ESCUELA SUPERIOR POLITÉCNICA DE CHIMBORAZO

FACULTAD DE INFORMÁTICA Y ELECTRÓNICA

FACULTAD DE INFORMÁTICA Y ELECTRÓNICA

INGENIERÍA EN SOFTWARE

APLICACIONES INFORMÁTICAS II

ESTUDIANTE: SCARLET CAYAPA – 7166

CURSO: OCTAVO

TEMA: DISEÑO ARQUITECTÓNICO DE LA APLICACIÓN

FECHA DE ENTREGA: 27/10/2025

DOCENTE: ING. JULIO SANTILLÁN

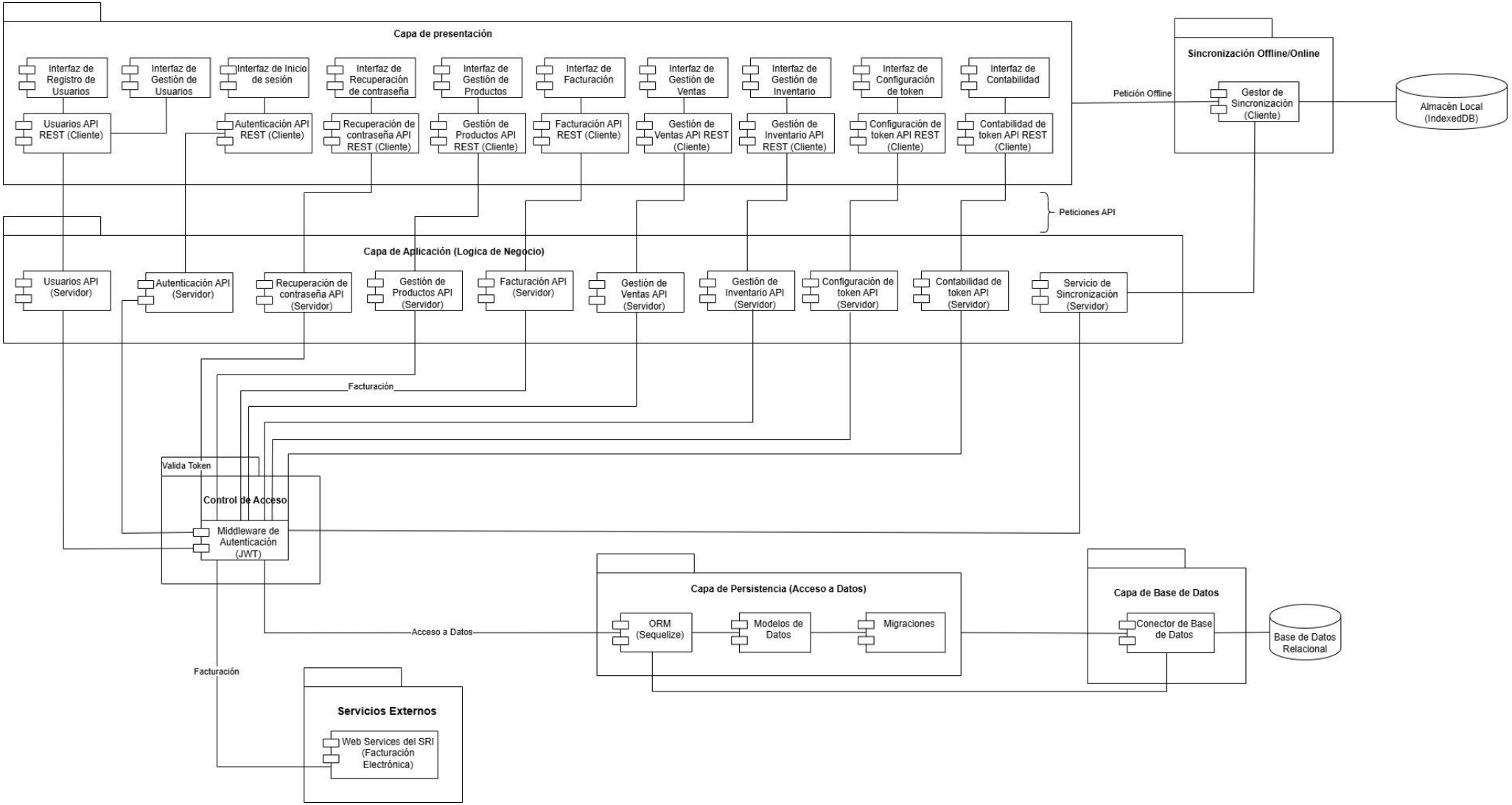
SEPTIEMBRE - FEBRERO 2026

1. Selección de la Arquitectura

Para el desarrollo de la aplicación, se ha seleccionado un estilo arquitectónico en capas (Layered Architecture). Esta elección se fundamenta en su capacidad para organizar el sistema en componentes lógicos bien definidos, facilitando la separación de responsabilidades, la mantenibilidad y la escalabilidad del proyecto.

