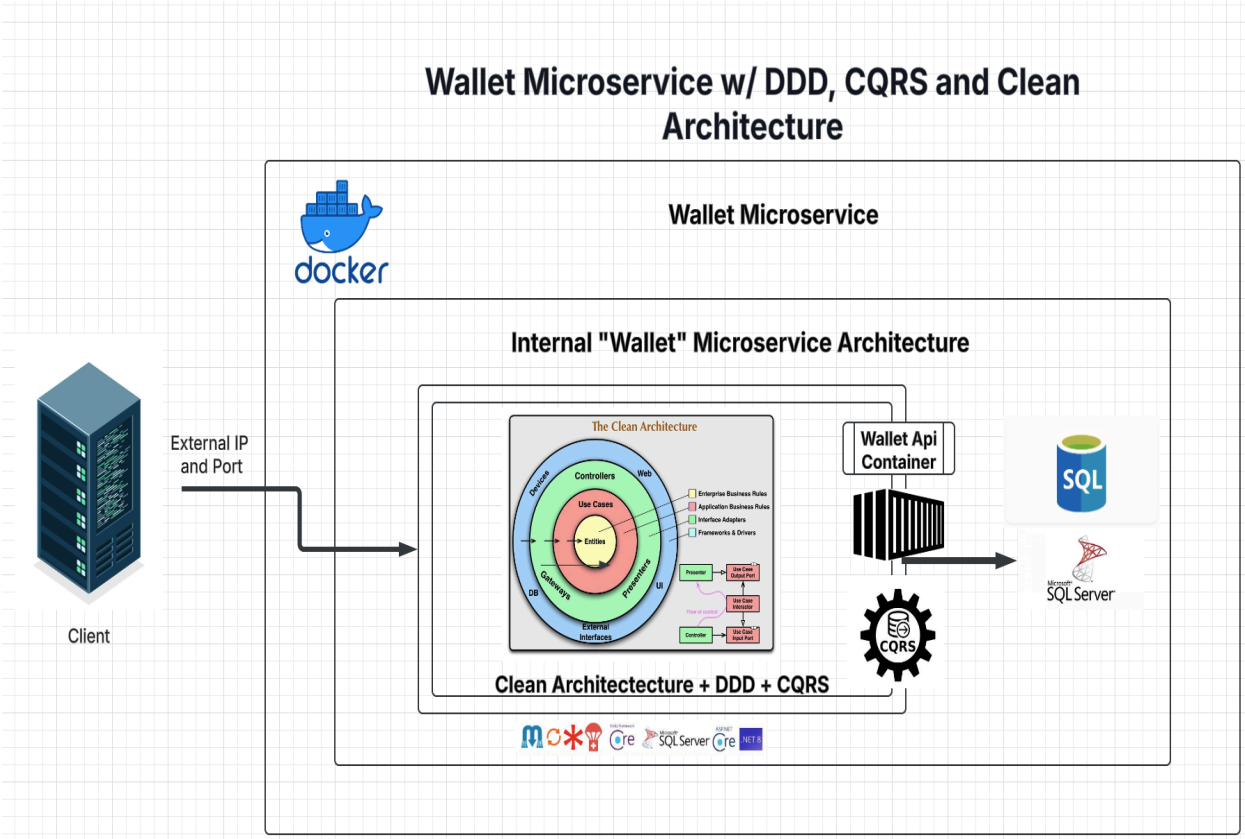
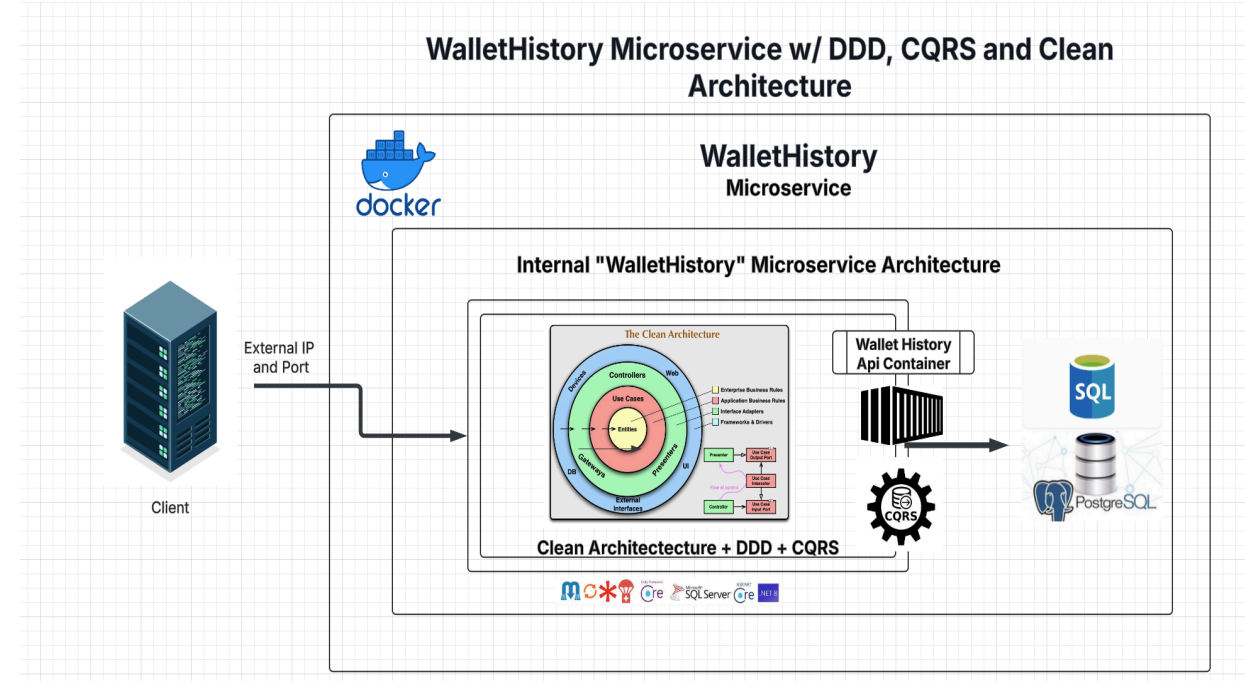
