**Actividad N. 2**

**Aplicación móvil en Kodular consumiendo Api Restful**

Presentado por:

**LUIS FELIPE DÍAZ CABANA**

**LUIGGI MAR SALEMI SOTO**

**Universidad de Cartagena**

**Ingeniería de software**

**Desarrollo de apps**

**2024**

**Tabla de Contenidos**

[Introducción 3](#_Toc148016971)

[Objetivo general 4](#_Toc148016972)

[Objetivos específicos 4](#_Toc148016973)

[Justificación 5](#_Toc148016974)

[Desarrollo 6](#_Toc148016975)

[Síntesis 58](#_Toc148016976)

[Bibliografía 59](#_Toc148016977)

# Introducción

En la era digital actual, el desarrollo de aplicaciones móviles se ha convertido en una herramienta fundamental para facilitar la interacción entre usuarios y servicios. En este contexto, Kodular es una plataforma poderosa que permite a usuarios con diversos niveles de experiencia en programación crear sus propias aplicaciones móviles de manera intuitiva y eficiente. En este trabajo, presentaremos el desarrollo de una aplicación móvil utilizando Kodular que interactúa con una API REST creada con Node.js y Express respaldada por una base de datos MySQL. Esta aplicación móvil está diseñada para ofrecer a los usuarios una experiencia fluida y personalizada al acceder a diferentes funcionalidades a través de la API.

# Objetivo general

Reutilizar la aplicación móvil realizada en Kodular para que ahora se comunique con una API REST implementada en Node.js y Express, con una base de datos MySQL para el almacenamiento de datos. Permitiendo acceder a diversas funcionalidades de manera eficiente.

# Objetivos específicos

* Desarrollar una API REST con Node.js y Express para gestionar las solicitudes del cliente y las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la base de datos MySQL.
* Establecer una comunicación segura entre la aplicación móvil y la API mediante solicitudes HTTP, garantizando la integridad y confidencialidad de los datos transmitidos.
* Permitir que los usuarios accedan a funcionalidades específicas de la aplicación, como iniciar sesión, registrar nuevos usuarios, consultar información y realizar acciones sobre recursos disponibles.

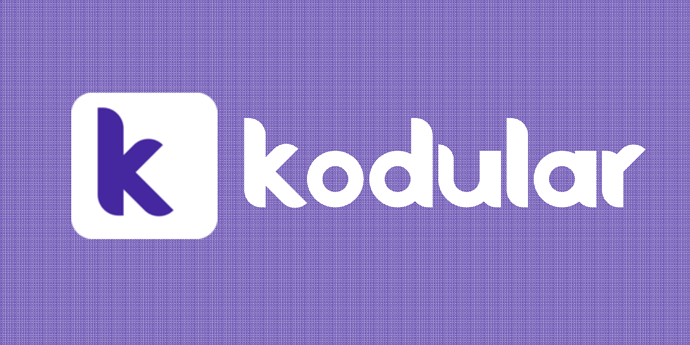
# Justificación

La implementación de una API REST creada con Node.js y Express para que la utilice la aplicación Android, se basa en su accesibilidad, facilidad de personalización y rápido desarrollo. Esta combinación tecnológica permite una rápida implementación de aplicaciones móviles con una interfaz visual intuitiva y funcionalidad personalizada. Además, la arquitectura basada en API REST garantiza la compatibilidad, escalabilidad y seguridad de la aplicación, brindando una experiencia de usuario segura y confiable. Por último, la integración con MySQL como base de datos relacional garantiza la persistencia y la integridad de los datos almacenados.

# Desarrollo

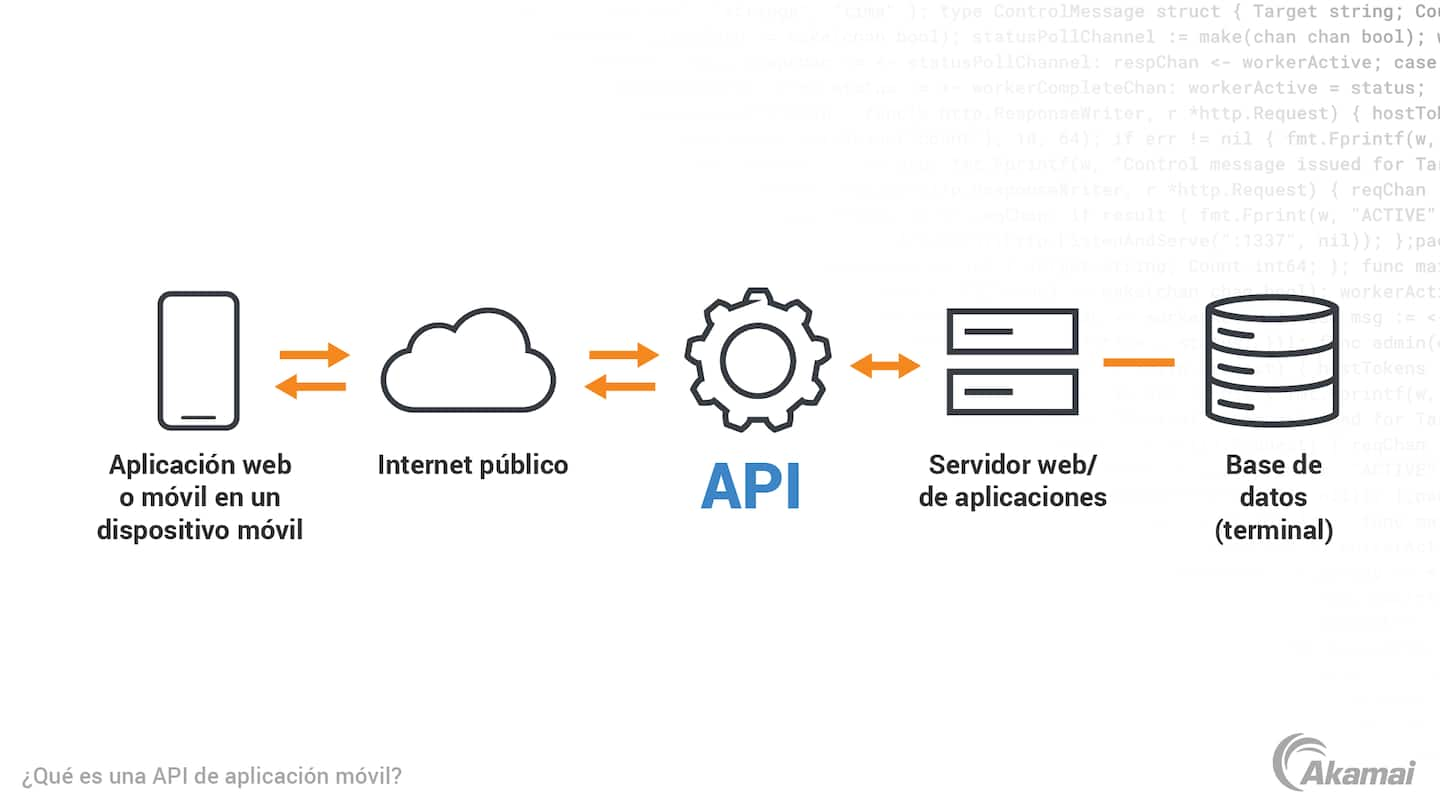
**Tecnologias utilizadas**

**Cliente movil android (frontend)**

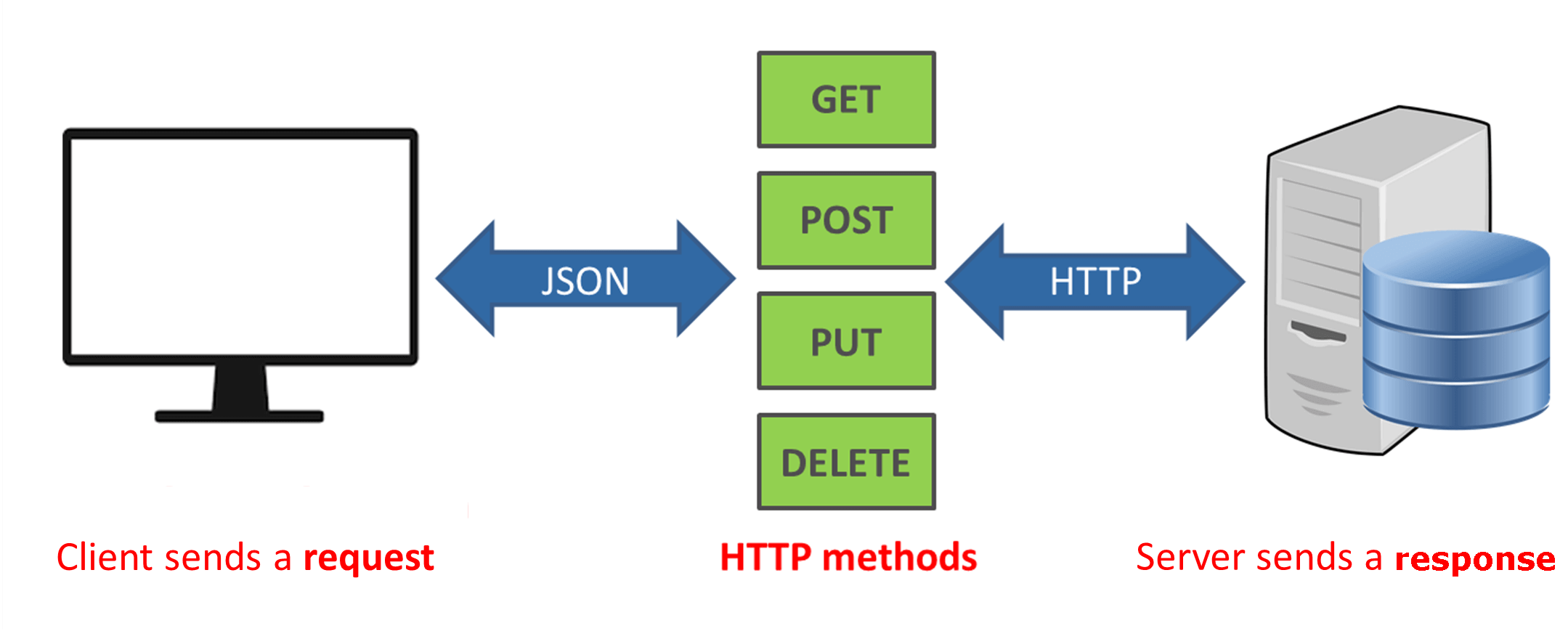


**Kodular:** es una plataforma en línea para el desarrollo de aplicaciones móviles. Su objetivo es proporcionar una forma sencilla de crear aplicaciones para Android, incluso sin conocimientos de programación.

**Arquitectura (backend)**



**Api:** las siglas API corresponden a Application Programming Interfaces, o en español, interfaz de programación de aplicaciones. Es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.

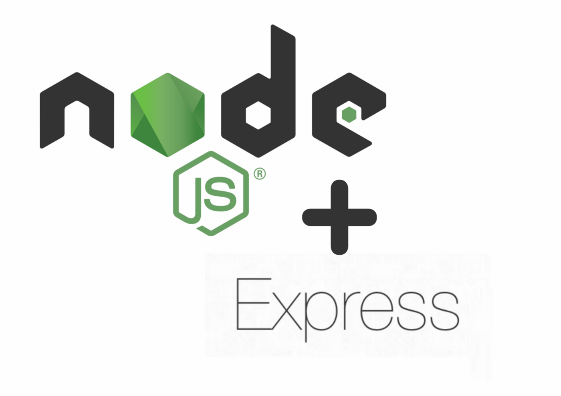


**Api restful:** es una interfaz que dos sistemas de computación utilizan para intercambiar información de manera segura a través de Internet. La mayoría de las aplicaciones para empresas deben comunicarse con otras aplicaciones internas o de terceros para llevar a cabo varias tareas.

**Entorno servidor (lenguaje y framework)**



**Node.js:** Node.js es un entorno en tiempo de ejecución multiplataforma para la capa del servidor basado en JavaScript. Se usa para crear aplicaciones escalables, sin subprocesos, con bucle de eventos y velocidad.



**Express:** es un framework de backend Node.js minimalista, rápido y similar a Sinatra, que proporciona características y herramientas robustas para desarrollar aplicaciones de backend escalables.

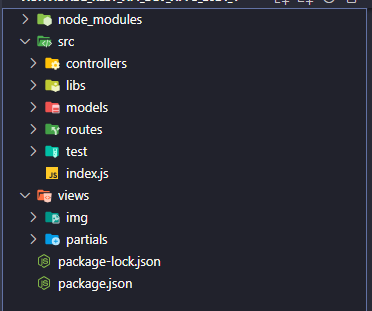
**Persistencia de datos (Motor de base de datos)**

****

**MySQL**: es un sistema de gestión de bases de datos relacionales de código abierto (RDBMS, por sus siglas en inglés) con un modelo cliente-servidor.

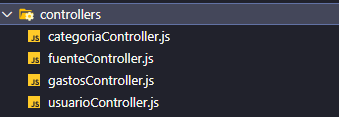
**Estructura del api rest (Backend)**

**Estructura del proyecto**



Como se puede observar, se sigue el patrón de arquitectura MVC para el desarrollo del proyecto.

**Carpeta Controllers**



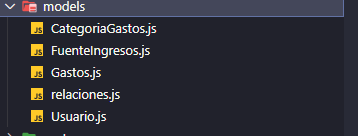
Contiene el controlador por cada modelo para la solicitudes y respuestas del cliente.

**Carpeta libs**



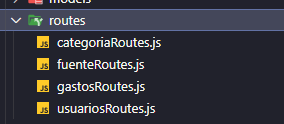
Contiene la conexión a la base de datos por medio de la librería o ORM “Sequelize” que permite el mapeo de las tablas a objetos.

**Carpeta models**



Contiene clases modelo, que representa cada una de las tablas de la base de datos.

**Carpeta routes**



Contiene las rutas de los endpoint para las solicitudes HTTP (Post,Get,Put,Delete) y respuesta de los controladores.

**Archivo index**



Este archivo permite la ejecución del servidor de node.js por lo tanto es el archivo principal para que pueda funcionar. Contiene las reglas y endpoint principales del api.

**Código de los controladores:**

**CategoriaController.js**

const CategoriaGastos = require('../models/CategoriaGastos');

*// Obtener todas las categorías de gastos*

const getAllCategorias = async (req, res) => {

try {

const categorias = await CategoriaGastos.findAll();

res.json(categorias);

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Obtener una categoría de gastos por su ID*

const getCategoriaById = async (req, res) => {

const { id } = req.params;

try {

const categoria = await CategoriaGastos.findByPk(id);

if (categoria) {

res.json(categoria);

} else {

res.status(404).send('Categoría de gastos no encontrada');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Crear una nueva categoría de gastos*

const createCategoria = async (req, res) => {

const { id\_usuario, fecha\_categoria, nombre\_categoria, descripcion\_categoria, icono } = req.body;

try {

const nuevaCategoria = await CategoriaGastos.create({

id\_usuario,

fecha\_categoria,

nombre\_categoria,

descripcion\_categoria,

icono

});

res.status(201).json({

message: 'Categoría de gastos creada exitosamente',

categoria: nuevaCategoria,

});

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Actualizar una categoría de gastos*

const updateCategoria = async (req, res) => {

const { id } = req.params;

const { id\_usuario, fecha\_categoria, nombre\_categoria, descripcion\_categoria, icono } = req.body;

try {

const categoria = await CategoriaGastos.findByPk(id);

if (categoria) {

await categoria.update({

id\_usuario,

fecha\_categoria,

nombre\_categoria,

descripcion\_categoria,

icono

});

res.json({

categoria: categoria,

mensaje: "Categoría de gastos actualizada exitosamente",

});

} else {

res.status(404).send('Categoría de gastos no encontrada');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Eliminar una categoría de gastos*

const deleteCategoria = async (req, res) => {

const { id } = req.params;

try {

const categoria = await CategoriaGastos.findByPk(id);

if (categoria) {

await categoria.destroy();

res.send('Categoría de gastos eliminada exitosamente');

} else {

res.status(404).send('Categoría de gastos no encontrada');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

module.exports = {

getAllCategorias,

getCategoriaById,

createCategoria,

updateCategoria,

deleteCategoria,

};

**EXPLICACION:**

Este controlador administra las operaciones relacionadas con las categorías de gastos en de la aplicación. Importa el modelo CategoriaGastos, que representa las categorías en la base de datos. Luego, ofrece funciones para diferentes acciones: recuperar todas las categorías, obtener una categoría por su ID, crear una nueva categoría, actualizar una existente y eliminar una categoría. Cada función maneja la interacción con la base de datos y devuelve respuestas apropiadas al cliente, ya sea devolviendo los datos solicitados o mensajes de éxito o error en caso de operaciones exitosas o fallidas, respectivamente.

