extrae nombre_usuario, email, password_usuario, e ingresos del cuerpo. Verifica que se proporcionen todos los campos obligatorios y encripta la nueva contraseña si se actualiza. Si no hay campos para actualizar, devuelve un error 400. Llama a la función de actualización en el modelo y maneja posibles errores.

```
const putUsuario = (req, res) => {
  const { usuario_id } = req.params;
  const { nombre_usuario, email, password_usuario, ingresos } = req.body;

  // Construir dinámicamente los datos a actualizar
  let updateData = {};

  if (nombre_usuario) {
    updateData.nombre_usuario = nombre_usuario;
  }

  if (email) {
    updateData.email = email;
  }

  if (password_usuario) {
    // Encriptar la nueva contraseña
    const hashedPassword = bcrypt.hashSync(password_usuario, 8);
    updateData.password_usuario = hashedPassword;
  }

  if (ingresos !== undefined) {
    // Asegurarse de incluir el campo ingresos si está en el cuerpo de la solicitud
    updateData.ingresos = ingresos;
  }

  // Verificar si hay campos para actualizar
  if (Object.keys(updateData).length === 0) {
    return res.status(400).json({ error: "No hay campos para actualizar" });
  }

  // Llamar al modelo para actualizar los datos del usuario
  Usuario.updateData(usuario_id, updateData, (err, result) => {
    if (err) {
      return res.status(500).json({ error: "Error al actualizar el usuario" });
    }

    if (result.affectedRows === 0) {
      return res.status(404).json({ message: "Usuario no encontrado" });
    }

    res.json(result);
  });
}
```

deleteUsuario: Este método elimina un usuario específico, identificado por `usuario_id`. Antes de eliminar el usuario, elimina todas las metas de ahorro y gastos asociados. Si ocurre un error durante la eliminación de las dependencias, se devuelve un mensaje de error con el código de estado 500. Si el usuario no se encuentra, se devuelve un código de estado 404. En caso de éxito, se devuelve un código de estado 200 junto con un mensaje de confirmación de la eliminación.

```
const deleteUsuario = (req, res) => {
  const { usuario_id } = req.params;

  // Eliminar las metas de ahorro relacionadas con el usuario
  const deleteMetasQuery = "DELETE FROM metas_de_ahorro WHERE usuario_id = ?";
  db.query(deleteMetasQuery, [usuario_id], (err) => {
    if (err) {
      console.error(`Error al eliminar metas de ahorro del usuario: ${err}`);
      return res
        .status(500)
        .json({ error: "Error al eliminar metas de ahorro del usuario" });
    }
  });

  // Eliminar los gastos relacionados con el usuario
  const deleteGastosQuery = "DELETE FROM gastos WHERE usuario_id = ?";
  db.query(deleteGastosQuery, [usuario_id], (err) => {
    if (err) {
      console.error(`Error al eliminar gastos del usuario: ${err}`);
      return res
        .status(500)
        .json({ error: "Error al eliminar gastos del usuario" });
    }
  });

  // Después de eliminar las dependencias, eliminar el usuario
  Usuario.delete(usuario_id, (err, data) => {
    if (err) {
      console.error(`Error al eliminar el usuario: ${err}`);
      return res
        .status(500)
        .json({ error: "Error al eliminar el usuario" });
    }

    if (data.affectedRows === 0) {
      return res.status(404).json({ message: "Usuario no encontrado" });
    }

    res.json(data);
  });
}
```

middlewareController

El controlador middlewareController.js maneja las funciones middleware que se utilizan para la verificación de tokens y permisos en el sistema. Su objetivo es asegurar que las rutas protegidas sean accesibles solo para usuarios autorizados.

verificarToken: Este middleware verifica la validez del token JWT enviado en la cabecera de autorización de la solicitud. Extrae el token de los encabezados y verifica si está presente. Si el token no se proporciona, se devuelve un código de estado 403 junto con un mensaje de error. Si el token está presente, se verifica utilizando la clave secreta almacenada en las variables de entorno. Si el token es válido, se decodifica y se almacenan el userId y el rol en el objeto de solicitud para su uso en las siguientes etapas de la solicitud. Si el token es invalido, se devuelve un código de estado 403.

```
// Middleware de verificación de token
const verificarToken = (req, res, next) => {
  const token = req.headers["authorization"]?.split(" ")[1];
  if (!token) return res.status(403).json({ error: "Token no proporcionado" });
}
```

verificarAdmin: Este middleware verifica si el usuario que hace la solicitud tiene el rol de "admin". Si el rol no es "admin", se devuelve un código de estado 403 junto con un mensaje de error que indica que no tiene permiso para acceder a la ruta. Si el rol es "admin", se permite el paso al siguiente middleware o controlador.

```
const verificarAdmin = (req, res, next) => {
  if (req.rol !== "admin") {
    return res.status(403).json({ error: "No tienes permiso para acceder a esta ruta" });
  }
  next();
};
```

Models

CategoríaGasto Model

El modelo CategoríaGasto maneja las operaciones relacionadas con las categorías de gasto en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las categorías de gasto.

getAll

Este método se utiliza para obtener todas las categorías de gasto de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla categorias_gasto. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el resultado al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

create

Este método permite crear una nueva categoría de gasto en la base de datos. Recibe un objeto data que debe contener el campo nombre_categoria. Ejecuta una consulta SQL para insertar la nueva categoría en la tabla categorias_gasto. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o data: Objeto que contiene la información de la nueva categoría.

o callback: Función que se ejecuta después de intentar crear la categoría, con dos parámetros: err y result.

update

Este método actualiza una categoría de gasto existente en la base de datos, identificada por su categoria_gasto_id. Recibe el ID de la categoría y un objeto data que debe contener el campo nombre_categoria. Ejecuta una consulta SQL para actualizar el nombre de la categoría correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o categoria_gasto_id: ID de la categoría que se desea actualizar.

o data: Objeto que contiene la nueva información de la categoría.

o callback: Función que se ejecuta después de intentar actualizar la categoría, con dos parámetros: err y result.

delete

Este método elimina una categoría de gasto de la base de datos, identificada por su categoria_gasto_id. Ejecuta una consulta SQL para eliminar la categoría correspondiente en la tabla categorias_gasto. Si hay un error durante la eliminación, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o categoria_gasto_id: ID de la categoría que se desea eliminar.

o callback: Función que se ejecuta después de intentar eliminar la categoría, con dos parámetros: err y result.

```
// Obtener todas las categorías de gasto
getAll: (callback) => {
  const query = "SELECT * FROM categorias_gasto";
  db.query(query, (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

// Crear una nueva categoría de gasto
create: (data, callback) => {
  const { nombre_categoria } = data;
  const query = "INSERT INTO categorias_gasto (nombre_categoria) VALUES (?)";
  db.query(query, [nombre_categoria], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

// Actualizar una categoría de gasto por ID
update: (categoria_gasto_id, data, callback) => {
  const { nombre_categoria } = data;
  const query =
    "UPDATE categorias_gasto SET nombre_categoria = ? WHERE categoria_gasto_id = ?";
  db.query(query, [nombre_categoria, categoria_gasto_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

// Eliminar una categoría de gasto por ID
delete: (categoria_gasto_id, callback) => {
  const query = "DELETE FROM categorias_gasto WHERE categoria_gasto_id = ?";
  db.query(query, [categoria_gasto_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

module.exports = CategoríaGasto;
```

CategoríaMeta Model

El modelo CategoríaMeta maneja las operaciones relacionadas con las categorías de metas en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las categorías de metas.

getAll

Este método se utiliza para obtener todas las categorías de metas de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla categorias_metas. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el resultado al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

create

Este método permite crear una nueva categoría de meta en la base de datos. Recibe un objeto data que debe contener el campo nombre_categoria. Ejecuta una consulta SQL para insertar la nueva categoría en la tabla categorias_metas. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o data: Objeto que contiene la información de la nueva categoría.

o callback: Función que se ejecuta después de intentar crear la categoría, con dos parámetros: err y result.

update

Este método actualiza una categoría de meta existente en la base de datos, identificada por su categoria_meta_id. Recibe el ID de la categoría y un objeto data que debe contener el campo nombre_categoria. Ejecuta una consulta SQL para actualizar el nombre de la categoría correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o categoria_meta_id: ID de la categoría que se desea actualizar.

o data: Objeto que contiene la nueva información de la categoría.

o callback: Función que se ejecuta después de intentar actualizar la categoría, con dos parámetros: err y result.

delete

Este método elimina una categoría de meta de la base de datos, identificada por su categoria_meta_id. Ejecuta una consulta SQL para eliminar la categoría correspondiente en la tabla categorias_metas. Si hay un error durante la eliminación, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o categoria_meta_id: ID de la categoría que se desea eliminar.

o callback: Función que se ejecuta después de intentar eliminar la categoría, con dos parámetros: err y result.

```
const CategoríaMeta = {
  // Obtener todas las categorías de metas
  getAll: (callback) => {
    const query = "SELECT * FROM categorias_metas";
    db.query(query, (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },
  // Crear una nueva categoría de meta
  create: (data, callback) => {
    const { nombre_categoria } = data;
    const query = "INSERT INTO categorias_metas (nombre_categoria) VALUES (?)";
    db.query(query, [nombre_categoria], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },
  // Actualizar una categoría de meta por ID
  update: (categoria_meta_id, data, callback) => {
    const { nombre_categoria } = data;
    const query =
      "UPDATE categorias_metas SET nombre_categoria = ? WHERE categoria_meta_id = ?";
    db.query(query, [nombre_categoria, categoria_meta_id], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },
  // Eliminar una categoría de meta por ID
  delete: (categoria_meta_id, callback) => {
    const query = "DELETE FROM categorias_metas WHERE categoria_meta_id = ?";
    db.query(query, [categoria_meta_id], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  };
};
```

Gasto Model

El modelo Gasto maneja las operaciones relacionadas con los gastos en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los gastos.

getAll

Este método se utiliza para obtener todos los gastos de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla gastos. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el resultado al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

getByIdUser

Este método obtiene todos los gastos asociados a un usuario específico, identificado por su usuario_id. Realiza una consulta SQL que une la tabla gastos con la tabla categorias_gasto para incluir el nombre de la categoría de gasto. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario cuyos gastos se desean obtener.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

getByCategory

Este método obtiene todos los gastos asociados a una categoría específica, identificada por su categoria_gasto_id. Ejecuta una consulta SQL para seleccionar los gastos correspondientes. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa y no hay resultados, se pasa null al callback; si hay resultados, se pasan al callback.

