GRADO EN INGENIERÍA INFORMÁTICA

PROGRAMACIÓN DE DISPOSITIVOS MÓVILES

UNIVERSIDAD
DE GRANADA

Práctica 2 : Aplicación Android: Gestión de gastos

David Serrano Domínguez (*davidserrano07@correo.ugr.es*)

2 de junio de 2025

Índice

Objetivos

La práctica consiste en desarrollar una aplicación móvil que funcione completamente de forma local, sin necesidad de conexión a un servidor externo. Para esta segunda entrega, se acordó implementar una app de gestión de gastos desde la que se pudiera añadir movimientos tanto de tipo ingreso como gasto, además de crear categorias que a su vez tuvieran subcategorias, poder filtrar por estas en la vista de los movimientos, además de buscar mediante el concepto del movimiento en un buscador.

Por último se decidió añadir la funcionalidad que mediante una foto poder incorporar un movimiento de tipo gasto proveniente de u ticket, sacando su concepto, fecha y importe total.

ENLACE AL REPOSITORIO DE GITHUB : Repositorio de las prácticas

1. Descripción de las funcionalidades de la aplicación

El primer paso a la hora de realizar esta práctica fue establecer las funcionalidades básicas que se deseaban implementar, es decir, lo que se conoce como el **MVP(Producto Mínimo Viable)**, se decidió que lo mínimo requerido sería poder gestionar los **ingresos** y **gastos**, así como las **categorias** y **subcategorías** de estos, con la capacidad de filtrar estos mismos.

Por otro lado, como funcionalidad adicional, se decidió incorporar un escáner de **tickets** que extraerá el **precio total, fecha y concepto** del mismo, con el fin de optimizar la gestión de gastos.

2. Tecnologias Usadas para la implementación

A continuación se explicará las tecnologías escogidas para la realización de la aplicación móvil y se explicará el motivo de la elección de las mismas.

Para el desarrollo de la app se decidió usar la librería de **Javascript** llamada **React**, concretamente **React Native** que es una versión de dicha librería enfocada al **desarrollo móvil** y que permite compilar las aplicaciones para ejecutarlas en **cualquier sistema operativo**.

Además, tras leer la **documentación oficial** ellos mismos recomiendan usar el framework de **Expo** para crear aplicaciones móviles de **React Native** de manera más sencilla y rápida; por tanto, se decidió seguir dicha recomendación y usar dicho **framework** para el desarrollo.

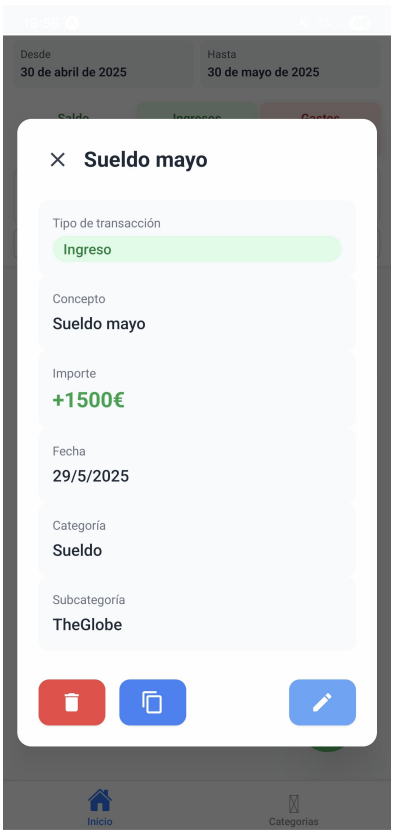
A continuación se deja la documentación de ambas tecnologías:

- **Documentación de React Native**
- **Documentación de Expo**

3. Pantallas principales de la aplicación e implementación

La aplicación tendría **dos** pantallas principales entre las que nos podríamos mover mediante el **navbar** inferior, por un lado tendríamos la página principal que nos permitiría ver nuestros **últimos movimientos** además de poder filtrar estos movimientos de distintas maneras.