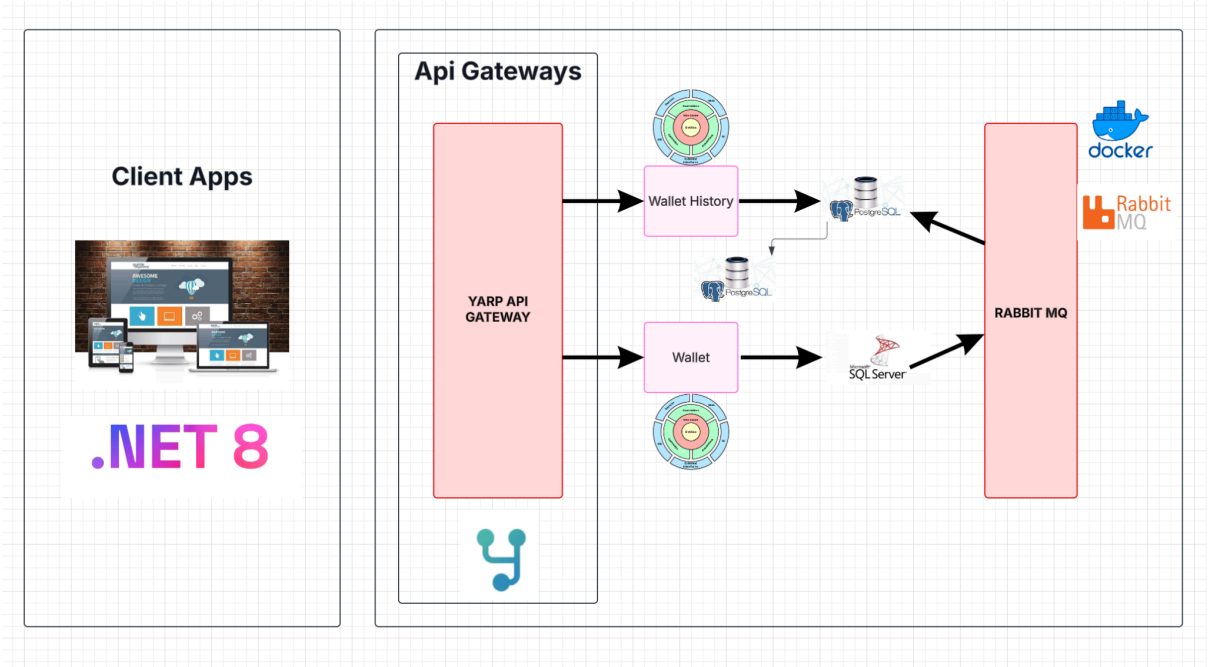