- Parámetros:

o categoria_gasto_id: ID de la categoría de gasto.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

create

Este método permite crear un nuevo gasto en la base de datos. Recibe el usuario_id y un objeto data que debe contener los campos monto, categoria_gasto_id, y descripción. Ejecuta una consulta SQL para insertar el nuevo gasto en la tabla gastos. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario que está creando el gasto.

o data: Objeto que contiene la información del nuevo gasto.

o callback: Función que se ejecuta después de intentar crear el gasto, con dos parámetros: err y result.

updateData

Este metodo actualiza un gasto existente en la base de datos, identificado por su id_gasto y el usuario_id. Recibe el ID del gasto y un objeto data que debe contener los campos monto, categoria_gasto_id, y descripcion. Ejecuta una consulta SQL para actualizar el gasto correspondiente. Si hay un error durante la actualizaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- parámetros:

o id_gasto: ID del gasto que se desea actualizar.

o usuario_id: ID del usuario que es propietario del gasto.

o data: Objeto que contiene la nueva informaci6n del gasto.

o callback: Funci6n que se ejecuta despu6s de intentar actualizar el gasto, con dos parámetros: err y result.

delete

Este metodo elimina un gasto de la base de datos, identificado por su id_gasto. Ejecuta una consulta SQL para eliminar el gasto correspondiente en la tabla gastos. Si hay un error durante la eliminaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- Parámetros:

o id_gasto: ID del gasto que se desea eliminar.

o callback: Funci6n que se ejecuta despu6s de intentar eliminar el gasto, con dos parámetros: err y result.

```
const CategoriaMeta = {
  // Obtener todas las categorías de metas
  getAll: (callback) => {
    const query = "SELECT * FROM categorias_metas";
    db.query(query, (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },

  // Crear una nueva categoría de meta
  create: (data, callback) => {
    const { nombre_categoria } = data;
    const query = "INSERT INTO categorias_metas (nombre_categoria) VALUES (?)";
    db.query(query, [nombre_categoria], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },

  // Actualizar una categoría de meta por ID
  update: (categoria_meta_id, data, callback) => {
    const { nombre_categoria } = data;
    const query =
      "UPDATE categorias_metas SET nombre_categoria = ? WHERE categoria_meta_id = ?";
    db.query(query, [nombre_categoria, categoria_meta_id], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },

  // Eliminar una categoría de meta por ID
  delete: (categoria_meta_id, callback) => {
    const query = "DELETE FROM categorias_metas WHERE categoria_meta_id = ?";
    db.query(query, [categoria_meta_id], (err, result) => {

```

ingreso Model

El modelo ingreso maneja las operaciones relacionadas con los ingresos de los usuarios en el sistema. Utiliza la conexión a la base de datos para realizar operaciones de lectura y actualización sobre los ingresos.

getIngresoByIdUserId

Este método obtiene el ingreso de un usuario específico, identificado por su `usuario_id`. Ejecuta una consulta SQL que selecciona el campo `ingresos` de la tabla `usuarios` correspondiente al ID del usuario. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el ingreso del usuario al callback.

- Parámetros:

o `usuario_id`: ID del usuario cuyo ingreso se desea obtener.

o `callback`: Función que se ejecuta después de completar la consulta, con dos parámetros: `err` y `results`.

updateIngreso

Este método actualiza el ingreso de un usuario existente en la base de datos, identificado por su `usuario_id`. Recibe el ID del usuario y el nuevo valor de `ingresos`. Ejecuta una consulta SQL para actualizar el campo `ingresos` en la tabla `usuarios`. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o `usuario_id`: ID del usuario cuyo ingreso se desea actualizar.

o `ingresos`: Nuevo valor de `ingresos` que se desea establecer.

o `callback`: Función que se ejecuta después de intentar actualizar el ingreso, con dos parámetros: `err` y `result`.

```
const Ingreso = {
    // Obtener el ingreso de un usuario por ID
    getIngresoByIdUserId: (usuario_id, callback) => {
        const query = "SELECT ingresos FROM usuarios WHERE usuario_id = ?";
        db.query(query, [usuario_id], (err, results) => {
            if (err) return callback(err, null);
            callback(null, results[0]); // Devuelve el ingreso del usuario
        });
    },

    // Actualizar el ingreso de un usuario por ID
    updateIngreso: (usuario_id, ingresos, callback) => {
        const query =
            `UPDATE usuarios
             SET ingresos = ?
             WHERE usuario_id = ?`;
        db.query(query, [ingresos, usuario_id], (err, result) => {
            if (err) return callback(err, null);
            callback(null, result);
        });
    },
};

module.exports = Ingreso;
```

MetaAhorro Model

El modelo MetaAhorro maneja las operaciones relacionadas con las metas de ahorro en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las metas de ahorro.

getAll

Este método se utiliza para obtener todas las metas de ahorro de la base de datos.

Ejecuta una consulta SQL que selecciona todas las entradas de la tabla metas_de_ahorro. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el resultado al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

getById

Este método obtiene todas las metas de ahorro asociadas a un usuario específico, identificado por su usuario_id. Ejecuta una consulta SQL que selecciona las metas correspondientes al ID del usuario. Si hay un error durante la consulta, se pasa el error al callback. Si no se encuentran resultados, se pasa null al callback; si hay resultados, se pasan al callback.

- Parámetros:

o usuario_id: ID del usuario cuyos metas de ahorro se desean obtener.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

getByCategoryId

Este método obtiene todas las metas de ahorro asociadas a una categoría específica, identificada por su categoria_meta_id. Ejecuta una consulta SQL que selecciona las metas correspondientes a la categoría. Si hay un error durante la consulta, se pasa el error al callback. Si no se encuentran resultados, se pasa null al callback; si hay resultados, se pasan al callback.

- Parámetros:

o categoria_meta_id: ID de la categoría de meta.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

create

Este método permite crear una nueva meta de ahorro en la base de datos. Recibe el usuario_id y un objeto data que debe contener los campos nombre_meta, monto_objetivo, fecha_limite, descripción, y categoria_meta_id. Ejecuta una consulta SQL para insertar la nueva meta en la tabla metas_de_ahorro. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario que está creando la meta.

o data: Objeto que contiene la información de la nueva meta.

o callback: Función que se ejecuta después de intentar crear la meta, con dos parámetros: err y result.

updateData

Este método actualiza una meta de ahorro existente en la base de datos, identificada por su meta_id. Recibe el ID de la meta y un objeto data que debe contener los campos nombre_meta, monto_objetivo, monto_actual, descripción, y estado_de_meta. Ejecuta una consulta SQL para actualizar la meta correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o meta_id: ID de la meta que se desea actualizar.

o data: Objeto que contiene la nueva información de la meta.

o callback: Función que se ejecuta después de intentar actualizar la meta, con dos parámetros: err y result.

updateMontoActual

Este metodo actualiza el manta actual de una meta de ahorro, identificada par su meta_id. Recibe el ID de la meta y el montoAdicional que se desea agregar al manta actual. Ejecuta una consulta SQL para incrementar el monto_actual de la meta. Si hay un error durante la actualizaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- Parámetros:

o meta_id: ID de la meta cuya cantidad se desea actualizar.

o montoAdicional: Manto que se agregara al monto_actual.

o callback: Funci6n que se ejecuta despu6s de intentar actualizar el manta, con dos parámetros: err y result.

delete

Este metodo elimina una meta de ahorro de la base de datos, identificada par su meta_id. Ejecuta una consulta SQL para eliminar la meta correspondiente en la tabla metas_de_ahorro. Si hay un error durante la eliminaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- Parámetros:

o meta_id: ID de la meta que se desea eliminar.

o callback: Funci6n que se ejecuta despu6s de intentar eliminar la meta, con dos parámetros: err y result.

```
const MetaAhorro = {
  // Obtener todas las metas de ahorro
  getAll: (callback) => {
    const query = "SELECT * FROM metas_de_ahorro";
    db.query(query, (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },

  // Obtener metas de ahorro por usuario
  getByUserId: (usuario_id, callback) => {
    const query = "SELECT * FROM metas_de_ahorro WHERE usuario_id = ?";
    db.query(query, [usuario_id], (err, result) => {
      if (err) return callback(err, null);
      if (result.length === 0) return callback(null, null);
      callback(null, result);
    });
  },

  // Obtener una meta de ahorro por categoria
  getByCategoriaId: (categoria_meta_id, callback) => {
    const query = "SELECT * FROM metas_de_ahorro WHERE categoria_meta_id = ?";
    db.query(query, [categoria_meta_id], (err, results) => {
      if (err) return callback(err, null);
      if (results.length === 0) return callback(null, null);
      callback(null, results);
    });
  },

  // Crear una nueva meta de ahorro
  create: (usuario_id, data, callback) => {
    const {
      nombre_meta,
      monto_objetivo,
      fecha_limite,
      descripcion,
      categoria_meta_id,
    } = data;
    const query = `INSERT INTO metas_de_ahorro (usuario_id, nombre_meta, monto_objetivo, fecha_limite, descripcion, categoria_meta_id)
VALUES (?, ?, ?, ?, ?, ?)`;
    db.query(
      query,
      [
        usuario_id,
        nombre_meta,
        monto_objetivo,
        fecha_limite,
        descripcion,
        categoria_meta_id,
      ],
      (err, result) => {
        if (err) return callback(err, null);
        callback(null, result);
      }
    );
  },

  // Actualizar una meta de ahorro
  updateData: (meta_id, data, callback) => {
    const {
      nombre_meta,
      monto_objetivo,
      monto_actual
    }
```

Notificación Model

El modelo Notificación maneja las operaciones relacionadas con las notificaciones en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las notificaciones.

getByIdUserId

Este método se utiliza para obtener todas las notificaciones de un usuario específico, identificado por su `usuario_id`. Ejecuta una consulta SQL que selecciona todas las notificaciones correspondientes al ID del usuario y las ordena por fecha en orden descendente. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

o `usuario_id`: ID del usuario cuyas notificaciones se desean obtener.

o `callback`: Función que se ejecuta después de completar la consulta, con dos parámetros: `err` y `result`.

create

Este método permite crear una nueva notificación en la base de datos. Recibe un objeto `data` que debe contener los campos `usuario_id`, `tipo`, y `mensaje`. Ejecuta una consulta SQL para insertar la nueva notificación en la tabla `notificaciones`. Devuelve una promesa que se resuelve con el resultado de la inserción o se rechaza con un error en caso de que ocurra uno.

- Parámetros:

o `data`: Objeto que contiene la información de la nueva notificación.

o Retorno: Promesa que se resuelve con el resultado de la inserción.

markAsRead

Este método marca una notificación como leída. Recibe la notificación `id` de la notificación que se desea actualizar. Ejecuta una consulta SQL que actualiza el campo `leido` de la notificación correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o `notificacion_id`: ID de la notificación que se desea marcar como leída.

o `callback`: Función que se ejecuta después de intentar marcar la notificación, con dos parámetros: `err` y `result`.

delete

Este método elimina una notificación de la base de datos, identificada por su `notificacion_id`. Ejecuta una consulta SQL para eliminar la notificación correspondiente en la tabla `notificaciones`. Si hay un error durante la eliminación, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o `notificacion_id`: ID de la notificación que se desea eliminar.

o `callback`: Función que se ejecuta después de intentar eliminar la notificación, con dos parámetros: `err` y `result`.

```
const Notificacion = {
  // Obtener una notificación por su ID
  getByIdUserId: (usuario_id, callback) => {
    const query =
      "SELECT * FROM notificaciones WHERE usuario_id = ? ORDER BY fecha DESC";
    db.query(query, [usuario_id], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },

  // Crear una nueva notificación
  create: async (data) => {
    const { usuario_id, tipo, mensaje } = data;
    const query =
      "INSERT INTO notificaciones (usuario_id, tipo, mensaje) VALUES (?, ?, ?)";

    return new Promise((resolve, reject) => {
      db.query(query, [usuario_id, tipo, mensaje], (err, result) => {
        if (err) {
          return reject(err); // Rechaza la promesa si hay un error
        }
        resolve(result); // Resuelve la promesa con el resultado de la inserción
      });
    });
  },

  // Marcar una notificación como leída
  markAsRead: (notificacion_id, callback) => {
    const query =
      "UPDATE notificaciones SET leido = TRUE WHERE notificacion_id = ?";
    db.query(query, [notificacion_id], (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    });
  },
};
```

Recordatorio Model

El modelo Recordatorio maneja las operaciones relacionadas con los recordatorios en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los recordatorios.

getByUserId

Este método obtiene todos los recordatorios asociados a un usuario específico, identificado por su `usuario_id`. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla `recordatorios` correspondientes al ID del usuario. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

`o usuario_id`: ID del usuario cuyos recordatorios se desean obtener.

