WorkPad

Nombre del fichero:	Proyecto Final PGL
Fecha de esta versión:	30/11/2025
Repositorio GitHub	https://github.com/Ari-dev-design/WorkPad

ÍNDICE

1 Introducción

WorkPad es una aplicación móvil desarrollada para ayudar a freelancers y trabajadores independientes a organizar su trabajo diario. La idea principal es centralizar en un mismo lugar la gestión de clientes, proyectos y facturas.

CONTEXTO DEL PROYECTO

El proyecto surge al identificar que muchos freelancers utilizan varias aplicaciones diferentes o incluso hojas de cálculo para llevar el control de su trabajo. Esta fragmentación dificulta tener una visión clara del negocio: quiénes son los clientes, qué proyectos están activos y cuánto dinero se ha facturado realmente.

Por eso se decide crear una solución más simple: una aplicación donde se puede consultar todo de un vistazo y gestionar los tres aspectos principales que necesita controlar un freelancer.

OBJETIVOS PRINCIPALES

Los objetivos que se plantean para este proyecto son:

1. Crear una aplicación completa donde se puedan añadir, consultar, modificar y eliminar clientes, proyectos y facturas. Todo conectado entre sí de forma lógica.

2. Aprovechar las funciones nativas del dispositivo:

   - Permitir capturar fotos de los logos de los clientes o subirlas desde la galería

   - Guardar la ubicación de cada cliente en un mapa interactivo

   - Garantizar que funcione en iPhone, Android y navegador web

3. Utilizar una base de datos real en la nube:

   - Se elige Supabase por ser gratuito y sencillo de configurar

   - Toda la información se almacena online y está siempre disponible

   - Las imágenes también se suben a la nube de forma automática

4. Desarrollar el código con TypeScript para reducir errores durante la programación.

5. Diseñar una interfaz visual limpia y fácil de usar, sin complicaciones innecesarias.

LO QUE HACE LA APLICACIÓN

Con **WorkPad** se puede:

- Guardar información de clientes: nombre, email, teléfono, logo y ubicación geográfica

- Crear proyectos asociados a cada cliente: con título, descripción, precio y fecha de entrega

- Generar facturas vinculadas a esos proyectos: con número, importe y estado de pago

- Consultar un resumen en la pantalla principal con el total de ingresos (suma de facturas pagadas)

- Buscar clientes rápidamente escribiendo su nombre o email

REQUISITOS CUMPLIDOS

Este proyecto cumple con todos los requisitos especificados en el enunciado:

Operaciones completas de crear, leer, modificar y eliminar en tres entidades diferentes

Relaciones entre entidades: cada cliente tiene sus proyectos, cada proyecto sus facturas

Conexión con una API REST para persistencia de datos

Integración con cámara y galería del dispositivo

Implementación de mapas para geolocalización

Sistema de navegación entre múltiples pantallas

2 Desarrollo

ARQUITECTURA GENERAL

La aplicación se estructura en tres capas principales:

1. CAPA DE PRESENTACIÓN (Frontend)
   - Desarrollada con React Native y Expo
   - Utiliza Expo Router para la navegación basada en archivos
   - Componentes reutilizables construidos con TypeScript
   - Estilos inline con StyleSheet de React Native
2. CAPA DE LÓGICA DE NEGOCIO (Servicios)
   - Archivo centralizado api.ts con todas las funciones de comunicación
   - Gestión de llamadas HTTP mediante Fetch API
   - Manejo de subida de imágenes a Supabase Storage
   - Validación y transformación de datos antes de enviarlos al backend
3. CAPA DE DATOS (Backend)
   - Supabase como Backend as a Service (BaaS)
   - Base de datos PostgreSQL con tres tablas relacionadas
   - Sistema de almacenamiento en la nube para imágenes
   - API REST generada automáticamente por Supabase

PATRÓN DE DISEÑO

Se utiliza el patrón de diseño basado en componentes con las siguientes características:

- Separación de responsabilidades: cada componente tiene una función específica

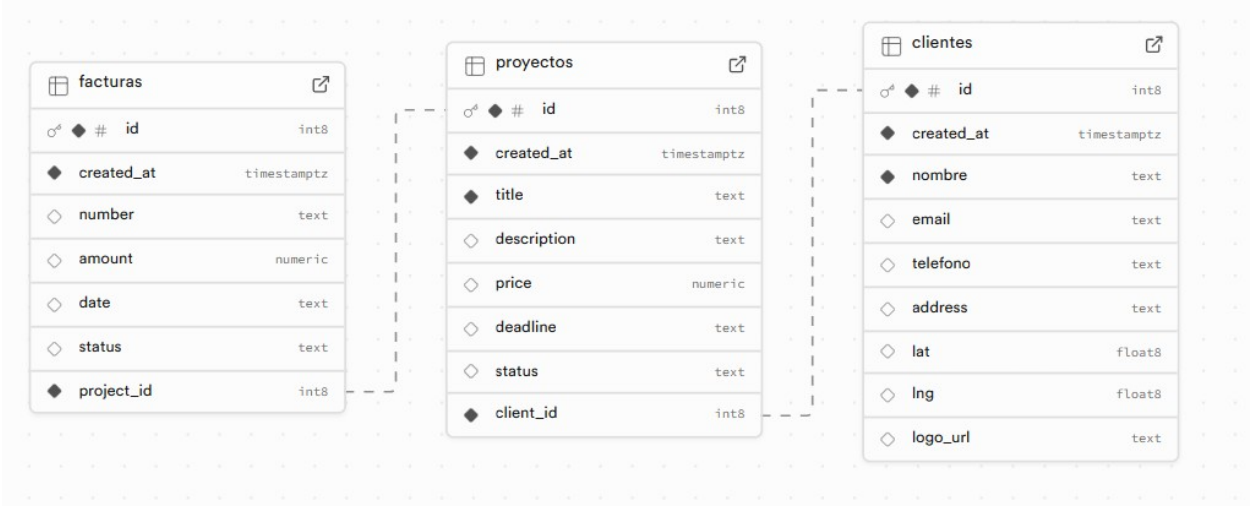
- Reutilización: componentes como Button, Input, ClientCard se usan en múltiples pantallas

- Estado local con hooks: useState para datos temporales, useFocusEffect para recargar al navegar

- Props tipadas con TypeScript para evitar errores en tiempo de ejecución

MODELO DE BASE DE DATOS

La base de datos se compone de tres tablas relacionadas:



Relaciones:

- clientes (1) → (N) proyectos: Un cliente puede tener múltiples proyectos

- proyectos (1) → (N) facturas: Un proyecto puede tener múltiples facturas

- Se implementa CASCADE DELETE: al eliminar un cliente, se eliminan automáticamente sus proyectos y facturas

Campos clave:

- clientes: almacena datos de contacto (nombre, email, teléfono), ubicación (lat, lng) y logo_url

- proyectos: incluye información del trabajo (title, description, price, deadline, status)

- facturas: registra datos de facturación (number, amount, date, status)

Estructura del Proyecto en VSCode

3 API Y Comunicación con Backend

CONFIGURACIÓN DE SUPABASE

La comunicación con el backend se realiza a través de Supabase REST API. Se configuran las credenciales y headers necesarios en el archivo services/api.ts:

```
WorkPad > services > JS api.js > ...
1   const SUPABASE_URL = "https://vxqalqfhfkavtvaijkcn.supabase.co";
2   const SUPABASE_KEY = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6InZ4cWF
3
4   const headers = {
5     "apikey": SUPABASE_KEY,
6     "Authorization": `Bearer ${SUPABASE_KEY}`,
7     "Content-Type": "application/json"
8   };
```