**FuenteController.js**

const FuenteIngresos = require("../models/FuenteIngresos");

const Usuario = require("../models/Usuario");

*// Obtener todas las fuentes de ingresos*

const obtenerTodasFuentes = async (req, res) => {

try {

const fuentes = await FuenteIngresos.findAll();

res.status(200).json({

code: "200",

message: "Fuentes de ingresos obtenidas correctamente",

fuentes,

});

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Obtener una fuente de ingresos por su ID*

const obtenerFuentePorId = async (req, res) => {

const { id\_ingreso } = req.params;

try {

const fuente = await FuenteIngresos.findByPk(id\_ingreso);

if (fuente) {

*// Enviar los datos de la fuente en la respuesta*

res.status(200).json({

code: "200",

message: "Ingreso encontrado",

fuente: {

id\_usuario: fuente.id\_usuario,

fecha\_recepcion\_fuente: fuente.fecha\_recepcion\_fuente,

nombre\_fuente: fuente.nombre\_fuente,

descripcion\_fuente: fuente.descripcion\_fuente,

valor: fuente.valor,

},

});

} else {

res

.status(404)

.json({ code: "404", message: "Fuente de ingresos no encontrada" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Crear una nueva fuente de ingresos*

const crearFuente = async (req, res) => {

const {

id\_usuario,

fecha\_recepcion\_fuente,

nombre\_fuente,

descripcion\_fuente,

valor,

} = req.body;

try {

*// Verificar si la cédula del usuario existe en la tabla de usuarios*

const usuarioExistente = await Usuario.findOne({

where: { cedula: id\_usuario },

});

if (!usuarioExistente) {

return res.status(400).json({

code: "400",

message: "La cédula ingresada no se encuentra registrada",

});

}

*// Crear la nueva fuente de ingresos asociada al usuario existente*

const nuevaFuente = await FuenteIngresos.create({

id\_usuario,

fecha\_recepcion\_fuente,

nombre\_fuente,

descripcion\_fuente,

valor,

});

res.status(201).json({

code: "201",

message: "Fuente de ingresos creada exitosamente",

fuente: nuevaFuente,

});

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Actualizar una fuente de ingresos*

const actualizarFuente = async (req, res) => {

const { id\_ingreso } = req.params;

const {

id\_usuario,

fecha\_recepcion\_fuente,

nombre\_fuente,

descripcion\_fuente,

valor,

} = req.body;

try {

const fuente = await FuenteIngresos.findByPk(id\_ingreso);

if (fuente) {

*// Verificar si el ingreso ya existe*

const ingresoExistente = await FuenteIngresos.findOne({

where: { nombre\_fuente },

});

if (ingresoExistente && ingresoExistente.id\_ingreso !== id\_ingreso) {

*// Si el ingreso ya existe y no es el mismo que se está actualizando*

return res.status(409).json({

code: "409",

message: "La fuente de ingresos ya existe",

});

}

await fuente.update({

id\_usuario,

fecha\_recepcion\_fuente,

nombre\_fuente,

descripcion\_fuente,

valor,

});

res.status(200).json({

code: "200",

message: "Fuente de ingresos actualizada exitosamente",

fuente: fuente,

});

} else {

res

.status(404)

.json({ code: "404", message: "Fuente de ingresos no encontrada" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Eliminar una fuente de ingresos*

const eliminarFuente = async (req, res) => {

const { id\_ingreso } = req.params;

try {

const fuente = await FuenteIngresos.findByPk(id\_ingreso);

if (fuente) {

await fuente.destroy();

res.status(200).json({

code: "200",

message: "Fuente de ingresos eliminada exitosamente",

});

} else {

res

.status(404)

.json({ code: "404", message: "Fuente de ingresos no encontrada" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

module.exports = {

obtenerTodasFuentes,

obtenerFuentePorId,

crearFuente,

actualizarFuente,

eliminarFuente,

};

**EXPLICACION:**

Este controlador se encarga de gestionar las operaciones relacionadas con las fuentes de ingresos en la aplicación. Comienza importando el modelo FuenteIngresos, que representa las fuentes de ingresos en la base de datos. Luego, proporciona funciones para diferentes acciones: recuperar todas las fuentes de ingresos, obtener una fuente por su ID, crear una nueva fuente, actualizar una existente y eliminar una fuente. Cada función maneja la interacción con la base de datos y devuelve respuestas apropiadas al cliente, ya sea devolviendo los datos solicitados o mensajes de éxito o error en caso de operaciones exitosas o fallidas, respectivamente.

**GastoController.js**

const Gastos = require('../models/Gastos');

const Usuario = require("../models/Usuario");

*// Obtener todos los gastos*

const getAllGastos = async (req, res) => {

try {

const gastos = await Gastos.findAll();

res.json(gastos);

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Obtener un gasto por su ID*

const getGastoById = async (req, res) => {

const { id } = req.params;

try {

const gasto = await Gastos.findByPk(id);

if (gasto) {

res.json(gasto);

} else {

res.status(404).send('Gasto no encontrado');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Crear un nuevo gasto*

const createGasto = async (req, res) => {

const { id\_usuario, fecha\_gasto, nombre\_gasto, valor\_gasto, categoria\_gasto, descripcion\_gasto } = req.body;

try {

const nuevoGasto = await Gastos.create({

id\_usuario,

fecha\_gasto,

nombre\_gasto,

valor\_gasto,

categoria\_gasto,

descripcion\_gasto

});

res.status(201).json({

message: 'Gasto creado exitosamente',

gasto: nuevoGasto,

});

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Actualizar un gasto*

const updateGasto = async (req, res) => {

const { id } = req.params;

const { id\_usuario, fecha\_gasto, nombre\_gasto, valor\_gasto, categoria\_gasto, descripcion\_gasto } = req.body;

try {

const gasto = await Gastos.findByPk(id);

if (gasto) {

await gasto.update({

id\_usuario,

fecha\_gasto,

nombre\_gasto,

valor\_gasto,

categoria\_gasto,

descripcion\_gasto

});

res.json({

gasto: gasto,

mensaje: "Gasto actualizado exitosamente",

});

} else {

res.status(404).send('Gasto no encontrado');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

*// Eliminar un gasto*

const deleteGasto = async (req, res) => {

const { id } = req.params;

try {

const gasto = await Gastos.findByPk(id);

if (gasto) {

await gasto.destroy();

res.send('Gasto eliminado exitosamente');

} else {

res.status(404).send('Gasto no encontrado');

}

} catch (error) {

console.error(error);

res.status(500).send('Error interno del servidor');

}

};

module.exports = {

getAllGastos,

getGastoById,

createGasto,

updateGasto,

deleteGasto,

};

**EXPLICACION:**

Este controlador se encarga de gestionar las operaciones relacionadas con los gastos en la aplicación. Comienza importando el modelo Gastos, que representa los gastos en la base de datos. Luego, proporciona funciones para diferentes acciones: recuperar todos los gastos, obtener un gasto por su ID, crear un nuevo gasto, actualizar un gasto existente y eliminar un gasto. Cada función maneja la interacción con la base de datos y devuelve respuestas apropiadas al cliente, ya sea devolviendo los datos solicitados o mensajes de éxito o error en caso de operaciones exitosas o fallidas, respectivamente.

**UsuarioController.js**

const Usuario = require("../models/Usuario");

*// Iniciar sesión*

const iniciarSesion = async (req, res) => {

const { cedula, contrasena } = req.body;

try {

if (!cedula) {

return res.status(400).json({

code: 400,

message: "La cédula es requerida para iniciar sesión",

});

}

*// Buscar al usuario en la base de datos por su cédula*

const usuario = await Usuario.findOne({ where: { cedula } });

*// Verificar si el usuario existe y la contraseña coincide*

if (usuario && usuario.contrasena === contrasena) {

res.status(200).json({

code: "200",

message: "Inicio de sesión exitoso",

primer\_nombre: usuario.primer\_nombre,

});

} else {

res

.status(401)

.json({ code: "401", message: "Credenciales incorrectas" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Registrar un nuevo usuario*

const registrarUsuario = async (req, res) => {

const {

cedula,

primer\_nombre,

segundo\_nombre,

primer\_apellido,

segundo\_apellido,

genero,

correo,

contrasena,

pregunta\_contra,

respuesta\_contra,

numero\_tel,

foto,

rol,

pais,

ciudad,

} = req.body;

try {

*// Crear el nuevo usuario en la base de datos*

const nuevoUsuario = await Usuario.create({

cedula,

primer\_nombre,

segundo\_nombre,

primer\_apellido,

segundo\_apellido,

genero,

correo,

contrasena,

pregunta\_contra,

respuesta\_contra,

numero\_tel,

foto,

rol,

pais,

ciudad,

});

res

.status(201)

.json({ code: "201", message: "Usuario registrado exitosamente" });

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Actualizar datos de un usuario*

const actualizarUsuario = async (req, res) => {

const { cedula } = req.params;

const {

primer\_nombre,

segundo\_nombre,

primer\_apellido,

segundo\_apellido,

genero,

correo,

pregunta\_contra,

respuesta\_contra,

numero\_tel,

foto,

rol,

pais,

ciudad,

} = req.body;

try {

*// Buscar el usuario por su cédula*

const usuario = await Usuario.findByPk(cedula);

*// Verificar si el usuario existe*

if (usuario) {

*// Actualizar los datos del usuario*

await usuario.update({

primer\_nombre,

segundo\_nombre,

primer\_apellido,

segundo\_apellido,

genero,

correo,

pregunta\_contra,

respuesta\_contra,

numero\_tel,

foto,

rol,

pais,

ciudad,

});

res

.status(200)

.json({ code: "200", message: "Usuario actualizado exitosamente" });

} else {

res.status(404).json({ code: "404", message: "Usuario no encontrado" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

const eliminarUsuarioPorCedula = async (req, res) => {

const { cedula } = req.params;

try {

*// Buscar el usuario por su cédula*

const usuario = await Usuario.findByPk(cedula);

*// Verificar si el usuario existe*

if (usuario) {

*// Eliminar el usuario de la base de datos*

await usuario.destroy();

*// contar usuarios existentes*

const remainingUsers = await Usuario.count();

res.status(200).json({

code: "200",

message: "Usuario eliminado exitosamente",

total: remainingUsers,

});

} else {

*// Si el usuario no se encuentra*

res.status(404).json({ code: "404", message: "Usuario no encontrado" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

*// Buscar un usuario por su cédula*

*// Buscar un usuario por su cédula*

const buscarUsuarioPorCedula = async (req, res) => {

const { cedula } = req.params; *// Obtener la cédula de los parámetros de la ruta*

try {

*// Buscar al usuario en la base de datos por su cédula*

const usuario = await Usuario.findOne({ where: { cedula } });

*// Verificar si el usuario existe*

if (usuario) {

*// Enviar los datos del usuario en la respuesta*

res.status(200).json({

code: "200",

usuario: {

cedula: usuario.cedula,

primer\_nombre: usuario.primer\_nombre,

primer\_apellido: usuario.primer\_apellido,

correo: usuario.correo,

pregunta\_contra: usuario.pregunta\_contra,

respuesta\_contra: usuario.respuesta\_contra,

numero\_tel: usuario.numero\_tel,

},

});

} else {

res.status(404).json({ code: "404", message: "Usuario no encontrado" });

}

} catch (error) {

console.error(error);

res

.status(500)

.json({ code: "500", message: "Error interno del servidor" });

}

};

module.exports = {

iniciarSesion,

registrarUsuario,

actualizarUsuario,

eliminarUsuarioPorCedula,

buscarUsuarioPorCedula,

};

**EXPLICACION:**

Este controlador gestiona usuarios de nuestra aplicacion. Ofrece una serie de funciones para interactuar con la base de datos de usuarios. La función iniciarSesion permite a los usuarios iniciar sesión proporcionando su cédula y contraseña, verificando si las credenciales son correctas y devolviendo un mensaje de éxito o error. La función registrarUsuario permite crear nuevos usuarios con una variedad de detalles, como nombre, género y correo electrónico, almacenando esta información en la base de datos y devolviendo un mensaje de éxito o error. actualizarUsuario permite modificar los datos de un usuario existente, buscándolo por su cédula, actualizando los campos especificados y devolviendo un mensaje de éxito o error. eliminarUsuarioPorCedula elimina un usuario basado en su cédula, también devolviendo un mensaje de éxito o error, y además muestra la cantidad total de usuarios restantes en la base de datos. Por último, buscarUsuarioPorCedula busca y devuelve los datos de un usuario específico según su cédula, proporcionando detalles como nombre, correo y número de teléfono en caso de éxito, o un mensaje de error si el usuario no se encuentra. Estas funciones forman la interfaz entre la aplicación y la base de datos, permitiendo la gestión completa de los usuarios.

**Códigos de las clases modelo**

const { DataTypes } = require('sequelize');

const sequelize = require('../libs/sequelize');

const Usuario = require('./Usuario');

const CategoriaGastos = sequelize.define('CategoriaGastos', {

id\_categoria: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true,

},

id\_usuario: {

type: DataTypes.STRING,

references: {

model: Usuario,

key: 'cedula',

},

},

fecha\_categoria: {

type: DataTypes.DATE,

},

nombre\_categoria: {

type: DataTypes.STRING,

},

descripcion\_categoria: {

type: DataTypes.STRING,

},

icono: {

type: DataTypes.STRING,

},

}, {

tableName: 'categoria\_gastos',

timestamps: false, *// Deshabilitar timestamps*

});

module.exports = CategoriaGastos;

**EXPLICACION:**  
  
Este archivo define el modelo CategoriaGastos que representa las categorías de gastos en la base de datos de la aplicación. Utiliza Sequelize, una biblioteca de Node.js para la gestión de bases de datos relacionales.

En este modelo, se especifican los diferentes campos que componen una categoría de gastos, como el identificador único (id\_categoria), el identificador del usuario al que pertenece (id\_usuario), la fecha de la categoría (fecha\_categoria), el nombre de la categoría (nombre\_categoria), la descripción de la categoría (descripcion\_categoria) y el icono asociado (icono).

Cada campo tiene un tipo de dato definido, como INTEGER para identificadores numéricos, STRING para cadenas de texto y DATE para fechas. Además, se establecen algunas restricciones, como la clave primaria (primaryKey) y la referencia al modelo Usuario para el campo id\_usuario.