`o callback`: Función que se ejecuta después de completar la consulta, con dos parámetros: `err` y `result`.

getAll

Este método se utiliza para obtener todos los recordatorios de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla `recordatorios`. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

`o callback`: Función que se ejecuta después de completar la consulta, con dos parámetros: `err` y `result`.

create

Este método permite crear un nuevo recordatorio en la base de datos. Recibe un objeto `data` que debe contener los campos `usuario_id`, `descripcion`, y `fecha_recordatorio`.

Ejecuta una consulta SQL para insertar el nuevo recordatorio en la tabla `recordatorios`.

Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

`o data`: Objeto que contiene la información del nuevo recordatorio.

`o callback`: Función que se ejecuta después de intentar crear el recordatorio, con dos parámetros: `err` y `result`.

updateData

Este método actualiza un recordatorio existente en la base de datos, identificado por su `recordatorio_id`. Recibe el ID del recordatorio y un objeto `data` que debe contener los campos `descripcion` y `fecha_recordatorio`. Ejecuta una consulta SQL para actualizar el recordatorio correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

`o recordatorio_id`: ID del recordatorio que se desea actualizar.

`o data`: Objeto que contiene la nueva información del recordatorio.

`o callback`: Función que se ejecuta después de intentar actualizar el recordatorio, con dos parámetros: `err` y `result`.

delete

Este método elimina un recordatorio de la base de datos, identificado por su `recordatorio_id`. Ejecuta una consulta SQL para eliminar el recordatorio correspondiente en la tabla `recordatorios`. Si hay un error durante la eliminación, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

`o recordatorio_id`: ID del recordatorio que se desea eliminar.

`o callback`: Función que se ejecuta después de intentar eliminar el recordatorio, con dos parámetros: `err` y `result`.

```
// Crear un recordatorio
create: (data, callback) => {
  ...
};

// Actualizar un recordatorio por ID
updateData: (recordatorio_id, data, callback) => {
  const { descripcion, fecha_recordatorio } = data;
  const query = "UPDATE recordatorios SET descripcion = ?, fecha_recordatorio = ? WHERE recordatorio_id = ?";
  db.query(query, [descripcion, fecha_recordatorio, recordatorio_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

// Eliminar un recordatorio por ID
delete: (recordatorio_id, callback) => {
  const query = "DELETE FROM recordatorios WHERE recordatorio_id = ?";
  db.query(query, [recordatorio_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};

module.exports = Recordatorio;
```

Reporte Model

El modelo Reporte maneja las operaciones relacionadas con los reportes en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los reportes.

getAll

Este método se utiliza para obtener todos los reportes de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla Reportes. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

getById

Este método obtiene todos los reportes asociados a un usuario específico, identificado por su usuario_id. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla Reportes correspondientes al ID del usuario. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

o usuario_id: ID del usuario cuyos reportes se desean obtener.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

create

Este método permite crear un nuevo reporte en la base de datos. Recibe el usuario_id y un objeto data que debe contener los campos título y descripción. Ejecuta una consulta SQL para insertar el nuevo reporte en la tabla Reportes. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario que está creando el reporte.

o data: Objeto que contiene la información del nuevo reporte.

o callback: Función que se ejecuta después de intentar crear el reporte, con dos parámetros: err y result.

update

Este método actualiza un reporte existente en la base de datos, identificado por su reporte_id. Recibe el ID del reporte y un objeto data que debe contener los campos título, descripción, y estado. Ejecuta una consulta SQL para actualizar el reporte correspondiente. Si hay un error durante la actualización, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o reporte_id: ID del reporte que se desea actualizar.

o data: Objeto que contiene la nueva información del reporte.

o callback: Función que se ejecuta después de intentar actualizar el reporte, con dos parámetros: err y result.

delete

Este método elimina un reporte de la base de datos, identificado por su reporte_id. Ejecuta una consulta SQL para eliminar el reporte correspondiente en la tabla Reportes. Si hay un error durante la eliminación, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o reporte_id: ID del reporte que se desea eliminar.

o callback: Función que se ejecuta después de intentar eliminar el reporte, con dos parámetros: err y result.

```

const Reporte = {
    // Devolver todos los reportes
    getAll(callback) => {
        const query = `SELECT * FROM Reportes`;
        db.query(query, (err, result) => {
            if (err) return callback(err, null);
            callback(null, result);
        });
    },
    // Obtener reportes por id de usuario
    getById(usuario_id, callback) => {
        const query = `SELECT * FROM Reportes WHERE usuario_id = ?`;
        db.query(query, [usuario_id], (err, result) => {
            if (err) return callback(err, null);
            callback(null, result);
        });
    },
    // Crear un nuevo reporte
    create(usuario_id, data, callback) => {
        const query = `INSERT INTO Reportes (usuario_id, titulo, descripcion) VALUES (?, ?, ?)`;
        db.query(query, [usuario_id, data.titulo, data.descripcion], (err, result) => {
            if (err) return callback(err, null);
            callback(null, result);
        });
    },
    module.exports = Reporte;
}

```

Usuario Model

El modelo Usuario maneja las operaciones relacionadas con los usuarios en el sistema. Utiliza la conexión a la base de datos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los usuarios.

getAll

Este método se utiliza para obtener todos los usuarios de la base de datos. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla usuarios. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

getTotalUsuarios

Este método obtiene el total de usuarios registrados en la base de datos. Ejecuta una consulta SQL que cuenta todas las entradas en la tabla usuarios. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el número total de usuarios al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y totalUsuarios.

getUsuariosMesActual

Este método obtiene el número de usuarios que se han registrado en el mes actual. Ejecuta una consulta SQL que cuenta todas las entradas en la tabla usuarios que tienen una fecha de registro correspondiente al mes y año actuales. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasa el número de usuarios registrados en el mes actual al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y usuariosMes.

getGrafica Usuarios

Este método obtiene datos sobre los usuarios registrados, agrupados por mes, para utilizarlos en una gráfica. Ejecuta una consulta SQL que agrupa los usuarios por mes y cuenta cuantos usuarios se registraron en cada uno. Si hay un error durante la consulta, se pasa el error al callback. Si la consulta es exitosa, se pasan los resultados al callback.

- Parámetros:

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y results.

getByEmail

Este método busca un usuario por su dirección de correo electrónico. Ejecuta una consulta SQL que selecciona todas las entradas de la tabla usuarios que coinciden con el correo proporcionado. Si hay un error durante la consulta, se pasa el error al callback. Si no se encuentra ningún usuario, se pasa null al callback. Si se encuentra un usuario, se pasan sus datos al callback.

- Parámetros:

o email: Dirección de correo electrónico del usuario que se desea buscar.

o callback: Función que se ejecuta después de completar la consulta, con dos parámetros: err y result.

create

Este método permite crear un nuevo usuario en la base de datos. Recibe un objeto data que debe contener los campos nombre_usuario, email, y password_usuario. Ejecuta una consulta SQL para insertar el nuevo usuario en la tabla usuarios. Si hay un error durante la inserción, se pasa el error al callback. Si la operación es exitosa, se pasa el resultado al callback.

- Parámetros:

o data: Objeto que contiene la información del nuevo usuario.

o callback: Función que se ejecuta después de intentar crear el usuario, con dos parámetros: err y result.

Backend

updateData

Este metodo actualiza los datos de un usuario existente en la base de datos, identificado por su usuario_id. Recibe el ID del usuario y un objeto data que puede contener nombre_usuario, email, password_usuario, e ingresos. Ejecuta una consulta SQL para actualizar los datos del usuario correspondiente. Si hay un error durante la actualizaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario que se desea actualizar.

o data: Objeto que contiene la nueva informaci6n del usuario.

o callback: Funci6n que se ejecuta después de intentar actualizar el usuario, con dos parámetros: err y result.

delete

Este metodo elimina un usuario de la base de datos, identificado por su usuario_id.

Ejecuta una consulta SQL para eliminar el usuario correspondiente en la tabla usuarios.

Si hay un error durante la eliminaci6n, se pasa el error al callback. Si la operaci6n es exitosa, se pasa el resultado al callback.

- Parámetros:

o usuario_id: ID del usuario que se desea eliminar.

o callback: Funci6n que se ejecuta después de intentar eliminar el usuario, con dos parámetros: err y result.

```
const Usuario = {
  getAll: (callback) => {
    const query = "SELECT * FROM usuarios";
    db.query(query, (err, results) => {
      if (err) return callback(err, null);
      callback(null, results);
    });
  },

  // Obtener el total de usuarios
  getTotalUsuarios: (callback) => {
    const query = "SELECT COUNT(*) AS totalUsuarios FROM usuarios";
    db.query(query, (err, results) => {
      if (err) return callback(err, null);
      callback(null, results[0].totalUsuarios);
    });
  },

  // Obtener los usuarios registrados en el mes actual
  getUsuariosMesActual: (callback) => {
    const query = `SELECT COUNT(*) AS usuariosMes
      FROM usuarios
      WHERE MONTH(fecha_registro) = MONTH(CURDATE())
      AND YEAR(fecha_registro) = YEAR(CURDATE())`;
    db.query(query, (err, results) => {
      if (err) return callback(err, null);
      callback(null, results[0].usuariosMes);
    });
  },
};

Usuario = {
  GraficaUsuarios: (callback) => {
    const query = `SELECT DATE_FORMAT(fecha_registro, '%Y-%m') AS month, COUNT(*) AS
      FROM usuarios
      GROUP BY month
      ORDER BY month`;
    db.query(query, (err, results) => {
      if (err) return callback(err, null);
      callback(null, results);
    });
  },
};

ByEmail: (email, callback) => {
  const query = "SELECT * FROM usuarios WHERE email = ?";
  db.query(query, [email], (err, result) => {
    if (err) return callback(err, null);
    if (result.length === 0) return callback(null, null);
    callback(null, result[0]);
  });
};

ate: (data, callback) => {
  const { nombre_usuario, email, password_usuario } = data;
  const query = `INSERT INTO usuarios (nombre_usuario, email, password_usuario)
    VALUES (?, ?, ?)`;
  db.query(
    query,
    [nombre_usuario, email, password_usuario],
    (err, result) => {
      if (err) return callback(err, null);
      callback(null, result);
    }
  );
};
```

Rutas de Categorías de Gastos

El archivo `categoriasGastosRoutes.js` define las rutas de la API para las operaciones relacionadas con las categorías de gastos. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `categoriaGastosController`.