Pantalla principal



Información detallada de movimiento

Categorias

En la página principal podríamos por un lado filtrar por **fechas**, **ingresos o gastos**, **categorias**, **subcategorias** y **nombre del movimiento**, además también se podría acceder a la información más especifica del movimiento haciendo click y veríamos la **modal de la segunda imagen**, que nos mostraría la información detallada de ese movimiento, que sería:

- **Tipo de movimiento**
- **Concepto**
- **Importe**
- **Fecha del importe**
- **Categoria**
- **Subcategoria**

Además desde esta modal podríamos **eliminar el movimiento**, **duplicarlo** o **editarlo**.

Una vez vistas las pantallas y vistas principales de nuestra aplicación vamos a pasar a ver los principales movimientos que nos van a permitir gestionar tanto los **movimientos**, como las **categorias y subcategorias** de nuestra aplicación

Formulario categoria

Formulario movimiento

Empezaremos hablando del **formulario de movimiento**, ya que este formulario será el eje central alrededor del cual girará el resto de nuestra aplicación.

El formulario de crear movimiento constaría de hasta **seis campos** que serían los mismos que vimos en la vista detallada de un movimiento, siendo todos obligatorios, excepto el de **descripción** que si sería opcional. Además de esto tendríamos arriba a la derecha la opción de **escanear**, que permitiría escanear un **ticket** y extraer de este el **concepto** que sería el nombre de la tienda a la que pertenece el ticket, el **importe** y la **fecha** del propio ticket. El resto de cmapos quedaría en responsabilidad del usuario rellenarlos.

Una vez visto el **formulario de movimiento**, pasaríamos al de **categorías** lo cual nos permitiría crear **nuevas categorias** o **subcategorias**.

El formulario sería mucho más sencillo, ya que solamente tendríamos como campo **obligatorio** uno único, que sería el de **nombre de la categoria**, dado que las **subcategorías** serían opcionales.

De esta manera habríamos visto por un lado las tres **pantallas** que componen nuestra aplicación y sus dos **formularios principales**.

3.1. Componentes principales

Vamos a explicar los dos componente principales de la aplicación que permiten su funcionamiento.

3.1.1. Formulario Movimiento

En este apartado vamos a profundizar en el componente de **formulario movimiento**, viendo su funcionamiento más específico a la para poder de añadir y editar **movimientos** a nuestra aplicación.

Empezaremos viendo la función encargada de **añadir movimientos** a nuestra aplicación, concretamente nos vamos a encontrar una función

```
const handleMovimiento = async (
  tipo: 'ingreso' | 'gasto',
  cantidad: number,
  concepto: string,
  descripcion: string,
  fecha: Date,
  categoria: Categoria,
  subcategoria: Subcategoria,
  movimientoExistente: Movimiento | null
) => {
  try {

    const cantidadFinal = tipo === 'gasto'
      ? (cantidad > 0 ? -cantidad : cantidad)
      : cantidad;

    console.log(fecha);

    const nuevoMovimiento: Movimiento = {
      id: movimientoExistente ? movimientoExistente.id : Date.now().toString(),
      concepto,
      descripcion,
      cantidad: cantidadFinal,
      fecha: fecha.toISOString(),
      categoria,
      subcategoria
    };

    let nuevoHistorico: Movimiento[];
    if (!movimientoExistente) {
      nuevoHistorico = [...historicoMovimientos, nuevoMovimiento];
    } else {
      nuevoHistorico = historicoMovimientos.map((current) =>
        current.id === movimientoExistente.id ? nuevoMovimiento : current
      );
    }

    await AsyncStorage.setItem('historico', JSON.stringify(nuevoHistorico));
    setHistoricoMovimientos(nuevoHistorico);


    const totalIngresos = nuevoHistorico
      .filter(mov => mov.cantidad > 0)
      .reduce((sum, mov) => sum + mov.cantidad, 0);

    const totalGastos = nuevoHistorico
      .filter(mov => mov.cantidad < 0)
      .reduce((sum, mov) => sum + Math.abs(mov.cantidad), 0);

    setIngresos(totalIngresos);
    setGastos(totalGastos);
    setSaldo(totalIngresos - totalGastos);
```

```
    } catch (error) {
      console.error('Error al guardar movimiento:', error);
    }
  };
```

Como vemos esta función recibirá por parámetros los valores necesarios para crear un objeto de tipo **movimiento**, y además actualizará en el **AsynStorage** el array de movimientos de nuestra apliación. Además actualizará los estados para así mostrar al usuario los cambios en los valores de **saldo**, **ingresos** y **gastos**.