El modelo se exporta al final del archivo para que pueda ser utilizado en otras partes de la aplicación, como los controladores y las migraciones de la base de datos. Esto facilita la creación, lectura, actualización y eliminación de categorías de gastos en la base de datos a través de Sequelize.

const { DataTypes } = require('sequelize');

const sequelize = require('../libs/sequelize');

const Usuario = require('./Usuario');

const FuenteIngresos = sequelize.define('FuenteIngresos', {

id\_ingreso: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true,

},

id\_usuario: {

type: DataTypes.STRING,

references: {

model: Usuario,

key: 'cedula',

},

},

fecha\_recepcion\_fuente: {

type: DataTypes.DATE,

},

nombre\_fuente: {

type: DataTypes.STRING,

},

descripcion\_fuente: {

type: DataTypes.STRING,

},

icono: {

type: DataTypes.STRING,

},

}, {

tableName: 'fuente\_ingresos',

timestamps: false, *// Deshabilitar timestamps*

});

module.exports = FuenteIngresos;

**EXPLICACION:**

El modelo FuenteIngresos representa las fuentes de ingresos en la base de datos de la aplicación. Utilizando Sequelize, una biblioteca de Node.js para la gestión de bases de datos relacionales, se especifican los campos que componen una fuente de ingresos.

Cada fuente de ingresos tiene un identificador único (id\_ingreso), el cual es autogenerado gracias a la configuración de autoIncrement y es la clave primaria de la tabla. Además, se registra el identificador del usuario al que pertenece la fuente de ingresos (id\_usuario), la fecha de recepción de la fuente (fecha\_recepcion\_fuente), el nombre de la fuente de ingresos (nombre\_fuente), su descripción (descripcion\_fuente), y el icono asociado (icono).

Para establecer la relación con el modelo de Usuario, se hace uso del campo id\_usuario, el cual hace referencia al campo cedula del modelo Usuario. De esta manera, se garantiza la integridad referencial en la base de datos.

Por último, se define el nombre de la tabla en la base de datos como fuente\_ingresos y se deshabilitan los timestamps, que son campos automáticos de registro de fecha de creación y actualización en la tabla.

Al exportar el modelo FuenteIngresos, este puede ser utilizado en otras partes de la aplicación para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las fuentes de ingresos en la base de datos utilizando Sequelize.

const { DataTypes } = require('sequelize');

const sequelize = require('../libs/sequelize');

const Usuario = require('./Usuario');

const Gastos = sequelize.define('Gastos', {

id\_gastos: {

type: DataTypes.INTEGER,

primaryKey: true,

autoIncrement: true,

},

id\_usuario: {

type: DataTypes.STRING,

references: {

model: Usuario,

key: 'cedula',

},

},

fecha\_gasto: {

type: DataTypes.DATE,

},

nombre\_gasto: {

type: DataTypes.STRING,

},

valor\_gasto: {

type: DataTypes.DOUBLE,

},

categoria\_gasto: {

type: DataTypes.STRING,

},

descripcion\_gasto: {

type: DataTypes.STRING,

},

}, {

tableName: 'gastos',

timestamps: false, *// Deshabilitar timestamps*

});

module.exports = Gastos;

**EXPLICACION:**

El modelo Gastos representa los gastos en la base de datos de la aplicación. Utilizando Sequelize, una biblioteca de Node.js para la gestión de bases de datos relacionales, se especifican los campos que componen un gasto.

Cada gasto tiene un identificador único (id\_gastos), que se autogenera gracias a la configuración de autoIncrement y es la clave primaria de la tabla. Además, se registra el identificador del usuario al que pertenece el gasto (id\_usuario), la fecha del gasto (fecha\_gasto), el nombre del gasto (nombre\_gasto), su valor (valor\_gasto), la categoría del gasto (categoria\_gasto), y una descripción del mismo (descripcion\_gasto).

Para establecer la relación con el modelo de Usuario, se utiliza el campo id\_usuario, que hace referencia al campo cedula del modelo Usuario. De esta manera, se garantiza la integridad referencial en la base de datos.

Por último, se define el nombre de la tabla en la base de datos como gastos y se deshabilitan los timestamps, que son campos automáticos de registro de fecha de creación y actualización en la tabla.

Al exportar el modelo Gastos, este puede ser utilizado en otras partes de la aplicación para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los gastos en la base de datos utilizando Sequelize.

const { DataTypes } = require('sequelize');

const sequelize = require('../libs/sequelize');

const Usuario = sequelize.define('Usuario', {

cedula: {

type: DataTypes.STRING,

primaryKey: true,

},

tipo\_identificacion: {

type: DataTypes.STRING,

},

contrasena: {

type: DataTypes.STRING,

},

pregunta\_contra: {

type: DataTypes.STRING,

},

respuesta\_contra: {

type: DataTypes.STRING,

},

primer\_nombre: {

type: DataTypes.STRING,

},

segundo\_nombre: {

type: DataTypes.STRING,

},

primer\_apellido: {

type: DataTypes.STRING,

},

segundo\_apellido: {

type: DataTypes.STRING,

},

genero: {

type: DataTypes.STRING,

},

correo: {

type: DataTypes.STRING,

},

numero\_tel: {

type: DataTypes.STRING,

},

foto: {

type: DataTypes.STRING,

},

rol: {

type: DataTypes.STRING,

},

pais: {

type: DataTypes.STRING,

},

ciudad: {

type: DataTypes.STRING,

},

}, {

tableName: 'usuarios',

timestamps: false, *// Deshabilitar timestamps*

});

module.exports = Usuario;

**EXPLICACION:**

Este archivo define el modelo Usuario, el cual representa a los usuarios de la aplicación en la base de datos.

Cada usuario tiene un identificador único (cedula), que se establece como clave primaria de la tabla. Además, se registran diversos campos como tipo\_identificacion para el tipo de documento de identificación, contrasena para la contraseña, pregunta\_contra y respuesta\_contra para la recuperación de contraseña, primer\_nombre, segundo\_nombre, primer\_apellido, segundo\_apellido para los nombres y apellidos, genero, correo, numero\_tel para el número de teléfono, foto para la URL de la foto de perfil, rol para el rol del usuario, y pais y ciudad para la ubicación del usuario.

El nombre de la tabla en la base de datos se establece como usuarios y se deshabilitan los timestamps, que son campos automáticos de registro de fecha de creación y actualización en la tabla.

Al exportar el modelo Usuario, este puede ser utilizado en otras partes de la aplicación para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los usuarios en la base de datos utilizando Sequelize.

**Código del archivo principal del servidor (index.js)**

const express = require('express');

const bodyParser = require('body-parser');

const sequelize = require('./libs/sequelize');

const usuarioRoutes = require('./routes/usuariosRoutes');

const gastosRoutes = require('./routes/gastosRoutes');

const categoriaRoutes = require('./routes/categoriaRoutes');

const fuenteRoutes = require('./routes/fuenteRoutes');

const app = express();

const port = process.env.PORT || 5000;

*// Middleware para analizar cuerpos de solicitud en formato JSON*

app.use(express.json());

*// Middleware para analizar cuerpos de solicitud en formato urlencode*

app.use(bodyParser.urlencoded({ extended: true }));

*// Establecer el encabezado Content-Type para todas las respuestas como JSON*

app.use((req, res, next) => {

res.setHeader('Content-Type', 'application/json');

next();

});

*// Bloque try-catch para manejar errores de sincronización*

(async () => {

try {

await sequelize.authenticate();

console.log('Conexión a la base de datos establecida correctamente.');

*// Sincronizar modelos después de la autenticación*

await sequelize.sync();

console.log('Modelos sincronizados correctamente');

*// Inicia el servidor después de la sincronización*

app.listen(port, () => {

console.log(`Servidor en http:*//localhost:${port}`);*

});

} catch (error) {

console.error('Error al sincronizar modelos o autenticar la base de datos:', error);

}

})();

*// Define las rutas para cada recurso*

app.use('/api/usuarios', usuarioRoutes);

app.use('/api/gastos', gastosRoutes);

app.use('/api/categorias', categoriaRoutes);

app.use('/api/fuentes', fuenteRoutes);

*// Ruta de prueba*

app.get('/', (req, res) => {

res.send('Todo OK');

});

**EXPLICACION:**

Este código configura el servidor Express para una aplicación web. Se establecen las dependencias necesarias, como Express, Body-parser para analizar cuerpos de solicitud y Sequelize para la interacción con la base de datos. Además, se importan los modelos de la base de datos y las rutas definidas para cada recurso.

El servidor se inicia en el puerto especificado en la variable port, que se establece en el puerto 5000 por defecto o se toma del entorno si está definido. Se utiliza un middleware para analizar cuerpos de solicitud en formato JSON y otro para analizar cuerpos en formato URL-Encoded.

También se define un middleware para establecer el encabezado Content-Type de todas las respuestas como JSON, lo cual es útil para estandarizar la API.

Se utiliza un bloque try-catch para manejar posibles errores de autenticación de la base de datos y sincronización de modelos. Si la autenticación y la sincronización son exitosas, se inicia el servidor Express y se registran las rutas para cada recurso. Además, se define una ruta de prueba que devuelve un mensaje "Todo OK" al acceder a la raíz del servidor.

Este código proporciona una estructura básica para una aplicación web que utiliza Express y Sequelize para interactuar con una base de datos relacional, y define las rutas necesarias para cada recurso en la API.

**Rutas de las solicitudes y respuestas HTTP:**

**usuariosRoute:**  
  
const express = require("express");

const router = express.Router();

const usuarioController = require("../controllers/usuarioController");

router.post("/login", usuarioController.iniciarSesion);

router.post("/registro", usuarioController.registrarUsuario);

router.put("/actualizar/:cedula", usuarioController.actualizarUsuario);

router.delete("/:cedula", usuarioController.eliminarUsuarioPorCedula);

router.get("/:cedula", usuarioController.buscarUsuarioPorCedula);

module.exports = router;

**EXPLICACION:**

Este código define las rutas para interactuar con la API de usuarios. Cuando un cliente envía una solicitud POST a "/login", se llama al controlador iniciarSesion del usuario para iniciar sesión. Si el cliente envía una solicitud POST a "/registro", se llama al controlador registrarUsuario para crear una nueva cuenta de usuario. Para actualizar la información de un usuario, se utiliza una solicitud PUT a "/actualizar/:cedula", donde ":cedula" es el identificador único del usuario. Cuando se envía una solicitud DELETE a "/:cedula", se llama al controlador eliminarUsuarioPorCedula para eliminar un usuario específico según su cédula. Finalmente, si el cliente realiza una solicitud GET a "/:cedula", se llama al controlador buscarUsuarioPorCedula para buscar y devolver la información de un usuario según su cédula. Todas estas rutas están definidas en el enrutador de Express y se exportan para que puedan ser utilizadas por la aplicación principal.

fuenteRoute:

const express = require("express");

const router = express.Router();

const fuenteController = require("../controllers/fuenteController");

router.get("/buscarTodo/", fuenteController.obtenerTodasFuentes);

router.get("/:id\_ingreso", fuenteController.obtenerFuentePorId);

router.post("/", fuenteController.crearFuente);

router.put("/actualizar/:id\_ingreso", fuenteController.actualizarFuente);

router.delete("/eliminar/:id\_ingreso", fuenteController.eliminarFuente);

module.exports = router;

**EXPLICACION:**

Este código define las rutas para interactuar con la API de fuentes de ingresos. Cuando un cliente envía una solicitud GET a "/buscarTodo/", se llama al controlador obtenerTodasFuentes para obtener todas las fuentes de ingresos disponibles. Si el cliente envía una solicitud GET a "/:id\_ingreso", se llama al controlador obtenerFuentePorId para obtener información sobre una fuente de ingresos específica según su ID. Para crear una nueva fuente de ingresos, se utiliza una solicitud POST a "/", donde se llama al controlador crearFuente. Para actualizar la información de una fuente de ingresos, se utiliza una solicitud PUT a "/actualizar/:id\_ingreso", donde ":id\_ingreso" es el identificador único de la fuente de ingresos. Cuando se envía una solicitud DELETE a "/eliminar/:id\_ingreso", se llama al controlador eliminarFuente para eliminar una fuente de ingresos específica según su ID. Todas estas rutas están definidas en el enrutador de Express y se exportan para que puedan ser utilizadas por la aplicación principal.

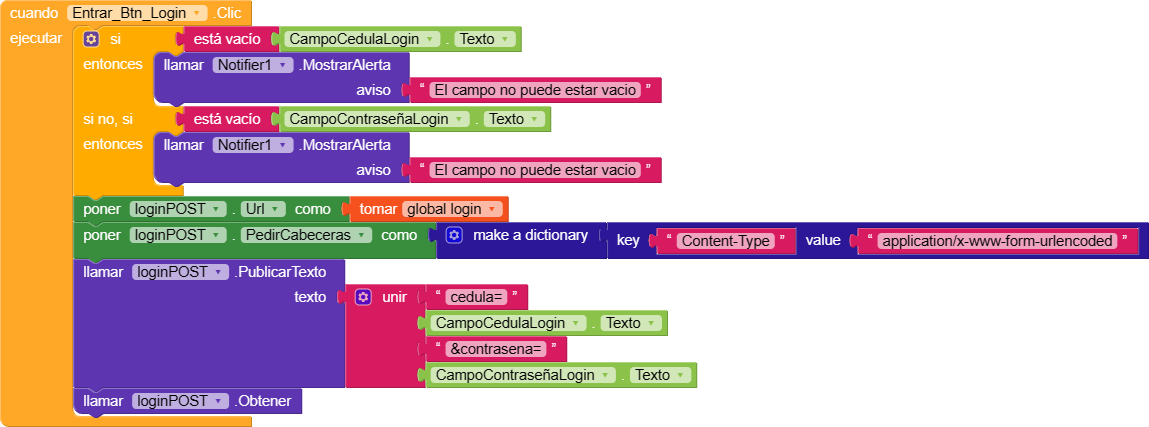
**Lógica en kodular**

**Lógica para ingresar usuarios:**

**Endpoint del api:**



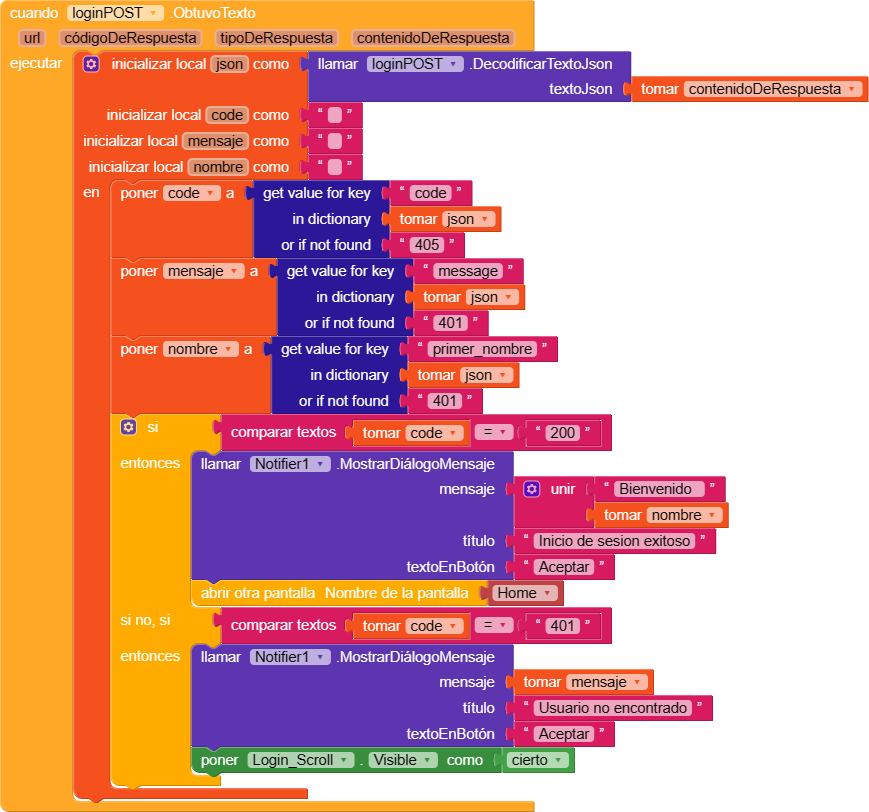
**Solicitud**



**EXPLICACION:**

Este bloque se activa cuando se hace clic en el botón “Entrar\_Btn\_Login”. Primero, verifica si los campos “CampoCedulaLogin” y “CampoContrasenaLogin” están vacíos. Si alguno de ellos está vacío, se muestra una alerta indicando que el campo no puede estar vacío. Si ambos campos están llenos, se realiza una solicitud POST a la URL almacenada en la variable global “login”. La solicitud POST incluye encabezados y un cuerpo de texto que contiene los datos ingresados en los campos “CampoCedulaLogin” y “CampoContrasenaLogin”, formateados adecuadamente para la solicitud.

