**Documentación del Proyecto: SpecuLabTeam**

**Sistema de Gestión de Propiedades SpecuLabTeam**

****

**Versión:** 1.0.0

**Autor(es):** Alejandro, Jorge, Tomás y Marco Salas.

**Fecha de Creación:** 09 de septiembre de 2025

**Url de Aplicación**: <https://speculab.netlify.app/>

**Repositorio de GitHub:** <https://github.com/alexgn12/SpecuLabTeam.git>

**Índice**

1. Resumen del Proyecto
2. Introducción
3. Tecnologías y Metodología

3.1. Contribuciones al equipo

4. Herramientas Utilizadas

5. Comunicación y Colaboración

1. Documentación del Backend (.NET API)

6.1. Arquitectura General

6.2. Estructura de Carpetas del Backend

6.3. Entidades y Modelos de Datos (DTOs)

6.4. Controladores (API Endpoints)

6.5. Servicios y Lógica de Negocio

6.6. Persistencia de Datos (Entity Framework Core)

6.7. Autenticación y Seguridad

6.8. Documentación de API (Swagger)

1. Documentación del Frontend (Angular)

7.1. Arquitectura y Tecnologías

7.2. Flujos de Usuario y Componentes Clave

1. Guía de Instalación y Despliegue

8.1. Instalación Local

8.2. Despliegue (CI/CD)

9. Login y Autenticación de Usuarios

       10. Manejo de Excepciones y Errores

11. Implementación de SignalR

**1. Resumen del Proyecto**

**SpecuLabTeam** es una plataforma integral diseñada para optimizar la gestión y el seguimiento de propiedades inmobiliarias y las solicitudes asociadas a ellas. Este sistema proporciona una solución robusta para administrar edificios, apartamentos, transacciones y el ciclo de vida de las solicitudes, ofreciendo una experiencia de usuario intuitiva y eficiente para los gestores de propiedades.

El proyecto utiliza una arquitectura moderna basada en una API monolítica bien estructurada, con un backend desarrollado en .NET 8 y un frontend interactivo en Angular. La integración continua y el despliegue se gestionan a través de GitHub Actions, asegurando un ciclo de desarrollo ágil y eficiente.

**2. Introducción**

El equipo de desarrollo está formado por cuatro integrantes con experiencia complementaria y multidisciplinar. Nuestro objetivo ha sido crear una aplicación robusta y eficiente siguiendo las mejores prácticas de la industria.

A lo largo del desarrollo del proyecto, nuestro equipo ha puesto en práctica los valores de la metodología Agile, utilizando Scrum para gestionar las iteraciones y entregas parciales. El seguimiento de tareas, asignación de historias y sprints se ha realizado a través de Azure DevOps, mientras que el control de versiones y colaboración en el código fuente ha estado centralizado en un repositorio de GitHub. El desarrollo se ha realizado en local, desplegando a producción solo una vez validada la calidad del producto. Las reuniones diarias (dailys) y los encuentros regulares con otros equipos han favorecido un trabajo transparente y colaborativo. Cada integrante ha realizado aportaciones valiosas y diferenciadas, complementando el éxito del proyecto.

**3. Metodología de Trabajo**

El desarrollo de SpecuLabTeam se llevó a cabo utilizando la metodología ágil Scrum. La gestión del proyecto, la planificación de sprints y el seguimiento del backlog se realizaron mediante Azure DevOps. Esta aproximación fomentó la colaboración, permitió una rápida adaptación a los cambios y aseguró una entrega constante de valor.

**3.1. Contribuciones al Equipo**

Tomás Orejas: En su rol de Product Owner y desarrollador full-stack, ha realizado una contribución integral al proyecto, participando activamente en el desarrollo del front-end y back-end. Destacó especialmente en la mejora del front-end con Angular, integrando nuevos componentes, optimizando la lógica de la página y desarrollando estilos para una web completamente responsive. Además, asumió el rol de Product Owner, definiendo y priorizando tareas para asegurar la alineación con los objetivos del producto. Además, implementó un agente de inteligencia artificial conectado a la base de datos, capaz de analizar la información y extraer conclusiones relevantes mediante la API de OpenAI.

