

# Documento de Diseño de la Arquitectura del Sistema



1. Introducción	3
1.1. Objetivo del documento	3
1.3. Visión general del producto	3
2. Arquitectura General del Sistema	4
2.1. Descripción de la arquitectura	4
2.2. Diagrama general de arquitectura	4
2.3. Justificación de la arquitectura elegida	5
<b>3. Componentes del Sistema</b>	<b>5</b>
3.1. Aplicación móvil (React Native)	5
3.2. Aplicación de escritorio (Electron)	7
3.3. Landing Page web	8
3.4. Base de Datos (SQLite)	9
4. Diseño de la Base de Datos	10
4.1. Modelo entidad-relación (ER)	10
4.2. Diagrama del modelo lógico (texto)	11
4.3. Tablas y campos	12
4.4. Políticas de integridad y restricciones	13
5. Seguridad del Sistema	14
5.1. Cifrado y protección de datos	14
5.2. Control de acceso	14
5.3. Riesgos y mitigaciones	15
6. Diseño de la Comunicación Interna	16
6.1. Módulos y flujo de datos	16
6.2. API interna / servicios	16
7. Requisitos Técnicos	17
7.1. Requisitos de hardware	17
7.2. Requisitos de software	18
7.3. Dependencias	18
8. Consideraciones de Rendimiento	19
8.1. Optimización esperada	19
8.2. Estrategias de eficiencia	19
9. Consideraciones de Mantenibilidad y Escalabilidad	20
9.1. Estructura modular	20
9.2. Posibles ampliaciones futuras	20
10. Conclusiones	21

# 1. Introducción

## 1.1. Objetivo del documento

Este documento cumple la función de proporcionar una estructura técnica suficiente para poder organizar todo el proyecto de manera unida y organizada desde la base de datos hasta la landing page. Toda la estructura estará detallada antes de poder organizar la programación y las posibles conexiones que se necesitaran.

## 1.2. Alcance del sistema

El sistema incluye una aplicación local-first que proporciona al usuario una seguridad real sobre sus contraseñas y que sea capaz de poder estar absolutamente protegido ante ataques y defendiéndose incluso una vez ya hackeado el ordenador. El propósito de la aplicación es la creación de un entorno absolutamente inaccesible para el usuario que le proporcione al usuario la capacidad de sentirse seguro.

## 1.3. Visión general del producto

Resumen del sistema, tipo de aplicación (móvil, escritorio, web), y público objetivo.

El sistema correrá de manera nativa en Android, donde aquí se trabajará todas las distintas versiones que se harán; Primero la de Iphone y después desktop. El usuario podrá acceder desde sus distintas plataforma a su aplicación y podrá ahí: gestionar, guardar y generar sus contraseñas de manera segura.