También como podemos observar esta función nos permitira poder modificar un **ingreso** ya existente en la apliación, an solo con recibir em sus parametros un objeto de tipo **movimiento** que no sea nulo nos dirá que debemos actualizarlo con los nuevos valores que recibamos por dichos parámetros.

3.1.2. Formulario Categoria

En el caso del formulario de categoria nos vamos a encontrar con algo similar, pero esta vez la función para añadir una categoría en nuestra aplicación la vamos a sacar de un contexto que tenemos, ya que las categorias al manejarlas desde distinta páginas y ventanas se consideor oportuno crear un contexto de categorias y por tanto las funciones para manejar dichas categorias se sacaría del contexto como vemos a continuación.

```
const context = useContext(CategoryContext);
const { handleAddCategoria, handleUpdateCategoria } = context as {
    categorias: Categoria[];
    handleAddCategoria: Function;
    handleUpdateCategoria: Function;
};
```

Como vemos del contexto **CategoryContexto** sacaríamos dos funciones que usaremos para **crear** y a **actualizar las categorias**.

Una vez visto como obtenemos estas funciones vamos a pasar a ver a continuación su implmentación.

```

const handleAddCategoria = async (nuevaCategoria: Categoria) => {
    try {
        const data = await AsyncStorage.getItem('categorias');
        if (data) {
            const categoriasActuales = await JSON.parse(data);
            const nuevasCategorias = [...categoriasActuales, nuevaCategoria];
            await AsyncStorage.setItem('categorias', JSON.stringify(nuevasCategorias));
            setCategorias(nuevasCategorias);
        } else {
            const nuevasCategorias = [nuevaCategoria];
            await AsyncStorage.setItem('categorias', JSON.stringify(nuevasCategorias));
            setCategorias(nuevasCategorias);
        }
    } catch (error) {
        console.log(error);
    }
}
```

Empezaremos viendo la implementación de la función de **añadir categoria**, que lo que hará será obtener del **AsyncStorage** el array de categorias actual, sumarle la nueva y una vez hecho esto modificar el array que contienen las categorias del **AsynStorage** y el estado del contexto con la nueva categoria.

```

const handleUpdateCategoria = async (categoriaEdidata: Categoria) => {
```

```
    try {
        const data = await AsyncStorage.getItem('categorias');
        if (data) {
            const categoriasActuales = await JSON.parse(data);
            const nuevasCategoria = categoriasActuales.map((categoria: Categoria) => {
                if (categoria.id === categoriaEdidata.id) {
                    return categoriaEdidata;
                } else {
                    return categoria;
                }
            });
            await AsyncStorage.setItem('categorias', JSON.stringify(nuevasCategoria));
            setCategorias(nuevasCategoria);
        }
    } catch (error) {
        console.log(error);
    }
}
```

En el caso de la función de actualizar categoria, nos vamos a encontrar con algo similar, solo que en este caso tras obtener el array de categorias del **AsyncStorage** obtenemos la categoría que vamos a modificar buscando por su **id** y posteriormente actualizamos con los nuevos valores de esa categoria, para después actualizar el **AsynStorage** y el estado del contexto de las categorias.

3.1.3. Ticket Scanner

Una vez vistos los dos formularios principales de nuestra aplicación vamos a pasar a ver uno de los más importantes que es el **componente** que va a permitir que podamos escanear tickets de los que sacar información como sería el **concepto**, **fecha** y **total**.

Para ello, nos encontraremos con **cuatro** funciones principales que nos permitirán obtener dicha información de la foto.

Vamos a empezar explicando que paquetes han sido necesarios para poder implementar este componente, además de que **api** hemos hecho uso para hacer el **ocr** a la imagen que nos ha permitido extraer el texto de la misma.

Para esta implementación han sido necesarias principalmente tres bibliotecas, que han sido:

- **Expo File System**
- **Expo Image Manipulator**
- **Expo Image Picker**

Gracias a estas tres bibliotecas se ha conseguido poder capturar la imagen y tratarla para posteriormente mandarla a la **API** pública de la que se ha hecho uso y hablaremos a continuación para poder pasarlo un **OCR** a la imagen.

Una vez vistas las bibliotecas vamos a pasar a ver la **API**, la cual se llama **OCR Space**. Esta es una api pública que nos permitira mediante una petición **POST**, pasarle la imagen en **base 64** y nos devolvera la información de la imagen.