**Respuesta:**

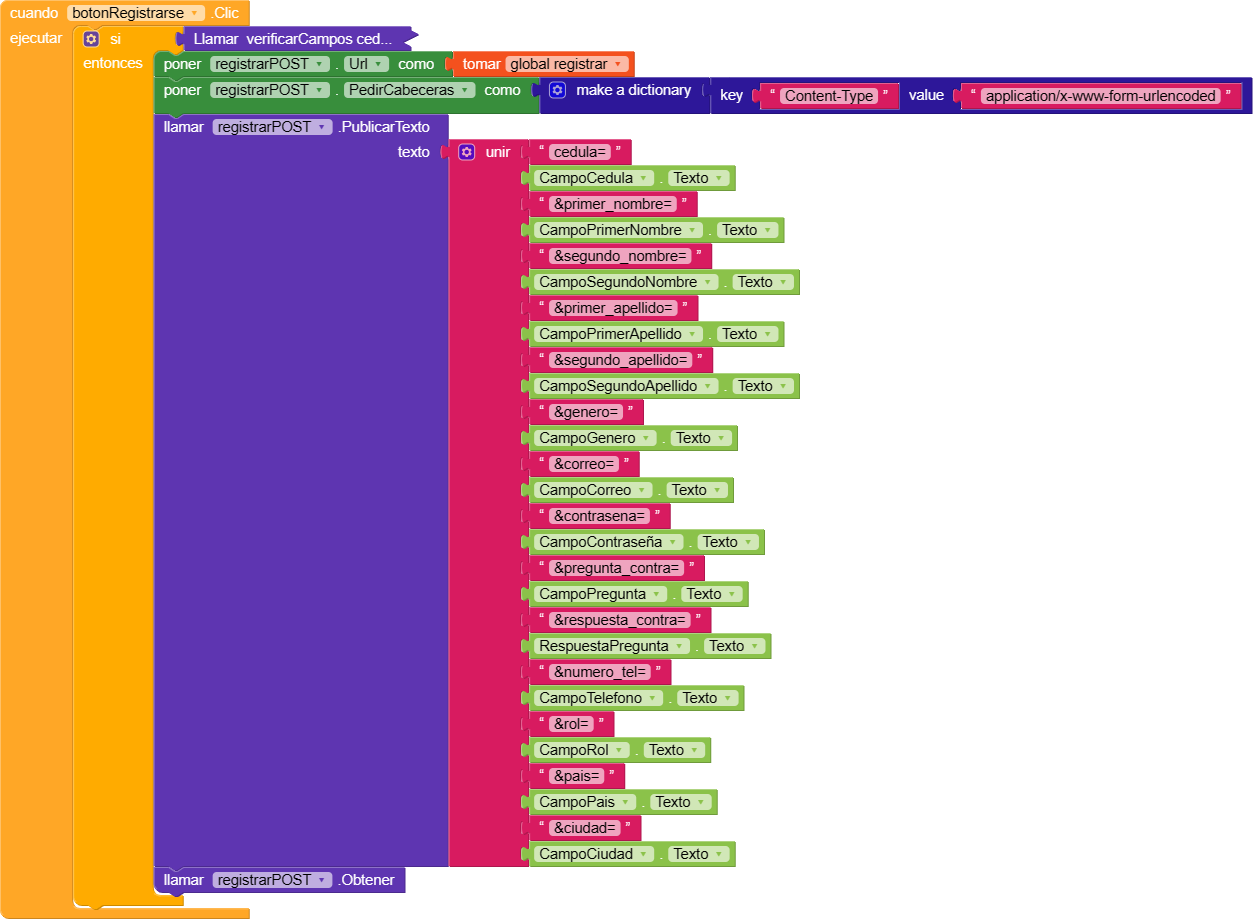


**EXPLICACION:**

Este bloque obtiene el texto de la solicitud POST “loginPOST”. Primero, decodifica el texto JSON de la respuesta y lo almacena en la variable local “json”. Luego, extrae los valores para las claves ‘code’, ‘message’ y ‘primer\_nombre’ del diccionario “json” y los almacena en las variables locales ‘code’, ‘mensaje’ y ‘nombre’, respectivamente. Si el código es ‘200’, muestra un mensaje de bienvenida con el primer nombre del usuario y abre la pantalla “Home”. Si el código es ‘401’, muestra un mensaje de error indicando que el usuario no fue encontrado y hace visible el componente “Login\_Scroll”.

**Lógica registrar usuarios:**

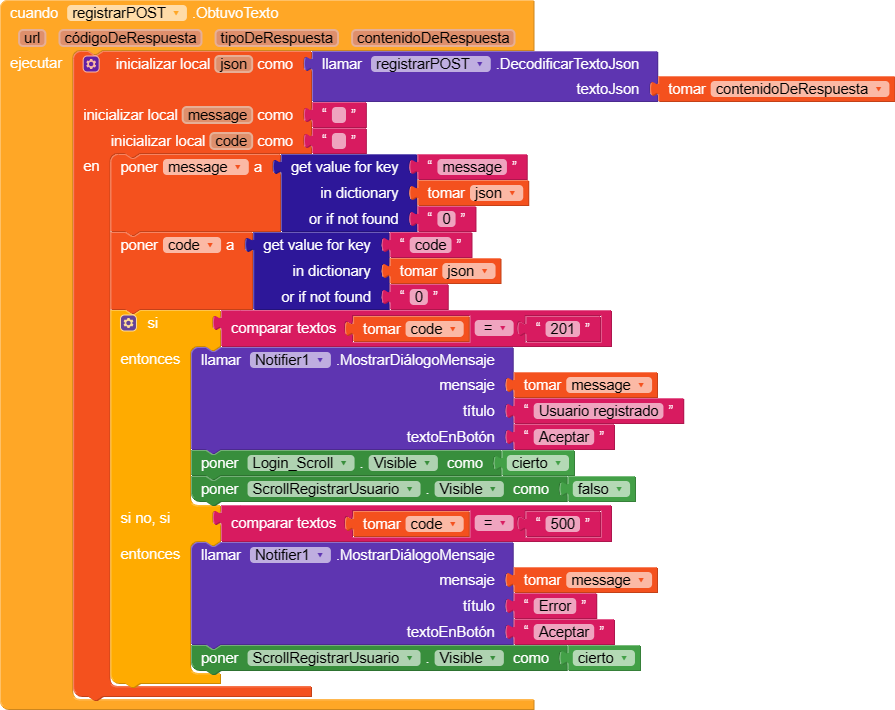
**Solicitud:**



**EXPLICACION:**

Este bloque se activa cuando se hace clic en el botón “botonRegistrarse”. Si la función “verificarCampos” devuelve verdadero, entonces configura la URL y los encabezados de la solicitud POST “registrarPOST” y realiza la solicitud. Los datos enviados en la solicitud POST incluyen los valores ingresados en varios campos del formulario de registro, como ‘cedula’, ‘primer-nombre’, ‘segundo-nombre’, ‘primer-apellido’, ‘segundo-apellido’, ‘genero’, ‘fecha-nacimiento’, ‘correo-e’, ‘telefono-fijo’, ‘celular’, ‘direccion-residencia’, ‘ciudad-residencia’, ‘usuario’, ‘numero-contra’, ‘contra-rep’, ‘rol’, ‘ciudad’, y ‘campus’. Estos datos se unen en una cadena de texto con el formato adecuado para la transmisión. Luego, obtiene la respuesta de la solicitud POST.

**Respuesta**



**EXPLICACION:**

Este bloque se activa cuando se obtiene la respuesta de la solicitud POST “registrarPOST”. Primero, decodifica el texto JSON de la respuesta y lo almacena en la variable local “json”. Luego, extrae los valores para las claves ‘message’ y ‘code’ del diccionario “json” y los almacena en las variables locales ‘message’ y ‘code’, respectivamente. Si el código es ‘201’, muestra un mensaje de bienvenida y hace visible el componente “Login\_Scroll” y oculta el componente “ScrollRegistrarUsuario”. Si el código es ‘500’, muestra un mensaje de error y hace visible el componente “ScrollRegistrarUsuario”.

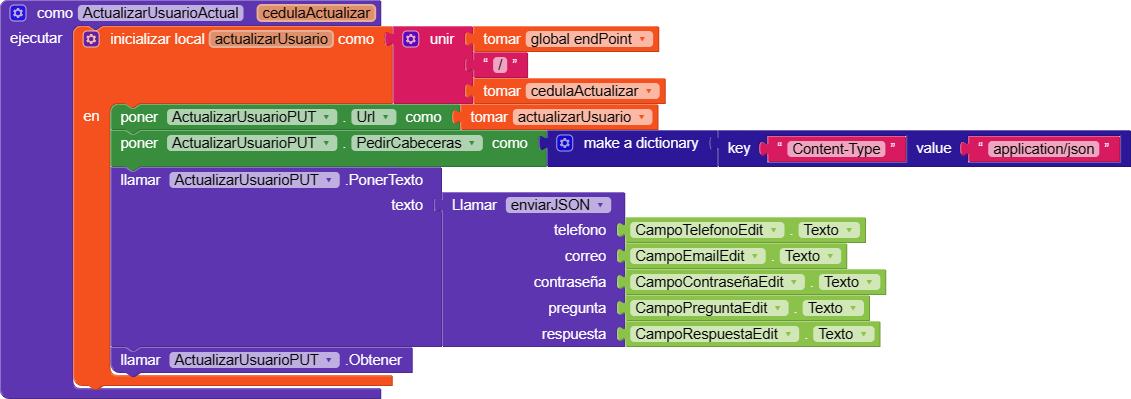
**Logica actualizar usuario**

**Endpoint**



**Solicitud**

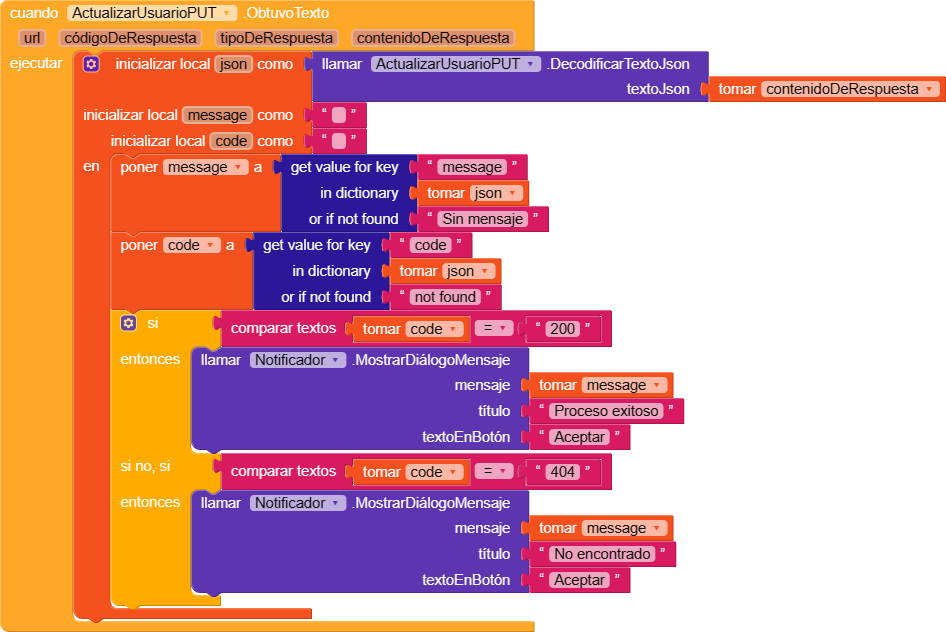




**EXPLICACION:**

Este bloque se activa cuando se hace clic en el botón “botonRegistrarse”. Primero, verifica si el campo “CampoBuscarCedula” está vacío. Si está vacío, muestra una alerta indicando que no se ha encontrado el usuario actual. Si el campo no está vacío, llama a la función “verificarCampos” con varios parámetros, incluyendo los textos de los campos “CampoTelefonoEdit”, “CampoEmailEdit”, “CampoContraseñaEdit”, “CampoPreguntaEdit”, y “CampoRespuestaEdit”. Si la función “verificarCampos” devuelve verdadero, entonces llama a la función “ActualizarUsuarioActual” con el texto del campo “CampoBuscarCedula” como parámetro.

**Respuesta**

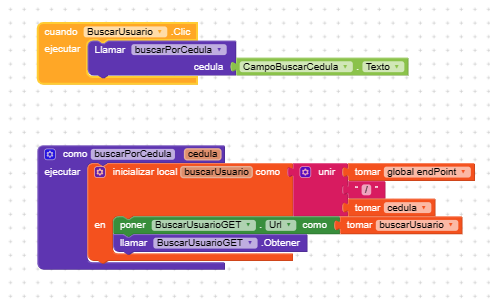


**EXPLICACION:**

Este bloque se habilita cuando se obtiene la respuesta de la solicitud PUT “ActualizarUsuarioPUT”. Primero, decodifica el texto JSON de la respuesta y lo almacena en la variable local “json”. Luego, extrae los valores para las claves ‘message’ y ‘code’ del diccionario “json” y los almacena en las variables locales ‘message’ y ‘code’, respectivamente. Si el código es ‘200’, muestra un mensaje de éxito. Si el código es ‘404’, muestra un mensaje indicando que no se encontró el recurso solicitado.

**Lógica buscar usuarios**

**Solicitud**



**EXPLICACION:**

Este bloque se activa cuando se hace clic en el botón “BuscarUsuario”. Llama a la función “buscarPorCedula” con el texto del campo “CampoBuscarCedula” como parámetro. La función “buscarPorCedula” inicializa la variable local “buscarUsuario” con la URL para buscar un usuario, que se compone de la variable global “endPoint” y el parámetro “cedulaActualizar”. Luego, configura la URL de la solicitud GET “BuscarUsuarioGET” con la variable “buscarUsuario” y realiza la solicitud GET.

**Respuesta**

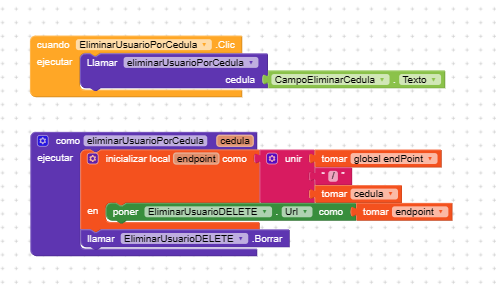


**EXPLICACION:**

Este bloque se activa cuando se obtiene la respuesta de la solicitud PUT “BuscarUsuarioGET”. Primero, decodifica el texto JSON de la respuesta y lo almacena en la variable local “json”. Luego, extrae los valores para las claves ‘codigo’ y ‘message’ del diccionario “json” y los almacena en las variables locales ‘codigo’ y ‘message’, respectivamente. Si el código es ‘200’, llena varios campos del formulario con los datos del usuario obtenidos del diccionario “json” y muestra un mensaje indicando que el usuario fue encontrado.