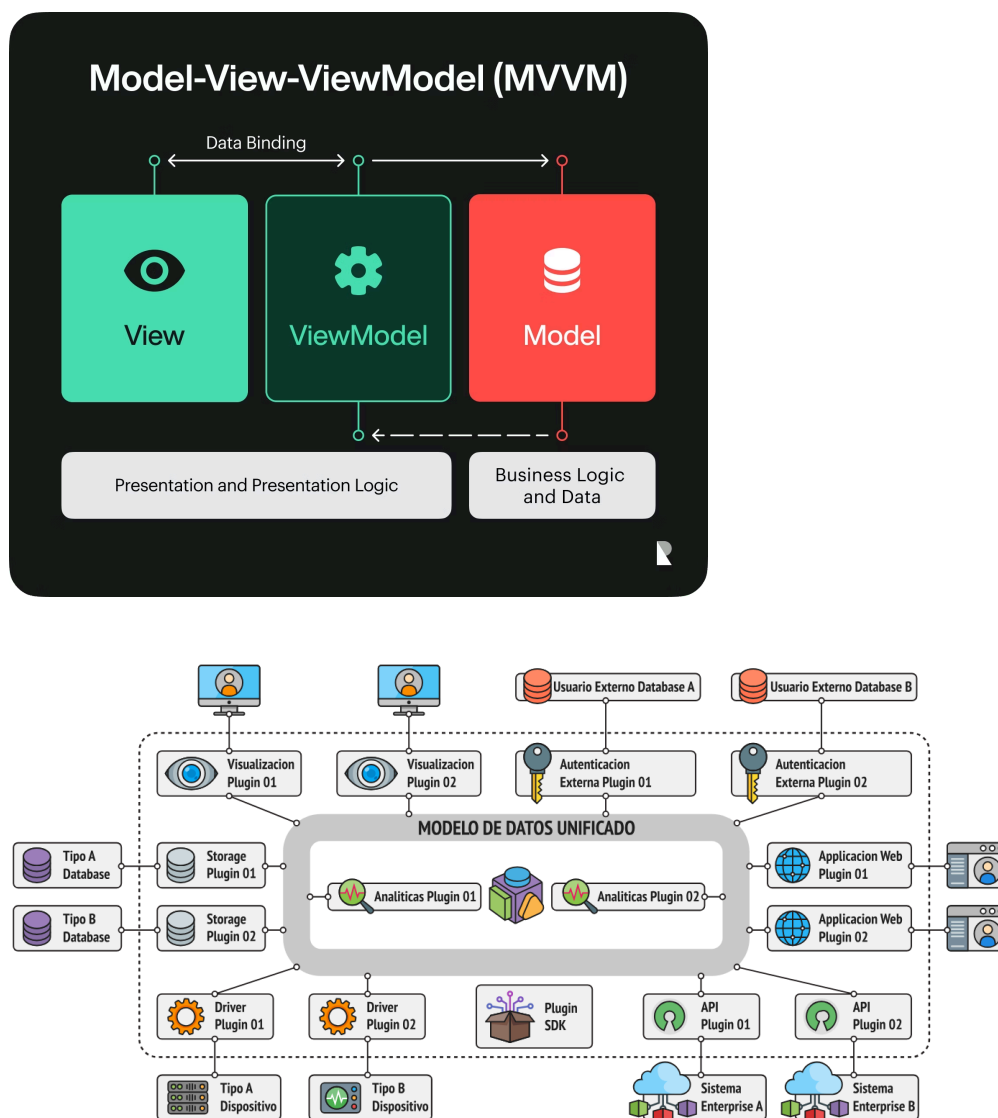
Mi público objetivo es el usuario común que solo quiera un gestor de contraseñas gratuito y que realmente sea seguro, con características open source. Pero, no descarto para posibles versiones la anidación a empresas ofreciendo sistemas a empresas.

## 2. Arquitectura General del Sistema

### 2.1. Descripción de la arquitectura

El sistema se estructura siguiendo una arquitectura modular combinada con el patrón MVVM, permitiendo una separación clara entre la lógica de negocio (ViewModel), la interfaz de usuario (View) y los datos (Model). Los módulos independientes —seguridad, persistencia, gestión de credenciales e interfaz— facilitan la escalabilidad, la mantenibilidad y la extensibilidad futura del proyecto.

### 2.2. Diagrama general de arquitectura



## 2.3. Justificación de la arquitectura elegida

He elegido una arquitectura **modular combinada con MVVM** porque se adapta muy bien a un proyecto que debe ser seguro, escalable y multiplataforma.

La arquitectura modular ya la he utilizado anteriormente en Python y me resulta intuitiva; me permite organizar el código en bloques independientes (seguridad, persistencia, gestión de credenciales, etc.), lo que facilita añadir nuevas funcionalidades en el futuro sin modificar lo que ya funciona.

