RealEstate – Documento de Arquitectura

Autor: David Navarro | Fecha: 2025-09-01

0) Resumen ejecutivo

Aplicación **RealEstate** (API .NET 8) para gestionar propiedades, dueños, imágenes y trazas de precio. Se prioriza **rendimiento**, **simplicidad** y **claridad** para una prueba técnica. Arquitectura en capas + CQRS ligero, EF Core con tipos **ObjectValue** para direcciones, endpoints REST, almacenamiento de imágenes local con URLs públicas.

---

1)Contexto & objetivos

- **Objetivo**. Crear una API clara y eficiente para una empresa inmobiliaria de gran tamaño que permita consultar información de propiedades en Estados Unidos, usando la base de datos existente. La API expondrá servicios para listar y filtrar con paginación, obtener el detalle (dueño, dirección, imágenes, trazas de precio), actualizar el precio registrando historial y gestionar imágenes mediante URLs.
- **Scope**: backend (REST), sin UI. Seguridad mínima (abierta por simplicidad de prueba).
- **Enfoque de arquitectura**: **DDD**: Domain Driven Desing con Entidades y **ObjectValue** (Value Objects), Repositorios, Unit of Work y CQRS ligero.
- **Requisitos clave**:
    - Filtros por Address/Price/Year con paginación.
    - Detalle con Owner, Images y Traces.
    - Carga/lectura de imágenes vía URL pública (no bytes en el detalle).

---

2)Decisiones de Arquitectura (ADRs)

Las decisiones de arquitectura son el hilo conductor del proyecto. Aquí explicamos qué elegimos, por qué lo hicimos y qué implicaciones tiene. La idea es que cualquier lector pueda entender el norte técnico sin bucear en el código.

- **ADR-001 – Estilo arquitectónico: DDD + CQRS ligero**. Optamos por separar claramente el **dominio** (entidades y ObjectValue), la **aplicación** (casos de uso), la **infraestructura** (EF Core, almacenamiento) y la **presentación** (API). Los comandos y queries son finos y directos, sin un bus externo ni orquestaciones complejas. La razón principal es la claridad, la facilidad para probar y la posibilidad de crecer hacia patrones más sofisticados si el producto lo pide. Descartamos un “servicio de aplicación” monolítico porque dificulta el testeo y tiende a acoplar demasiado.
- **ADR-002 – Persistencia con EF Core (SqlServer)**. Elegimos EF Core para reflejar el modelo de dominio con naturalidad, mapear **ObjectValue** (direcciones) en la misma

tabla y usar conversores DateOnly. Frente a alternativas como Dapper, ganamos productividad y expresividad a costa de un rendimiento *teórico* algo menor en lecturas extremas, que mitigamos con índices, proyecciones y AsNoTracking().

- **ADR-003 – Imágenes: URL en JSON; bytes por endpoint separado**. En las respuestas devolvemos **URLs públicas** y evitamos embutir bytes en el DTO de detalle. Esto reduce payloads, mejora la caché del navegador/CDN y permite migrar fácilmente a S3/Azure sin romper contratos.
- **ADR-004 – Unit of Work con transacciones explícitas**. Implementamos un Unit of Work basado en DbContextTransaction para garantizar consistencia entre operaciones coordinadas (por ejemplo, actualizar una propiedad y registrar su traza de precio). Evitamos EnableRetryOnFailure() o, si se requiere, envolvemos la transacción en CreateExecutionStrategy().
- **ADR-005 – Sin migraciones, EnsureCreated() + seed**. En una prueba técnica priorizamos ir al grano. Creamos el esquema con EnsureCreated() y cargamos datos de ejemplo. Es una decisión consciente: para producción se incorporarían **migraciones** y versionado del esquema.
- **ADR-006 – Orden estable de colecciones**. Las trazas se entregan con los **eventos más recientes primero**; las imágenes, por id (o una futura columna ImageOrder). La intención es dar una UX determinista y facilitar los tests.

---

3)Performance & escalabilidad

El rendimiento se ataca desde el diseño. En lectura usamos AsNoTracking() para evitar costos de *tracking*, y limitamos los Include a lo estrictamente necesario. El detalle de una propiedad se recupera en un solo viaje con AsSingleQuery(); si el tamaño de las colecciones creciera mucho, podemos cambiar a AsSplitQuery() para reducir consumo de memoria.

Los filtros y la paginación se ejecutan en la base de datos, apoyados por índices pensados para los patrones de acceso reales (precio, año, ciudad/estado). También cuidamos el tamaño de las respuestas: las imágenes no viajan como bytes, sino como URLs cacheables, lo que descarga al servidor y habilita CDN. Finalmente, instrumentamos los casos de uso con métricas y logs de latencia para detectar regresiones.

