ARQUITECTURA PROYECTO GESTIÓN DE EVENTOS COMUNITARIOS

Arquitectura Elegida: Arquitectura Hexagonal

¿Qué es?

La arquitectura hexagonal separa el núcleo de la aplicación (reglas y casos de uso) de todo lo externo (web, base de datos, email, colas, etc.).

La arquitectura hexagonal separa el núcleo de negocio de los detalles de infraestructura mediante puertos (interfaces) y adaptadores (implementaciones). Los controladores HTTP y el frontend son adaptadores entrantes que invocan casos de uso (puertos de entrada). La persistencia, emails y servicios externos son adaptadores salientes que implementan puertos de salida. Las dependencias apuntan siempre hacia el centro (aplicación/dominio), lo que permite cambiar tecnologías, mejorar testabilidad y evolucionar el sistema sin tocar las reglas de negocio.

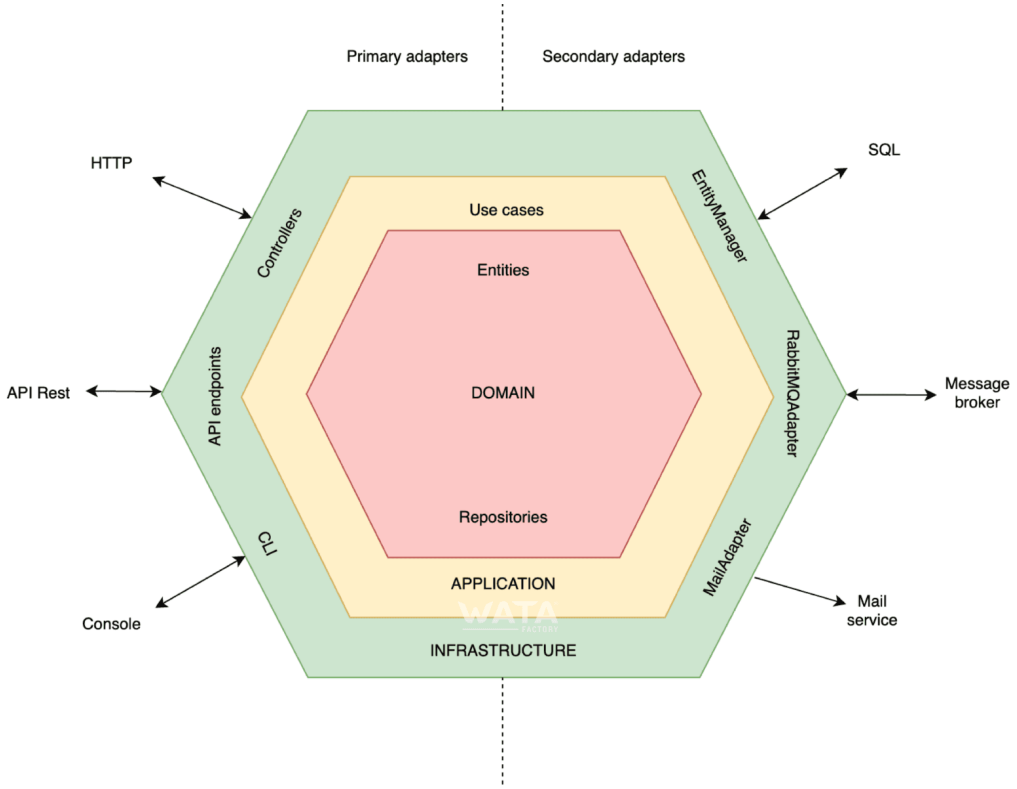
Principios esenciales

1. **Independencia del dominio**: las reglas de negocio no dependen de frameworks ni de detalles técnicos.
2. **Dependencias hacia adentro**: los adaptadores conocen los puertos, pero el núcleo no conoce los adaptadores.
3. **Sustituibilidad**: puedes cambiar una BD, proveedor de email o protocolo sin tocar el núcleo.
4. **Testabilidad**: los casos de uso se prueban con dobles (mocks/fakes) de los puertos, sin levantar servidores ni BD.

Piezas y cómo encajan

1. **Dominio**: modelos y reglas del negocio (clases como Evento, Inscripcion, políticas, validaciones).
2. **Aplicación (Casos de uso)**: orquestan el flujo (crear evento, inscribir ciudadano, aprobar presupuesto). Exponen Puertos de Entrada (interfaces) que invocarás desde el mundo exterior.
3. **Puertos de Salida**: interfaces que la aplicación necesita para realizar acciones externas (guardar, enviar email, publicar calendario).
4. **Adaptadores Entrantes**: reciben peticiones del exterior y llaman a los puertos de entrada (ej. controladores REST).
5. **Adaptadores Salientes**: implementan los **puertos de salida** (repositorios JPA, clientes HTTP para Finanzas, SMTP, etc.).