Además, al combinar esta estructura con **MVVM**, consigo una separación aún más clara entre la interfaz y la lógica interna del sistema. Esto es especialmente útil en un proyecto que tiene versiones para escritorio y móvil, ya que la lógica (ViewModel y Model) se mantiene coherente, mientras que cada plataforma puede tener su propia capa visual.

La forma en la que lo imagino es como una tienda:

el usuario interactúa únicamente con el mostrador (la interfaz), mientras que detrás hay estanterías con herramientas organizadas por módulos. Si algún día quiero añadir una nueva estantería —es decir, una nueva función o componente— puedo hacerlo sin alterar la experiencia del usuario ni romper lo ya existente.

---

## 3. Componentes del Sistema

### 3.1. Aplicación móvil (React Native)

#### Estructura:

- App basada en MVVM:
  - **View**: pantallas de React Native
  - **ViewModel**: lógica de validación, cifrado y manejo de datos
  - **Model**: objetos (credencial, usuario, metadatos)
- Arquitectura modular: seguridad, persistencia, UI, utilidades.

**Navegación:**

- Navegación mediante **React Navigation**.
- Stack principal:
  1. **Pantalla de desbloqueo** (contraseña maestra)
  2. **Listado de contraseñas**
  3. **Detalle de credencial**
  4. **Crear / editar contraseña**
  5. **Ajustes / exportación / importación**

**Módulos y pantallas:**

- Módulo de autenticación
- Módulo de seguridad (cifrado AES, hashing PBKDF2/Argon2)
- Módulo de gestión de contraseñas
- Pantallas de gestión, edición y ajustes
- Módulo de backup/exportación.

**Integración con la base de datos local:**

- Base de datos local mediante **SQLite** + librería “react-native-sqlite-storage”.
- Acceso a la BD siempre **después de descifrar clave interna**.
- Todas las credenciales se almacenan **cifradas** antes de insertarse.

## 3.2. Aplicación de escritorio (Electron)

### Estructura:

- Proceso **main**: control de ventanas, seguridad, APIs internas.
- Proceso **renderer**: interfaz (React o HTML/JS).
- Módulos separados: seguridad, persistencia, interfaz, utilidades.
- BD SQLite integrada en el sistema de archivos local.

### Comunicación entre procesos (main ↔ renderer):

- Comunicación mediante **IPC (inter-process communication)**.
- El renderer nunca manipula archivos directamente; manda solicitudes al main.

### Gestión de archivos:

- Lectura/escritura solo permitida desde el main.
- Importación/exportación cifrada (por ejemplo, archivos *.pretorian*).
- Sistema sandbox del renderer activado para reducir riesgos.

### 3.3. Landing Page web

#### Tecnologías utilizadas:

- HTML + CSS + JS
- Deploy estático: GitHub Pages / Netlify / Vercel
- No almacena datos, solo contenido informativo.

#### Estructura de la página:

1. **Portada**
  - Nombre del proyecto
  - Lema / objetivo
2. **Descripción de la aplicación**
  - Características principales
3. **Descargas**
  - APK Android
  - App de escritorio
4. **Tutoriales y documentación**
5. **Aviso sobre privacidad y seguridad**
6. **Enlaces al repositorio Open Source**

### 3.4. Base de Datos (SQLite)

#### Motivos de elección:

- Permite almacenamiento **local-first**.
- Ligera, rápida y multiplataforma.
- Muy fácil de integrar en React Native y Electron.
- No requiere servidor (cumple tu requisito de privacidad total).

#### Flujo de acceso a datos:

1. La app se desbloquea con contraseña maestra.
2. Se genera/descifra la clave interna.
3. Se descifran los registros que se consultan.
4. La BD nunca contiene textos planos, solo cifrado AES-256.
5. Las operaciones CRUD pasan por un módulo de seguridad.

