Decisiones arquitectónicas del proyecto PAE

ADR: Architecture Decision Records

Documento que registra las decisiones de arquitectura clave y su justificación.

ADR-001: Arquitectura Multicapa (Clean Architecture)

Fecha: Inicial

Decisión

El proyecto PAE utiliza **Clean Architecture** con separación clara entre:

- **Presentación** (Composable UI)
- **Dominio** (Business logic)
- **Datos** (Persistencia y APIs)

Contexto

- Proyecto grande con dos aplicaciones Android independientes
- Necesidad de reutilizar código entre Machine y RutaPAE
- Facilitar testing y mantenimiento a largo plazo

Alternativas consideradas

1. **MVP/MVVM simple**: menos testeable, acoplamiento UI-DB
2. **Arquitectura monolítica**: difícil de escalar
3. **Arquitectura hexagonal**: más compleja, innecesaria para este scope

Consecuencias

 Positivas:

- Testabilidad: cada capa se prueba independientemente
- Reusabilidad: módulos pueden reutilizarse en otros proyectos
- Mantenibilidad: cambios aislados a cada capa
- Escalabilidad: fácil agregar nuevas funcionalidades

 Negativas:

- Más archivos y carpetas
- Requiere mayor disciplina en separación de responsabilidades
- Curva de aprendizaje para nuevos desarrolladores

ADR-002: Máquina de Estados para Entregas

Decisión

El flujo de captura de entrega en Machine se implementa como una **máquina de estados** lineal con 8 estados predefinidos.

```
WaitingForWeight → CaptureImages → CaptureFace → ComparingWeights →
GenerateEmbedding → VerifyInDatabase → SaveDelivery → WaitForWeightRemoved
```

Contexto

- Proceso de entrega tiene múltiples pasos ordenados
- Cada paso puede fallar independientemente
- Necesidad de reintentos y recuperación de errores
- UI necesita mostrar progreso y estado actual

Alternativas consideradas

1. **Procedimientos lineales**: sin recuperación de errores
2. **Callbacks anidados**: “callback hell”, difícil mantener
3. **Eventos (event bus)**: desorden de eventos async
4. **Corrutinas simples**: sin persistencia de estado

Consecuencias

 Positivas:

- Flujo predecible y serializable
- Recuperación de errores clara (retry)
- Fácil seguimiento del progreso
- Estado persistible entre sesiones

 Negativas:

- Agregar nuevos estados requiere cambio en muchos lugares
- Estados deben ser completamente independientes
- Difícil cambiar orden después de deployment

ADR-003: Repository Pattern para acceso a datos

Decisión

Se implementa el **Repository Pattern** en `MachineData` y `RutaPAEData` para abstraer acceso a persistencia.

```
interface Repository {
    val deliveries: Table<Delivery>
    val beneficiaries: Table<Beneficiary>
    fun create(entity: Entity): Entity
    fun update(entity: Entity): Boolean
    fun delete(entity: Entity): Boolean
}
```

Contexto

- Base de datos SQLite con índices vectoriales (VectorDB)
- Múltiples tipos de datos con operaciones de lectura/escritura
- Posible migración futura a base de datos diferente
- Necesidad de testing sin BD real

Alternativas consideradas

1. **Acceso directo a BD**: acoplamiento fuerte
2. **ORM genérico**: framework heavy (Room sería más pesado)
3. **DAO simple**: menos abstracción
4. **SQLite directo**: sin abstracción

Consecuencias

 Positivas:

- Fácil cambiar implementación de BD
- Mock para testing
- API consistente para todas las entidades
- Independencia de framework

 Negativas:

- Más código de boilerplate
- Mapeo manual entre DTOs y entidades
- Overhead de abstracción

ADR-004: P2P con DirectLink (Wi-Fi Direct)

Decisión

La comunicación entre Machine y RutaPAE usa **DirectLink** (Wi-Fi Direct/hotspot) como transporte primario.

Contexto

- Máquinas en zonas rurales sin conectividad consistente
- Necesidad de sincronización sin depender de internet
- RutaPAE debe descubrir máquinas automáticamente
- Backup a conexión HTTP si P2P no disponible

Alternativas consideradas

1. **Solo HTTP**: depende de conectividad
2. **Bluetooth**: rango limitado
3. **Mesh networks**: complejidad innecesaria
4. **Firebase/Cloud Messaging**: requiere internet
5. **Bluetooth + HTTP**: peor que P2P + HTTP

Consecuencias

 Positivas:

- Funciona sin internet
- Local, sin latencia de cloud
- Rápido para datos grandes (fotos)
- Seguro: sin tráfico por servidor externo

 Negativas:

- Requiere permisos adicionales en Android
- No funciona con todas las mezclas de dispositivos
- Implica manejo de hotspot en Machine

ADR-005: VectorDB con embeddings 512d

Decisión

Se implementa **VectorDB ORM** con índices vectoriales para almacenar embeddings de beneficiarios.

```
@VectorColumn(dimensions = 512, distanceMetric = DistanceMetric.COSINE)
val embedding: FloatArray
```

Contexto

- Necesidad de búsqueda por similitud facial
- Operaciones locales sin enviar fotos al servidor
- Requisitos de privacidad: fotos nunca salen del dispositivo
- Precisión importante para matching correcto

Alternativas consideradas

1. **Solo búsqueda por nombre**: impreciso
2. **Fotos y búsqueda remota**: viola privacidad
3. **Hashing de imágenes**: pérdida de información
4. **Face API remota**: depende de conectividad
5. **Almacenar embeddings sin índice**: búsqueda O(n)

Consecuencias

 Positivas:

- Privacidad: datos nunca salen del dispositivo
- Rapidez: búsqueda vectorial local
- Precisión: embeddings mantiened información suficiente
- Offline-first: funciona sin conectividad

 Negativas:

- Requiere framework adicional (VectorDB)
- Consumo de memoria: 512 floats × N beneficiarios
- Aprendizaje de embeddings centralizado (necesita backend)

ADR-006: Corrutinas de Kotlin para concurrencia

Decisión

Se usa **Kotlin Coroutines** para todas las operaciones asíncronas.

```
suspend fun syncDeliveriesFromMachine() { ... }
```

```
LaunchedEffect {
    StateNameEmitter.collect { state -> ... }
}
```

Contexto

- Proyecto 100% Kotlin
- UI Composable requiere suspensión
- Operaciones I/O: HTTP, BD, P2P
- Necesidad de cancelación limpia

Alternativas consideradas

1. **Callbacks**: callback hell
2. **RxJava/RxKotlin**: framework pesado
3. **Threads**: manual tedioso, error-prone
4. **AsyncTask**: deprecated
5. **Futures/CompletableFuture**: verboso

Consecuencias

 Positivas:

- Sintaxis limpia y legible
- Integración nativa con Composable
- Manejo de excepciones familiar
- Cancelación automática
- Bajo overhead

 Negativas:

- Curva de aprendizaje (suspend, async, launch)
- Debugging de flow más complejo
- Compilación más lenta

ADR-007: Emisores para comunicación inter-módulos

Decisión

Se usa patrón **Observer con Flow/Emitters** para comunicación entre módulos.

```
StateNameEmitter.emit("CapturingFace")
```

```
StateNameEmitter.collect { name -> ... }
```

Contexto

- Comunicación desacoplada entre Dominio y Presentación
- Múltiples observadores posibles
- Reactividad necesaria
- Necesidad de serialización mínima

Alternativas consideradas

1. **Callbacks directos**: acoplamiento fuerte
2. **Live Data**: deprecado, Framework Android
3. **Event Bus (EventBus/Otto)**: reflexión, overhead
4. **StateFlow global**: estado mutable shared
5. **Intent (broadcast)**: para cross-process, innecesario aquí

Consecuencias

 Positivas:

- Desacoplamiento: productor no conoce consumidores
- Múltiples observadores posibles
- Type-safe
- Cancelación automática con scope
- Performante

 Negativas:

- Debugging: flujo de datos menos visible
- Posibles memory leaks si no se cancela
- Test debe mockear Flow

ADR-008: Inyección de dependencias diferenciada

Decisión

- Machine usa **Dagger2/Hilt**
- RutaPAE usa **inyección manual**
- MachineDomain/RutaPAEDomain usan **inyección por constructor**

Contexto

- Machine: app compleja con muchos componentes

- RutaPAE: app más simple, control centralizado en DomainManager
- Módulos de lógica: necesitan flexibilidad en testing

Alternativas consideradas

1. **Hilt en todas partes**: overhead para RutaPAE
2. **Service Locator**: anti-pattern
3. **Singleton global**: testing difícil
4. **Factory methods**: boilerplate

Consecuencias

 Positivas:

- Apropiado para cada caso (complejidad vs simplicidad)
- Control fino en RutaPAE
- Automatización en Machine
- Testeable

 Negativas:

- Inconsistencia entre apps
- Desarrolladores aprenden dos patrones

ADR-009: Modelos P2P en Contract separado

Decisión

Se crea módulo `Contract` con modelos P2P compartidos entre Machine y RutaPAE.

```
P2PMachineState
P2PDeliveryData
P2PMachineConfiguration
// ...
```