El hexágono (núcleo) en el centro; alrededor, conectores que se pueden enchufar o desenchufar.



Regla de dependencias (la ley del hexágono)

1. Adaptadores → Aplicación → Dominio (nunca al revés).
2. El **Dominio** no importa nada de Spring, JPA, HTTP, etc.
3. La **Aplicación** depende del Dominio y **de interfaces** (puertos), no de implementaciones.
4. Los **Adaptadores** dependen de los puertos para implementar el "cómo".

¿Cómo se ve en Java Spring Boot + React?

1. **Frontend (React)**: es un adaptador entrante más: hace llamadas HTTP a la API.
2. **Controllers Spring**: adaptadores entrantes; traducen HTTP ↔ DTOs ↔ **puertos de entrada**.
3. **Casos de uso (services de aplicación)**: implementan la lógica de orquestación. Usan **puertos de salida** (interfaces) para persistencia, email, etc.
4. **Repositorios JPA / WebClient**: adaptadores salientes; implementan puertos de salida.
5. **BD / SMTP / Proveedores externos**: detalles reemplazables.

Flujos típicos en el sistema (“Gestión de Eventos Comunitarios”)

A) Crear un evento

1. POST /eventos (Controller) → valida DTO y llama RegistrarEvento (puerto de entrada).
2. RegistrarEvento aplica reglas (fechas válidas, capacidad ≥ 0).
3. Llama a EventoRepository (puerto de salida) para guardar.
4. Devuelve resultado al Controller → responde HTTP.

B) Inscribir ciudadano a un evento

1. POST /inscripciones (Controller) → llama GestionarInscripcion.
2. Caso de uso verifica cupo/estado del evento.
3. Llama a InscripcionRepository y, si procede, a EmailSender (confirmación).
4. Responde al Controller.

C) Decisión de Finanzas (aprobación/rechazo)

1. POST /finanzas/webhook (Controller) → llama ProcesarDecisionFinanzas.
2. Caso de uso actualiza estado del evento y notifica por EmailSender/PushSender.
3. Persiste cambios vía repos y responde.

Beneficios concretos

- **Cambiar tecnología sin dolor**: ¿Postgres hoy y Mongo mañana? Solo cambias el adaptador.
- **Pruebas rápidas**: simulas EventoRepository o EmailSender con mocks sin levantar nada.
- **Evolutivo**: puedes añadir canales (GraphQL, CLI, Jobs) sin tocar la lógica central.
- **Menos acoplamiento**: cada cosa en su lugar, dependencias ordenadas.

Qué no es hexagonal (errores comunes)

- Poner JPA annotations y lógica HTTP dentro de las clases de negocio.
- Que los casos de uso llamen directamente a JpaRepository (eso es detalle, debe ir en adaptadores).
- Controllers grandes que contengan reglas de negocio.

Decisiones prácticas (Spring Boot)

- **Paquetes**:
    - …domain (clases del negocio)
    - …application (casos de uso + puertos)
    - …infrastructure (JPA entities, repos, HTTP clients, email, mappings)
    - …api (controllers, DTOs, validaciones HTTP)

- **DTOs vs Modelo**:
    - DTOs (API) ≠ Clases de dominio. Usa mapeadores para traducir.
- **Transacciones**: anotar transacciones en **aplicación** o en adaptadores con cuidado (no en dominio).
- **Validación**:
    - Validación sintáctica en controllers (Bean Validation).
    - Reglas de negocio en aplicación/dominio.

Porque elegimos Arquitectura Hexagonal

- **Propósito**: por qué hexagonal (independencia, testabilidad, reemplazo de tecnología).
- **Capas y roles**: dominio, aplicación (puertos), adaptadores (entrantes/salientes).
- **Regla de dependencia**: hacia adentro, jamás al revés.
- **Trazabilidad**: mapea los casos de uso (crear evento, inscribir, aprobar finanzas) a puertos de entrada y salidas implicadas.
- **Estrategia de pruebas**: casos de uso con dobles de puertos; adaptadores con tests de integración.
- **Decisiones técnicas**: empaquetado, DTOs, transacciones, mapeo, logging/observabilidad.