#### Capa de persistencia en cada plataforma

- **React Native:**
  - SQLite mediante librería
  - Acceso asíncrono (promesas)
- **Electron:**
  - SQLite mediante driver Node (better-sqlite3 o sqlite3)
  - Acceso sincrónico en el main para mayor control

## 4. Diseño de la Base de Datos

### 4.1. Modelo entidad-relación (ER)

El sistema tendrá dos entidades principales:

#### 1. Usuario

- Representa a cada persona que utiliza la aplicación.
- Cada usuario tiene una **contraseña maestra** (cifrada) y datos de perfil.
- Relación: un usuario puede tener **muchas credenciales**.

#### 2. Credencial

- Representa cada cuenta, servicio o aplicación que el usuario gestiona.
- Contiene usuario del servicio, contraseña cifrada, notas y fecha de creación.
- Relación: pertenece a un solo usuario.

**Relación:**

- **1 Usuario → N Credenciales** (uno a muchos)

Descripción textual: cada usuario puede almacenar múltiples credenciales, pero cada credencial pertenece únicamente a un usuario.

## 4.2. Diagrama del modelo lógico (texto)

Usuario

-----

id\_usuario (PK)

nombre\_usuario

hash\_contrasena\_maestra

email

fecha\_creacion

Credencial

-----

id\_credencial (PK)

id\_usuario (FK)

servicio

usuario\_servicio

contrasena\_cifrada

notas

fecha\_creacion

- **PK** → Clave primaria
- **FK** → Clave foránea, referencia a Usuario

## 4.3. Tablas y campos

**Tabla: Usuario**

Campo	Tipo	Descripción
id_usuario	INTEGER	Identificador único de usuario, autoincremental
nombre_usuario	TEXT	Nombre del usuario
hash_contrasena_maestra	TEXT	Contraseña maestra cifrada / hash
email	TEXT	Correo electrónico (opcional)
fecha_creacion	DATETIME	Fecha de registro del usuario

**Tabla: Credencial**

Campo	Tipo	Descripción
id_credencial	INTEGER	Identificador único de la credencial, autoincremental
id_usuario	INTEGER	Clave foránea que relaciona la credencial con un usuario
servicio	TEXT	Nombre del servicio (ej: Gmail, Facebook)
usuario_servicio	TEXT	Usuario o correo usado en el servicio
contrasena_cifrada	TEXT	Contraseña cifrada del servicio

notas	TEXT	Notas adicionales
fecha_creacion	DATETIME	Fecha de creación de la credencial

---

## 4.4. Políticas de integridad y restricciones

- **Integridad referencial:**
    - Credencial.id\_usuario → FK → Usuario.id\_usuario
    - Al borrar un usuario, sus credenciales se borran en cascada (ON DELETE CASCADE).
  - **Restricciones:**
    - PK en ambas tablas (id\_usuario, id\_credencial).
    - usuario\_servicio + servicio puede ser UNIQUE si quieres evitar duplicados.
  - **Índices recomendados:**
    - Índice por id\_usuario en Credencial para búsquedas rápidas.
    - Índice por servicio si se hace búsqueda por nombre de servicio.
  - **Seguridad y cifrado:**
    - Ninguna contraseña se guarda en texto plano, solo cifrada con AES-256.
    - Contraseña maestra almacenada como hash seguro (PBKDF2, bcrypt o Argon2).
-

## 5. Seguridad del Sistema

### 5.1. Cifrado y protección de datos

#### Algoritmos utilizados

- **AES-256 (Advanced Encryption Standard):**
  - Para cifrar todas las contraseñas de los servicios.
  - Modo de operación recomendado: **GCM** (autenticación y confidencialidad).
- **PBKDF2 / Argon2 / bcrypt:**
  - Para almacenar la contraseña maestra del usuario en forma de **hash seguro**.
  - Evita que la contraseña maestra pueda ser revertida a texto plano.