Características de la Arquitectura en capas			
Claridad y organización	**Facilidad de testing**	**Flexibilidad para cambios**	**Adecuación al contexto del proyecto**
Permite dividir el sistema en módulos independientes (presentación, lógica de negocio, persistencia), lo que simplifica el desarrollo y la comprensión del código.	Cada capa puede ser probada de forma aislada, lo cual es crucial para garantizar la calidad del software.	El desacoplamiento entre capas permite modificar una sin afectar significativamente a las demás (por ejemplo, cambiar la base de datos o el frontend).	Dado que el proyecto es de mediana complejidad y será desarrollado por un solo integrante, esta arquitectura ofrece un equilibrio ideal entre estructura y simplicidad, evitando la sobrecarga de estilos más complejos como microservicios o event-driven.

2. Diagrama Arquitectónico



3. Análisis Comparativo (Benchmarking) de patrones de diseño

Se presenta un análisis comparativo de los patrones más relevantes para cada capa del sistema, considerando las necesidades específicas de una PWA con funcionalidades de gestión comercial y analítica predictiva.

3.1 Tabla de Benchmarking: Patrones para el Frontend (React.js)

Criterio de Decisión	Container-Presenter	MVC Clásico	Flux/Redux Puro
Adecuación a React	Alta. Se alinea con la filosofía de componentes y hooks.	Media. React no es un framework MVC; forzarlo genera complejidad innecesaria.	Alta, pero con sobrecarga si no se usa Redux Toolkit.
Gestión de Estado Global	Requiere complementarse con Zustand/Redux	Baja. No resuelve estado global por sí solo.	Alta. Diseñado específicamente para estado global.
Mantenibilidad	Alta. Separa lógica de presentación de lógica de negocio.	Media. Puede generar acoplamiento si no se implementa con disciplina.	Alta, pero verboso sin herramientas modernas.
Soporte para PWA (offline, sincronización)	Alta. Fácil integración con IndexedDB y eventos.	Media. Posible, pero menos flexible.	Alta. Middleware permite persistir estado offline.
Curva de Aprendizaje	Baja. Intuitivo para desarrolladores de React.	Media. Requiere adaptar conceptos MVC a componentes.	Moderada. Requiere entender acciones, reducers, store.

3.2 Tabla de Benchmarking: Patrones para el Backend (Express.js)

Criterio de Decisión	Repository + Service	Active Records (Solo Sequelize)	CQRS
Desacoplamiento de lógica y datos	Alta. Repositorios abstraen la base de datos.	Baja. La lógica se mezcla con los modelos.	AAlta, pero excesiva para este caso.
Testeabilidad	Alta. Fácil mockear repositorios en tests unitarios.	Media. Requiere base de datos real o mocks complejos.	Alta, pero con mayor complejidad.
Escalabilidad	Alta. Permite evolucionar capas independientemente.	Media. Difícil escalar si la lógica crece.	Muy alta, pero innecesaria para volúmenes moderados.
Integración con modelo predictivo	Alta. El DemandPredictionService encapsula la lógica.	Media. Lógica dispersa en controladores.	Compleja. Sobrediseño para este contexto.

Curva de Aprendizaje	Moderada. Bien documentado y usado en la industria.	Baja. Más simple, pero menos robusto.	Alta. Requiere arquitectura avanzada.

3.3 Tabla de Benchmarking: Patrones para Funcionalidad Predictiva

Criterio de Decisión	Strategy	Factory	Template metod
Flexibilidad para cambiar algoritmos	Alta. Cada algoritmo es una estrategia intercambiable.	Media. Crea instancias, pero no encapsula lógica ejecutable.	Baja. Define un algoritmo fijo con pasos variables.
A/B Testing de modelos	Sí. Fácil comparar resultados de distintas estrategias.	Posible, pero menos directo.	No. Algoritmo central no cambia.
Extensibilidad futura	Alta. Añadir nuevos modelos no rompe el código existente.	Alta, pero no resuelve ejecución dinámica.	Media. Requiere herencia.
Alineación con principios SOLID	Cumple Open/Closed y Single Responsibility.	Cumple Single Responsibility.	Menos flexible ante cambios.