Importaciones

- `express`: Se importa el módulo express para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `categoriaGastosController`: Se importa el controlador que maneja la lógica de negocio relacionada con las categorías de gastos.

Rutas Definidas

1. GET/

o Descripción: Obtiene todas las categorías de gastos.

o Controlador: `getCategoriaGastos`

o Respuesta Esperada: Retorna un código de estado 200 y una lista de categorías de gastos en formato JSON.

1. POST /create

o Descripción: Crea una nueva categoría de gasto.

o Controlador: `postCategoriaGastos`

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el campo `nombre_categoria`.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:categoria_gasto_id

o Descripción: Actualiza una categoría de gasto existente para su ID.

o Controlador: `putCategoriaGastos`

o Parámetros:

§ `categoria_gasto_id`: ID de la categoría de gasto que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el campo `nombre_categoria`.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:categoria_gasto_id

o Descripción: Elimina una categoría de gasto existente para su ID.

o Controlador: `deleteCategoriaGastos`

o Parámetros:

§ `categoria_gasto_id`: ID de la categoría de gasto que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

Exportación

El enrutador se exporta para ser utilizado en el archivo principal de la aplicación, permitiendo que otras partes de la aplicación utilicen estas rutas

```
st express = require("express");
st router = express.Router();
st categoriaGastosController = require("../controllers/categoriaGastosController");

ter.get("/", categoriaGastosController.getCategoriasGastos);
ter.post("/create", categoriaGastosController.postCategoriaGastos);
ter.put("/update/:categoria_gasto_id", categoriaGastosController.putCategoriaGastos);
ter.delete("/delete/:categoria_gasto_id", categoriaGastosController.deleteCategoriaGastos);

ule.exports = router;
```

Rutas de Categorías de Metas

El archivo categoriasMetasRoutes.js define las rutas de la API para las operaciones relacionadas con las categorías de metas. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador categoriaMetasController.

Importaciones

- express: Se importa el módulo express para crear un enrutador.
- router: Se crea una instancia de express.Router() que se utiliza para definir las rutas de la API.
- categoriaMetasController: Se importa el controlador que maneja la lógica de negocio relacionada con las categorías de metas.

Rutas Definidas

1. GET /

- o Descripción: Obtiene todas las categorías de metas.
- o Controlador: getcategoriasMetas
- o Respuesta Esperada: Retorna un código de estado 200 y una lista de categorías de metas en formato JSON.

1. POST /create

- o Descripción: Crea una nueva categoría de meta.
- o Controlador: postcategoriasMetas
- o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el campo nombre_categoria.
- o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:categoria_meta_id

- o Descripción: Actualiza una categoría de meta existente por su ID.
- o Controlador: putcategoriasMetas
- o Parámetros:
 - § categoria_meta_id: ID de la categoría de meta que se desea actualizar (proporcionado en la URL).
- o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el campo nombre_categoria.
- o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:categoria_meta_id

- o Descripción: Elimina una categoría de meta existente por su ID.
- o Controlador: deletecategoriasMetas
- o Parámetros:
 - § categoria_meta_id: ID de la categoría de meta que se desea eliminar (proporcionado en la URL).
- o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

Exportación

- El enrutador se exporta para ser utilizado en el archivo principal de la aplicación, permitiendo que otras partes de la aplicación utilicen estas rutas.

```
const router = express.Router();
const categoriaMetasController = require('../controllers/categoriaMetasController')

router.get("/", categoriaMetasController.getcategoriasMetas)
router.post("/create", categoriaMetasController.postcategoriasMetas)
router.put("/update/:categoria_meta_id", categoriaMetasController.putcategoriasMetas)
router.delete("/delete/:categoria_meta_id", categoriaMetasController.deletecategoriasMetas)
```

Rutas de Gastos

El archivo `gastosRoutes.js` define las rutas de la API para las operaciones relacionadas con los gastos. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `gastosController`. Además, incluye un middleware para la verificación del token.

Importaciones

- `express`: Se importa el módulo `express` para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `gastosController`: Se importa el controlador que maneja la lógica de negocio relacionada con los gastos.
- `token`: Se importa el middleware de autenticación que verifica la validez del token **JWT**.

Rutas Definidas

1. GET /

o Descripción: Obtiene todos los gastos.

o Controlador: `getGastos`

o Respuesta Esperada: Retorna un código de estado 200 y una lista de gastos en formato JSON.

1. GET /user/:usuario_id

o Descripción: Obtiene todos los gastos de un usuario específico para su ID.

o Controlador: `getGastoByUserId`

o Parámetros:

§ `:usuario_id`: ID del usuario cuyos gastos se desean obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de gastos del usuario en formato JSON.

1. GET /category/:categoria_gasto_id

o Descripción: Obtiene todos los gastos de una categoría específica para su ID.

o Controlador: `getGastoByCategoria`

o Parámetros:

§ `:categoria_gasto_id`: ID de la categoría de gastos que se desea obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de gastos de la categoría en formato JSON.

1. POST /create

o Descripción: Crea un nuevo gasto.

o Controlador: `postGasto`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para crear el gasto.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:id_gasto

o Descripción: Actualiza un gasto existente para su ID.

o Controlador: `putGasto`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Parámetros:

§ `:id_gasto`: ID del gasto que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:id_gasto

o Descripción: Elimina un gasto existente para su ID.

o Controlador: `deleteGasto`

o Parámetros:

§ `:id_gasto`: ID del gasto que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

Backend

Rutas de ingreso

El archivo `ingresoRoutes.js` define las rutas de la API para las operaciones relacionadas con los ingresos de los usuarios. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `ingresoController`. También incluye un middleware para la verificación del token.

Importaciones

- `express`: Se importa el módulo `express` para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `ingresoController`: Se importa el controlador que maneja la lógica de negocio relacionada con los ingresos.
- `token`: Se importa el middleware de autenticación que verifica la validez del token JWT.

Rutas Definidas

1. GET /usuario/:usuario_id

○ Descripción: Obtiene el ingreso de un usuario específico para su ID.

○ Controlador: `getIngresoById`

○ Parámetros:

§ `usuario_id`: ID del usuario cuyo ingreso se desea obtener (proporcionado en la URL).

○ Respuesta Esperada: Retorna un código de estado 200 y el ingreso del usuario en formato JSON.

1. PUT /usuario/:usuario_id

○ Descripción: Actualiza el ingreso de un usuario específico para su ID.

○ Controlador: `updateIngreso`

○ Middleware: `token.verificarToken` (requiere un token válido para acceder).

○ Parámetros:

§ `usuario_id`: ID del usuario cuyo ingreso se desea actualizar (proporcionado en la URL).

○ Cuerpo de Solicitud: Se espera un objeto JSON que contenga el nuevo ingreso del usuario.

○ Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

```
const router = express.Router();
const ingresoController = require("../controllers/ingresoController");
const token = require("../middleware/authMiddleware")

router.get("/usuario/:usuario_id", ingresoController.getIngresoById);
router.put("/usuario/:usuario_id", token.verificarToken, ingresoController.updateIngreso);

module.exports = router;
```

Rutas de Metas de Ahorro

El archivo `metasAhorroRoutes.js` define las rutas de la API para las operaciones relacionadas con las metas de ahorro de los usuarios. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `metasAhorroController`. Además, incluye un middleware para la verificación del token.

importaciones

- `express`: Se importa el módulo `express` para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `metasAhorroController`: Se importa el controlador que maneja la lógica de negocio relacionada con las metas de ahorro.
- `token`: Se importa el middleware de autenticación que verifica la validez del token JWT.

Rutas Definidas

1. GET /

o Descripción: Obtiene todas las metas de ahorro.

o Controlador: `getMetasAhorro`

o Respuesta Esperada: Retorna un código de estado 200 y una lista de metas de ahorro en formato JSON.

1. GET /user/:usuario_id

o Descripción: Obtiene todas las metas de ahorro de un usuario específico para su ID.

o Controlador: `getMetasByUserId`

o Parámetros:

§ `usuario_id`: ID del usuario cuyas metas se desean obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de metas del usuario en formato JSON.

1. GET /category/:categoria_meta_id

o Descripción: Obtiene todas las metas de ahorro de una categoría específica para su ID.

o Controlador: `getMetasByCategoria`

o Parámetros:

§ `categoria_meta_id`: ID de la categoría de metas que se desea obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de metas de la categoría en formato JSON.

1. POST /create

o Descripción: Crea una nueva meta de ahorro.

o Controlador: `postMetaAhorro`

o Middleware: `token.verifyToken` (requiere un token válido para acceder).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para crear la meta de ahorro.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:meta_id

o Descripción: Actualiza una meta de ahorro existente para su ID.

o Controlador: `putMetaAhorro`

o Parámetros:

§ `meta_id`: ID de la meta que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. PUT /updateMonto/:meta_id

o Descripción: Actualiza el monto actual de una meta de ahorro.

o Controlador: putMontoActual

o Parámetros:

§ meta_id: ID de la meta cuyo monto se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el monto adicional.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:meta_id

o Descripción: Elimina una meta de ahorro existente por su ID.

o Controlador: deleteMetaAhorro

o Parámetros:

§ meta_id: ID de la meta que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

```
metasAhorroRoutes.js > ...
const express = require("express");
const router = express.Router();
const metasAhorroController = require("../controllers/metaAhorroController");
const token = require("../middleware/authMiddleware");

outer.get("/", metasAhorroController.getMetasAhorro);
outer.get("/user/:usuario_id", metasAhorroController.getMetasByUserId);
outer.get("/category/:categoria_meta_id", metasAhorroController.getMetasByCategory);
outer.post("/create", token.verificarToken, metasAhorroController.postMetaAhorro);
outer.put("/update/:meta_id", metasAhorroController.putMetaAhorro);
outer.put("/updateMonto/:meta_id", metasAhorroController.putMontoActual);
outer.delete("/delete/:meta_id", metasAhorroController.deleteMetaAhorro);

module.exports = router;
```

Rutas de Notificaciones

El archivo `notificacionesRoutes.js` define las rutas de la API para las operaciones relacionadas con las notificaciones de los usuarios. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `notificacionesController`. También incluye un middleware para la verificación del token **JWT**.

Importaciones

- `express`: Se importa el módulo express para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `notificacionController`: Se importa el controlador que maneja la lógica de negocio relacionada con las notificaciones.
- `token`: Se importa el middleware de autenticación que verifica la validez del token **JWT**.

Rutas Definidas

1. GET /user/:usuario_id

o Descripción: Obtiene todas las notificaciones de un usuario específico por su ID.

o Controlador: `getNotificacionesByUserId`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Parámetros:

§ `usuario_id`: ID del usuario cuyas notificaciones se desean obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de notificaciones en formato JSON.

