# DOCUMENTO TÉCNICO: UNVEIL UGC PLATFORM


> De: Pablo Candela
> Contacto:
> candelappablo@gmail.com / +54 9 3364 54-0036


## 1. ARQUITECTURA UTILIZADA

**Patrón:** Clean Architecture + Context API

**Descripción:**

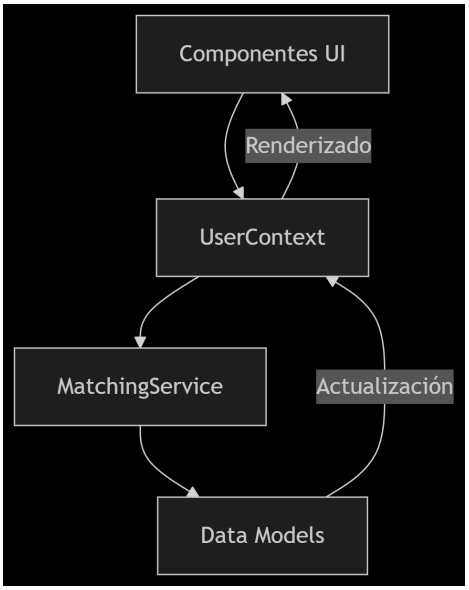
La aplicación se organiza en capas claramente separadas:

- **Modelos:** Definidos en `src/models` (User, Offer, Campaign)
- **Servicios:** Lógica de negocio en `src/services` (matchingService, mockData)
- **Componentes:** Elementos UI reutilizables en `src/components`
- **Pantallas:** Vistas principales en `src/screens`

El estado global se gestiona mediante Context API para:

- Datos de usuario
- Progreso de onboarding
- Autenticación

**Diagrama:**




**Justificación:**

- Escalabilidad para añadir nuevas features
- Fácil mantenimiento del código
- Testing independiente por capas

---

## 2. DECISIONES TÉCNICAS CLAVE

| Área | Decisión | Justificación Técnica |
|---|---|---|
| **Gestión Estado** | Context API | Ideal para complejidad media y flujos centralizados |
| **Navegación** | React Navigation v6 | Soporte para anidación y transiciones personalizadas |
| **Mock Data** | Servicios locales | Simula comportamiento real de API con latencia controlada |
| **Validación** | Yup + Lógica Custom | Combina validación estándar con reglas específicas |
| **UI Components** | Biblioteca reutilizable | Reduce duplicación y asegura consistencia |

---

## 3. Flujo de datos del Onboarding: estructura y lógica

### 1. Estructura general

El onboarding de tu app está compuesto por varias pantallas (steps), cada una encargada de recolectar una parte de la información del usuario. Ejemplo típico de pasos:

- Step 1: Datos personales
- Step 2: Redes sociales
- Step 3: Intereses
- Step 4: Confirmación

Cada pantalla es un componente independiente, pero todas comparten y actualizan un **estado centralizado** que vive en el contexto global de usuario ( `UserContext` ).

---

### 2. ¿Cómo se almacena la información?

- Existe un objeto `onboardingData` en el contexto, con una estructura tipo:

```
onboardingData = {
  step1: { ... },
  step2: { ... },
  step3: { ... },
  // etc
}
```

- Cada pantalla del onboarding solo se encarga de llenar/modificar su parte correspondiente.

---

### 3. Lógica central: `updateOnboardingData`

Esta función es la clave para mantener la información **acumulada, organizada y actualizable** paso a paso.

```
typescriptCopyInsertconst updateOnboardingData = (step: keyof OnboardingData, data: any) => {
if (onboardingData) {
  setOnboardingData({
    ...onboardingData,
    [step]: {
```

```
      ...onboardingData[step],
      ...data,
    },
  });
 }
};
```

**¿Cómo funciona?**

1. **Recibe el nombre del paso** (`step`, por ejemplo `"step2"`) y los nuevos datos de ese paso (`data`).
2. **Verifica** que ya exista un objeto `onboardingData` (es decir, que hay datos previos).
3. **Actualiza solo el paso correspondiente**:
   - Usa el spread operator para copiar todo el estado anterior.
   - Para el paso que corresponde (`[step]`), también hace un spread de lo que ya había y lo sobrescribe/agrega con los nuevos datos.
   - Así, si el usuario vuelve atrás y edita algo, solo se modifica ese paso, sin perder lo demás.
4. **Ejemplo visual**:
   - Antes:

     ```
     js
     CopyInsert
     onboardingData = {
       step1: { nombre: "Ana" },
       step2: { instagram: "@ana" }
     }
     ```

   - Llamada:

     ```
     pdateOnboardingData("step2", { tiktok: "@anaTikTok" })
     ```

   - ◦ Después:

     ```
     onboardingData = {
       step1: { nombre: "Ana" },
       step2: { instagram: "@ana", tiktok: "@anaTikTok" }
     }
     ```

**Ventajas de este enfoque**

- **Inmutabilidad:** No se modifica el estado anterior, siempre se crea uno nuevo.
- **Escalabilidad:** Puedes agregar más pasos sin cambiar la lógica.
- **Flexibilidad:** Permite que el usuario navegue hacia adelante y atrás, editando cualquier paso.