**Lógica para eliminar usuarios por cedula**

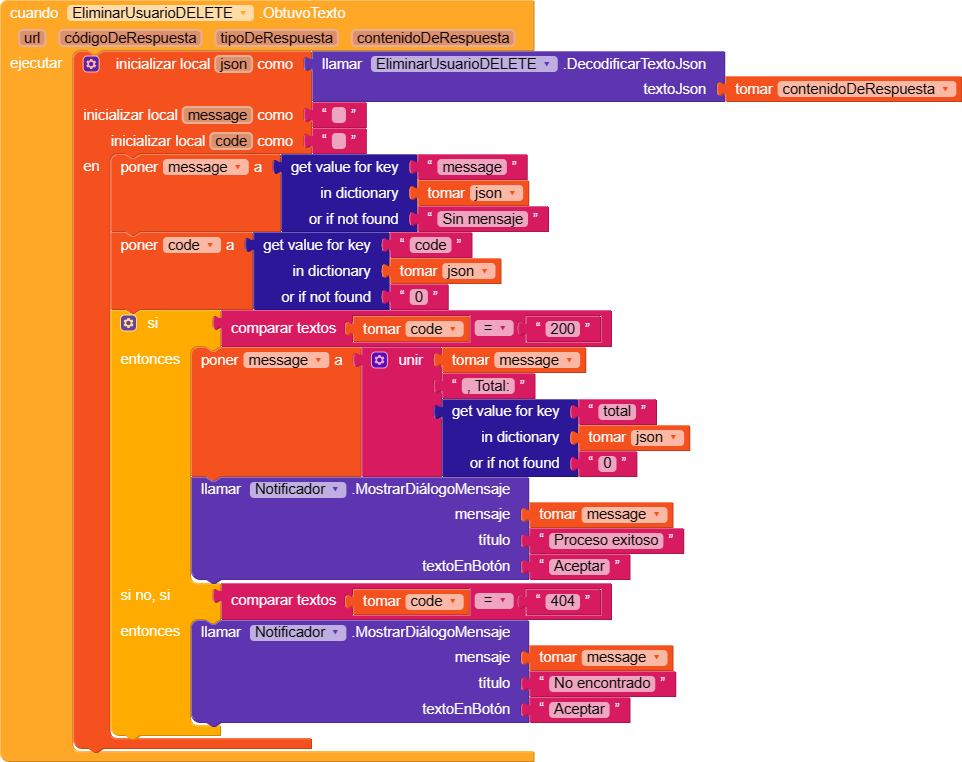
**Solicitud**



**EXPLICACION:**

Este se activa cuando se hace clic en el botón “EliminarUsuarioPorCedula”. Llama a la función “oEliminarUsuarioPorCedula” con el texto del campo “CampoEliminarCedula” como parámetro. La función “oEliminarUsuarioPorCedula” inicializa la variable local “endpoint” con la URL para eliminar un usuario, que se compone de la variable global “endPoint” y el parámetro “cedulaActualizar”. Luego, configura la URL de la solicitud DELETE “EliminarUsuarioDELETE” con la variable “endpoint” y realiza la solicitud DELETE.

**Respuesta**



**EXPLICACION:**

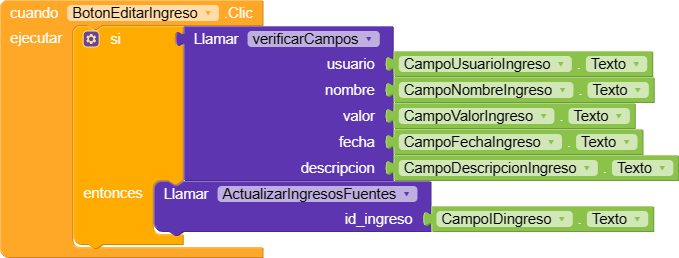
Este bloque se activa cuando se obtiene la respuesta de la solicitud DELETE “EliminarUsuarioDELETE”. Primero, decodifica el texto JSON de la respuesta y lo almacena en la variable local “json”. Luego, extrae los valores para las claves ‘message’ y ‘code’ del diccionario “json” y los almacena en las variables locales ‘message’ y ‘code’, respectivamente. Si el código es ‘200’, muestra un mensaje de éxito y añade el total de usuarios eliminados al mensaje. Si el código es ‘404’, muestra un mensaje indicando que no se encontró el recurso solicitado.

**Lógica para la gestión de la fuente de ingresos del usuario:**

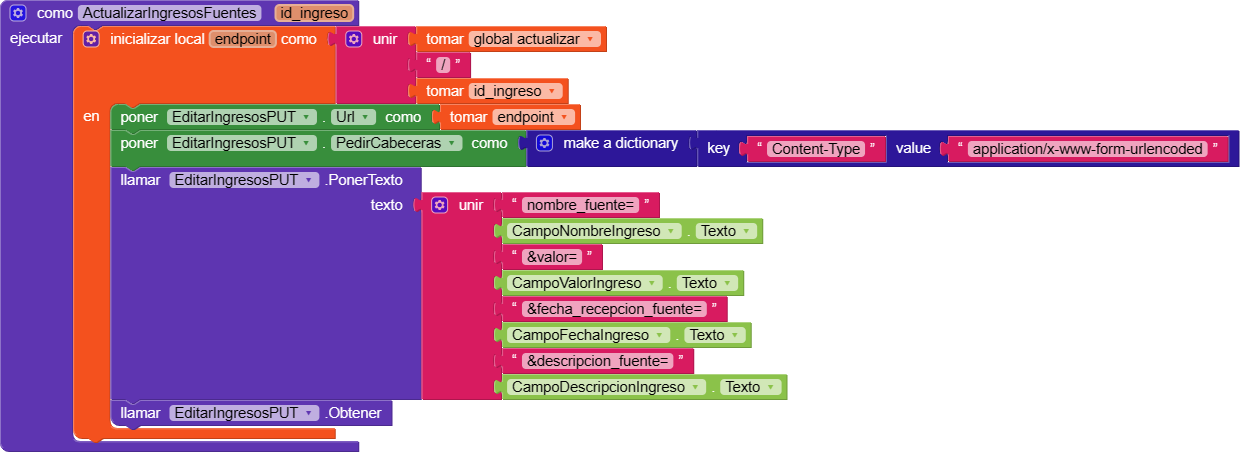
**Actualizar fuente de ingresos**



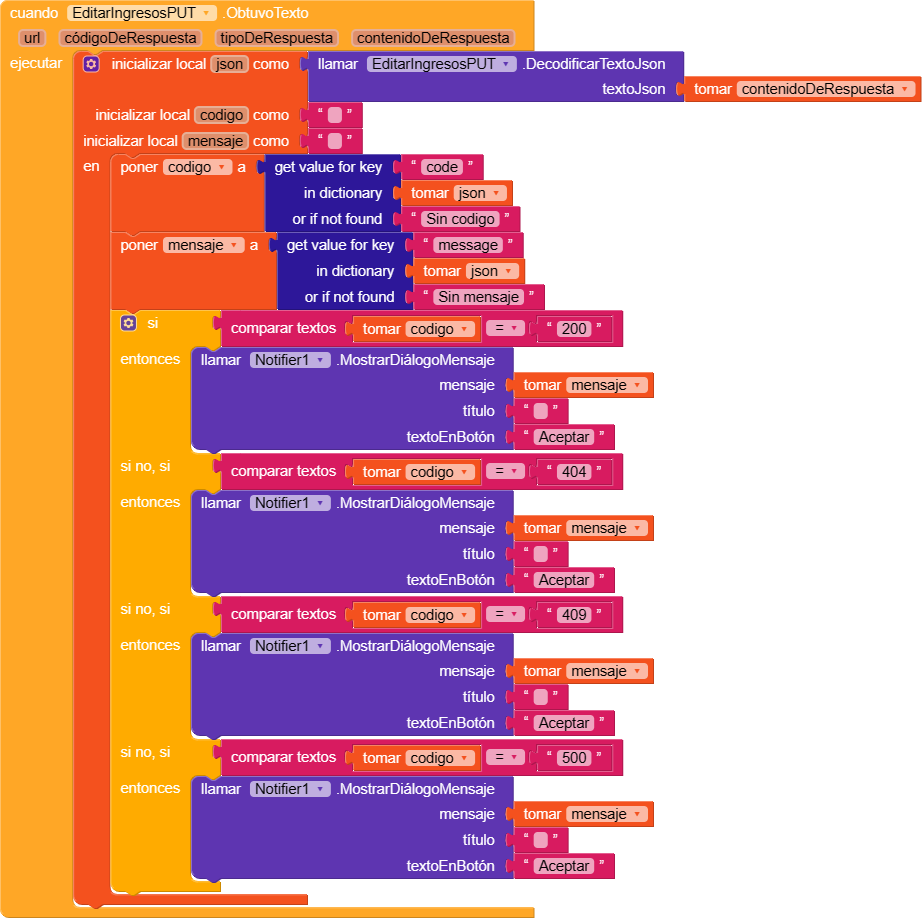
Este es el endpoint para la solicitudes y respuestas del api de actualizar fuente de ingresos.



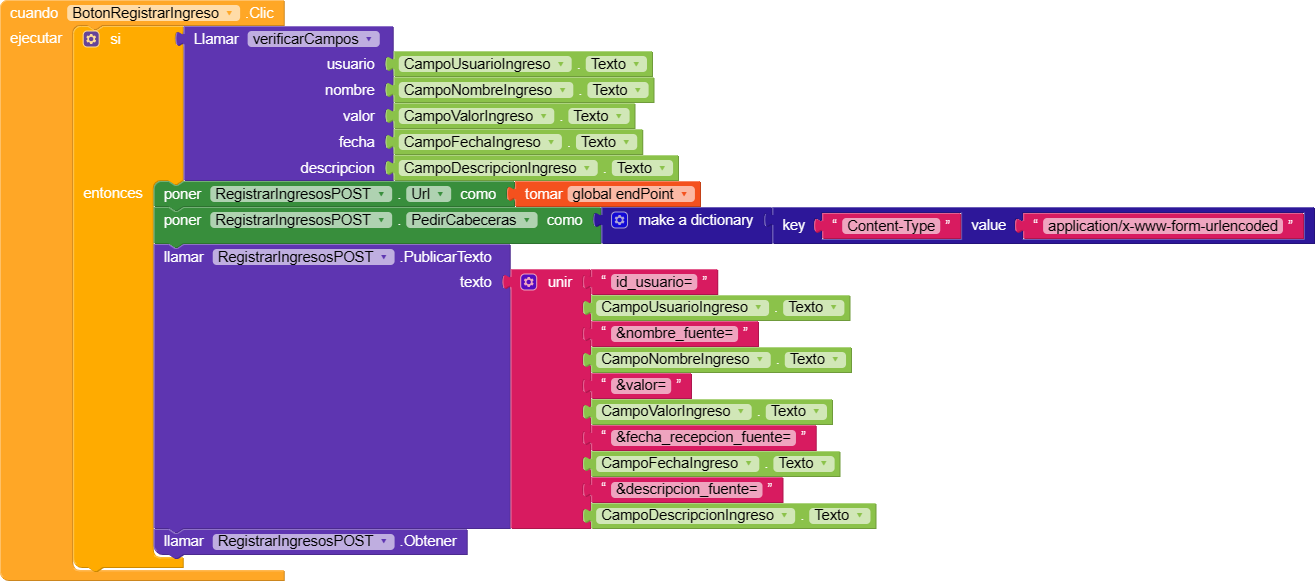
Esta función del botón se activa cuando se hace clic y permite la búsqueda de la fuente de ingreso.



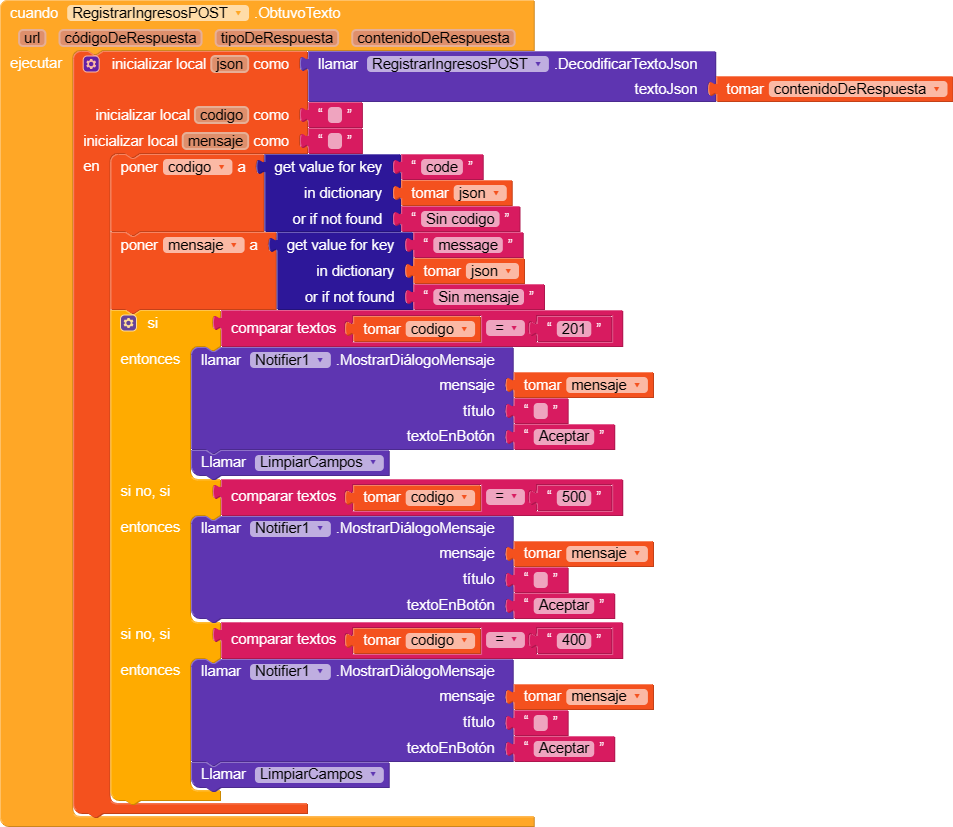
La función permite realizar el proceso de actualizar la fuente de ingreso.



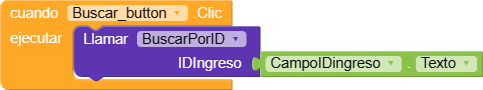
Este bloque representa la respuesta de la api.



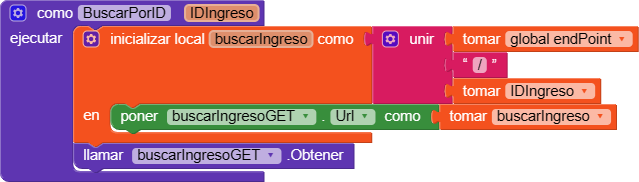
Esta función se ejecuta cuando se hace clic en el botón para realizar la solicitud de registrar una nueva fuente de ingresos.