Función getClients

Esta función realiza una petición GET y ordena los resultados por fecha de creación descendente.

```
export const getClients = async () => {
  try {
    const response = await fetch(`${SUPABASE_URL}/rest/v1/clientes?select=*&order=created_at.desc`
    return response.ok ? await response.json() : [];
  } catch (e) { return []; }
};
```

Función insertClient

Antes de insertar, se sube la imagen (si existe) y se obtiene su URL pública.

```
export const insertClient = async (clientData) => {
  try {
    let finalLogoUrl = null;
    if (clientData.logo) finalLogoUrl = await uploadImage(clientData.logo);

    const body = {
      nombre: clientData.name,
      email: clientData.email,
      telefono: clientData.phone,
      lat: clientData.lat,
      lng: clientData.lng,
      logo_url: finalLogoUrl
    };

    const response = await fetch(`${SUPABASE_URL}/rest/v1/clientes`, {
      method: 'POST',
      headers: headers,
      body: JSON.stringify(body)
    });
    return response.ok;
  } catch (e) { return false; }
};
```

Función deleteClient
Gracias al **CASCADE DELETE** configurado en la base de datos, al eliminar un cliente se borran automáticamente sus proyectos y facturas.
Se implementa una función auxiliar para gestionar la subida de imágenes al bucket de Supabase Storage:

```
export const deleteClient = async (id) => {
    const response = await fetch(`${SUPABASE_URL}/rest/v1/clientes?id=eq.${id}`, { method: 'DELETE', headers });
    return response.ok;
};
```

Función uploadImage

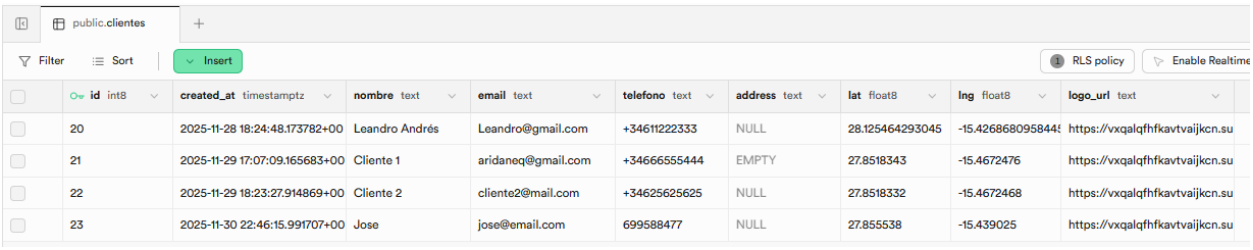
Esta función:
1. Extrae el nombre del archivo de la URI local
2. Crea un FormData con el archivo
3. Sube la imagen al bucket "logos"
4. Retorna la URL pública de la imagen

```
const uploadImage = async (localUri) => {
  try {
    if (!localUri) return null;
    const filename = localUri.split('/').pop();
    const match = /\.(\w+)$/.exec(filename);
    const type = match ? `image/${match[1]}` : `image`;

    const formData = new FormData();
    formData.append('file', {
      uri: localUri,
      name: filename,
      type: type
    });

    const upload = await fetch(`${SUPABASE_URL}/storage/v1/object/logos/${filename}`, {
      method: 'POST',
      headers: {
        "apikey": SUPABASE_KEY,
        "Authorization": `Bearer ${SUPABASE_KEY}`,
      },
      body: formData
    });

    if (!upload.ok) throw new Error('Falló la subida');
    return `${SUPABASE_URL}/storage/v1/object/public/logos/${filename}`;
  } catch (e) {
    console.error("Error en uploadImage:", e);
    return null;
  }
};
```

Tabla clientes completa en Supabase
Se observan todas las columnas de la tabla:

- id: Identificador único autogenerado (int8)
- created_at: Marca de tiempo de creación
- nombre: Nombre del cliente (text)
- email: Correo electrónico (text)
- telefono: Número de teléfono (text)
- address: Dirección postal (text, nullable)
- lat: Latitud GPS (float8)
- lng: Longitud GPS (float8)
- logo_url: URL pública de la imagen en Supabase Storage (text)

Los registros mostrados fueron creados mediante la función insertClient() desde la aplicación móvil. Se puede observar cómo cada cliente tiene su logo almacenado en Storage y sus coordenadas GPS guardadas correctamente.

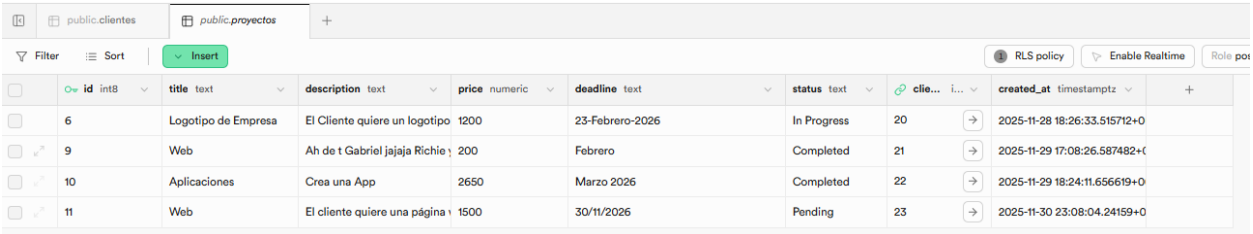


id int8	created_at timestamptz	nombre text	email text	telefono text	address text	lat float8	lng float8	logo_url text
20	2025-11-28 18:24:48.173782+00	Leandro Andrés	Leandro@gmail.com	+34611222333	NULL	28.125464293045	-15.4268680958445	https://vxqalqfhfkavtvaijkcn.su
21	2025-11-29 17:07:09.165683+00	Cliente 1	aridaneq@gmail.com	+34666555444	EMPTY	27.8518343	-15.4672476	https://vxqalqfhfkavtvaijkcn.su
22	2025-11-29 18:23:27.914869+00	Cliente 2	cliente2@mail.com	+34625625625	NULL	27.8518332	-15.4672468	https://vxqalqfhfkavtvaijkcn.su
23	2025-11-30 22:46:15.991707+00	Jose	jose@email.com	699588477	NULL	27.855538	-15.439025	https://vxqalqfhfkavtvaijkcn.su

Tabla proyectos en Supabase
Estructura de la tabla proyectos:

- id: Identificador único del proyecto (int8)
- title: Título del proyecto (text)
- description: Descripción detallada (text)
- price: Presupuesto del proyecto (numeric)
- deadline: Fecha límite de entrega (text)
- status: Estado del proyecto (text: Pending, In Progress, Completed)
- client_id: Clave foránea que referencia a clientes.id (int8)
- created_at: Fecha de creación

La columna client_id establece la relación: cada proyecto pertenece a un único cliente. Por ejemplo, el proyecto "Logotipo de Empresa" (id: 6) está asociado al cliente con id: 20 (Leandro Andrés). Esta relación está configurada con CASCADE DELETE, por lo que al eliminar un cliente se borran automáticamente todos sus proyectos.