Wallet Microservice w/ DDD, CQRS and Clean Architecture
Wallet Microservice
docker
Internal "Wallet" Microservice Architecture
External IP and Port
Client
The Clean Architecture
Controllers
Use Cases
Entities
Web
UI
Presenters
Gateways
DB
Devices
External Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Wallet Api Container
CQRS
SQL
SQL Server
Clean Architectecture + DDD + CQRS


WalletHistory Microservice w/ DDD, CQRS and Clean Architecture
WalletHistory
Microservice
docker
Internal "WalletHistory" Microservice Architecture
External IP and Port
Client
The Clean Architecture
Controllers
Use Cases
Entities
Web
UI
Presenters
Gateways
DB
Devices
External Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Wallet History Api Container
CQRS
SQL
PostgreSQL
Clean Architectecture + DDD + CQRS



Architectures, Patterns, Libraries and Best Practices

Architectures

- Layered Architecture
- DDD
- Clean Architecture

Patterns & Principles

- SOLID, DI
- CQRS
- Mediator, Proxy, Decorator, Options
- Pub/Sub, Caching
- API Gateway

Databases

- Postgres
- SQL Server

Libraries

- Carter
- Marten
- MediatR
- Mapster
- MassTransit
- FluentValidation

- EF Core

1. ¿Cómo tú implementación puede ser escalable a miles de transacciones?

Para manejar miles de transacciones concurrentes, aplicaría:

- **Balanceo de carga** con un **API Gateway (NGINX, AWS API Gateway, Azure API Management)** para distribuir las solicitudes.
- **Persistencia eficiente** con **Dapper** para consultas rápidas o **Entity Framework Core** con estrategias de indexación y particionamiento en la base de datos.
- **Colas de procesamiento asíncrono** usando **RabbitMQ o Azure Service Bus** para procesar transacciones sin bloquear solicitudes.
- **Caching con Redis o Memcached** para reducir la carga en la base de datos.
- **Base de datos escalable** con PostgreSQL (con particionamiento y réplicas) o una solución NoSQL como DynamoDB/CosmosDB para mayor concurrencia.

2. ¿Cómo tú implementación asegura el principio de idempotencia?

Para evitar transacciones duplicadas, implementaría:

- **Idempotency Keys**: Cada solicitud incluiría un identificador único (UUID) para asegurar que se procese solo una vez.
- **Registros de transacciones en la BD**: Se almacenaría un historial de operaciones con validaciones de duplicidad.
- **Manejo adecuado de concurrencia**: Uso de bloqueos optimistas con **row versioning** o transacciones ACID en bases SQL.
- **Mensajería con deduplicación**: Uso de **RabbitMQ con mensajes persistentes o Azure Service Bus con detección de duplicados**.

3. ¿Cómo protegerías tus servicios contra ataques de DoS, SQL Injection y CSRF?

- **DoS/DDoS**:
    - Limitar solicitudes por IP con **Rate Limiting (ASP.NET Middleware, API Gateway, WAF en AWS/Azure)**.
    - **CDN como CloudFront/Azure Front Door** para mitigar tráfico malicioso.
- **SQL Injection**:
    - Uso de **ORMs seguros (EF Core/Dapper con consultas parametrizadas)**.
    - Políticas de **Principle of Least Privilege (PoLP)** en la base de datos.
- **CSRF**:
    - Implementación de **Tokens CSRF** en endpoints críticos.
    - **CORS correctamente configurado** para restringir orígenes no confiables.
    - Uso de **JWT o OAuth2 con tokens de refresco**.

4. ¿Cuál sería tu estrategia para migrar un monolito a microservicios?

1. **Identificación de módulos independientes**: Extraer servicios como **billetera y transacciones** en bases de código separadas.
2. **API Gateway como punto de entrada**: Para enrutar solicitudes sin modificar clientes actuales.

3. **Desacoplamiento de la base de datos**:
    - Si es SQL → Estrategia de **Database-per-Service** con sincronización de eventos.
    - Si es NoSQL → Separación de datos en **DynamoDB, CosmosDB o MongoDB**.
4. **Mensajería asíncrona**: Uso de **Event-Driven Architecture con RabbitMQ o Azure Event Grid** para comunicación entre servicios.
5. **Despliegue progresivo**: Usar **Docker/Kubernetes** para manejar los servicios sin afectar la disponibilidad del sistema.