1. POST /create

o Descripción: Crea una nueva notificación.

o Controlador: `postNotificacion`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para crear la notificación.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:notificacion_id

o Descripción: Actualiza una notificación existente por su ID.

o Controlador: `putNotificacion`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Parámetros:

§ `notificacion_id`: ID de la notificación que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:notificacion_id

o Descripción: Elimina una notificación existente por su ID.

o Controlador: `deleteNotificacion`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Parámetros:

§ `notificacion_id`: ID de la notificación que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

```
const express = require("express");
const router = express.Router();
const notificacionController = require("../controllers/notificacionesController");
const token = require("../middleware/authMiddleware");

router.get("/user/:usuario_id", token.verificarToken, notificacionController.getNotificacionesByUserId);
router.post("/create", token.verificarToken, notificacionController.postNotificacion);
router.put("/update/:notificacion_id", token.verificarToken, notificacionController.putNotificacion);
router.delete("/delete/:notificacion_id", token.verificarToken, notificacionController.deleteNotificacion);

module.exports = router;
```

Rutas de Recordatorios

El archivo recordatoriosRoutes.js define las rutas de la API para las operaciones relacionadas con los recordatorios de los usuarios. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador recordatorioController. También incluye un middleware para la verificación del token JWT.

Importaciones

- express: Se importa el módulo express para crear un enrutador.
- router: Se crea una instancia de express.Router() que se utiliza para definir las rutas de la API.
- recordatorioController: Se importa el controlador que maneja la lógica de negocio relacionada con los recordatorios.
- token: Se importa el middleware de autenticación que verifica la validez del token JWT.

Rutas Definidas

1. GET/

o Descripción: Obtiene todos los recordatorios.

o Controlador: getRecordatorios

o Respuesta Esperada: Retorna un código de estado 200 y una lista de recordatorios en formato JSON.

1. GET /user/:usuario_id

o Descripción: Obtiene todos los recordatorios de un usuario específico por su ID.

o Controlador: getRecordatoriosByUserId

o Middleware: token.verificarToken (requiere un token válido para acceder).

o Parámetros:

§ usuario_id: ID del usuario cuyos recordatorios se desean obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de recordatorios del usuario en formato JSON.

1. POST /create

o Descripción: Crea un nuevo recordatorio.

o Controlador: postRecordatorio

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para crear el recordatorio.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:recordatorio_id

o Descripción: Actualiza un recordatorio existente por su ID.

o Controlador: putRecordatorio

o Middleware: token.verificarToken (requiere un token válido para acceder).

o Parámetros:

§ recordatorio_id: ID del recordatorio que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:recordatorio_id

o Descripción: Elimina un recordatorio existente por su ID.

o Controlador: deleteRecordatorio

o Parámetros:

§ recordatorio_id: ID del recordatorio que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

```
const express = require("express");
const router = express.Router();
const recordatorioController = require("../controllers/recordatorioController");
const token = require("../middleware/authMiddleware");

router.get("/", recordatorioController.getRecordatorios);
router.get("/user/:usuario_id", token.verificarToken, recordatorioController.getRecordatoriosByUserId);
router.post("/create", recordatorioController.postRecordatorio);
router.put("/update/:recordatorio_id", token.verificarToken, recordatorioController.putRecordatorio);
router.delete("/delete/:recordatorio_id", recordatorioController.deleteRecordatorio);

module.exports = router;
```

Backend

Rutas de Reportes

El archivo `reportesRoutes.js` define las rutas de la API para las operaciones relacionadas con los reportes. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `reportesController`. Además, incluye un middleware para la verificación del token JWT en ciertas rutas.

Importaciones

- `express`: Se importa el módulo `express` para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `reportesController`: Se importa el controlador que maneja la lógica de negocio relacionada con los reportes.
- `token`: Se importa el middleware de autenticación que verifica la validez del token JWT.

Rutas Definidas

1. GET /

o Descripción: Obtiene todos los reportes.

o Controlador: `getReportes`

o Respuesta Esperada: Retorna un código de estado 200 y una lista de reportes en formato JSON.

1. GET /user/:usuario_id

o Descripción: Obtiene todos los reportes de un usuario específico por su ID.

o Controlador: `getReportesByUserId`

o Parámetros:

§ `usuario_id`: ID del usuario cuyos reportes se desean obtener (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y una lista de reportes del usuario en formato JSON.

1. POST /create

o Descripción: Crea un nuevo reporte.

o Controlador: `postReporte`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para crear el reporte.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si la creación es exitosa.

1. PUT /update/:reporte_id

o Descripción: Actualiza un reporte existente por su ID.

o Controlador: `putReporte`

o Middleware: `token.verificarToken` (requiere un token válido para acceder).

o Parámetros:

§ `reporte_id`: ID del reporte que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:reporte_id

o Descripción: Elimina un reporte existente por su ID.

o Controlador: `deleteReporte`

o Parámetros:

§ `reporte_id`: ID del reporte que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

```
const express = require("express");
const router = express.Router();
const reportesController = require("../controllers/reportesController");
const token = require("../middleware/authMiddleware");

// Obtener todos los reportes
router.get("/", reportesController.getReportes);

// Obtener reportes por ID de usuario
router.get("/user/:usuario_id", reportesController.getReportesByUserId);

// Crear un nuevo reporte (requiere autenticación)
router.post("/create", token.verificarToken, reportesController.postReporte);

// Actualizar un reporte por ID (requiere autenticación)
router.put("/update/:reporte_id", token.verificarToken, reportesController.putReporte);

// Eliminar un reporte por ID
router.delete("/delete/:reporte_id", reportesController.deleteReporte);

module.exports = router;
```

Rutas de Usuarios

El archivo `usuariosRoutes.js` define las rutas de la API para las operaciones relacionadas con los usuarios. Utiliza el framework Express para gestionar las solicitudes HTTP y vincula las rutas a los métodos del controlador `usuariosController`.

Importaciones

- `express`: Se importa el módulo `express` para crear un enrutador.
- `router`: Se crea una instancia de `express.Router()` que se utiliza para definir las rutas de la API.
- `usuariosController`: Se importa el controlador que maneja la lógica de negocio relacionada con los usuarios.

Rutas Definidas

1. GET /

o Descripción: Obtiene todos los usuarios registrados en el sistema.

o Controlador: `getUsuarios`

o Respuesta Esperada: Retorna un código de estado 200 y una lista de usuarios en formato JSON.

1. GET /info

o Descripción: Obtiene información adicional sobre los usuarios, como el total de usuarios y estadísticas del mes actual.

o Controlador: `getInfoUsuarios`

o Respuesta Esperada: Retorna un código de estado 200 y un objeto JSON que contiene información detallada sobre los usuarios.

1. POST /

o Descripción: Registra un nuevo usuario en el sistema.

o Controlador: `createUsuarios`

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos necesarios para el registro del usuario.

o Respuesta Esperada: Retorna un código de estado 201 y un mensaje de éxito si el registro es exitoso.

1. POST /login

o Descripción: Inicia sesión de un usuario existente.

o Controlador: `loginUsuario`

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga el email y la password_usuario del usuario.

o Respuesta Esperada: Retorna un código de estado 200, un mensaje de éxito y un token de autenticación si las credenciales son correctas.

1. PUT /update/:usuario_id

o Descripción: Actualiza la información de un usuario existente para su ID.

o Controlador: `putUsuario`

o Parámetros:

§ `usuario_id`: ID del usuario que se desea actualizar (proporcionado en la URL).

o Cuerpo de Solicitud: Se espera un objeto JSON que contenga los campos a actualizar.

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la actualización es exitosa.

1. DELETE /delete/:usuario_id

o Descripción: Elimina un usuario existente para su ID.

o Controlador: `deleteUsuario`

o Parámetros:

§ `usuario_id`: ID del usuario que se desea eliminar (proporcionado en la URL).

o Respuesta Esperada: Retorna un código de estado 200 y un mensaje de éxito si la eliminación es exitosa.

```

1 const express = require('express');
2 const router = express.Router();
3 const usuariosController = require('../controllers/usuariosController')
4
5 router.get("/", usuariosController.getUsuarios);
6 router.get("/info", usuariosController.getInfoUsuarios);
7 router.post("/register", usuariosController.createUsuarios);
8 router.post("/login", usuariosController.loginUsuario)
9 router.put("/update/:usuario_id", usuariosController.putUsuario)
10 router.delete("/delete/:usuario_id", usuariosController.deleteUsuario)
11
12 module.exports = router

```

Backend

Documentación del Servidor en server.js

Este archivo establece y configura el servidor utilizando Express.js. Se encarga de manejar las rutas de la aplicación, la configuración de middleware, la gestión de CORS y la conexión con APIs externas.

Importaciones

- express: Framework utilizado para crear el servidor y gestionar las rutas.
- dotenv: Módulo para cargar variables de entorno desde un archivo .env.
- node-fetch: Permite realizar solicitudes HTTP desde el servidor.
- cors: Middleware para habilitar CORS (Cross-Origin Resource Sharing).
- ./cronJobs: Se requiere para ejecutar tareas programadas automáticamente.

Configuración del Servidor

1. Creación de la Aplicación:

o Se inicializa una instancia de Express mediante const app = express();;

1. Carga de Variables de Entorno:

o Se carga el archivo .env para acceder a variables sensibles como las credenciales de la base de datos y las claves de las APIs.

1. Configuración de CORS:

o Se habilita CORS para permitir solicitudes desde el frontend, especificando el origen permitido y los métodos aceptados.

o Se configura para permitir el uso de credenciales (como cookies) en las solicitudes.

1. Middleware:

o Se utiliza express.json() para que el servidor pueda parsear el cuerpo de las solicitudes con formato JSON.

```
const app = express();
require("dotenv").config();
const fetch = (...args) => import("node-fetch").then(({ default: fetch }) => fetch(...args));
const cors = require("cors");
require("./cronJobs");
const port = process.env.PORT;
const NEWS_API_KEY = process.env.NEWSDATA_API_KEY;
```

Backend

Rutas

- Las rutas se importan desde sus respectivos controladores y se asignan a prefijos específicos en la aplicación. Las rutas disponibles son:

- o Usuarios: /api/usuarios
- o Categorías de Metas: /api/categoriasMetas
- o Categorías de Gastos: /api/categoriasGastos
- o Gastos: /api/gastos
- o Metas de Ahorro: /api/metas
- o Recordatorios: /api/recordatorios
- o Reportes: /api/reportes
- o Ingresos: /api/ingresos
- o Notificaciones: /api/notificaciones

Funcionalidad de API Externa

- Ruta para Obtener Artículos:

- o Endpoint: /api/articles

o Descripción: Esta ruta permite a los usuarios obtener artículos de la API de NewsData.io utilizando una palabra clave.

- o Manejo de Solicitudes:

§ Se extrae la palabra clave de la consulta (por defecto, "finance").

§ Se construye la URL para realizar la solicitud a la API externa.

§ Se registra la solicitud en la consola para verificación.

