TEMA

JUSTIFICACION

PRÁCTICA DE INVESTIGACIÓN — DISEÑO DE ARQUITECTURA DE SOFTWARE PARA UNA APLICACIÓN MÓVIL

ALUMNOS

ALEXIS BERNAL NIETO

JANET LEILANI CARAPIA HERNANDEZ

LEONARDO JESÚS MORALES TREJO

ABIGAIL RODRIGUEZ FERMIN

HELEN VEGA RESENDIZ

GRUPO

DS03SV-24

CARRERA

INGENIERÍA EN DESARROLLO Y GESTIÓN DE SOFTWARE

CUATRIMESTRE

DECIMO CUATRIMESTRE

PROFESOR

SALDAÑA BENITEZ HECTOR

MATERIA

DESARROLLO MÓVIL INTEGRAL

LUGAR: SAN JUAN DEL RIO, QRO. **FECHA:** 18/09/2025

Justificación — Selección de arquitectura (MVVM + Clean)

Proyecto: SmartStorage Garage (App móvil)

Contexto del proyecto

- **Tipo de app:** Flutter (Android/iOS) para arrendatarios y, en segunda fase, operadores. Funciones: reservar unidad, abrir/cerrar con credenciales temporales (QR/NFC/PIN), ver facturas, pagar, recibir alertas (humedad/intrusión) y notificaciones push.
- **Fuentes de datos:** API REST (Core backend); repositorios con sincronización local (SQLite/Isar/Hive); notificaciones FCM. Los eventos IoT (MQTT) se gestionan en el backend y se reflejan vía REST/push.
- **Offline:** alta necesidad. Debe mostrar estado, facturas y **AccessGrant** cacheados; colas de acciones con reintento cuando vuelve la red.
- **Notificaciones:** Push para pagos/alertas; in-app para estados de acceso/sensores.
- **Crecimiento esperado:** +3 módulos/mes (reservas, inventario de objetos, plan Business). Uso de *feature toggles* y modularidad.
- **Equipo y experiencia: 5 integrantes**. Sugerido: **2 móviles Flutter (1 lead + 1 mid), 1 backend (API/DB), 1 IoT/Edge (gateway, MQTT, cámaras), 1 QA/DevOps (CI, pruebas, releases)**. Objetivo: mantenibilidad y testabilidad con *ramp-up* rápido y entregas en paralelo por vertical.

Criterio	MVC	MVVM	Clean (capas + MVVM)
Responsabilidades	Controller concentra navegación y lógica de presentación (riesgo “God Controller”).	ViewModel expone estado/acciones; View pasiva; *binding* reactivo.	**Domain** puro (entidades/use cases). Presentation desacoplada. Data con repositorios.
Flujo de datos	Bidireccional View↔Model vía Controller.	Unidireccional: VM como fuente de verdad de UI.	Unidireccional con **regla de dependencia** hacia Domain.
Testabilidad	Media (controllers acoplados al framework).	Alta (ViewModel testeable sin framework).	Muy alta: Domain 100% testeable; repos mockeables.
Complejidad inicial	Baja/Media.	Media.	Media/Alta (más clases/abstracciones).
Curva de aprendizaje	Baja (apps pequeñas).	Media (estado reactivo).	Media/Alta (disciplina en capas y mapeos DTO↔Entidad).
Dependencia de frameworks	Alta (controllers/activities/fragments).	Media (VM + librería de estado).	**Baja en Domain**; Presentation/Data pueden variar sin tocar Domain.

Selección

Elegimos: *MVVM + Clean Architecture* en Flutter, con *Riverpod* para estado/DI ligera.

Criterios de decisión

1. **Mantenibilidad:** separación clara (Presentation, Domain, Data). El **Domain** es independiente de Flutter/APIs; cambios de UI o backend no rompen reglas de negocio.
2. **Escalabilidad:** repositorios definidos como **interfaces** en Domain permiten añadir nuevas fuentes (REST/Firestore/cache) y casos de uso sin reescribir UI.
3. **Testabilidad:** Use Cases y ViewModels se prueban en aislamiento con dobles de prueba (mocks/fakes) de Repositories.
4. **Offline-first:** repositorios implementan *single source of truth* con caché local y sincronización; Use Cases orquestan políticas de lectura/escritura y reintentos.
5. **Eventos/Realtime:** el dominio permanece estable; adaptadores en Data traducen MQTT/FCM a repositorios/streams sin acoplar UI al transporte.

Riesgos y mitigaciones

- **Boilerplate / sobre-arquitectura en MVP.** *Mitigación:* aplicar **vertical slices** por *feature*; *codegen* (freezed/json_serializable).
- **ViewModels con demasiada lógica.** *Mitigación:* mover reglas a **Use Cases**; dejar VM como orquestador de estado.
- **Violación de la regla de dependencia** (UI conoce Data). *Mitigación:* depender de **interfaces** en Domain e inyectar implementaciones en Data.
- **Complejidad de sincronización offline.** *Mitigación:* políticas claras (write-through / write-behind), colas de operaciones y reconciliación por *timestamp*/versión.

Estado en Flutter (gestión de estado)

- **Estrategia: Riverpod** para gestión de estado y DI ligera (Providers para UseCases/Repos). Ventajas: testabilidad, devtools, menos boilerplate.
- **Alternativas: BLoC** si el equipo prefiere flujos de eventos estrictos; **Provider** para equipos junior/POC.
- **Pruebas:** unit (use cases, mappers), widget (pantallas críticas), integración (repos y sincronización).

Referencias

- Android Developers. (n.d.). *Guide to app architecture*. https://developer.android.com/topic/architecture
- Android Developers. (n.d.). *Architecture recommendations*. https://developer.android.com/topic/architecture/recommendations
- Fowler, M. (n.d.). *GUI architectures*. martinfowler.com. https://martinfowler.com/eaaDev/uiArchs.html
- Martin, R. C. (2012, August 13). *The clean architecture*. The Clean Coder Blog. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- Flutter. (n.d.). *State management*. https://docs.flutter.dev/data-and-backend/state-mgmt
- Flutter. (n.d.). *Local caching / Offline-first*. https://docs.flutter.dev/get-started/fundamentals/local-caching
- Riverpod. (n.d.). *Riverpod documentation*. https://riverpod.dev/
- OWASP Foundation. (2024). *OWASP Mobile Top 10 – 2024*. https://owasp.org/www-project-mobile-top-10/