Contexto

- Necesidad de versioning de API P2P
- Ambas apps deben conocer el mismo formato
- Cambios coordin

ados entre apps

Alternativas consideradas

1. **Modelos en Machine, RutaPAE importa**: acoplamiento
2. **Modelos duplicados**: desincronización futura
3. **Serialización ad-hoc**: error-prone
4. **GraphQL schema**: overhead innecesario

Consecuencias

 Positivas:

- Single source of truth
- Ambas apps sincronizadas
- Versionable
- Testing de serialización centralizado

 Negativas:

- Módulo adicional
- Cambios coordinados entre apps

ADR-010: WorkManager para sync en background

Decisión

Se usa **WorkManager** en RutaPAE para sincronización de entregas en background.

```
DeliverySyncWorker : CoroutineWorker {
    override suspend fun doWork(): Result
}

DeliverySyncScheduler.enqueue(context, machineId)
```

Contexto

- Sincronización debe ocurrir incluso con app cerrada
- Límites de background execution en Android 8+
- Necesidad de reintentos automáticos
- Batching de requests

Alternativas consideradas

1. **Foreground Service**: requiere notificación
2. **Handler/Timer**: impreciso

3. **Alarm Manager**: timing inflexible
4. **Firebase Job Dispatcher**: obsoleto
5. **Sync Adapter**: para providers

Consecuencias

 Positivas:

- Respeta límites de Android
- Reintentos automáticos
- Batching eficiente
- Diagnóstico integrado

 Negativas:

- Ejecución no está garantizada (timing)
- Debugging más complejo
- Requiere WorkManager dependency

ADR-011: UI con Jetpack Compose

Decisión

Ambas apps usan **Jetpack Compose** exclusivamente para UI.

```
@Composable
fun LaboratoryScreen(viewModel: LaboratoryViewModel) { ... }
```

Contexto

- Proyecto nuevo: sin legacy XML layouts
- Mejor productividad
- Hot reload en desarrollo
- UI declarativa más mantenible

Alternativas consideradas

1. **XML layouts**: más verbose
2. **Flutter**: cambio de stack
3. **SwiftUI**: solo iOS
4. **Hybrid (Compose+XML)**: inconsistencia

Consecuencias

 Positivas:

- Sintaxis limpia
- Hot reload
- Menos boilerplate
- Preview en IDE

 Negativas:

- Composability learning curve
- Compilación más lenta
- Debugging de recomposition complejo

ADR-012: Dos aplicaciones separadas (Machine + RutaPAE)

Decisión

Se mantienen **dos aplicaciones Android separadas** con sus propios builds y releases.

Contexto

- Diferentes funcionalidades
- Diferentes workflows
- Diferentes versiones
- Diferentes firmas de certificado

Alternativas consideradas

1. **Una app con modos**: complejidad, conflicto de UX
2. **Shared codebase con build variants**: difícil mantener
3. **PWA + Android**: scope diferente

Consecuencias

 Positivas:

- Liberación independiente
- Claridad de propósito
- Sin complejidad condicional

 Negativas:

- Duplicación de código base
- Cambios deben aplicarse a ambas

ADR-013: SQLite + VectorDB ORM

Decisión

Se usa **SQLite con VectorDB ORM** en lugar de Room o Firebase.

Contexto

- Necesidad de índices vectoriales
- Control fino de esquema
- Sin dependencias grandes
- Offline-first

Alternativas consideradas

1. **Room**: no tiene soporte vectorial
2. **Firebase**: requiere conectividad
3. **Realm**: propietario
4. **Raw SQLite**: sin ORM
5. **PostgreSQL embeddo**: demasiado pesado

Consecuencias

 Positivas:

- Índices vectoriales nativos
- Control completo
- Ligero
- Offline

 Negativas:

- API customizada
- Menos tooling
- Migraciones manuales

Principios guía

Estos son los principios que guían las decisiones arquitectónicas en PAE:

1. **Offline-first**: el sistema funciona sin conectividad
2. **Privacy-first**: datos sensibles nunca salen del dispositivo
3. **Responsabilidad única**: cada módulo hace una cosa bien
4. **Inversión de dependencias**: abstractas, no concretas
5. **Open/Closed**: abierto para extensión, cerrado para modificación
6. **DRY (Don't Repeat Yourself)**: evitar duplicación
7. **KISS (Keep It Simple, Stupid)**: no over-engineer
8. **Testeable**: todo debe testearse fácilmente
9. **Performance**: observar impacto en batería y memoria
10. **User experience**: la arquitectura debe servir al usuario, no al revés

---

Futuras decisiones pendientes

- Migración a Kotlin Multiplatform (iOS, Backend)
- Agregación de analytics (privado, local)
- Expansión a más modalidades de entrega
- Integración con más proveedores de verificación biométrica