§ Se maneja la respuesta y los posibles errores, enviando los artículos al cliente o un mensaje de error en caso de fallo.

```
app.use("/api/usuarios", usuariosRoutes);
app.use("/api/categoriasMetas", categoriasMetasRoutes);
app.use("/api/categoriasGastos", categoriasGastosRoutes);
app.use("/api/gastos", gastosRoutes);
app.use("/api/metas", metasAhorroRoutes);
app.use("/api/recordatorios", recordatoriosRoutes);
app.use("/api/reportes", reportesRoutes);
app.use("/api/ingresos", ingresoRoutes);
app.use("/api/notificaciones", notificacionesRoutes);

// Ruta para obtener artículos de NewsData.io
app.get("/api/articles", async (req, res) => {
  const { keyword = "finance" } = req.query;
  const url = `https://newsdata.io/api/1/news?apikey=${NEWS_API_KEY}&q=${keyword}&language=es`;

  try {
    console.log("Solicitando artículos de NewsData.io:", url); // Log para verificar URL
    const response = await fetch(url);

    // Verifica si la respuesta es válida
    if (!response.ok) {
      const errorText = await response.text();
      throw new Error(`Error fetching articles: ${errorText}`);
    }

    const data = await response.json();
    console.log("Artículos recibidos de NewsData.io:", data.results); // Log de los resultados
    res.json(data.results); // Envía los artículos al frontend
  } catch (error) {
    console.error("Error en la solicitud a NewsData.io:", error);
    res.status(500).json({ error: "Error fetching articles" });
  }
})
```

Backend

Gastoscontroller Modificado.

El componente gastosController.js gestiona las operaciones relacionadas con los gastos de los usuarios en el sistema. Su objetivo es permitir la creación, actualización, eliminación y consulta de los gastos registrados, tanto a nivel general como por usuario o categoría. Además, se incorpora un sistema de puntos asociado a cada gasto añadido. A continuación, se describe la funcionalidad y los métodos disponibles.

Métodos:

1. getGastos

Descripción:

Este método obtiene todos los gastos registrados en el sistema.

Flujo de operación:

- Llama a Gasto.getAll para obtener todos los gastos.
- Si ocurre un error durante la consulta, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y los datos obtenidos.

Ruta:

GET /gastos

```
const Gasto = require("../models/gastos");

const getGastos = (req, res) => {
  Gasto.getAll((err, data) => {
    if (err) {
      return res.status(500).json({ error: "Error al obtener los Gastos" });
    }
    res.status(200).json(data);
  });
};
```

getGastoByCategoria

Descripción:

Este método obtiene todos los gastos relacionados con una categoría específica, identificada por categoria_gasto_id.

Flujo de operación:

- Llama a Gasto.getByCategoriaId con el categoria_gasto_id recibido en los parámetros.
- Si ocurre un error, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y los datos de los gastos en esa categoría.

Ruta:

GET /gastos/categoría/:categoria_gasto_id

Respuesta exitosa:

json

Copiar código

```
27
28   Gasto.getByCategoriaId(categoría_gasto_id, (err, data) => {
29     if (err) {
30       return res.status(500).json({ error: "Error al obtener los gastos" });
31     }
32     res.status(200).json(data);
33   });
34 };
35
36 const postGasto = (req, res) => {
37   const { monto, categoria_gasto_id, descripción } = req.body;
38   const usuario_id = req.userId; // Obtenemos usuario_id del token
39
40   // Datos para crear el nuevo gasto
41   const nuevoGasto = {
42     monto,
43     categoria_gasto_id,
44     descripción,
45   };
46
47   // Llamamos al modelo para crear el gasto
```

Backend

postGasto

Descripción:

Este método permite crear un nuevo gasto para el usuario autenticado, que se obtiene del token de usuario. Además, el usuario recibe puntos para añadir un gasto.

Flujo de operación:

- Recibe las parámetros monto, categoria_gasto_id y descripción desde el cuerpo de la solicitud.
- Crea el nuevo gasto utilizando Gasto.create, asociando el gasto al usuario_id obtenido del token.
- Despues de crear el gasto, asigna 10 puntos al usuario usando Gasto.agregarPuntos.
- Si hay algún error durante la creación del gasto o la actualización de puntos, responde con un error apropiado.
- Si ambos procesos son exitosos, responde con un código de estado 201 y un mensaje de éxito junta con los puntos obtenidos.

```
const postGasto = (req, res) => {
  const { monto, categoria_gasto_id, descripcion } = req.body;
  const usuario_id = req.userid; // Obtenemos usuario_id del token

  // Datos para crear el nuevo gasto
  const nuevoGasto = [
    monto,
    categoria_gasto_id,
    descripcion,
  ];
};
```

putGasto

Descripción:

Este método permite actualizar un gasto existente para el usuario autenticado.

Flujo de operación:

- Recibe el id_gasto en las parámetros de la URL y los nuevos valores monto, categoria_gasto_id y descripción en el cuerpo de la solicitud.
- Si alguno de los campos es invalido o falta, responde con un error de 400.
- Llama a Gasto.updateData para actualizar el gasto en la base de datos.
- Si ocurre un error o no se encuentra el gasto, responde con un mensaje adecuado.
- Si la actualización es exitosa, responde con un mensaje de éxito.

Ruta:

PUT /gastos/:id_gasto

```
77 // Extrae el usuario_id del token en la solicitud
78 const usuario_id = req.userId;
79
80 if (!usuario_id || !monto || !categoria_gasto_id || !descripcion) {
81   return res.status(400).json({ error: "Todos los campos son obligatorios" });
82 }
83
84 const updateData = { monto, categoria_gasto_id, descripcion };
85
86 Gasto.updateData(id_gasto, usuario_id, updateData, (err, result) => {
87   if (err) {
88     return res
89       .status(500)
90       .json({ error: "Error al actualizar gasto", detalles: err });
91   }
92
93   if (result.affectedRows === 0) {
94     return res.status(403).json({
```

Gastoscontroller Modificado

El componente metasAhorroController.js gestiona las operaciones relacionadas con las metas de ahorro de los usuarios en el sistema. Su objetivo es permitir la creación, actualización, eliminación, consulta y seguimiento de las metas de ahorro, así como la asignación de puntos por completar o eliminar metas.

Métodos:

1. getMetasAhorro

Descripción:

Este método obtiene todas las metas de ahorro registradas en el sistema.

Flujo de operación:

- Llama a MetaAhorro.getAll para obtener todas las metas de ahorro.
- Si ocurre un error durante la consulta, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y los datos obtenidos.

Ruta:

GET /metas-ahorro

```

3  const getMetasAhorro = (req, res) => {
4    MetaAhorro.getAll((err, data) => {
5      if (err) {
6        return res.status(500).json({ error: "Error al obtener las metas" });
7      }
8      res.status(200).json(data);
9    });
10  };
11

```

getMetasByUserId

Descripción:

Este método obtiene las metas de ahorro de un usuario específico basado en el usuario_id proporcionado.

Flujo de operación:

- Llama a MetaAhorro.getByUserId con el usuario_id recibido en los parámetros.
- Si ocurre un error, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y los datos de las metas del usuario.

Ruta:

GET /metas-ahorro/usuario/:usuario_id

Respuesta exitosa:

```

const getMetasByUserId = (req, res) => {
  const { usuario_id } = req.params;

  MetaAhorro.getByUserId(usuario_id, (err, data) => {
    if (err) {
      return res
        .status(500)
        .json({ error: "Error al obtener las metas del usuario" });
    }
    res.status(200).json(data);
  });
};

```

Backend

getMetasByCategoria

Descripción:

Este método obtiene todas las metas de ahorro relacionadas con una categoría específica, identificada por categoria_meta_id.

Flujo de operación:

- Llama a MetaAhorro.getByCategoriaId con el categoria_meta_id recibido en las parámetros.
- Si ocurre un error, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y las datos de las metas en esa categoría.

Ruta:

GET /metas-ahorro/categoría/:categoria_meta_id

```
28 ✓  MetaAhorro.getByCategoriaId(categoria_meta_id, (err, data) => {
29 ✓    if (err) {
30      return res.status(500).json({ error: "Error al obtener las metas" });
31    }
32    res.status(200).json(data);
33  });
34  };
35
36 ✓ const postMetaAhorro = (req, res) => {
37 ✓  const {
38    nombre_meta,
39    monto_objetivo,
40    fecha_limite,
41    descripcion,
42    categoria_meta_id,
43  } = req.body;
44  const usuario_id = req.userId; // Obtenemos usuario_id del token
45
46  // Validación de campos obligatorios
47 ✓  if (!nombre_meta || !monto_objetivo || !fecha_limite || !categoria_meta_id) {
48 ✓    return res.status(400).json({
```

postMetaAhorro

Descripción:

Este método permite crear una nueva meta de ahorro para el usuario autenticado.

Flujo de operación:

- Recibe los parámetros nombre_meta, monto_objetivo, fecha_limite, descripción y categoria_meta_id desde el cuerpo de la solicitud.
- Verifica que todos los campos obligatorios estén presentes.
- Crea la nueva meta utilizando MetaAhorro.create y la asocia al usuario_id obtenido del token.
- Despues de crear la meta, asigna 20 puntos al usuario usando MetaAhorro.agregarPuntos.
- Si todo es exitoso, responde con un código de estado 201 y las datos de la meta junta con las puntos obtenidos.

Ruta:

POST /metas-ahorro

```
st postMetaAhorro = (req, res) => {
  const {
    nombre_meta,
    monto_objetivo,
    fecha_limite,
    descripcion,
    categoria_meta_id,
  } = req.body;
  const usuario_id = req.userId; // Obtenemos usuario_id del token

  // Validación de campos obligatorios
  if (!nombre_meta || !monto_objetivo || !fecha_limite || !categoria_meta_id)
    return res.status(400).json({
      error: "Todos los campos obligatorios deben ser proporcionados",
    });
```

Backend

putMetaAhorro

Descripción:

Este método permite actualizar una meta de ahorro existente para el usuario autenticado.

Flujo de operación:

- Recibe el `meta_id` en los parámetros de la URL y los nuevos valores `nombre_meta`, `monto_objetivo`, `monto_actual`, `descripcion`, y `estado_de_meta` en el cuerpo de la solicitud.
- Verifica que `meta_id` sea válido y que los campos de actualización sean proporcionados.
- Llama a `MetaAhorro.updateData` para actualizar la meta en la base de datos.
- Si ocurre un error o no se encuentra la meta, responde con un mensaje adecuado.
- Si la actualización es exitosa, responde con un mensaje de éxito.

Ruta:

PUT /metas-ahorro/:meta_id

Cuerpo de la solicitud:

json

Copiar código

```
const putMetaAhorro = (req, res) => {
  const { meta_id } = req.params;

  const {
    nombre_meta,
    monto_objetivo,
    monto_actual,
    descripcion,
    estado_de_meta,
  } = req.body;

  // Verificar que meta_id sea un número válido
  if (!meta_id || isNaN(meta_id)) {
    return res.status(400).json({ error: "ID de la meta no es válido" });
  }

  // Filtrar solo los campos que están presentes en el cuerpo de la solicitud
  let updateData = {};
```

UsuariosController Modificado

Documentación del Componente de Gestión de Usuarios

El componente `usuariosController.js` gestiona las operaciones relacionadas con los usuarios en el sistema. Su objetivo es permitir la creación, actualización, eliminación, consulta de usuarios, la gestión de puntos y la autenticación a través de JWT. Además, se incorpora un sistema de racha de login y recompensas premium basadas en puntos. A continuación, se detalla la funcionalidad y los métodos disponibles.

Métodos:

1. getUsers

Descripción:

Este método obtiene todos los usuarios registrados en el sistema.