#### Gestión de claves

- La contraseña maestra genera la **clave de cifrado local**.
  - La clave AES se deriva dinámicamente usando PBKDF2/Argon2 con sal única por usuario.
  - Las claves **no se guardan en disco**, solo se mantienen en memoria mientras la aplicación está abierta.
- 

### 5.2. Control de acceso

#### Validación

- Validación de contraseña maestra al iniciar la app.
- Comprobaciones de integridad de la base de datos y datos cifrados antes de cualquier operación.

#### Autenticación

- Usuario inicia sesión con contraseña maestra o biometría.
- Cada sesión genera un token temporal en memoria que se destruye al cerrar la app.

#### Persistencia segura

- Todos los datos almacenados en **SQLite** cifrados.
  - Las operaciones de lectura/escritura pasan por el módulo de seguridad (nunca directo).
  - Archivos de backup exportados también cifrados con la misma clave derivada.
-

### 5.3. Riesgos y mitigaciones

Riesgo / Vulnerabilidad	Mitigación
Robo del dispositivo	Todos los datos cifrados; contraseña maestra requerida.
Ataques de fuerza bruta	Hash seguro con PBKDF2/Argon2, iteraciones altas y sal única.
Fuga de datos por backup	Backups siempre cifrados con AES-256; contraseña maestra necesaria para importar.
Acceso directo a archivos SQLite	Cifrado completo de la base de datos; acceso controlado por módulo de seguridad.
Malware o keyloggers	Se recomienda usar apps de seguridad en el sistema.
Corrupción de datos	Validaciones de integridad antes de operaciones críticas; exportación/importación segura.

## 6. Diseño de la Comunicación Interna

### 6.1. Módulos y flujo de datos

El sistema está organizado en módulos independientes siguiendo la arquitectura **modular + MVVM**, lo que permite que cada componente se comunique de forma clara y segura.

#### Flujo general de datos:

1. **Usuario** interactúa con la **Vista** (UI en móvil o escritorio).
2. La **Vista** envía acciones al **ViewModel**, que contiene la lógica de negocio.
3. El **ViewModel** llama a los **módulos de seguridad y persistencia** según la acción:
  - Cifrado / descifrado de datos
  - Operaciones CRUD sobre la base de datos SQLite
  - Generación de backups o restauración de datos
4. Los datos procesados regresan al **ViewModel**, que actualiza la **Vista**.

Así, la **Vista** nunca accede directamente a los datos ni a los ficheros, garantizando seguridad y separación de responsabilidades.

---

### 6.2. API interna / servicios

Para mantener la comunicación organizada y segura, se implementa una **API interna entre módulos**:

#### Electron

- Comunicación entre **main** ↔ **renderer** mediante **IPC (Inter-Process Communication)**.
- Funciones expuestas por el main (seguras y controladas):

#### React Native

- Módulos de acceso a SQLite encapsulados en funciones o servicios:

- Funciones de cifrado integradas antes de guardar y al leer datos.

### Beneficios de esta arquitectura de comunicación

- **Seguridad:** la Vista no toca archivos ni claves.
  - **Reutilización:** el mismo módulo de seguridad/persistencia puede compartirse entre móvil y escritorio.
  - **Mantenibilidad:** cambios en un módulo no afectan directamente a otros.
  - **Escalabilidad:** se pueden añadir nuevos módulos (p.ej., sincronización cifrada) sin alterar el flujo existente.
- 

## 7. Requisitos Técnicos

### 7.1. Requisitos de hardware

- **Móvil (React Native)**
    - Android 7.0 o superior
    - iOS 12 o superior
    - 2 GB de RAM mínimo
    - Espacio libre aproximado: 50 MB
  - **Escritorio (Electron)**
    - Windows 10 o superior / macOS 10.15 o superior / Linux moderno
    - 4 GB de RAM mínimo
    - Espacio libre aproximado: 100 MB
  - **Para desarrollo**
    - PC con 8 GB de RAM o superior
    - CPU moderna (quad-core recomendado)
    - Conexión a internet para instalar dependencias
-