---

**4. Resumen del flujo completo**

1. **Cada pantalla** del onboarding muestra un formulario y, al avanzar, llama a `updateOnboardingData` con los datos de ese paso.
2. **El contexto** guarda y acumula todos los datos.

3.
**Al finalizar**, se consolidan todos los datos y se guarda el usuario registrado en el contexto global.
4.
**El resto de la app** puede acceder a los datos completos del usuario desde el contexto.

---

**5. Visual del flujo**

```
CopyInsertStep1Screen → updateOnboardingData("step1", datosStep1)
      ↓
Step2Screen → updateOnboardingData("step2", datosStep2)
      ↓
Step3Screen → updateOnboardingData("step3", datosStep3)
      ↓
Step4Screen (confirmación/finalización)
      ↓
UserContext (usuario registrado y listo para usar en toda la app)
```

---

# 4. SISTEMA DE MATCHING

El sistema de matching en la app conecta ofertas (campañas de negocios) con creadores de contenido (usuarios) según varios criterios de compatibilidad y relevancia. Aquí te explico cómo funciona:

---

# 1. ¿Dónde está la lógica?

Toda la lógica está en el archivo:
`CopyInsertsrc/services/matchingService.ts`

Las funciones principales son:

- `getMatchingOffersForUser(user)`
- `getMatchingOffersWithDetailsForUser(user)`

---

# 2. ¿Cómo se realiza el matching?

**a) Filtrado inicial**

- Se descartan ofertas de negocios con los que el usuario ya completó o canceló campañas, para evitar repeticiones.

**b) Cálculo de score (puntuación de compatibilidad)**

Cada oferta recibe un puntaje basado en varios factores. Los criterios y sus pesos aproximados son:

- **Nivel del creador** (25%): Si el usuario cumple o supera el nivel requerido por la oferta.
- **Intereses** (20%): Si los intereses seleccionados por el usuario coinciden con la categoría de la oferta.
- **Plataformas requeridas** (15%): Si el usuario usa las redes/plataformas que la oferta solicita (Instagram, TikTok, etc).
- **Tipo de contenido** (15%): Si el usuario crea el tipo de contenido que la oferta necesita (story, reel, video, etc).
- **Ubicación** (10%): Si el usuario está en el país o ciudad que la oferta requiere.

- **Asistencia a evento** (5%): Si la oferta requiere presencia física y el usuario está en la ciudad indicada.
- **Exclusividad** (5%): Si la oferta es exclusiva (por ahora suma siempre).

**c) Orden y presentación**

- Se suman los puntos de cada criterio.
- Las ofertas se ordenan de mayor a menor score.
- Solo se muestran las ofertas activas y relevantes para el usuario.

---

## 3. ¿Qué función usar?

- `getMatchingOffersForUser(user)` devuelve solo el array de ofertas ordenadas.
- `getMatchingOffersWithDetailsForUser(user)` devuelve además el score y un array con los criterios que hicieron match, útil para mostrar detalles al usuario.

---

## 4. Ejemplo de criterios evaluados

Supón que una oferta pide:

- Nivel: Intermedio
- Categoría: Belleza
- Plataformas: Instagram, TikTok
- Ciudad: Madrid

Si el usuario es de nivel avanzado, le interesa Belleza, usa Instagram y vive en Madrid, recibirá el máximo score posible.

---

## 5. USO DE HERRAMIENTAS DE IA

| Herramienta | Aplicación | Resultado Obtenido |
|---|---|---|
| **ChatGPT & DeepSeek** | - Consultas rápidas sobre tecnologías nuevas (React Navigation, Context API)<br>- Generación de ejemplos básicos de código para componentes- Explicación de conceptos técnicos en lenguaje sencillo | - Aprendí a implementar navegación entre pantallas en muy poco tiempo<br>- Ejemplos listos para copiar/pegar y adaptar, evitando "bloqueos técnicos" |
| | - Redacción de documentación técnica<br>- Organización de secciones del README | - Documentación clara y profesional sin perder tiempo en formato<br>- Textos explicativos listos en minutos |
| **Claude** | - Optimización de funciones complejas<br>- Simplificación de lógica redundante<br>- Limpieza de código "espagueti" | - Código más legible y mantenible<br>- Eliminación de código residual e innecesario. |