3.4 Tabla de Benchmarking: Patrones para Gestión Offline (PWA)

Criterio de Decisión	Command + Background Sync	Simple Cache (LocalStorage)	Event Sourcing
Resiliencia offline	Alta. Guarda operaciones completas para reintentar.	Baja. No soporta operaciones complejas (ej. ventas con múltiples entidades).	Muy alta, pero compleja.
Sincronización confiable	Sí. Usa Background Sync API del navegador.	No. No hay mecanismo de reintento automático.	Sí, pero con sobrecarga.
Mantenibilidad	Alta. Código claro y modular.	Alta, pero limitado.	Baja. Alta complejidad conceptual.
Integración con arquitectura en capas	Sí. Se integra con Services y Repositories.	Parcial. Solo frontend.	Difícil. Requiere replantear persistencia.

4. Consideraciones Prácticas en la Selección de Patrones de Diseño

Dado que el desarrollo de la Aplicación Web Progresiva (PWA) para la Asociación KALLARI será llevado a cabo por un único desarrollador, se realizó una evaluación crítica de los patrones propuestos inicialmente en el análisis comparativo. Si bien todos los patrones mencionados como CQRS, Event Sourcing, DTOs estrictos, Adapter para integraciones simples, o Redux puro son válidos desde una perspectiva arquitectónica ideal, su implementación simultánea no resulta recomendable en este contexto.

La aplicación de un número elevado de patrones de diseño incrementa significativamente la complejidad del sistema, la curva de aprendizaje y el tiempo de desarrollo. En un entorno de equipo reducido específicamente, un desarrollador individual con conocimientos intermedios en patrones de diseño, priorizar la simplicidad, la rapidez de iteración y la entrega funcional es más estratégico que adherirse a una arquitectura teóricamente óptima pero difícil de mantener en la práctica. Además, muchos de estos patrones resuelven problemas que aún no existen en la fase inicial del proyecto (por ejemplo, alta concurrencia, múltiples fuentes de datos o cambios frecuentes en APIs externas).

Por estas razones, se optó por una estrategia pragmática basada en el principio de "hazlo simple primero, mejóralo después" (You Aren't Gonna Need It – YAGNI). Se seleccionaron únicamente aquellos patrones que ofrecen un equilibrio claro entre beneficio inmediato y esfuerzo razonable, permitiendo construir un MVP funcional sin sacrificar la posibilidad de evolucionar la arquitectura en fases posteriores.

5. Patrones seleccionados y su justificación:

5.1 Frontend:

Se adopta el patrón Container-Presenter, complementado con Zustand para la gestión de estado global. Esta combinación se alinea naturalmente con la filosofía de React, facilita la separación de lógica de presentación y negocio, y permite una integración directa con IndexedDB para soporte offline. Se evita Redux puro por su verbosidad innecesaria en una aplicación de tamaño mediano.

5.2 Backend:

Se implementa una capa de Service sobre Sequelize, sin introducir el patrón Repository en esta etapa inicial. Esto permite encapsular la lógica de negocio (ventas, contabilidad, predicción) de forma clara y testeable, sin añadir abstracciones prematuras. El patrón Strategy sí se aplica en el módulo predictivo, ya que se prevé la evaluación de múltiples algoritmos y la necesidad de compararlos dinámicamente.

5.3 Funcionalidad Offline (PWA):

Se utiliza una estrategia híbrida: operaciones críticas (como ventas) se serializan como objetos simples y se almacenan en IndexedDB, con sincronización manual o automática mediante la Background Sync API. Esto brinda resiliencia offline sin la complejidad del

patrón Command completo, que se considerará solo si la lógica de sincronización se vuelve más compleja.

5.4 Seguridad y Extensibilidad:

Se mantiene el uso de middleware JWT para autenticación por roles, por su simplicidad y eficacia. En cambio, patrones como Adapter o DTOs estrictos se posponen: mientras la integración con el SRI sea estable y la API exponga pocos endpoints, una función encapsulada y objetos de respuesta simples son suficientes.

Esta selección permite iniciar el desarrollo con una base sólida pero ligera, priorizando la entrega de valor funcional a la Asociación KALLARI. La arquitectura está diseñada para ser refactorizable: a medida que surjan necesidades reales (más algoritmos, mayor volumen de datos, cambios en APIs externas), se introducirán los patrones adicionales de forma incremental y justificada.