id int8	title text	description text	price numeric	deadline text	status text	clie... i...	created_at timestamptz
6	Logotipo de Empresa	El Cliente quiere un logotipo	1200	23-Febrero-2026	In Progress	20	2025-11-28 18:26:33.515712+0
9	Web	Ah de t Gabriel jajaja Richie	200	Febrero	Completed	21	2025-11-29 17:08:26.587482+(
10	Aplicaciones	Crea una App	2650	Marzo 2026	Completed	22	2025-11-29 18:24:11.656619+0
11	Web	El cliente quiere una página	1500	30/11/2026	Pending	23	2025-11-30 23:08:04.24159+0

Tabla facturas en Supabase
Estructura de la tabla facturas:

- id: Identificador único de la factura (int8)
- number: Número de factura generado automáticamente (text, formato INV-XXXX)
- amount: Importe de la factura (numeric)
- date: Fecha de emisión (text)
- status: Estado de pago (text: Pending, Paid, Cancelled)
- project_id: Clave foránea que referencia a proyectos.id (int8)

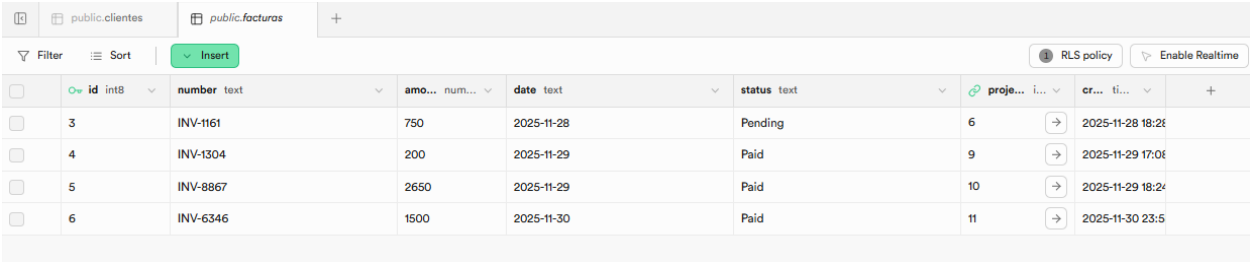
- created_at: Fecha de creación (timestamptz)

La columna project_id establece la relación: cada factura pertenece a un único proyecto. Por ejemplo, la factura INV-1161 (id: 3) está asociada al proyecto con id: 6 ("Logotipo de Empresa").

CADENA COMPLETA DE RELACIONES:
Cliente (id: 20) → Proyecto (id: 6) → Factura (id: 3)

Esto permite que al eliminar un cliente, se borren en cascada todos sus proyectos y, a su vez, todas las facturas de esos proyectos.

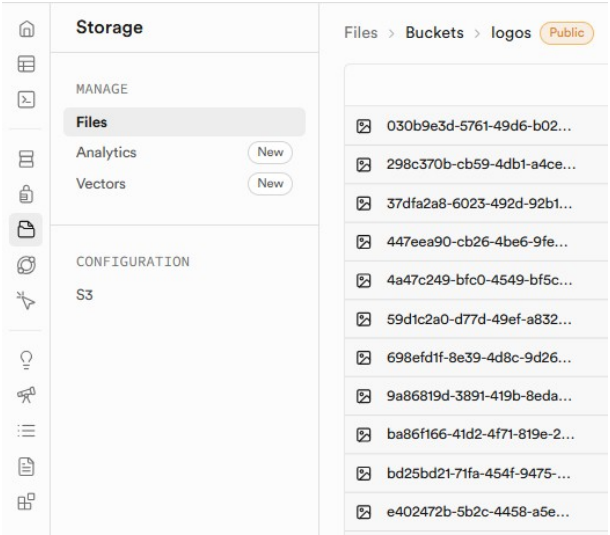
CÁLCULO DE INGRESOS:
El dashboard calcula el total de ingresos sumando únicamente las facturas con status: "Paid". En este ejemplo: 200 + 2650 + 1500 = 4350 €



Bucket logos en Supabase Storage

Configuración del Storage:

- Bucket: logos (público)
- Ubicación: /storage/v1/object/logos/
- Tipo de archivos: Imágenes (JPEG, PNG)
- Acceso: Público (cualquiera puede ver las imágenes mediante URL)
Funcionamiento:



Cuando un usuario crea o edita un cliente y selecciona una imagen (desde cámara o galería):

1. La función uploadImage() en api.ts crea un FormData con la imagen
2. Realiza una petición POST a Supabase Storage
3. Supabase guarda la imagen con un nombre único (hash)
4. Retorna la URL pública:
https://vxqalqfhfkavtvaijkcn.supabase.co/storage/v1/object/public/logos/{filename}
5. Esta URL se guarda en la columna logo_url de la tabla clientes

De esta forma, las imágenes no se almacenan directamente en la base de datos , sino que se guardan en Storage y solo se referencia su URL en la tabla.

4 Componentes Reutilizables

La aplicación se construye utilizando componentes reutilizables que se emplean en múltiples pantallas. Esto facilita el mantenimiento del código y garantiza consistencia visual en toda la aplicación.

COMPONENTE: Button

Botón personalizable con soporte para diferentes variantes visuales, estados de carga e iconos.

Propiedades (Props):

- title: Texto que se muestra en el botón

- onPress: Función que se ejecuta al presionar

- loading: Muestra un indicador de carga

- variant: Define el estilo (primary, secondary, danger)

- icon: Icono de Ionicons (opcional)

- disabled: Desactiva el botón

```tsx
interface ButtonProps {
  title: string;
  onPress: () => void;
  loading?: boolean;
  variant?: "primary" | "secondary" | "danger";
  icon?: keyof typeof Ionicons.glyphMap;
  style?: ViewStyle;
  disabled?: boolean;
}
```

Componente Button (implementación principal)

El componente Button implementa lógica condicional para cambiar colores según la variante y el estado (disabled, loading).

```tsx
export default function Button({
  title,
  onPress,
  loading,
  variant = "primary",
  icon,
  style,
  disabled,
}: ButtonProps) {
  const getBackgroundColor = () => {
    if (disabled) return "#E5E7EB";
    if (variant === "danger") return "#FEE2E2";
    if (variant === "secondary") return "#EFF6FF";
    return "#7C3AED"; // Primary (Morado)
  };

  const getTextColor = () => {
```

Componente ClientCard
Este componente utiliza renderizado condicional: si el cliente tiene logo_url, muestra la imagen; si no, muestra un icono genérico.

```tsx
export default function ClientCard({ client, onPress }: ClientCardProps) {
  return (
    <Pressable onPress={onPress} style={styles.card}>
      {/* FOTO */}
      {client.logo_url ? (
        <Image source={{ uri: client.logo_url }} style={styles.clientImage} />
      ) : (
        <View style={styles.iconBox}>
          <Ionicons name="business" size={24} color="#7C3AED" />
        </View>
      )}

      {/* TEXTOS */}
      <View style={{ flex: 1 }}>
        <Text style={styles.cardTitle}>{client.nombre}</Text>

        {/* Teléfono Formateado */}
        <Text style={styles.cardSubtitle}>{formatPhone(client.telefono)}</Text>

        <Text style={styles.cardEmail}>{client.email || "No email"}</Text>
      </View>

      <Ionicons name="chevron-forward" size={20} color="#b6b7b9ff" />
    </Pressable>
  );
}
```