Una vez vistas las **bibliotecas** y la **API** usada vamos a pasar a ver más en detalle cada una de las **cuatro funciones** que forman parte de este proceso desde que capturamos la imagen hasta que obtenemos los datos deseados.

```
const pickImage = async () => {
    try {
        const result = await ImagePicker.launchCameraAsync({
            mediaTypes: ImagePicker.MediaTypeOptions.Images,
            allowsEditing: true,
            quality: 1,
        });


        if (!result.canceled) {
            setIsProcessing(true);
            await processImage(result.assets[0].uri);
        }
    } catch (error) {
        console.error('Error al tomar la foto:', error);
        setIsProcessing(false);
    }
};
```

Como podemos obervar esta función se encarga de capturar la foto, haciendo uso de la libreria de **ImagepPicker** del ticket y en caso de no haber error se encarga de llamar a la función **processImage** para procesar dicha imagen y pasarla a **base 64**, a continuación veremos esta función.

```
const processImage = async (imageUri: string) => {
    try {
        const manipulatedImage = await ImageManipulator.manipulateAsync(
            imageUri,
            [
                { resize: { width: 800 } },
            ],
            {
                compress: 0.7,
                format: ImageManipulator.SaveFormat.JPEG
            }
        );

        const base64 = await FileSystem.readAsStringAsync(manipulatedImage.uri, {
            encoding: FileSystem.EncodingType.Base64,
        });


        const sizeInBytes = (base64.length * 3) / 4;
        const sizeInKB = sizeInBytes / 1024;

        if (sizeInKB > 900) {
            const smallerImage = await ImageManipulator.manipulateAsync(
                imageUri,
                [
                    { resize: { width: 600 } },
                ],
                {
                    compress: 0.5,
                    format: ImageManipulator.SaveFormat.JPEG
                }
            );

            const newBase64 = await FileSystem.readAsStringAsync(smallerImage.uri, {
                encoding: FileSystem.EncodingType.Base64,
            });

            return await sendToOcr(newBase64);
        }

```

```
        return await sendToOcr(base64);

    } catch (error) {
        console.error('Error procesando la imagen:', error);
        Alert.alert(
            "Error",
            "No se pudo procesar el ticket. Por favor, intentalo de nuevo.",
            [{ text: "OK" }]
        );
    } finally {
        setIsProcessing(false);
    }
};
```

Esta función se encargará de reducir por un lado el **ancho y tamaño** de la imagen para así evitar que supere los **900 kilobytes** ya que el máximo que soporta la **API** es **1024 kilobytes**, en caso de que con el primer ajuste supere los **900 kilobytes** se reducira aún mas su tamaño y posteriormente se tranformara en **base 64** y se mandará a el función que se encarga de hacer la petición a la **API** que le hará el **OCR**.

Además toda la función esta envuelta en un **try catch** para que en caso de cualquier error, avisar al usuario mediante el uso de un **alert**.

```

const sendToOcr = async (base64: string) => {
    try {
        const formData = new FormData();
        formData.append('apikey', 'K81796944888957');
        formData.append('language', 'spa');
        formData.append('isOverlayRequired', 'false');
        formData.append('base64Image', `${base64}`);


        const response = await fetch('https://api.ocr.space/parse/image', {
            method: 'POST',
            headers: {
                'apikey': 'K81796944888957',
            },
            body: formData
        });

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

        const result = await response.json();

        if (result.ParsedResults && result.ParsedResults.length > 0) {
            const text = result.ParsedResults[0].ParsedText;
            const data = parseTicketText(text);
            onTicketProcessed(data);
        } else {
            throw new Error('No se pudo extraer texto de la imagen');
        }
    } catch (error) {
        console.error('Error en OCR:', error);
        throw error;
    } finally {
        setIsProcessing(false);
    }
};
```