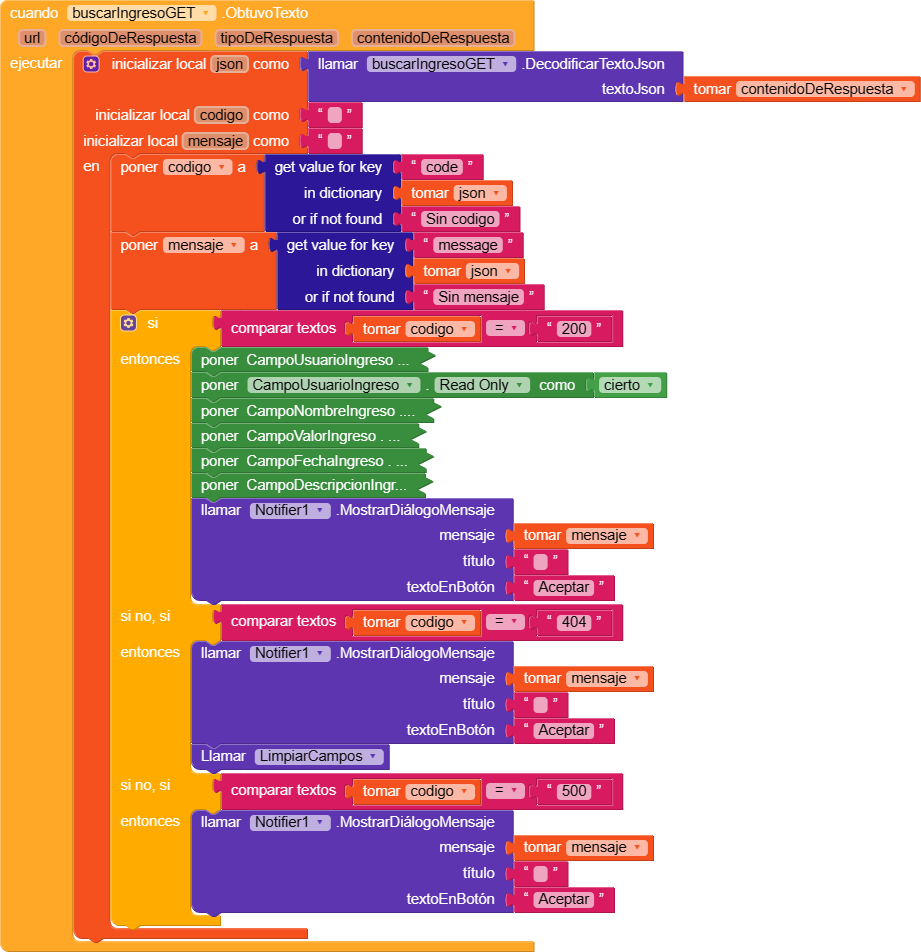
Esta es la lógica para capturar la respuesta a la solicitud.



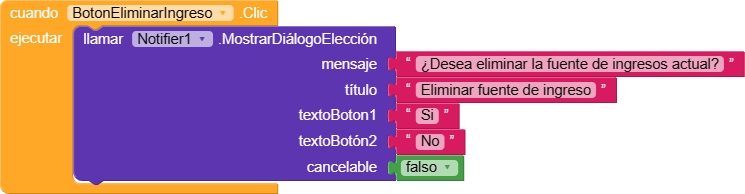
Acción del botón buscar



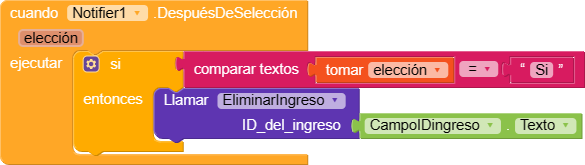
Función para hacer la solicitud de la búsqueda



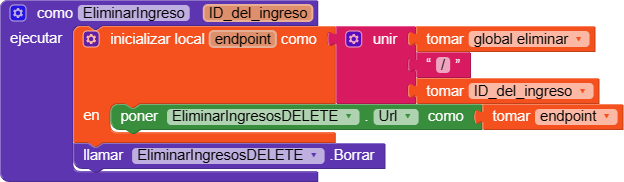
Respuesta de la búsqueda de la fuente de ingreso.



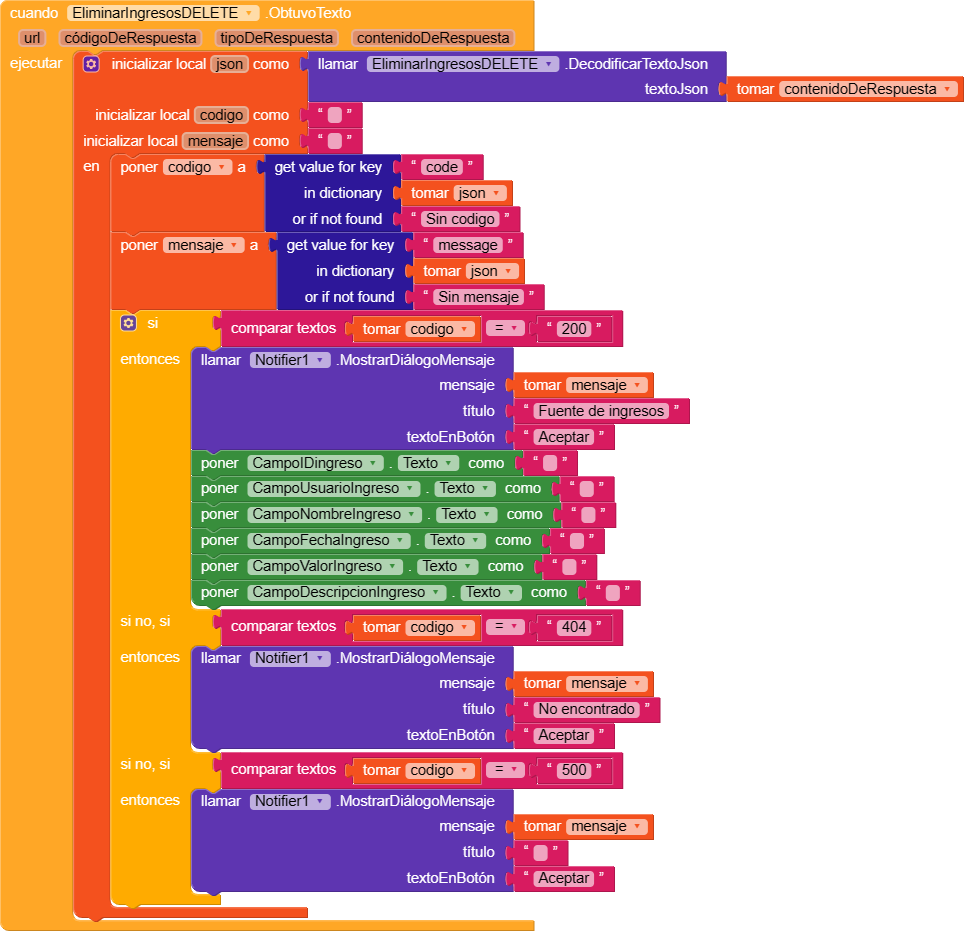
Evento del boton al presionar eliminar fuente de ingresos del usuario que brinda dos opciones para elegir.



Si la elección es “Si” entonces llama a la función que realizara la solicitud.



Función que se le pasa al evento de la notificación para eliminar fuente de ingresos.



Respuesta de la api al eliminar la fuente de ingresos.

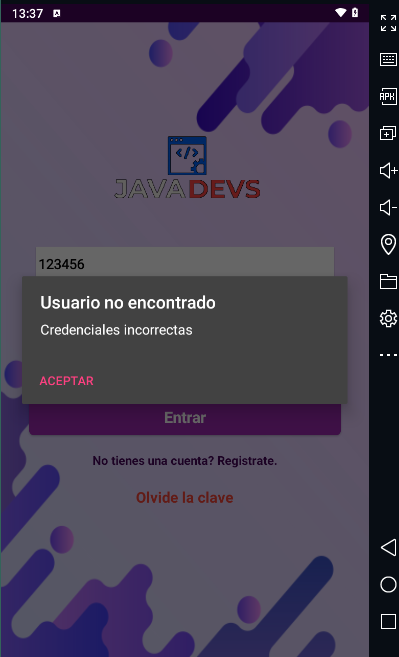
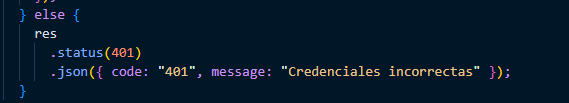
**Funcionamiento de la aplicación**



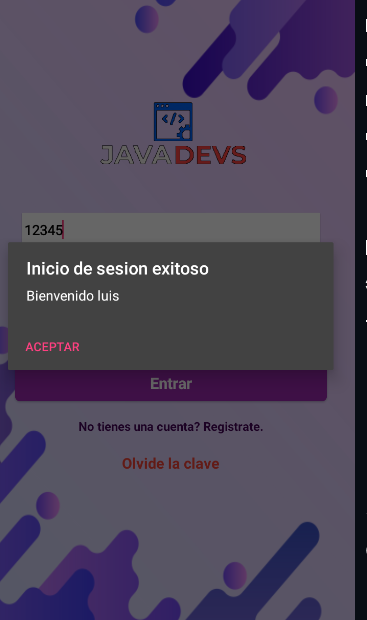
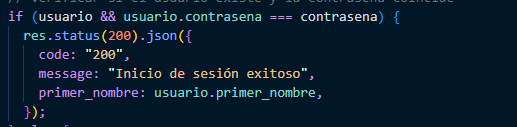
Este el login de la app



Ingresamos la cedula del usuario y contraseña validas

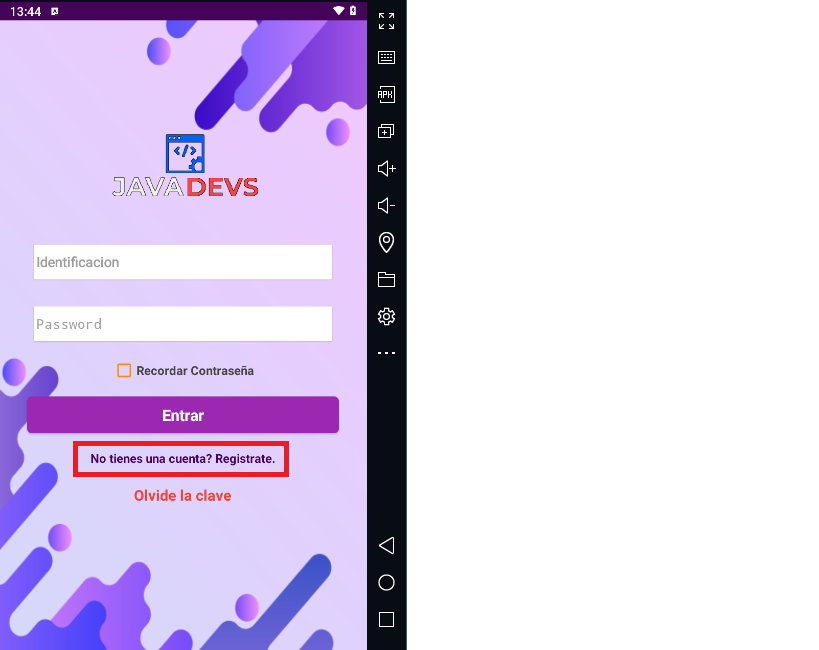


En caso de que la contraseña o cedula sean incorrectas, la aplicación lanza un mensaje diciendo que no fue posible encontrar al usuario.

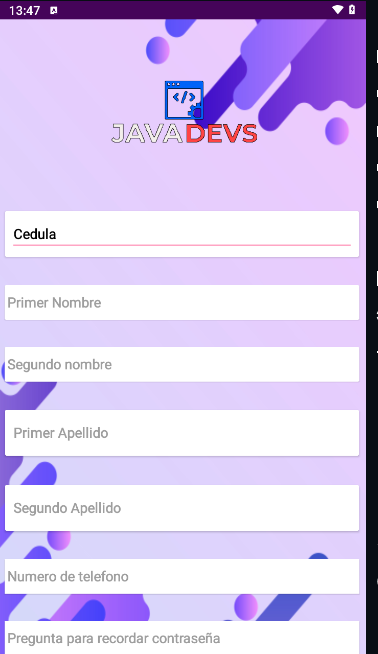


Si las credenciales son válidas, aparecerá un mensaje exitoso junto al primer nombre del usuario registrado

En caso tal de que aun no este registrado, presionamos en “No tienes una cuenta? Registrate.” Para iniciar el proceso de registro.



Se abrirá un formulario

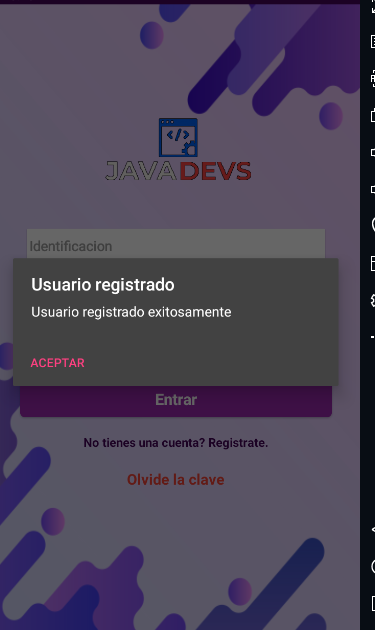
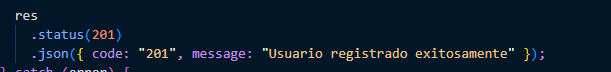


Ingresamos los datos para el nuevo usuario:



Si los datos son aceptables, la aplicación muestra un mensaje de proceso de registro exitoso y lo devolverá al menú de inicio de sesion.

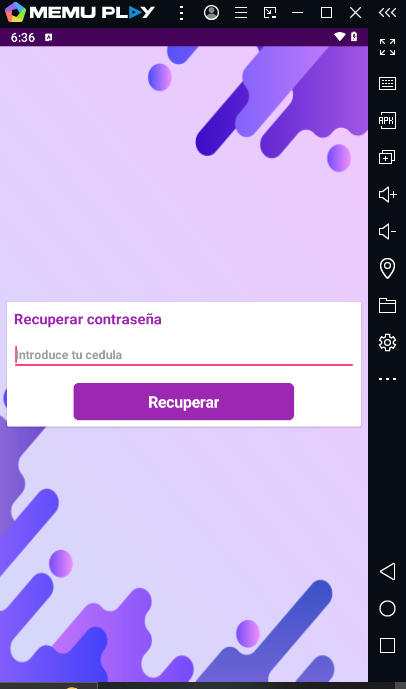


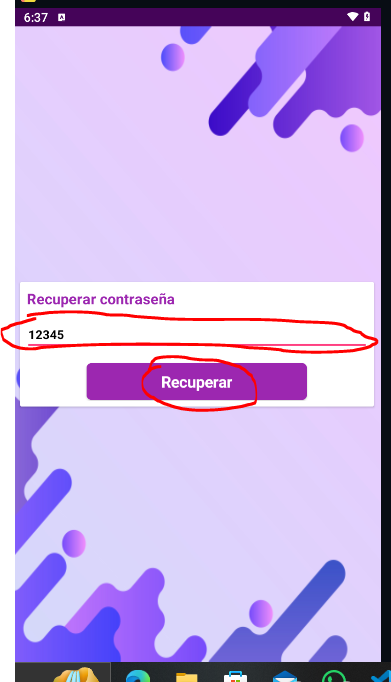


También se encuentra la opción por si se olvido la clave

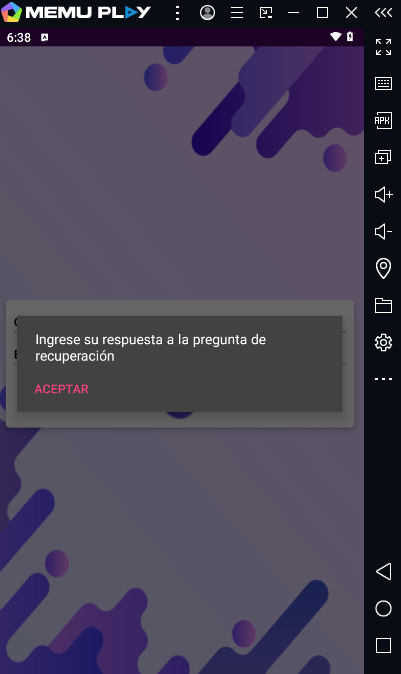
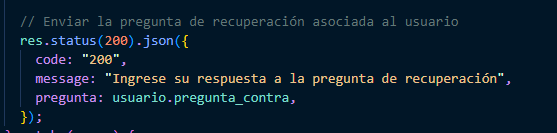