Componente Input

El componente hereda todas las propiedades de TextInput mediante el spread operator (...props), lo que permite utilizarlo como un TextInput normal pero con estilos consistentes.

```tsx
interface InputProps extends TextInputProps {
  label: string;
  error?: string;
}

export default function Input({ label, style, ...props }: InputProps) {
  return (
    <View style={styles.container}>
      <Text style={styles.label}>{label}</Text>
      <TextInput
        style={[styles.input, style]}
        placeholderTextColor="#9CA3AF"
        {...props}
      />
    </View>
  );
}
```

Lógica de estado del proyecto

Esta función retorna un porcentaje del estado del proyecto, lo que permite mostrar visualmente el progreso.

```
export default function ProjectCard({ project, onPress }: ProjectCardProps) {
  const getStatusColor = (status: string) => {
    if (status === "Completed")
      return { bg: "#D1FAE5", text: "#059669", fill: "#10B981", percent: 100 };
    if (status === "In Progress")
      return { bg: "#DBEAFE", text: "#1E40AF", fill: "#3B82F6", percent: 50 };
    return { bg: "#FEF3C7", text: "#92400E", fill: "#F59E0B", percent: 0 }; // Pending
  };
```

Componente InvoiceCard
Se utiliza una constante isPaid para aplicar estilos condicionales: verde si está pagada, naranja si está pendiente.

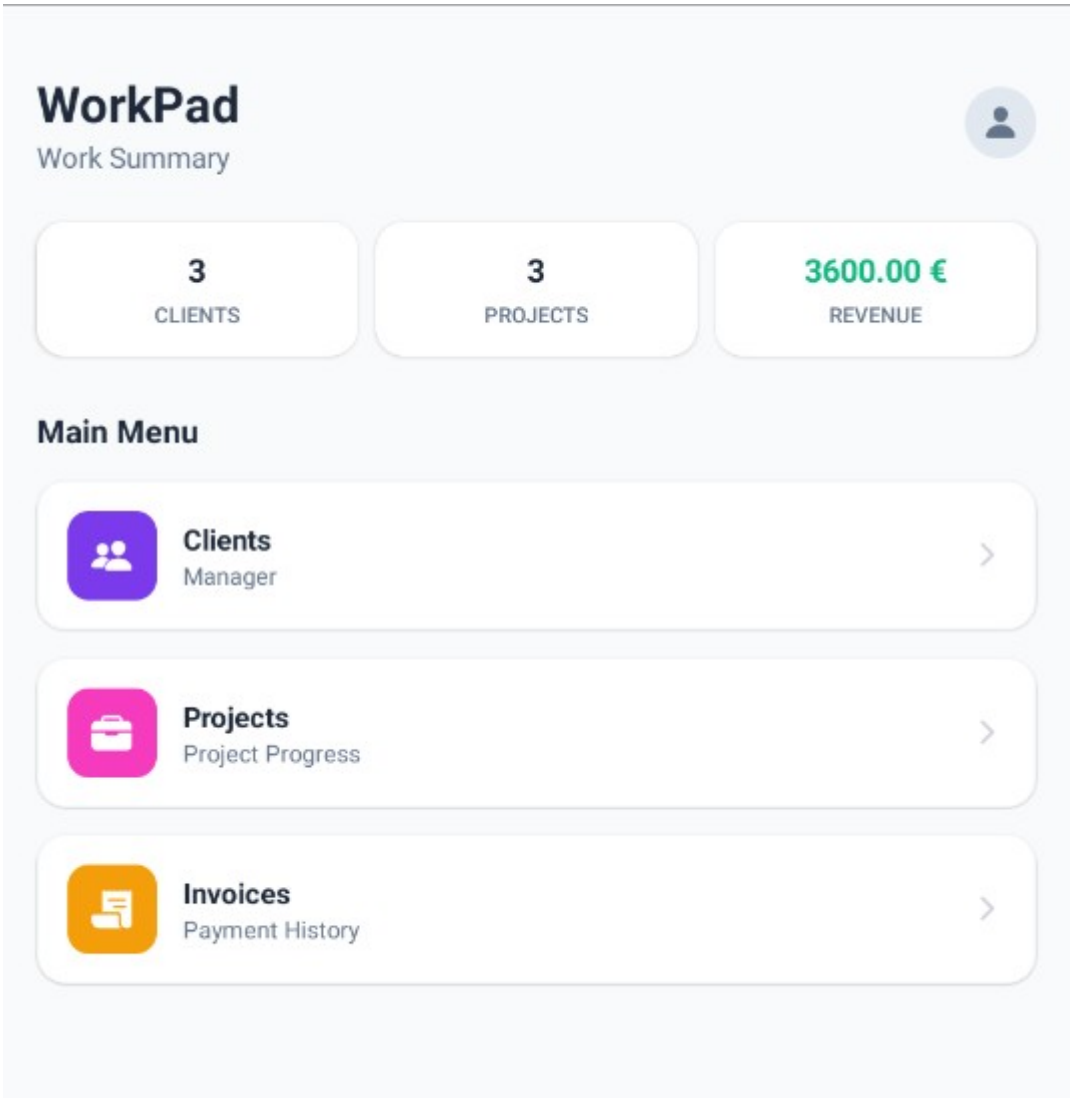
```
export default function InvoiceCard({ invoice, onPress }: InvoiceCardProps) {
  const isPaid = invoice.status === "Paid";

  return (
    <Pressable onPress={onPress} style={styles.card}>
      <View style={styles.leftCol}>
        <Text style={styles.number}>{invoice.number}</Text>
        <Text style={styles.date}>{invoice.date}</Text>
      </View>

      <View style={styles.rightCol}>
        <Text style={styles.amount}>{invoice.amount} €</Text>
        <Text
          style={[styles.status, isPaid ? styles.textGreen : styles.textOrange]}
        >
          {invoice.status}
        </Text>
      </View>

      <Ionicons
        name="chevron-forward"
        size={16}
        color="#D1D5DB"
        style={{ marginLeft: 8 }}
      />
    </Pressable>
  );
}
```

5 Funcionalidades y Casos de Uso

En esta sección se presentan las principales funcionalidades de la aplicación con ejemplos visuales de su funcionamiento.

CASO DE USO 1: DASHBOARD PRINCIPAL

Al iniciar la aplicación, se muestra un dashboard con información resumida:



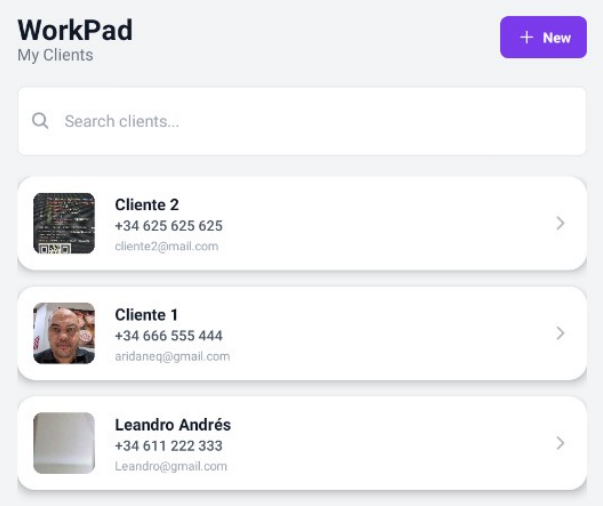
Elementos visuales:

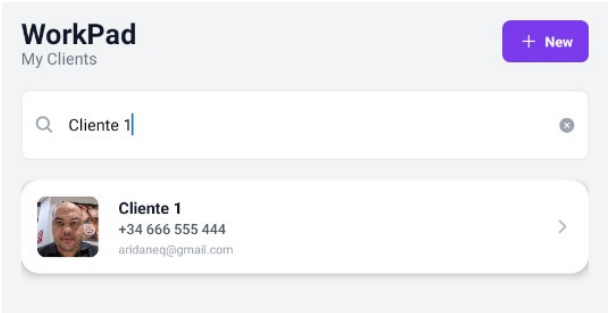
- Total de ingresos calculado sumando las facturas con estado "Paid"
- Contadores de clientes, proyectos y facturas totales
- Botones de navegación rápida a cada sección
- Actualización automática al volver a la pantalla mediante useFocusEffect

Dashboard con useFocusEffect
El hook useFocusEffect se ejecuta cada vez que se navega a esta pantalla, asegurando que los datos estén actualizados.

```
export default function Dashboard() {
  const [stats, setStats] = useState({
    clients: 0,
    projects: 0,
    revenue: "0 €",
  });
  const [loading, setLoading] = useState(true);
```

CASO DE USO 2: GESTIÓN DE CLIENTES
Listar clientes con búsqueda. En la pantalla de clientes permite:
- Ver todos los clientes en formato de tarjetas
- Buscar en tiempo real por nombre o email
- Botón para crear nuevo cliente

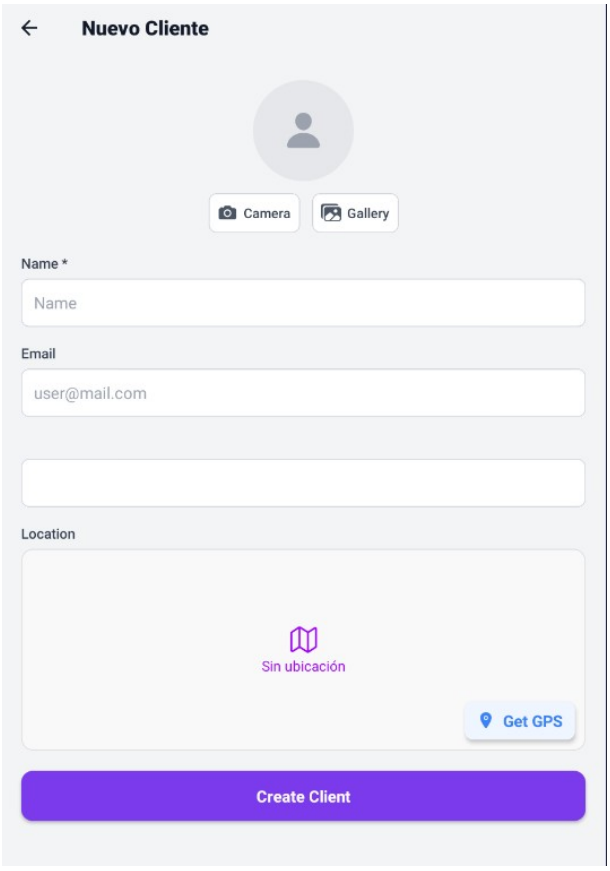


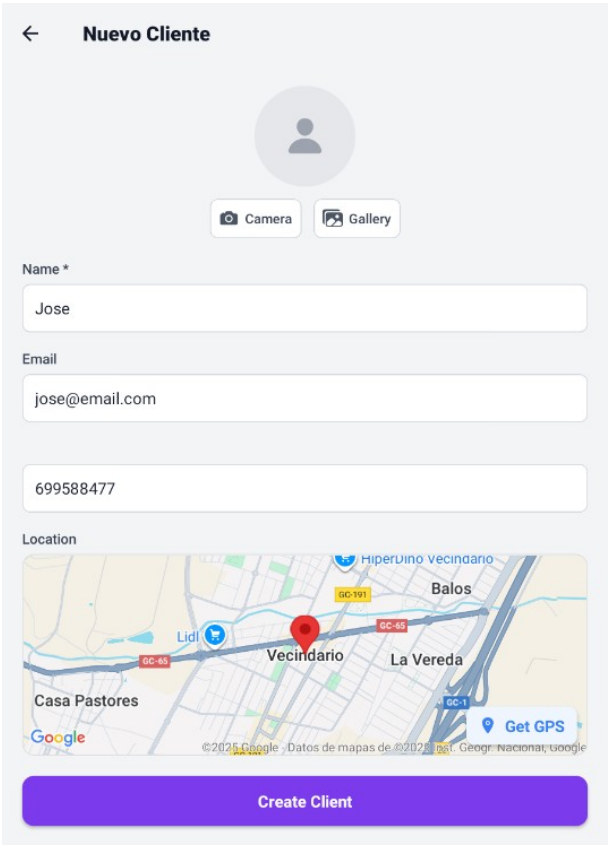


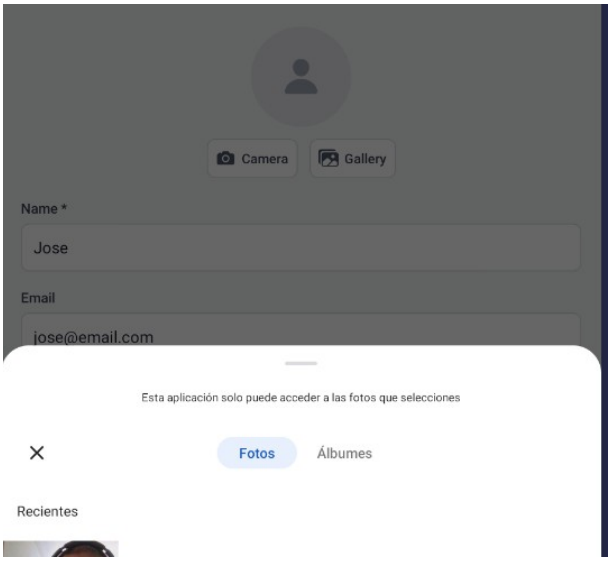
Búsqueda de clientes
implementacion;

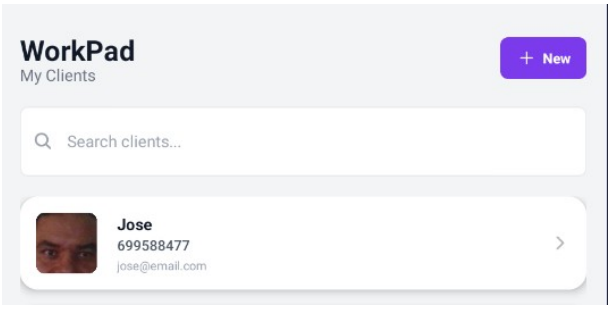
```
const filteredClients = clients.filter((client) => {
  const term = searchText.toLowerCase();

  const name = (client.nombre || "").toLowerCase();
  const email = (client.email || "").toLowerCase();

  return name.includes(term) || email.includes(term);
});
```

Caso de uso 3: Crear nuevo cliente
Proceso de creación:
1. Se completan los campos: nombre, email, teléfono
2. Se selecciona una imagen desde cámara o galería
3. Se elige la ubicación en el mapa tocando sobre él
4. Al guardar, se sube la imagen a Supabase Storage
5. Se crea el registro en la base de datos









Selección de imagen:

Se implementa la función pickImage que permite al usuario seleccionar una imagen desde la galería del dispositivo utilizando expo-image-picker. Esta función se invoca al presionar el botón de selección de imagen en el formulario de creación de cliente.

```
const pickImage = async (mode: "camera" | "gallery") => {
  const { status } =
    mode === "camera"
      ? await ImagePicker.requestCameraPermissionsAsync()
      : await ImagePicker.requestMediaLibraryPermissionsAsync();

  if (status !== "granted") {
    Alert.alert(
      "Permiso denegado",
      "Necesitamos acceso a la cámara/galería."
    );
    return;
  }

  let result =
    mode === "camera"
      ? await ImagePicker.launchCameraAsync({
          allowsEditing: true,
          aspect: [1, 1],
          quality: 0.5,
        })
      : await ImagePicker.launchImageLibraryAsync({
          allowsEditing: true,
          aspect: [1, 1],
          quality: 0.5,
        });
```

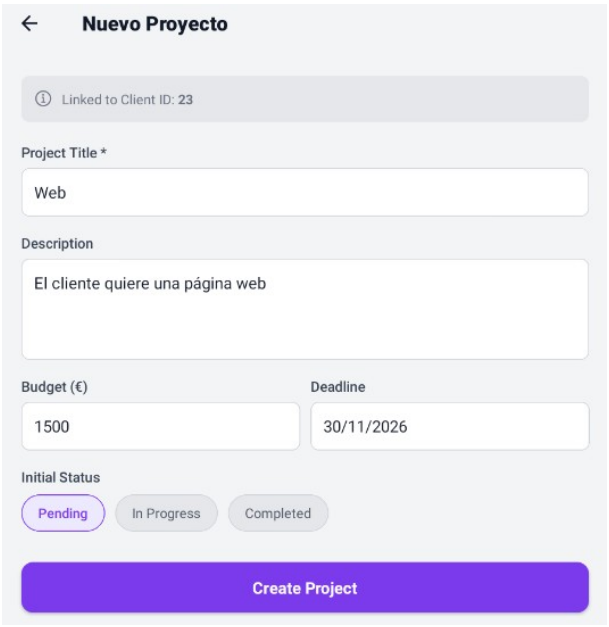
Renderizado del mapa

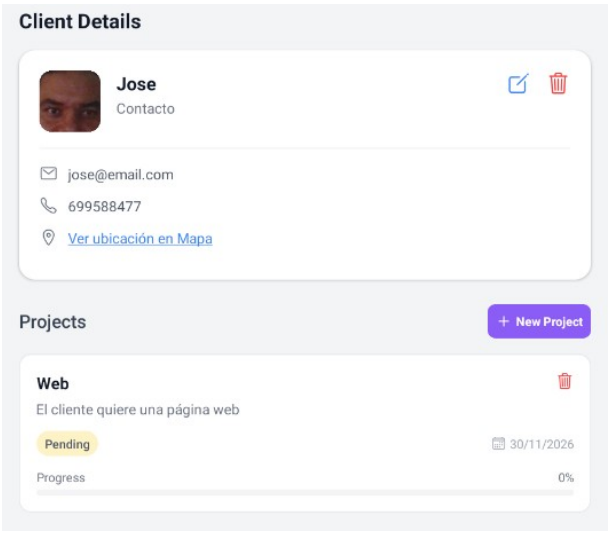
Se implementa el renderizado condicional del mapa en el formulario de creación de cliente. Si existen coordenadas guardadas (lat y lng), se muestra el MapView con un marcador; si no, se muestra un placeholder indicando que no hay ubicación. Incluye un botón flotante para obtener la ubicación GPS actual.

```
<Text style={styles.labelMap}>Location</Text>
<View style={styles.mapContainer}>
  {form.lat && form.lng ? (
    <MapView
      style={styles.map}
      region={{
        latitude: form.lat,
        longitude: form.lng,
        latitudeDelta: 0.01,
        longitudeDelta: 0.01,
      }}
    >
      <Marker
        coordinate={{ latitude: form.lat, longitude: form.lng }}
        title="Nuevo Cliente"
      />
    </MapView>
  ) : (
    <View style={styles.noMap}>
      <Ionicons name="map-outline" size={30} color="#9f03faff" />
      <Text style={{ color: "#9f03faff" }}>Sin ubicación</Text>
    </View>
  )}
```

Caso de Uso 4: Creación de un proyecto

El proceso de creación de un proyecto permite asociar un nuevo trabajo a un cliente existente. El usuario debe completar un formulario con título, descripción, precio, fecha límite y estado inicial. Al guardar, se validan los datos, se convierten los tipos necesarios y se envía la información a la API. Si la operación es exitosa, se navega automáticamente a la pantalla anterior.





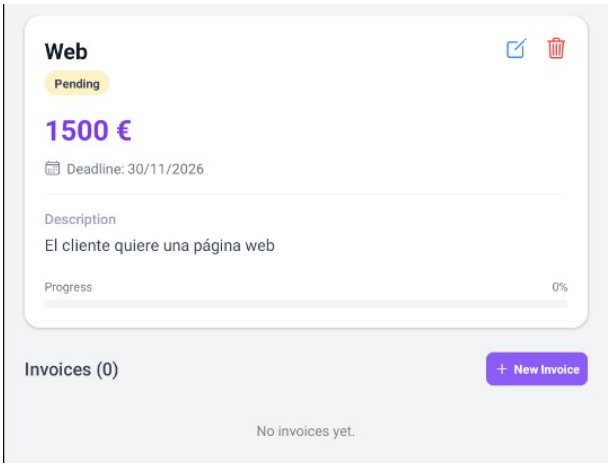
Creación de proyecto

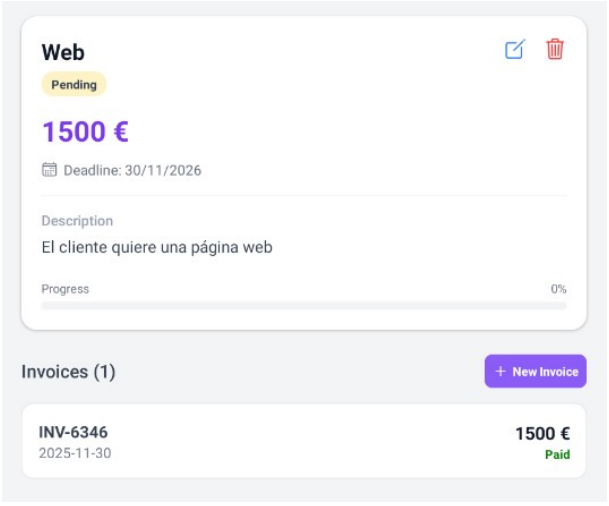
La función handleSave gestiona el proceso completo de creación:

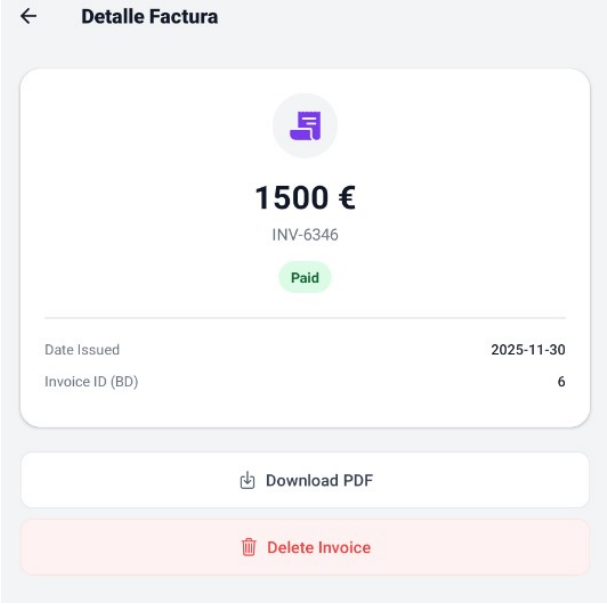
1. Valida que el título del proyecto no esté vacío (campo obligatorio)
2. Si falta el título, muestra un Alert y detiene la ejecución
3. Activa el estado de carga (setSaving(true)) para deshabilitar el botón
4. Envía el formulario completo a la función insertProject de la API
5. Desactiva el estado de carga
6. Si la operación es exitosa, muestra un Alert de confirmación y navega hacia atrás
7. Si falla, muestra un Alert de error

```
const handleSave = async () => {
  if (!form.title) {
    Alert.alert("Error", "Please enter a project title");
    return;
  }

  setSaving(true);
  const success = await insertProject(form);
  setSaving(false);

  if (success) {
    Alert.alert("Success", "Project created successfully!", [
      { text: "OK", onPress: () => router.back() },
    ]);
  } else {
    Alert.alert("Error", "Could not save project");
  }
};
```

Caso de uso 5: Crear facturas:
El proceso de creación de facturas permite generar documentos de cobro asociados a un proyecto específico. El sistema genera automáticamente un número de factura único, establece la fecha actual por defecto y permite al usuario configurar el importe y el estado. Se validan los datos antes de guardar, asegurando que el importe sea un número válido.







El proceso de validación y guardado se divide en dos funciones:
validateForm():
- Verifica que el campo amount no esté vacío
- Comprueba que el importe sea un número válido usando isNaN()
- Muestra alertas específicas para cada tipo de error
- Retorna true solo si todas las validaciones pasan

handleSave():
- Ejecuta validateForm() y detiene el proceso si la validación falla
- Activa el estado de carga para deshabilitar el botón

- Llama a insertInvoice() con los datos del formulario
- Muestra un Alert de confirmación si es exitoso y navega hacia atrás
- Muestra un Alert de error si la operación falla

```
const validateForm = () => {
  if (!form.amount) {
    Alert.alert("Error", "Debes poner un importe (€).");
    return false;
  }
  if (isNaN(Number(form.amount))) {
    Alert.alert(
      "Error",
      "El importe debe ser un número (usa punto para decimales)."
    );
    return false;
  }
  return true;
};

const handleSave = async () => {
  if (!validateForm()) return;

  setSaving(true);
  const success = await insertInvoice(form);
  setSaving(false);

  if (success) {
    Alert.alert("Éxito", "Factura generada", [
      { text: "OK", onPress: () => router.back() },
    ]);
  } else {
    Alert.alert("Error", "No se pudo guardar la factura");
  }
};
```

Generación de número de factura:

Se genera automáticamente un número único con formato INV-XXXX donde XXXX es un número aleatorio entre 1000 y 9999.

number: `INV-${Math.floor(1000 + Math.random() * 9000)}`

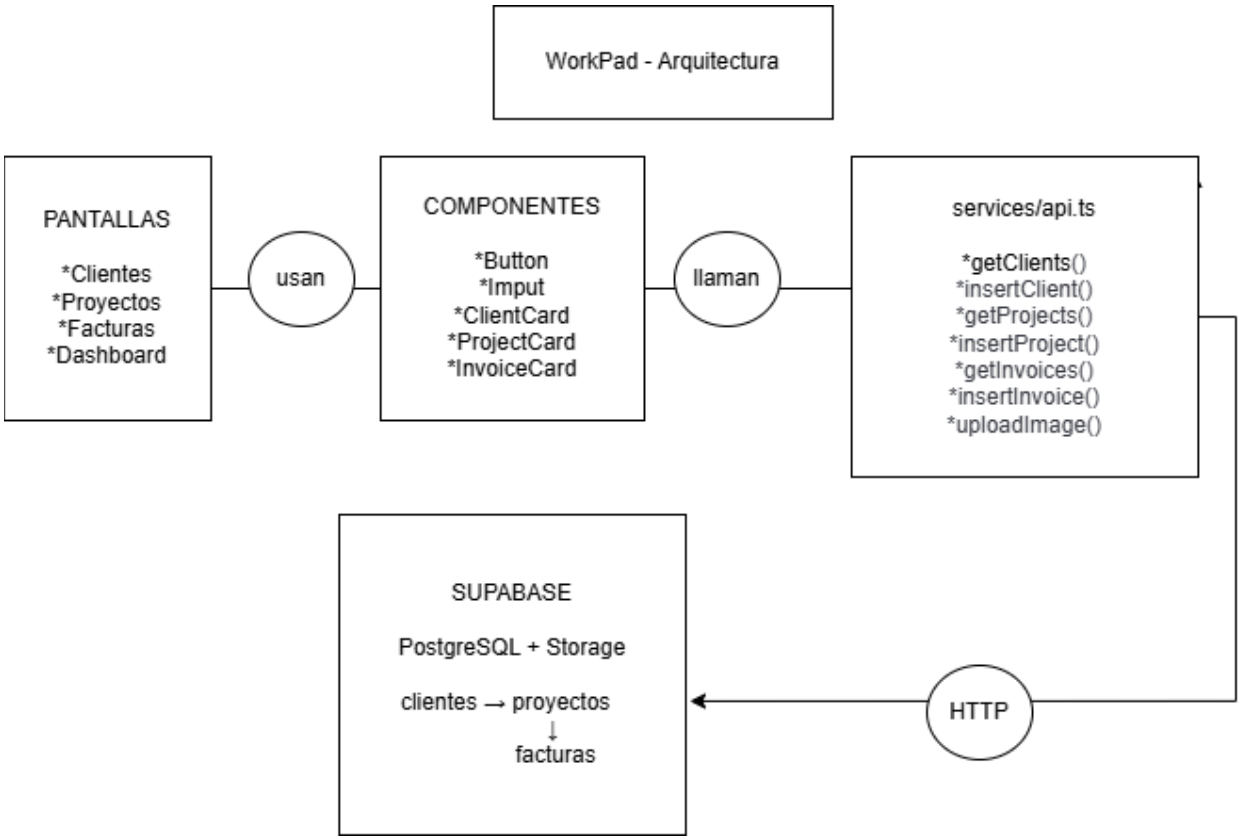
Fecha Automática:

Se establece automáticamente la fecha actual en formato ISO (YYYY-MM-DD).

date: new Date().toISOString().split("T")[0]

6 Arquitectura General y Diagrama de Componentes

La aplicación WorkPad se estructura siguiendo un patrón de arquitectura en capas que separa claramente las responsabilidades entre presentación, lógica de negocio y datos. Este diseño facilita el mantenimiento, permite la reutilización de código y asegura una comunicación eficiente con el backend. El siguiente diagrama muestra la arquitectura completa de la aplicación y las relaciones entre sus diferentes capas:



1. CAPA DE PRESENTACIÓN – PANTALLAS

Contiene todos los módulos de vistas organizados por funcionalidad:

• Clientes: Gestión completa de clientes (lista, detalle, crear, editar)

• Proyectos: Administración de proyectos vinculados a clientes

• Facturas: Control de facturación asociada a proyectos

• Dashboard: Pantalla principal con métricas y resumen de ingresos

Cada módulo implementa operaciones CRUD completas siguiendo el patrón de Expo Router con navegación basada en archivos.