---

4)Roadmap / trabajos futuros

- Autenticación JWT + roles; rate limiting en imágenes.
- ImageOrder, miniaturas (resize) y CDN/S3 con URLs firmadas.
- Migraciones EF; validaciones con FluentValidation.
- Observabilidad (OpenTelemetry), métricas y tracing.
- Ordenamiento configurable en listados.

---

5)Estructura de solución

RealEstate.sln

```
├─ RealEstate.Api              // ASP.NET Core Web API
├─ RealEstate.Application      // Casos de uso, CQRS, contratos/DTOs
├─ RealEstate.Domain           // Entidades de dominio (POCOs)
├─ RealEstate.Infrastructure   // EF Core, repos, DbContext, DI, storage
└─ RealEstate.Tests            // Pruebas: controllers, handlers y repos
```

La solución refleja el enfoque **DDD**: el **dominio** no conoce a la infraestructura, y la **aplicación** define contratos (repositorios y Unit of Work) que la infraestructura implementa. La **API** es una capa delgada que valida lo mínimo, hace *model binding* y delega el comportamiento a los *handlers*.

El proyecto **RealEstate.Tests** prueba el sistema desde tres ángulos. Los *handlers* se validan con pruebas unitarias usando *mocks* de repositorios y también con integración sobre SQLite in-memory para asegurar que los filtros, la paginación y las proyecciones funcionan. Los **repositorios** se prueban con consultas reales para garantizar que los índices se aprovechan y que los órdenes son estables. Los **controladores** se ejercitan con WebApplicationFactory para comprobar rutas, códigos de estado (201/404/400), CreatedAtAction y *model binding*.

---

6)Modelo de dominio

Entidades

- Owner { IdOwner, Name, Address?, Photo?, Birthday? }
- Property { IdProperty, Name, Address?, Price, CodeInternal?, Year?, IdOwner, Owner, Images, Traces }
- PropertyImage { IdPropertyImage, IdProperty, File, Enabled }
- PropertyTrace { IdPropertyTrace, IdProperty, DateSale, Name, Value, Tax }

Value Objects (en la **misma tabla** como *ObjectValue types*):

- OwnerAddress { Street, City, State, ZipCode } (opcional)
- PropertyAddress { Street, City, State, ZipCode } (opcional)

Fechas

- DateOnly en Owner.Birthday y PropertyTrace.DateSale con **conversores** EF a date.

---

7)Persistencia & base de datos

Usamos **SQL Server** en contenedor local y **EF Core** como ORM. El esquema es intencionalmente simple: Owner, Property, PropertyImage y PropertyTrace. Las direcciones

(OwnerAddress y PropertyAddress) se modelan como **ObjectValue** y se guardan en la misma tabla de sus agregados para mantener cohesión y evitar *joins* innecesarios.

Los **índices** responden a cómo consultamos los datos. El de **precio** en Property (Price) acelera rangos típicos de búsquedas(MinPrice/MaxPrice). El de **año** (Property(Year)) mejora filtros por igualdad y ordenaciones. Para la ubicación, el índice compuesto en **ciudad/estado** (Property(Address.City, Address.State)) cubre búsquedas frecuentes por zona; con patrones LIKE 'Miami%' el motor puede hacer busqueda por prefijo. En imágenes, el índice PropertyImage(IdProperty, Enabled) apunta al recorrido caliente de listar fotos activas de una propiedad. Para trazas, PropertyTrace(IdProperty, DateSale) permite leer el histórico ya preparado para ordenar por fecha.

La creación del esquema se hace con EnsureCreated() y la carga inicial con un *seeder* de Miami. En **Program.cs**:

await app.Services.CreateAndSeedAsync(reset: false); // pon true la 1ª vez para recrear todo

El parámetro reset define la estrategia: con **true** se elimina y crea la base desde cero (solo en local o la primera vez y con permisos adecuados), mientras que con **false** se asegura el esquema y se evita duplicar datos si ya existen. Si el servidor deniega CREATE DATABASE, se debe crear la base con un usuario privilegiado y otorgar permisos db_owner al usuario de la aplicación.

---

8)Acceso a datos & Unit of Work

- **Patrón**: Repositorios finos + **Unit of Work EF**.
- **Transacciones**: EfUnitOfWork con Begin/Commit/Rollback **o** ExecuteInTransactionAsync (si se habilita *execution strategy* con reintentos).
- **Lecturas**: AsNoTracking() por defecto.
- **Detalle**: Include(Owner, Images, Traces) + AsSingleQuery(); orden de colecciones en memoria.