Al darle clic se abre la siguiente vista, donde se colocara la cedula registrada

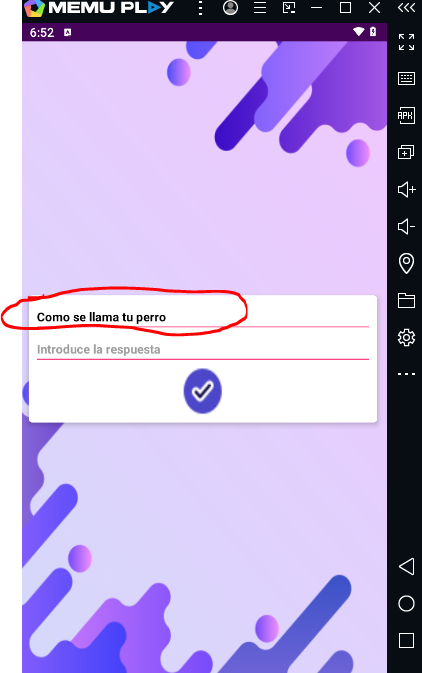
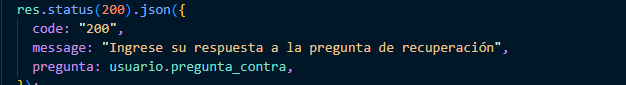




Si la cedula existe o es válida, se abre otra vista con un mensaje de proceso exitoso y la pregunta de recuperación asociada a la cedula cargada en un campo de solo lectura

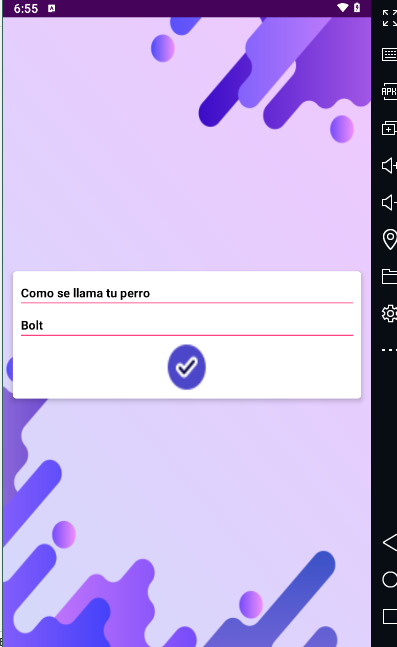




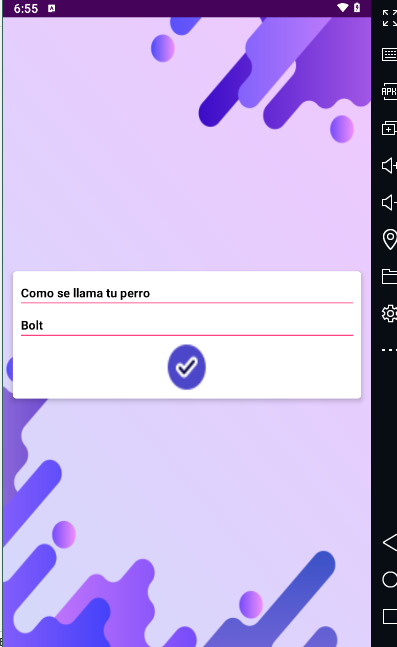




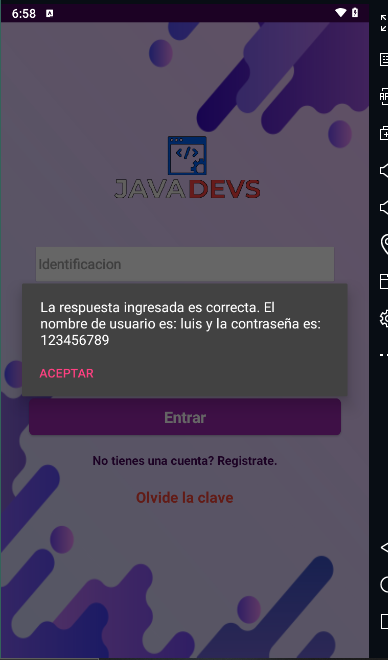
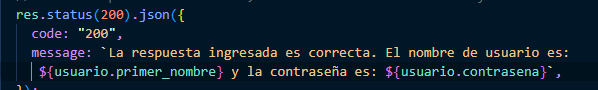
Colocamos la respuesta asociada a esa pregunta



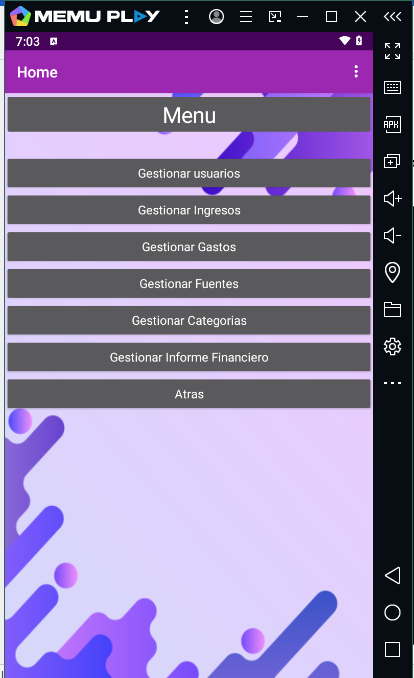
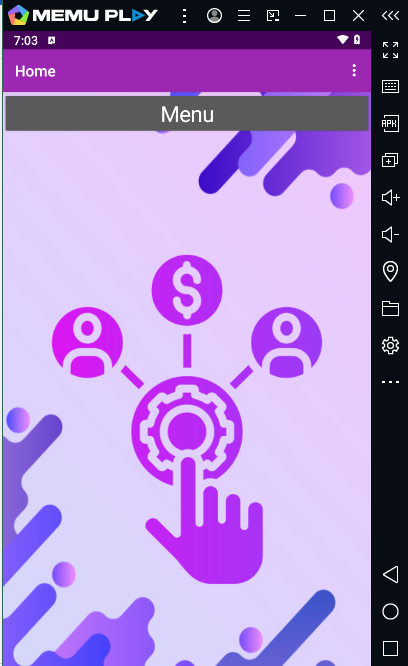
Validamos y si coincide con la del usuario, se debe mostrar un mensaje exitoso con el nombre y contraseña del usuario asociado.

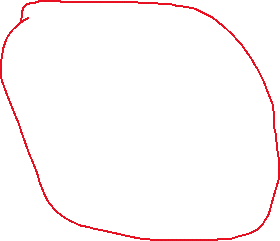




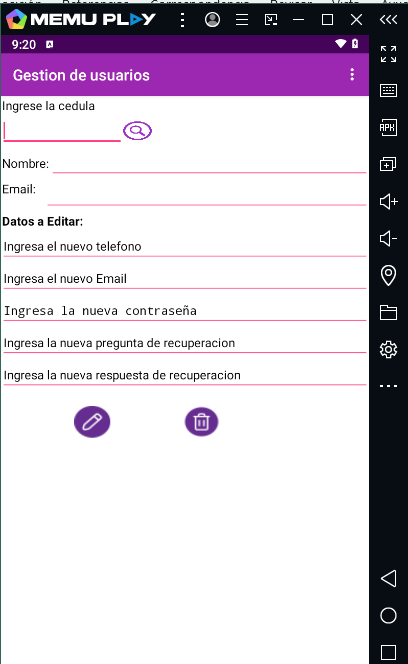


Una vez dentro de la aplicación, nos aparece una opción que si le damos clic se despliega un menú de más opciones

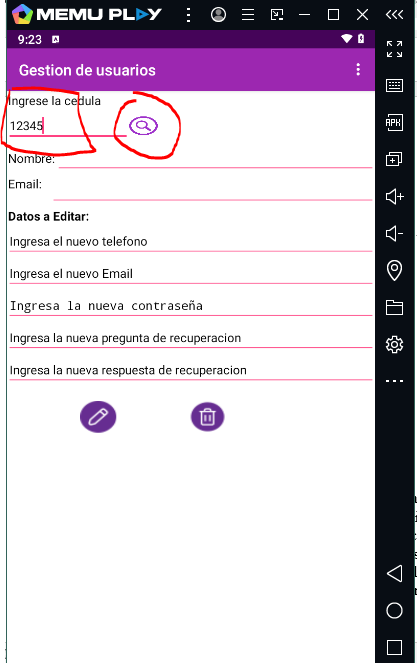




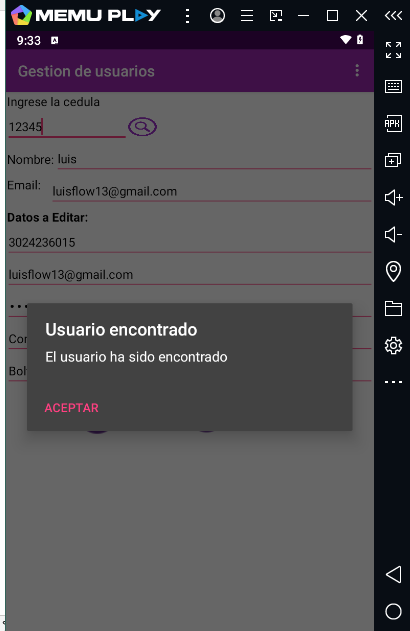
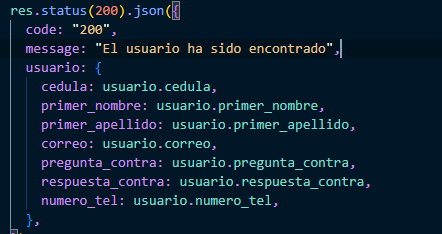
Vamos con la opción para gestionar los usuarios y una ves que le damos clic se abre la siguiente pantalla

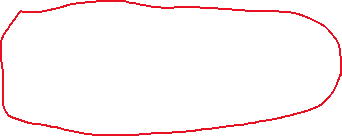


Como se puede observar posee varias funciones para buscar, editar y eliminar usuarios de la aplicación. Vamos a probar primero el buscar con una cedula registrada.

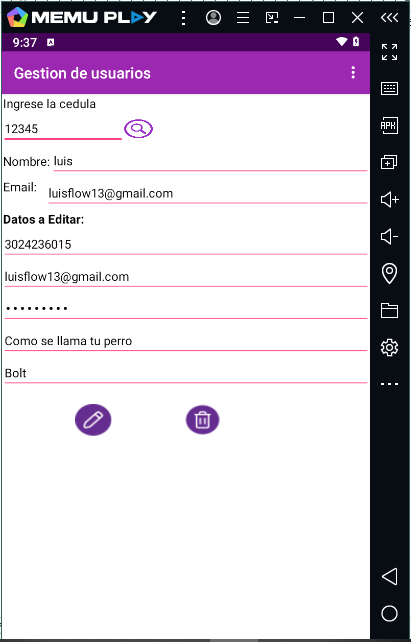


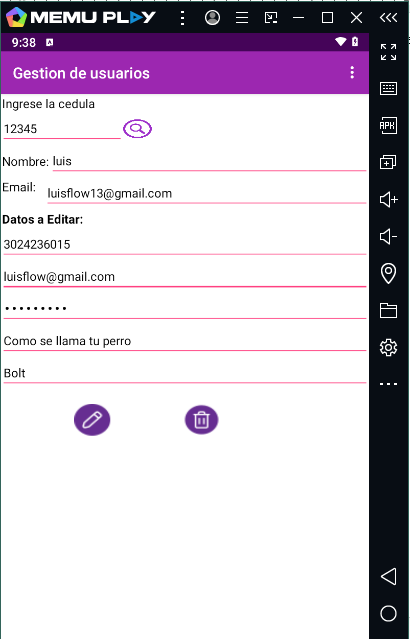
Al darle clic al botón en forma de lupa, mostrara un mensaje exitoso si existe y se cargan en los campos de abajo con la información encontrada y que se puede editar del usuario:





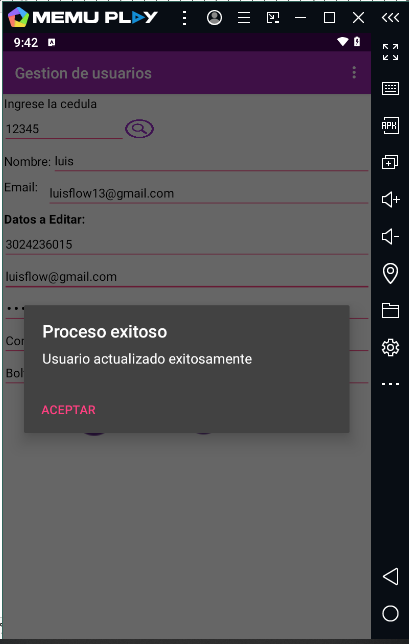
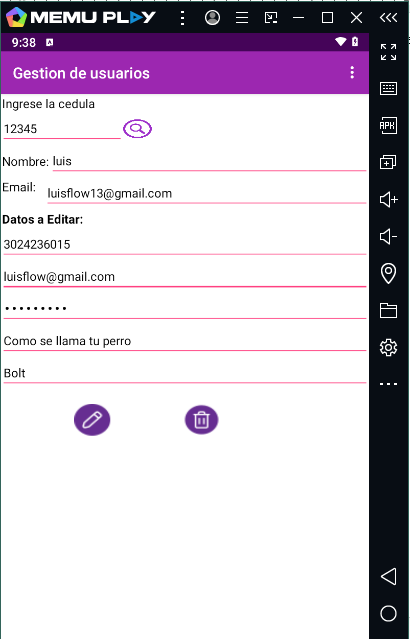
Ahora procederemos a modificar un dato del usuario, en este caso modificare el correo de la siguiente manera







Ahora procederemos actualizar el correo. Para ello presionamos el botón de actualizar, luego debe mostrar un mensaje de exitoso si el dato es correcto o no hay ningún problema con la información.

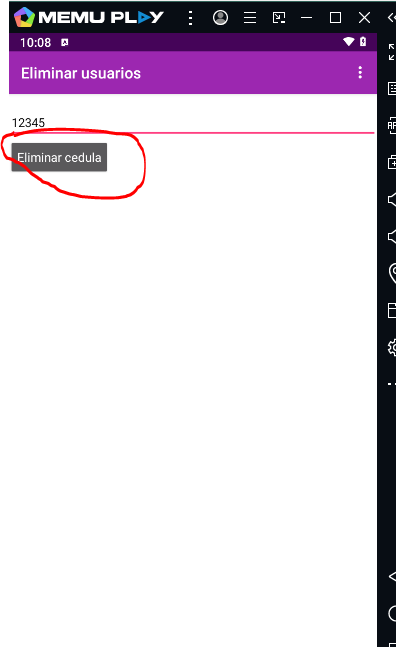


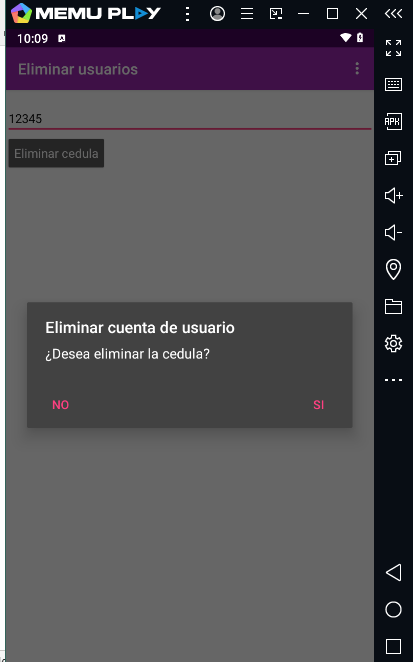




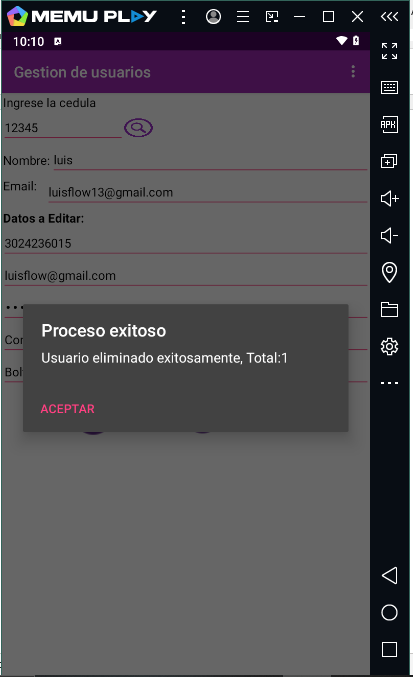


Ahora probemos el eliminar cuenta del usuario. Al darle clic al botón eliminar, debe aparecer una nueva pantalla donde digitaremos la cedula a eliminar y luego presionamos al botón de eliminar cedula y este debe lanzar una notificación de selección donde podemos elegir entre eliminar o no eliminar la cuenta.