Esta función será la encargada de hacer la petición a la **API** que hará el **OCR** a nuestra imagen, para ello mandaremos en la petición aparte de la **API KEY** otros parametros, y nuestra imagen en **base 64**. En caso de que la respuesta no sea **200** lanzará un error; en caso contrario, obtendremos el resultado en **json**, extraeremos el

texto del resultado y se lo enviaremos a la función encargada de **parsear** el ticket y obtener la información que deseamos.

```
const parseTicketText = (text: string) => {
    const lines = text.split('\n').map(l => l.trim()).filter(l => l !== '');
    let total = 0;
    let concepto = '';
    let fecha: Date | null = null;

    for (const line of lines) {
        if (/^[A-Z\s]+$/.test(line) && line.length > 3) {
            concepto = line;
            break;
        }
    }

    for (const line of lines) {
        const dateMatch = line.match(/(\d{2})[\/\-\.](\d{2})[\/\-\.](\d{4})/);
        if (dateMatch) {
            const [, day, month, year] = dateMatch;
            fecha = new Date(parseInt(year), parseInt(month) - 1, parseInt(day));
            break;
        }
    }

    for (let i = 0; i < lines.length; i++) {
        if (lines[i].toUpperCase().includes('TOTAL')) {
            let candidates: number[] = [];
            const priceMatches = lines[i].match(/(\d+[.,]\d{2})/g);
            if (priceMatches) {
                candidates.push(...priceMatches.map(p => parseFloat(p.replace(',', '.'))));
            }
            for (let j = 1; j <= 2; j++) {
                if (lines[i + j]) {
                    const nextMatches = lines[i + j].match(/(\d+[.,]\d{2})/g);
                    if (nextMatches) {
                        candidates.push(...nextMatches.map(p => parseFloat(p.replace(',', '
.'))));
                    }
                }
            }
            if (candidates.length > 0) {
                total = Math.max(...candidates);
                break;
            }
        }
    }

    if (!fecha) fecha = new Date();
    if (!total) total = 0;

    return {
        concepto,
        cantidad: parseFloat(total.toFixed(2)),
        fecha
    };
};
```

La función mostrada se encarga de analizar el contenido de un ticket en formato texto con el objetivo de extraer tres datos clave: el nombre del comercio (concepto), la fecha de la compra y el importe total.

Para ello, en primer lugar se divide el texto en líneas, eliminando aquellas vacías o formadas solo por espacios. A continuación, se realiza un recorrido inicial buscando una línea que contenga únicamente letras mayúsculas y espacios, lo que comúnmente representa el nombre del establecimiento. Esta línea se toma como el **concepto**.

Seguidamente, se analiza cada línea en busca de una fecha con el formato día/mes/año, permitiendo separadores como barra **/**, guion **-** o punto **.**, aceptando fechas del estilo **10/03/2025**, **10-03-2025** o **10.03.2025**. Una vez encontrada la primera coincidencia válida, se convierte a un objeto **Date**.

Finalmente, se busca el importe total del ticket. Para ello se recorren las líneas en busca de la palabra clave **TOTAL**. Una vez localizada, se extraen los posibles valores numéricos en esa línea y en las dos siguientes, considerando formatos con punto o coma decimal (por ejemplo: **23.45** o **23,45**). De entre todos los valores encontrados, se asume que el mayor representa el importe total del ticket.

Si durante el análisis no se identifica una fecha o un total, se proceden a asignar valores por defecto: la fecha actual y un total de **0**.

El resultado es un objeto que incluye los **concepto, la cantidad y la fecha** extraídos del ticket, los cuales se añadirán al formulario de movimiento.

4. Conclusiones

La aplicación desarrollada resulta ser una herramienta **altamente útil**, ya que cumple de manera efectiva con su funcionalidad principal: la gestión de ingresos y gastos. Permite al usuario registrar movimientos económicos, asignarles una categoría y subcategoría, así como filtrarlos fácilmente para un mejor seguimiento y análisis. Además, se ofrece la posibilidad de gestionar estas categorías y subcategorías de forma completa, permitiendo su creación, edición y eliminación según las necesidades del usuario.

Como valor añadido, se ha incorporado una funcionalidad adicional que mejora notablemente la experiencia de uso: la posibilidad de crear nuevos movimientos a partir de una imagen de un ticket. Gracias a esta característica, el sistema es capaz de extraer automáticamente el **concepto**, la **fecha** y el **importe total** del ticket mediante técnicas de análisis de texto, permitiendo así registrar un gasto de forma rápida, cómoda y precisa.

En conjunto, estas funcionalidades convierten la aplicación en una solución práctica y versátil para el control financiero personal.