2. CAPA DE COMPONENTES REUTILIZABLES

Librería de componentes UI compartidos que garantizan consistencia visual:

• Button: Botón con variantes (primary, secondary, danger) y estados de carga

• Input: Campo de entrada con etiqueta y validación integrada

• ClientCard: Tarjeta para visualizar información de clientes en listas

• ProjectCard: Tarjeta con badge de estado y barra de progreso

• InvoiceCard: Tarjeta con código de colores según estado de pago

Relación: Las pantallas USAN estos componentes para construir la interfaz.

3. CAPA DE SERVICIOS – services/api.ts

Centraliza toda la comunicación con el backend mediante funciones específicas:

API de Clientes:

• getClients() - Obtener lista completa

• insertClient() - Crear nuevo cliente

• updateClient() - Modificar datos

• deleteClient() - Eliminar (cascada)

API de Proyectos:

• getProjects() - Listar proyectos

• insertProject() - Crear proyecto

• updateProject() - Actualizar proyecto

API de Facturas:

• getInvoices() - Obtener facturas

• insertInvoice() - Generar factura

• updateInvoice() - Modificar factura

Utilidades:

• uploadImage() - Subir imágenes a Supabase Storage

Relación: Los componentes LLAMAN a estas funciones para obtener/modificar datos.

4. CAPA DE DATOS – SUPABASE

Backend as a Service que proporciona:

• PostgreSQL: Base de datos relacional con tres tablas

- clientes (tabla principal)

- proyectos (vinculada a clientes mediante client_id)

- facturas (vinculada a proyectos mediante project_id)

• Storage: Almacenamiento en la nube para imágenes (bucket "logos")

• API REST: Generada automáticamente por Supabase

Relaciones entre tablas:

- clientes (1) → proyectos (N) con CASCADE DELETE

- proyectos (1) → facturas (N) con CASCADE DELETE

Relación: services/api.ts se comunica mediante HTTP/REST con Supabase.

FLUJO DE DATOS EN LA APLICACIÓN

Un ejemplo típico del flujo de datos sería:

1. Usuario accede a la pantalla "Nuevo Cliente"

2. La pantalla renderiza componentes Input y Button

3. Usuario completa el formulario y pulsa "Guardar"

4. La pantalla llama a insertClient() de services/api.ts

5. api.ts realiza petición HTTP POST a Supabase

6. Supabase inserta el registro en la tabla "clientes"

7. Retorna confirmación a api.ts

8. La pantalla recibe el resultado y navega hacia atrás

Este patrón se repite para todas las operaciones CRUD en la aplicación.

VENTAJAS DE ESTA ARQUITECTURA

- Separación de responsabilidades: cada capa tiene una función específica

- Reutilización de código: componentes compartidos en múltiples pantallas

- Mantenibilidad: cambios centralizados en servicios afectan a toda la app

- Escalabilidad: fácil añadir nuevos módulos siguiendo el mismo patrón

- Testabilidad: cada capa puede probarse de forma independiente

7 Conclusiones

LO QUE HE APRENDIDO

Durante el desarrollo de WorkPad he aprendido a trabajar con tecnologías que no conocía:

- React Native y Expo para crear aplicaciones móviles multiplataforma

- TypeScript para tener código más seguro y con menos errores

- Supabase como backend en la nube (base de datos y almacenamiento)

- Integración de funcionalidades nativas: cámara, galería y mapas GPS

- Arquitectura en capas separando vistas, lógica y datos

Lo más valioso ha sido entender cómo estructurar una aplicación completa: desde diseñar la base de datos con sus relaciones hasta crear la interfaz de usuario. Ver cómo un cliente puede tener proyectos y estos tener facturas, todo conectado, ha sido muy satisfactorio.

También he comprendido la importancia de crear componentes reutilizables. Crear Button, Input y las Cards me ahorró mucho tiempo y hace el código más limpio.

DIFICULTADES ENCONTRADAS

Los principales problemas que tuve fueron:

- Subir imágenes a Supabase: no formateaba bien el FormData.

- Expo Router: es diferente a React Navigation tradicional. Me costó entender la navegación por carpetas y el paso de parámetros.

- TypeScript: muchos errores de tipos hasta que definí bien todas las interfaces.

- CASCADE DELETE: tuve que probar varias veces en Supabase hasta configurarlo correctamente.

QUÉ MEJORARÍA

Si tuviera más tiempo añadiría:

- Sistema de autenticación de usuarios

- Modo offline con sincronización

- Exportar facturas a PDF

- Gráficos de evolución de ingresos

- Notificaciones para fechas límite

También mejoraría aspectos técnicos como validaciones más robustas, mejor manejo de errores y añadir tests.

VALORACIÓN FINAL

Estoy satisfecho con el resultado. La aplicación cumple todos los requisitos del módulo y funciona correctamente en dispositivos reales.

Lo mejor del proyecto ha sido ver cómo todas las partes se conectan y funcionan juntas. Lo más complicado fue la curva de aprendizaje inicial con Expo Router y Supabase.

Este proyecto me ha enseñado la importancia de planificar antes de programar y leer bien la documentación oficial. WorkPad es una base sólida que podría evolucionar a una aplicación de producción real.

8 Referencias

- Android Studio

https://developer.android.com/studio

Emulador de Android para pruebas

- ChatGPT

https://chat.openai.com/

Asistente de inteligencia artificial utilizado para consultas y resolución de dudas técnicas

- Claude

https://claude.ai/

Asistente de IA utilizado para apoyo en desarrollo y documentación

- **Expo Documentation**

https://docs.expo.dev/

Documentación oficial de Expo para desarrollo de aplicaciones React Native

- **Expo Go**

https://expo.dev/client

Aplicación móvil para testing en dispositivo físico

- **Expo Image Picker**

https://docs.expo.dev/versions/latest/sdk/imagepicker/

Documentación para acceso a cámara y galería

- **Expo Location**

https://docs.expo.dev/versions/latest/sdk/location/

Documentación para geolocalización

- **Expo Router Documentation**

https://docs.expo.dev/router/introduction/

Guía de navegación basada en archivos con Expo Router

- **GitHub Issues - Expo**

https://github.com/expo/expo/issues

Solución de problemas específicos de Expo

- **React Native Documentation**

https://reactnative.dev/docs/getting-started

Guía oficial de React Native

- **React Native Maps**

https://github.com/react-native-maps/react-native-maps

Librería para integración de mapas

- **React Navigation**

https://reactnavigation.org/docs/getting-started

Documentación de React Navigation

- **Stack Overflow**

https://stackoverflow.com/

Resolución de errores y consultas técnicas

- **Supabase Community**

https://github.com/supabase/supabase/discussions

Comunidad y ejemplos de uso de Supabase

- **Supabase Dashboard**

  https://app.supabase.com/

  Panel de administración de la base de datos

- **Supabase Documentation**

  https://supabase.com/docs

  Documentación de Supabase para base de datos y almacenamiento

- **TypeScript Handbook**

  https://www.typescriptlang.org/docs/

  Manual oficial de TypeScript

- **Visual Studio Code**

  https://code.visualstudio.com/

  Editor de código utilizado en el desarrollo

Historial de revisiones

Fecha	Descripción	Autor
30/11/2025	Versión inicial	Aridane Quevedo Cabrera