Flujo de operación:

- Llama a `Usuario.getAll` para obtener todos los usuarios.
- Si ocurre un error durante la consulta, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con un código de estado 200 y los datos obtenidos.

Ruta:

GET /usuarios

```
6
7  const getUsers = (req, res) => {
8    Usuario.getAll((err, data) => {
9      if (err) {
10        return res.status(500).json({ error: "Error al obtener usuarios" });
11      }
12      res.status(200).json(data);
13    });
14  };
15
```

Backend

getInfoUsuarios

Descripción:

Este método obtiene información sobre el total de usuarios, usuarios del mes actual y datos para generar una gráfica de usuarios.

Flujo de operación:

- Llama a `Usuario.getTotalUsuarios` para obtener el total de usuarios.
- Llama a `Usuario.getUsuariosMesActual` para obtener los usuarios registrados en el mes actual.
- Llama a `Usuario.getGraficaUsuarios` para obtener los datos de la gráfica.
- Si ocurre un error, responde con un código de estado 500 y un mensaje de error.
- Si la consulta es exitosa, responde con los datos combinados.

Ruta:

GET /usuarios/info

```
26     if (err) {
27       return res
28         .status(500)
29         .json({ error: "Error al obtener usuarios del mes actual" });
30     }
31
32     Usuario.getGraficaUsuarios((err, graficaUsuarios) => {
33       if (err) {
34         return res
35           .status(500)
36           .json({ error: "Error al obtener datos para la gráfica" });
37     }
38   }
```

createUsuarios

Descripción:

Este método permite crear un nuevo usuario en el sistema. La contraseña se encripta utilizando bcrypt antes de almacenarla.

Flujo de operación:

- Recibe los parámetros `nombre_usuario`, `email`, y `password_usuario` desde el cuerpo de la solicitud.
- Encripta la contraseña con `bcrypt.hashSync`.
- Llama a `Usuario.create` para crear al nuevo usuario.
- Si la creación es exitosa, responde con un código de estado 201 y un mensaje de éxito.

Ruta:

POST /usuarios

loginUsuario

Descripción:

Este método permite a un usuario iniciar sesión. Verifica las credenciales y genera un token JWT para autenticar al usuario.

Flujo de operación:

- Recibe el `email` y `password_usuario` desde el cuerpo de la solicitud.
- Busca el usuario en la base de datos utilizando `Usuario.getByEmail`.
- Verifica la contraseña utilizando `bcrypt.compareSync`.
- Si la autenticación es exitosa, genera un token JWT.
- Responde con un código de estado 200 y el token generado.

Ruta:

POST /usuarios/login

```
  console.error(`El error es ${err}`);
  return res.status(404).json({ error: "Usuario no encontrado" });
}

// Verifica que el campo de contraseña no sea undefined
if (!user.password_usuario) {
  return res
    .status(500)
    .json({ error: "Error en el servidor: contraseña no definida" });
}
```

Backend

putUsuario

Descripción:

Este método permite actualizar los datos de un usuario existente. Se pueden actualizar los campos como nombre, email, contraseña y otros.

Flujo de operación:

- Recibe el usuario_id en los parámetros de la URL y los datos a actualizar en el cuerpo de la solicitud.
- Encripta la nueva contraseña (si se proporciona).
- Llama a Usuario.updateData para actualizar los datos del usuario.
- Si la actualización es exitosa, responde con un mensaje de éxito.

Ruta:

PUT /usuarios/:usuario_id

```
if (updateErr) {
    console.error("Error al actualizar la racha:", updateErr);
    return res.status(500).json({ error: "Error al actualizar la racha" });
}

// Generamos el token JWT
const token = jwt.sign(
    { id: user.usuario_id, rol: user.rol }, // Incluye el rol en el token
    process.env.JWT_SECRET,
    {
        expiresIn: 86400, // 24 horas
    }
);

res.status(200).json({
    message: "Login exitoso",
```

getPuntosUsuario

Descripción:

Este método obtiene la cantidad de puntos de un usuario.

Flujo de operación:

- Recibe el usuario_id en los parámetros de la URL.
- Consulta los puntos del usuario en la base de datos.
- Si el usuario es encontrado, responde con los puntos del usuario.

Ruta:

GET /usuarios/:usuario_id/puntos

```
db.query(query, [usuario_id], (err, result) => {
    if (err) {
        console.error(`Error al obtener los puntos del usuario: ${err}`);
        return res
            .status(500)
            .json({ error: "Error al obtener los puntos del usuario" });
    }

    if (result.length === 0) {
        return res.status(404).json({ message: "Usuario no encontrado" });
    }
```

Backend

canjearRecompensaPremium

Descripción:

Este metodo permite a los usuarios canjear una recompensa premium si tienen suficientes puntos acumulados.

Flujo de operación:

- Verifica que el usuario tenga al menos 100 puntos.
- Si tiene suficientes puntos, los resta y actualiza el tipo de suscripción a premium, añadiendo una nueva fecha de vencimiento.
- Si el proceso es exitoso, responde con un mensaje de éxito.

Ruta:

POST /usuarios/canjear-recompensa

```
245 // Comprobar si el usuario tiene suficientes puntos
246 const query = `SELECT puntos FROM usuarios WHERE usuario_id = ?`;
247 db.query(query, [usuario_id], (err, results) => {
248   if (err) {
249     console.error("Error al obtener los puntos del usuario:", err);
250     return res.status(500).json({ error: "Error al obtener los puntos" });
251   }
252
253   const puntosUsuario = results[0].puntos;
254   if (puntosUsuario < puntosRequeridos) {
255     return res.status(400).json({
256       error: "No tienes suficientes puntos para canjear esta recompensa.",
257     });
258   }
259
260   // Restar los puntos y actualizar la suscripción
261   const nuevaFechaVencimiento = new Date();
262   nuevaFechaVencimiento.setDate(nuevaFechaVencimiento.getDate() + 2); // Agregar 2 días
263
264   const updateQuery = `UPDATE usuarios SET puntos = puntos - ?, tipo_suscripcion = 'premium', fecha_vencimiento_premium = ? WHERE usuario_id = ?`;
265   db.query(updateQuery, [puntosRequeridos, nuevaFechaVencimiento, usuario_id], (err, results) => {
266     if (err) {
267       console.error("Error al actualizar la suscripción:", err);
268       return res.status(500).json({ error: "Error al actualizar la suscripción" });
269     }
270     res.json({ message: "Recompensa canjeadas exitosamente." });
271   });
272 });
273 
```

Modelogastos Modificado.

El modelo Gasto.js gestiona las operaciones relacionadas con los gastos en la base de datos. Su objetivo es permitir la obtención, creación, actualización, eliminación y gestión de los gastos de los usuarios, así como la asignación de puntos a los usuarios cuando se añade un gasto.

Métodos:

1. getAll

Descripción:

Este metodo obtiene todos los gastos registrados en la base de datos.

Flujo de operación:

- Ejecuta la consulta SQL para seleccionar todos los registros de la tabla gastos.
- Si hay un error, se ejecuta el callback con el error.
- Si la consulta es exitosa, devuelve los resultados.

```
1: (callback) => {
2:   const query = "SELECT * FROM gastos";
3:   db.query(query, (err, results) => {
4:     if (err) return callback(err, null);
5:     callback(null, results);
6:   });
7:
8:
9: // Obtener todos los gastos de un usuario por ID
10: getUserId: (usuario_id, callback) => {
11:   const query = `
12:     SELECT gastos.*, categorias_gasto.nombre_categoria
13:     FROM gastos
14:     JOIN categorias_gasto ON gastos.categoría_gasto_id = categorias_gasto.categoría_gasto_id
15:     WHERE gastos.usuario_id = ?`;
16:
17:   db.query(query, [usuario_id], (err, results) => {
18:     if (err) return callback(err, null);
19:     callback(null, results);
20:   });
21: }
```

Backend

getByIdUser

Descripción:

Este método obtiene todos los gastos de un usuario específico según(el usuario_id). Además, obtiene el nombre de la categoría del gasto.

Flujo de operación:

- Ejecuta una consulta SQL para obtener los gastos de un usuario, uniendo la tabla gastos con la tabla categorias_gasto para obtener el nombre de la categoría.
- Si hay un error, se ejecuta el callback con el error.
- Si la consulta es exitosa, devuelve los resultados.

```
// Acciona: Devuelve los gastos de un usuario por su id
getByIdUser: (usuario_id, callback) => {
  const query = `
    SELECT gastos.*, categorias_gasto.nombre_categoria
    FROM gastos
    JOIN categorias_gasto ON gastos.categoría_gasto_id = categorias_gasto.categoría_gasto_id
    WHERE gastos.usuario_id = ?
  `;

  db.query(query, [usuario_id], (err, results) => {
    if (err) return callback(err, null);
    callback(null, results);
  });
},
```

getByIdCategoria

Descripción:

Este método obtiene todos los gastos de una categoría específica, identificada por el categoria_gasto_id.

Flujo de operación:

- Ejecuta una consulta SQL para seleccionar los gastos cuyo categoria_gasto_id coincide con el valor proporcionado.
- Si hay un error, se ejecuta el callback con el error.
- Si no se encuentran resultados, devuelve null.
- Si la consulta es exitosa, devuelve los resultados.

```
getByIdCategoria: (categoria_gasto_id, callback) => {
  const query = "SELECT * FROM gastos WHERE categoria_gasto_id = ?";
  db.query(query, [categoria_gasto_id], (err, results) => {
    if (err) return callback(err, null);
    if (results.length === 0) return callback(null, null);
    callback(null, results);
  });
},
```

create

Descripción:

Este método permite crear un nuevo gasto en la base de datos.

Flujo de operación:

- Recibe los datos del gasto (monto, categoria, descripción) y el usuario_id.
- Ejecuta una consulta SQL para insertar un nuevo registro en la tabla gastos.
- Si hay un error, se ejecuta el callback con el error.
- Si la consulta es exitosa, devuelve el resultado de la operación.

```
const query = `
  INSERT INTO gastos (usuario_id, monto, categoria_gasto_id, fecha, descripción)
  VALUES (?, ?, ?, NOW(), ?)
`;
db.query(
  query,
  [usuario_id, monto, categoria_gasto_id, descripción],
  (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  }
},
```

Backend

updateData

Descripción:

Este método permite actualizar un gasto existente utilizando su id_gasto y el usuario_id para asegurarse de que el gasto pertenece al usuario correcto.

Flujo de operación:

- Recibe el id_gasto, usuario_id, y los nuevos datos para actualizar (monto, categoría, descripción).
- Ejecuta una consulta SQL para actualizar el gasto en la base de datos.
- Si hay un error, se ejecuta el callback con el error.
- Si la consulta es exitosa, devuelve el resultado de la operación.

```
const { monto, categoria_gasto_id, descripcion } = data;
const query = `
    UPDATE gastos
    SET monto = ?, categoria_gasto_id = ?, descripcion = ?
    WHERE id_gasto = ? AND usuario_id = ?
`;
db.query(
    query,
    [monto, categoria_gasto_id, descripcion, id_gasto, usuario_id],
    (err, result) => {
        if (err) return callback(err, null);
        callback(null, result);
    }
);
```

agregar Puntos

Descripción:

Este método permite actualizar los puntos de un usuario después de agregar un gasto.