Repos

- IOwnerRepository { GetAsync, Exists }
- IPropertyRepository { CreateAsync, GetPropertyByFiltersAsync, GetDetailAsync, ChangePriceAsync, AddImageAsync, GetImagesAsync }

---

9)CQRS ligero (Application)

- **Queries**
    - GetPropertyListQuery → PagedDtos<PropertySummaryDto> (filtros PropertyFilters con AddressDto)
    - GetPropertyDetailsQuery → PropertyDetailDto

- **Commands**
    - CreatePropertyBuildingCommand → PropertyDetailDto
    - UpdatePropertyCommand → bool
    - ChangePriceCommand → bool
    - AddPropertyImageCommand → PropertyImageDto

---

10)DTOs & mapeo (AutoMapper)

- PropertyDetailDto incluye OwnerDto, AddressDto?, Images (Url), Traces.
- PropertySummaryDto con Owner y Address como string (ToString del VO) o OwnerDto según perfil.
- **Profiles**:
    - OwnerProfiles: Owner → OwnerDto (Address VO → DTO).
    - PropertyDetailProfile: Property → PropertyDetailDto (Owner, Address, Images, Traces).
    - PropertySummaryProfile: Property → PropertySummaryDto (Owner.Name y Address.ToString()).
    - PropertyCommandProfile: Create/Update*Command → Property (ignora navs, mapea VO Address).

---

11)API (endpoints)

PropertiesController

- GET /api/properties?Page=&PageSize=&Address.Street=&City=&State=&ZipCode=&MinPrice=&MaxPrice=&Year= → paginado.
- GET /api/properties/{id} → detalle (incluye **URLs** de imágenes, no bytes).
- PATCH /api/properties/{id}/price → cambia precio, agrega PropertyTrace.
- PUT /api/properties/{id} → actualiza propiedad (si cambia precio, genera trace).
- POST /api/properties → crea propiedad (opcional: imágenes iniciales).
- POST /api/properties/{id}/images → agrega imagen (multipart **o** JSON base64).
- GET /api/properties/{id}/images → **lista imágenes** (JSON con **Url**).
- GET /api/properties/{id}/images/{imageId}/content → (opcional) **bytes** con PhysicalFile + cache.

Swagger

- Model binding de filtros con **propiedades públicas** ([FromQuery] en Query object). Campos como *textbox*.
- Consumes("multipart/form-data") en el endpoint que recibe IFormFile.

---

12)Imágenes: almacenamiento & URLs

- **Almacenamiento actual**: sistema de archivos local en wwwroot/uploads/properties/{id}/....
- **Abstracción**: IImageStorage (actual: LocalImageStorage), preparada para cambiar a S3/Azure.
- **Exposición**:
  - En detalles/listas retornamos **URL pública** (no bytes).
  - IImageUrlBuilder compone https://{host}/uploads/... (o firma S3 más adelante).
- **Static files**: app.UseStaticFiles().

---

13)Validación, errores y logging

- Validaciones simples en controller (Name/Address/Price/IdOwner > 0).
- NotFound para recursos ausentes, BadRequest para payload inválido, 201 CreatedAtAction en creación.
- Logs informativos en consultas (latencias) y comandos.

---

14)Seguridad (simplificada para la prueba)

- **Actual**: sin auth.
- **Futuro**: JWT (Bearer), autorización por rol; URLs de imagen **protegidas** vía endpoint content o URLs firmadas temporales en CDN.

---

15)Pruebas

- Framework: **xUnit** + FluentAssertions + Moq + Microsoft.NET.Test.Sdk + xunit.runner.visualstudio.
- **InMemory/SQLite** para integración de repos.
- Pitfall resuelto: *“The instance of entity type 'Owner' cannot be tracked...”* → evitar adjuntar dos instancias con misma PK (usar solo FK o reutilizar instancia). db.ChangeTracker.Clear() entre bloques si hace falta.

16)Operación & despliegue

- **SQL Server**: contenedor local. Usuario RealStateApp con db_owner (o sysadmin en dev, no recomendado en prod).
- **Inicialización**: EnsureCreated() + DbInitializer.CreateAndSeedAsync.
    - En Program.cs:
    - await app.Services.CreateAndSeedAsync(reset: false); // pon true la 1ª vez para recrear todo
    - **Uso**:
        - reset: true → borra y crea la base **desde cero** y luego hace seed (solo en dev/local o la primera vez).
        - reset: false → asegura esquema y hace seed **solo si** no hay datos (idempotente en ejecuciones normales).
        - No usar reset: true en entornos con datos reales.
- **Configuración**: appsettings.json con DefaultConnection.
- **Static files**: wwwroot presente; carpetas de uploads creadas por el storage.

---

17)Cómo ejecutar (dev)

1. Levantar SQL Server (docker) y crear DB/usuario si hace falta.
2. Configurar appsettings.json con DefaultConnection.
3. dotnet run en RealEstate.Api.
4. Swagger en /swagger.