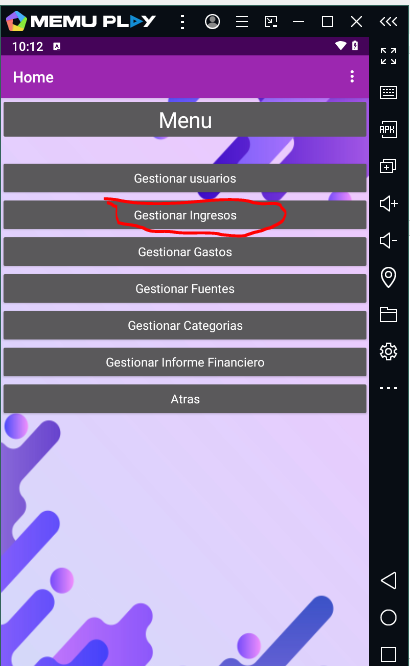
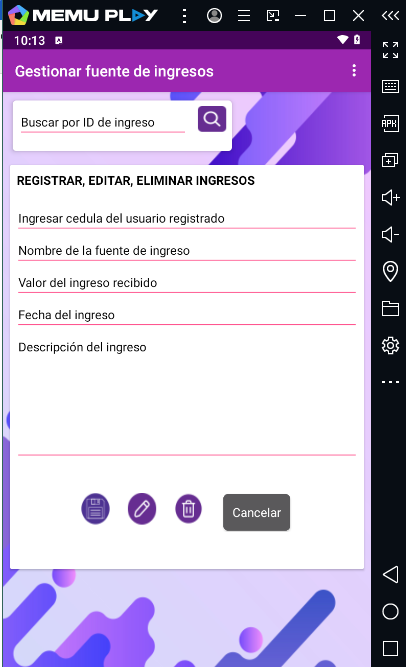




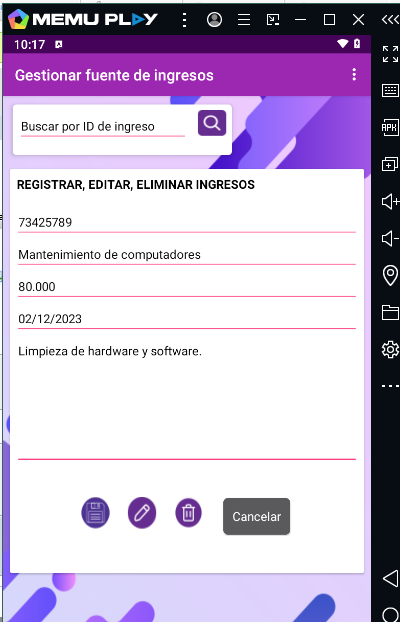
Si presionamos “si” deberá aparecer un mensaje exitoso de la eliminación más la cantidad de usuarios o cedulas disponibles y regresar a la pantalla de gestión de usuario.



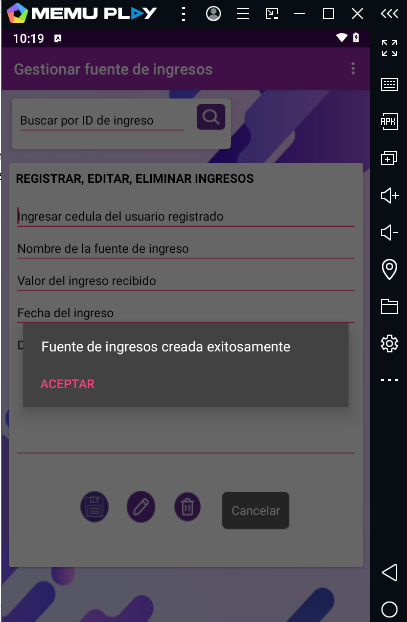
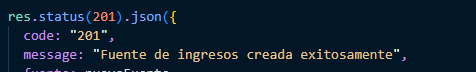
Vamos a probar una función específica para los usuarios que la fuente de ingresos. Al darle clic se deberá abrir una nueva pantalla con la información para gestionar la fuente de ingresos de los usuarios.



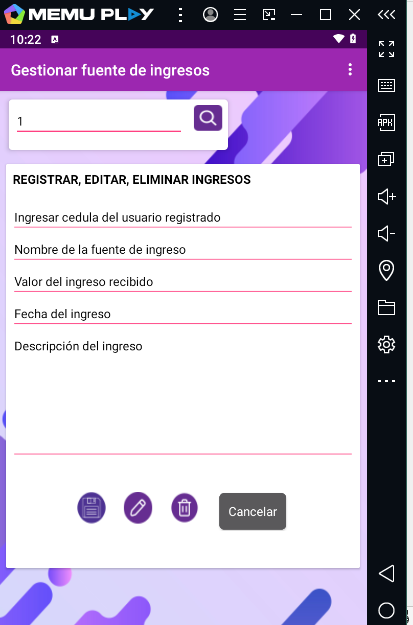
Como se puede observar tiene la opción para buscar fuentes de igreso a través de su ID y se puede registrar, editar y eliminar fuentes de ingresos asociados al usuario encontrado.Primero probemos creando una nueva fuente de ingresos llenado los campos con la información correspondiente.

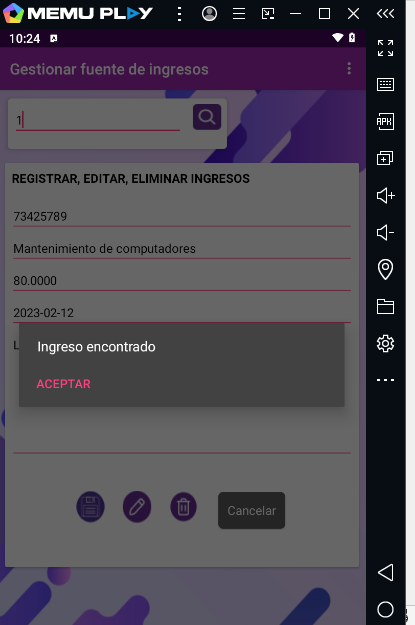


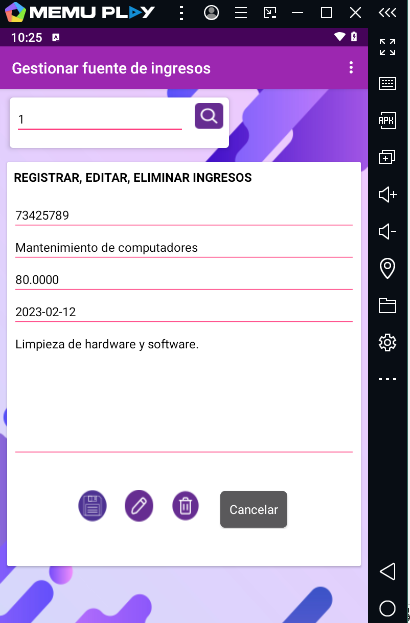
Si presionamos el botón de guardar, deberá registrar este nuevo ingreso al usuario cuya cedula es 73425789. Se deberán de limpiar los campos y aparecerá una notificación de proceso exitoso.

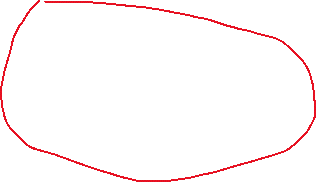


Para comprobar si realmente se creó, pues procedemos a buscarla por su id que por lógica deberá ser 1 ya que es la primera fuente que se crea. Al presionar el botón de buscar deberá mostrar primero una notificación de éxito y segundo se cargaron los campos de abajo con la información asociada a esa fuente de ingresos.

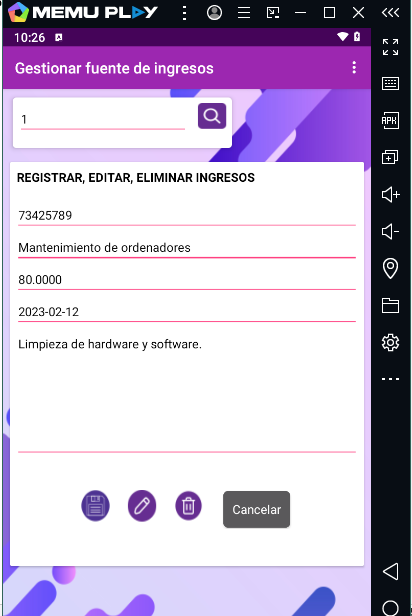
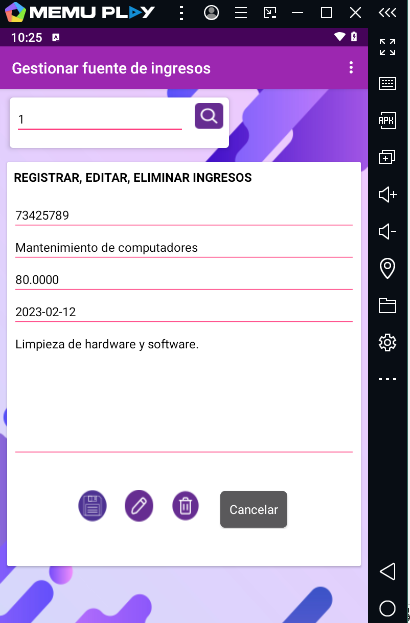




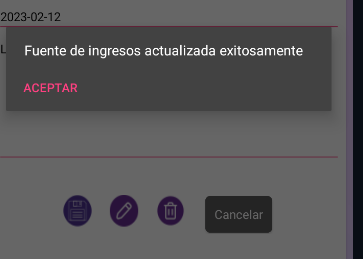


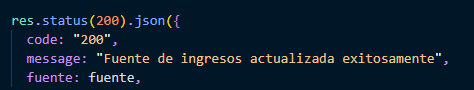


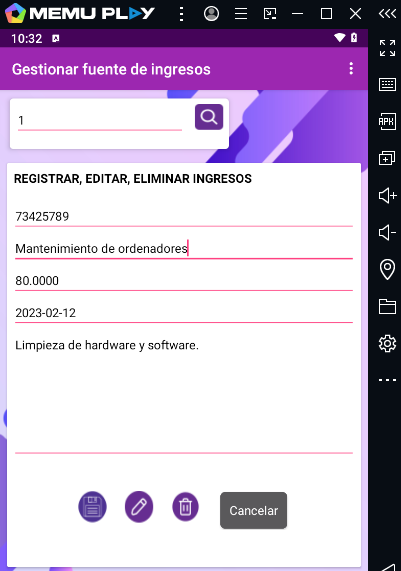
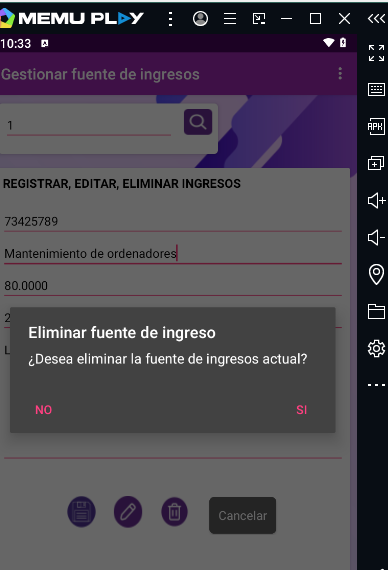
Ahora procederemos a actualizar y eliminar el ingreso para culminar la demostración del funcionamiento de la aplicación.



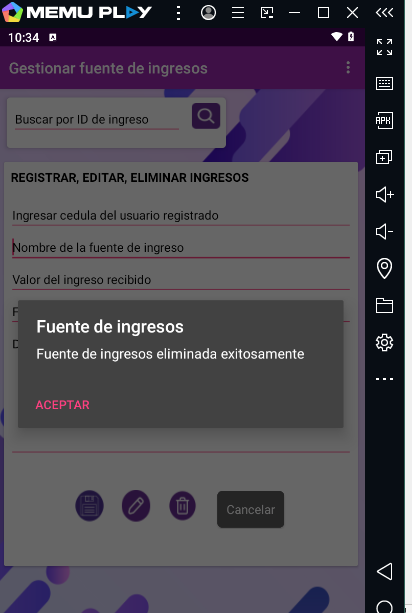


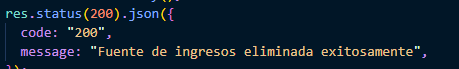


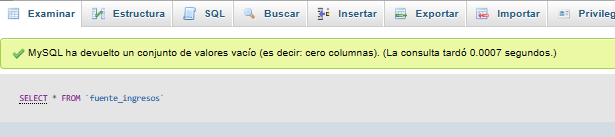












# Síntesis

El trabajo escrito tiene como objetivo desarrollar una aplicación móvil para la gestión de finanzas personales en dispositivos Android. La justificación se basa en la necesidad de proporcionar a los usuarios una herramienta eficiente para organizar sus recursos económicos. La aplicación permitirá registrar ingresos, gastos, categorías y fuentes de ingresos, así como generar informes financieros detallados. Se busca mejorar el bienestar financiero de los usuarios y promover una mayor conciencia en la administración de sus recursos económicos en un entorno digitalizado.

# Bibliografía

Rd, S. [@saloom\_rd]. (s/f). Kodular crud by saloom. Youtube. Recuperado el 7 de abril de 2024, de <http://www.youtube.com/playlist?list=PLkf5Qxv5kxO_cnVsWsfBxZD4tYBC-1-vJ>

Pérez, E. [@edrperez]. (2021, diciembre 24). Prueba de REST en MIT App Inventor 2. Youtube. <https://www.youtube.com/watch?v=-SXg6BSR1XI>

Aviles, J. [@2000jaime]. (2023, octubre 3). App Inventor 2 MIT-2 consumo Apis parte1. Youtube. <https://www.youtube.com/watch?v=XGrd9ukE7As>

Gtt, R. [@RMTGTT]. (2018, febrero 7). Add web API access to MIT App Inventor2 apps PART 2. Youtube. <https://www.youtube.com/watch?v=Nd4PYqCiXXE>

Koder, M. [@mrkoder8819]. (2022, febrero 15). How to use API in kodular | MIT App Inventor. Youtube. <https://www.youtube.com/watch?v=wcJX7TBRxGI>

Wisesa, L. [@lintangwisesa]. (2019, junio 24). Android App Inventor & @FAVORIOT REST API. Youtube. <https://www.youtube.com/watch?v=pFlpfQpe6vI>