## 7.2. Requisitos de software

- **Móvil (React Native)**
    - Node.js  $\geq 18.x$
    - NPM o Yarn
    - Expo CLI o React Native CLI
    - Android Studio / Xcode (según plataforma)
  - **Escritorio (Electron)**
    - Node.js  $\geq 18.x$
    - NPM o Yarn
    - Electron  $\geq 26.x$
  - **Base de datos**
    - SQLite 3
    - Librerías de acceso:
      - react-native-sqlite-storage para móvil
      - better-sqlite3 o sqlite3 para escritorio
  - **Landing page**
    - HTML/CSS/JS
    - Servidor estático GitHub Pages / Netlify / Vercel
- 

## 7.3. Dependencias

- **JavaScript / TypeScript**: base para Electron y React Native
  - **React Navigation**: navegación móvil
  - **crypto-js / bcryptjs / argon2**: cifrado y hashing
  - **SQLite**: persistencia de datos local
  - **TailwindCSS / Shadcn UI / Material UI**: diseño de interfaces
  - **Electron-builder**: empaquetado y generación de ejecutables de escritorio
  - **Git**: control de versiones
- 

### Notas adicionales

- Todo el software utilizado es **gratuito y open source**, compatible con la filosofía del proyecto.
- Las versiones de las librerías serán definidas al inicio del desarrollo para garantizar **compatibilidad y estabilidad**.

- La arquitectura modular permite **actualizar dependencias o cambiar librerías** sin afectar el núcleo del sistema.
- 

## 8. Consideraciones de Rendimiento

### 8.1. Optimización esperada

- Aplicación móvil y de escritorio:
  - Tiempo de carga inicial reducido mediante lazy loading de componentes y pantallas.
  - Consultas a la base de datos optimizadas con índices en campos clave (id\_usuario, servicio).
  - Cifrado/descifrado ejecutado solo cuando es necesario, evitando mantener grandes volúmenes de datos en memoria.
- Landing page:
  - Minimalista, con recursos estáticos optimizados (imágenes comprimidas, JS/CSS minificado).
- Backups y exportaciones:
  - Operaciones en segundo plano para no bloquear la interfaz.

### 8.2. Estrategias de eficiencia

- Módulos independientes: cada uno optimiza su lógica sin afectar otros módulos.
  - Procesamiento asíncrono: lectura/escritura de SQLite en móvil con promesas, evitando congelar la UI.
  - Caching temporal: datos descifrados solo en memoria mientras sea necesario.
  - Evitar duplicación de cálculos: por ejemplo, generación de claves y hashing solo cuando se cambia la contraseña maestra.
-

## 9. Consideraciones de Mantenibilidad y Escalabilidad

### 9.1. Estructura modular

- Separación clara de responsabilidades:
  - Seguridad, persistencia, gestión de credenciales, interfaz.
- Permite añadir nuevos módulos (p.ej., sincronización cifrada, soporte para múltiples dispositivos) sin alterar el núcleo.
- Cada módulo puede ser testeado y actualizado de forma independiente.

### 9.2. Posibles ampliaciones futuras

- Sincronización cifrada entre dispositivos (opcional, respetando filosofía local-first).
- Soporte multiplataforma ampliado: Linux, versiones futuras de Android/iOS.
- Integración con gestores de contraseñas existentes vía import/export cifrado.
- Mejoras en UI/UX: personalización de temas, Dark Mode.
- Notificaciones y recordatorios: alertas sobre contraseñas débiles o expiradas.

## 10. Conclusiones

Es un proyecto con mucho futuro y que podrá arreglar todo lo que se le propone de manera más o menos eficiente, con interfaces moderna, conexiones correctas y tecnologías que le proporcionen la potencia suficiente para que cualquier dispositivo pueda funcionar de manera más o menos decente y el usuario pueda estar protegido, tanto como con sus contraseñas como en la seguridad de la información añadida que pueda llegar albergar.

---