Flujo de operación:

- Recibe el usuario_id y el número de puntos a agregar.
- Ejecuta una consulta SQL para actualizar los puntos del usuario en la base de datos.

```
82 const query =
83     "UPDATE usuarios SET puntos = puntos + ? WHERE usuario_id = ?";
84 db.query(query, [puntos, usuario_id], (err, result) => {
85     if (err) return callback(err, null);
86     callback(null, result);
87 });
88 },
```

Backend

MetasAhorro Modificado.

El modelo MetaAhorro.js gestiona las operaciones relacionadas con las metas de ahorro de los usuarios en la base de datos. Permite obtener, crear, actualizar, eliminar metas de ahorro y asociar puntos a los usuarios cuando se modifican o completan metas. A continuación, se detallan los métodos disponibles y su funcionalidad.

Métodos:

updateData

Descripción:

Este método permite actualizar una meta de ahorro existente en la base de datos.

Flujo de operación:

- Recibe el `meta_id` y los nuevos datos (`nombre_meta`, `monto_objetivo`, `monto_actual`, `descripcion`, `estado_de_meta`).
- Ejecuta una consulta SQL para actualizar los datos de la meta en la tabla `metas_de_ahorro`.
- Si hay un error, se ejecuta el callback con el error.
- Si la consulta es exitosa, devuelve el resultado de la operación.

```
// METADEAHORRO.js
const data = {
  nombre_meta,
  monto_objetivo,
  monto_actual,
  descripcion,
  estado_de_meta,
} = data;
const query = `
  UPDATE metas_de_ahorro
  SET nombre_meta = ?, monto_objetivo = ?, monto_actual = ?, descripcion = ?, estado_de_meta = ?
  WHERE meta_id = ?
`;
db.query(
  query,
  [
    nombre_meta,
    monto_objetivo,
    monto_actual,
```

updateMontoActual

Descripción:

Este método permite actualizar el monto actual de una meta de ahorro.

Flujo de operación:

- Recibe el `meta_id` y el `montoAdicional` a agregar al monto actual.
- Ejecuta una consulta SQL para actualizar el monto actual de la meta en la base de datos.

```
// METADEAHORRO.js
updateMontoActual: (meta_id, montoAdicional, callback) => {
  const query = `
    UPDATE metas_de_ahorro
    SET monto_actual = monto_actual + ?
    WHERE meta_id = ?
  `;
  db.query(query, [montoAdicional, meta_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
},
```

Backend

agregarPuntos

Descripción:

Este método permite agregar puntos a un usuario asociado a una meta de ahorro.

Flujo de operación:

- Recibe el usuario_id y la cantidad de puntos a agregar.
- Ejecuta una consulta SQL para actualizar las puntos del usuario en la base de datos.

Consulta SQL:

```
agregarPuntos: (usuario_id, puntos, callback) => {
  const query =
    "UPDATE usuarios SET puntos = puntos + ? WHERE usuario_id = ?";
  db.query(query, [puntos, usuario_id], (err, result) => {
    if (err) return callback(err, null);
    callback(null, result);
  });
};
```

Modelo usuario Modificado.

El modelo Usuario.js gestiona las operaciones relacionadas con los usuarios en la base de datos. Permite realizar consultas, crear, actualizar y eliminar usuarios, así como gestionar la información de login y la generación de estadísticas relacionadas con los usuarios. A continuación, se detallan los métodos disponibles y su funcionalidad.

Métodos:

getById

Descripción:

Este método obtiene un usuario específico para su usuario_id.

Flujo de operación:

- Ejecuta una consulta SQL para seleccionar el usuario cuyo usuario_id coincide con el valor proporcionado.
- Si hay un error, se ejecuta el callback con el error.
- Si el usuario existe, devuelve el primer registro que coincide con el usuario_id.

•

•

•

```
"SELECT * FROM usuarios WHERE usuario_id = ?",
[id],
(err, results) => {
  if (err) {
    return callback(err, null);
  }
  callback(null, results[0]); // Devuelve el primer usuario que coincide con el ID
};
```

• updateLoginInfo

• Descripción:

Este método permite actualizar la información de login de un usuario, como las fechas de primer login, último login y la racha de inicio de sesión.

• Flujo de operación:

- Recibe usuarioid, firstlogin, lastlogin y racha.
- Ejecuta una consulta SQL para actualizar las datos de login del usuario.

```
// Método para actualizar la información de login
updateLoginInfo: (usuarioid, firstLogin, lastLogin, racha, callback) => {
  const query = `UPDATE usuarios SET first_login = ?, last_login = ?, racha = ? WHERE usuario_id = ?`;
  const values = [firstLogin, lastLogin, racha, usuarioid];
  db.query(query, values, callback);
};
```

Backend

Documentación de la Ruta y Método cumplirMetaAhorro Modificación

```
router.Delete("/delete/metaexitsa/:meta_id", token.verificarToken,  
metasAhorroController.cumplirMetaAhorro);
```

Descripción General:

Esta ruta permite eliminar una meta de ahorro exitosa y otorgar puntos al usuario cuando ha cumplido la meta. La operación es gestionada por el controlador metasAhorroController.cumplirMetaAhorro, y requiere autenticación mediante un token JWT proporcionado por el middleware token.verificarToken.

Este endpoint elimina la meta de ahorro correspondiente (indicado por meta_id) y, como parte del proceso, otorga puntos al usuario para premiar su éxito en el cumplimiento de la meta.

Método HTTP:

DELETE

Parámetros:

- meta_id (en la URL): El ID de la meta de ahorro que se considera cumplida. Este parámetro es necesario para identificar y eliminar la meta correspondiente.

Middleware:

- token.verificarToken: Se asegura de que el usuario esté autenticado antes de permitir el acceso a la operación. Este middleware valida el token JWT enviado en la solicitud.

Controlador:

- metasAhorroController.cumplirMetaAhorro: Este controlador ejecuta la lógica para marcar la meta como exitosa, eliminarla de la base de datos y otorgar los puntos correspondientes al usuario.

Flujo de operación:

1. Autenticación: Antes de que el controlador maneje la solicitud, el middleware token.verificarToken valida que el usuario esté autenticado mediante un token JWT. Si el token no es válido o no está presente, el acceso es denegado.
2. Verificación de meta_id: El parámetro meta_id es extraído de la URL de la solicitud. Este ID es usado para localizar la meta de ahorro en la base de datos.
3. Eliminación de la meta de ahorro: Si la meta de ahorro existe, se elimina de la base de datos.
4. Asignación de puntos: Despues de eliminar la meta, se agregan puntos al usuario como recompensa por haber cumplido la meta de ahorro. Los puntos son sumados a la cuenta del usuario que está asociado con la meta eliminada.
5. Respuesta:
 - Si la operación es exitosa, el servidor devuelve un código de estado 200 OK con un mensaje indicando que la meta ha sido eliminada exitosamente y los puntos fueron otorgados al usuario.
 - Si ocurre algún error durante la operación, se envía una respuesta con un código de error adecuado (por ejemplo, 500 Internal Server Error si la eliminación falla o 404 Not Found si la meta no existe).

```
        return res.status(500).json({ error: "Error al eliminar la meta" });
    }

    if (data.affectedRows === 0) {
        return res.status(404).json({ message: "Meta no encontrada" });
    }

    // Agregar puntos al usuario después de eliminar la meta
    const puntos = 50; // Los puntos que gana el usuario por completar la meta
    MetaAhorro.agregarPuntos(usuario_id, puntos, (err, result) => {
        if (err) {
```

Backend

Documentación de las Rutas de usuariosController.js

A continuación, se documentan las rutas que interactúan con los métodos de usuariosController para realizar operaciones como el canje de recompensas premium, la consulta de puntos del usuario y la obtención de la información de un usuario.

1. Ruta: Canjear Recompensa Premium

```
  "usuario_id", "verificarToken", usuariosController.canjearRecompensaPremium]
```

Descripción:

Esta ruta permite que un usuario canjee sus puntos para una suscripción premium. El canje solo es posible si el usuario tiene suficientes puntos acumulados. Si la transacción es exitosa, el usuario recibe la suscripción premium y los puntos se actualizan en la base de datos.

Método HTTP:

POST

Parámetros:

- `usuario_id` (en la URL): El ID del usuario que está canjeando la recompensa.

Middleware:

- `verificarToken`: Este middleware valida el token JWT del usuario para asegurarse de que está autenticado antes de proceder con el canje.

Controlador:

- `usuariosController.canjeRecompensaPremium`: Gestiona la lógica de verificar si el usuario tiene suficientes puntos y realiza el canje de la recompensa premium.

Flujo de operación:

1. El usuario envía una solicitud de canje de recompensa premium.
2. El middleware verifica el token JWT para asegurar que el usuario esté autenticado.
3. El controlador verifica si el usuario tiene suficientes puntos para canjear la recompensa.
4. Si el usuario tiene suficientes puntos, se realiza la actualización de la suscripción y la fecha de vencimiento.
5. Responde con un mensaje de éxito si la operación es exitosa.

2. Ruta: Obtener Puntos del Usuario

```
  "usuario_id", "verificarToken", usuariosController.getPuntosUsuario]
```

Descripción:

Esta ruta permite consultar los puntos acumulados de un usuario específico, identificando al usuario mediante su `usuario_id`.

Método HTTP:

GET

Parámetros:

- `usuario_id` (en la URL): El ID del usuario para el que se desea obtener la información de puntos.

Middleware:

- `verificarToken`: Este middleware valida el token JWT del usuario para garantizar que la solicitud proviene de un usuario autenticado.

Controlador:

- `usuariosController.getPuntosUsuario`: Gestiona la consulta de puntos del usuario en la base de datos y devuelve la cantidad de puntos que el usuario tiene.

Flujo de operación:

1. El usuario realiza una solicitud para consultar sus puntos.
2. El middleware verifica el token JWT para asegurar que el usuario esté autenticado.
3. El controlador consulta la base de datos para obtener los puntos del usuario.
4. Responde con el número de puntos que el usuario tiene.

Backend

3. Ruta: Obtener Información del Usuario

```
info-user/:id', verificarToken , usuariosController.getUsuarioById]
```

Descripción:

Esta ruta permite obtener información detallada de un usuario específico utilizando su `usuario_id`. Solo se permitirá acceder a la información si el usuario que realiza la solicitud está autenticado y su `usuario_id` coincide con el `usuario_id` proporcionado en la URL.

Método HTTP:

GET

Parámetros:

- `id` (en la URL): El ID del usuario cuya información se desea consultar.

Middleware:

- `verificarToken`: Este middleware valida el token JWT para asegurar que la solicitud proviene de un usuario autenticado y que tiene permisos para consultar los datos del usuario.

Controlador:

- `usuariosController.getUsuarioById`: Gestiona la lógica para obtener la información del usuario de la base de datos.

Flujo de operación:

1. El usuario realiza una solicitud para obtener su propia información.
2. El middleware verifica el token JWT para autenticar al usuario.
3. El controlador verifica que el `usuario_id` en el token coincida con el `id` proporcionado en la URL.
4. Si los IDs coinciden, consulta la base de datos para obtener los datos del usuario.
5. Responde con los datos del usuario.