---

**Impacto en el Proyecto:**

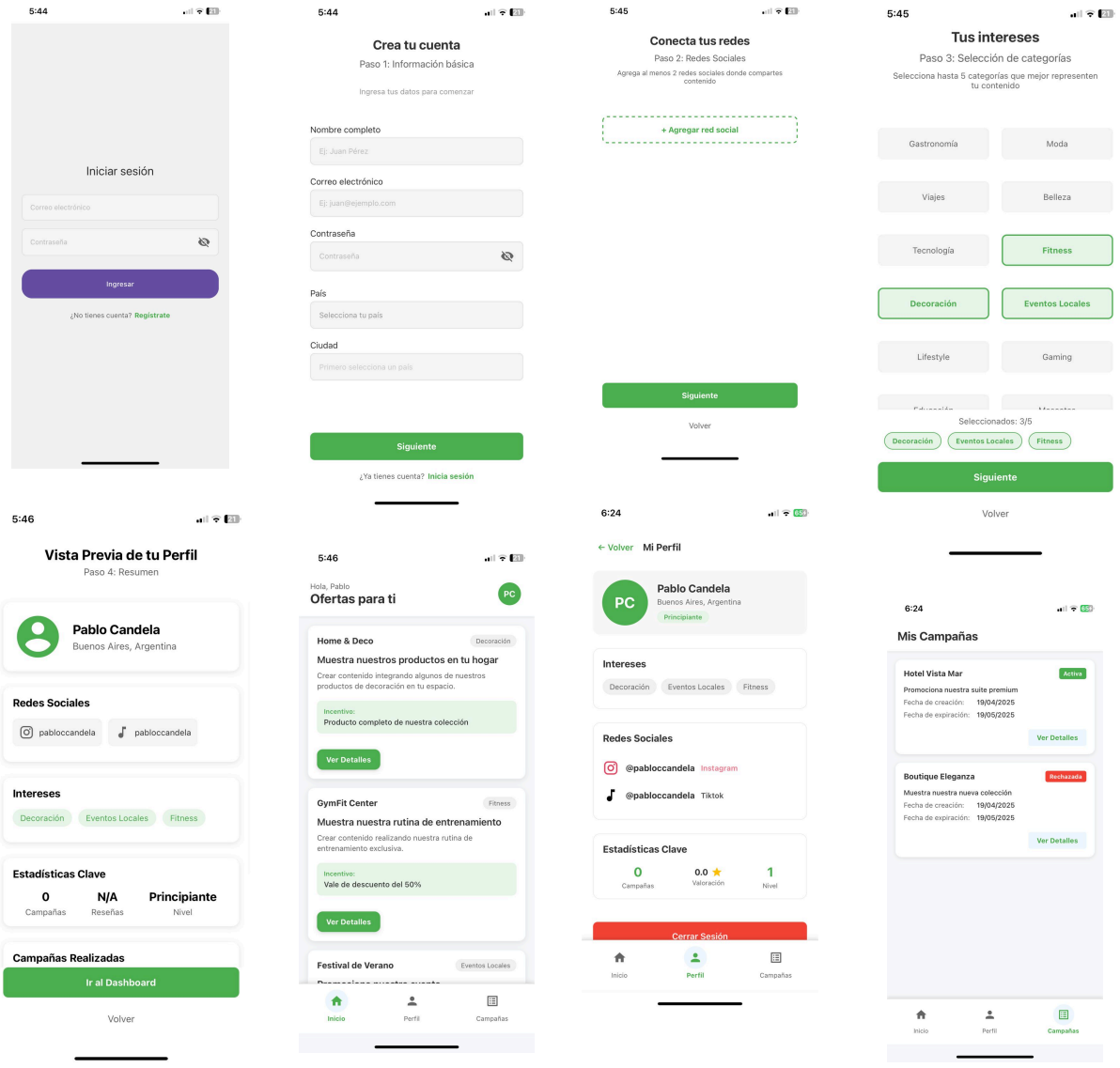
En resumen el uso de la IA me señaló partes donde usaba soluciones complicadas y propuso alternativas simples. También Sugirió buenas prácticas que desconocía (ej: cómo estructurar los

contextos para evitar renders innecesarios).Y me ahorro tiempo en la documentación lo que me permitió enfocarme en codificar.

---

# 6. Visuales Principales

Estas son algunas de las visuales principales de Unveil App




---

# 7. APRENDIZAJES CLAVE

## 7.1. Integración de Patrones Arquitectónicos

Combinación efectiva para mantener separación de responsabilidades mientras se maneja estado global.

- **Clean Architecture + Context API**:CopyDownload

plaintext

```
UI → Context (Middleware) → Servicios → Modelos
```

- **Patrón Observer**:

  Suscripción automática de componentes a cambios específicos del contexto.

## 7.2. Desarrollo Ágil con React Native

- **Logro Principal**:

  Construcción completa de app móvil funcional en **<7 días** usando:

  - Expo para build rápido
  - TypeScript para calidad de código
  - React Navigation para UX profesional

**Habilidades Demostradas:**

```
| Frontend Mobile | Gestión de Estado | Integración de API | Debugging Avanzado |
|-----------------|-------------------|--------------------|--------------------|
| React Native    | Context API       | Mock Services      | React DevTools     |
```

## 7.3. Impacto Profesional

- Capacidad para entregar soluciones técnicas robustas bajo plazos ajustados
- Aprendizaje rápido y adaptabilidad
- Dominio práctico de arquitecturas escalables
- Habilidad para diagnosticar y resolver problemas complejos de sincronización estado/UI

---