• Jorge Martinez : En su rol de coordinador y desarrollador full-stack, ha liderado la gestión interna del equipo y la comunicación con otros grupos, facilitando la coordinación general. Ha contribuido activamente al desarrollo tanto del backend como del frontend, destacando la implementación del registro de logs, la configuración de AutoMapper y el diseño de servicios en segundo plano para edificios, así como la lógica principal de la sección de peticiones en el frontend. Además, se ha encargado de implementar la integración y conexión de la aplicación con los sistemas desarrollados por otros equipos.

• Alejandro Guerra: En su rol de Scrum Master y desarrollador backend, ha impulsado la organización y eficiencia del equipo, facilitando la comunicación entre los integrantes y apoyando en los distintos procesos de desarrollo de la aplicación. Ha centrado sus esfuerzos en la arquitectura del backend, destacando la introducción de MediatR bajo un enfoque de Clean Architecture, la implementación de un repositorio genérico y la colaboración en el background service encargado de la comunicación con los demás equipos.

Además, ha desarrollado el sistema de notificaciones en tiempo real mediante SignalR, ha contribuido a la implementación de la autenticación con JWT en el login y ha participado en la integración de estas funcionalidades en el frontend, garantizando una experiencia más fluida y segura para los usuarios.

• Marco Salas: En su rol de desarrollador full-stack y administrador de bases de datos, realizó las siguientes contribuciones al equipo:

• Desarrollo completo del módulo de Historial: Implementó tanto el backend (.NET 8 API) como el frontend (Angular) para la gestión y visualización del historial de solicitudes, asegurando la correcta trazabilidad de los cambios de estado y operaciones realizadas.

• Gestión de la base de datos: Se encargó de la creación de la primera migración, así como de la edición y subida de la estructura inicial de la base de datos, facilitando la persistencia y el acceso eficiente a los datos del sistema.

• Documentación: Elaboró la documentación técnica y funcional del proyecto, incluyendo la descripción de la arquitectura, los flujos de trabajo, la guía de instalación y despliegue, y la explicación detallada de los endpoints y módulos principales.

• Gestión de excepciones y logging: Implementó un manejo centralizado de excepciones en la API, asegurando respuestas consistentes y seguras ante errores. Aplicó el patrón Result para encapsular los resultados de las operaciones, diferenciando claramente entre respuestas exitosas y fallidas, y facilitando la propagación de errores de forma controlada. Además, integró Serilog como sistema de logging estructurado, permitiendo el registro detallado de eventos, errores y trazas de ejecución, lo que mejora la observabilidad y el diagnóstico de problemas en el sistema.

**4. Herramientas Utilizadas**

• Azure DevOps:

Usado para monitorizar el estado de las tareas, asignarlas a cada miembro y realizar el seguimiento de sprints y entregas.

• GitHub:

Repositorio centralizado de código, donde cada integrante realizó sus aportaciones usando ramas y pull requests para fomentar la colaboración y el control de versiones.

• Desarrollo Local:

El trabajo principal se realizó en entornos locales de desarrollo hasta que la aplicación estuvo lista para pasar a producción, garantizando calidad en cada etapa.

**5. Comunicación y Colaboración**

• Dailys:

Reuniones diarias breves para compartir avances, identificar posibles bloqueos y coordinar el trabajo del equipo.

• Sincronización con otros equipos:

Además de las reuniones internas, se organizaron encuentros regulares con equipos de otras áreas del proyecto para asegurar alineación global y resolver dependencias técnicas.

**6. Documentación del Backend (.NET API)**

El backend de SpecuLabTeam es una API RESTful desarrollada con ASP.NET Core (.NET 8). Es el núcleo del sistema, gestionando la lógica de negocio, la persistencia de datos y exponiendo los servicios para el frontend Angular.

**6.1. Arquitectura General**

El backend sigue un patrón de **arquitectura en capas**, separando claramente las responsabilidades:

* **Capa de Presentación:** Controladores API que gestionan las peticiones HTTP.
* **Capa de Lógica de Negocio:** Servicios que implementan la lógica del negocio.
* **Capa de Acceso a Datos:** Repositorios que interactúan directamente con la base de datos.
* **Capa de Dominio:** Entidades que representan los objetos de negocio.

+---------------------+

| Frontend (Angular)|

+---------------------+

|

v (HTTP/REST API)

+---------------------+

| Backend (.NET 8 API)|

+---------------------+

|

v (ORM/EF Core)

+---------------------+

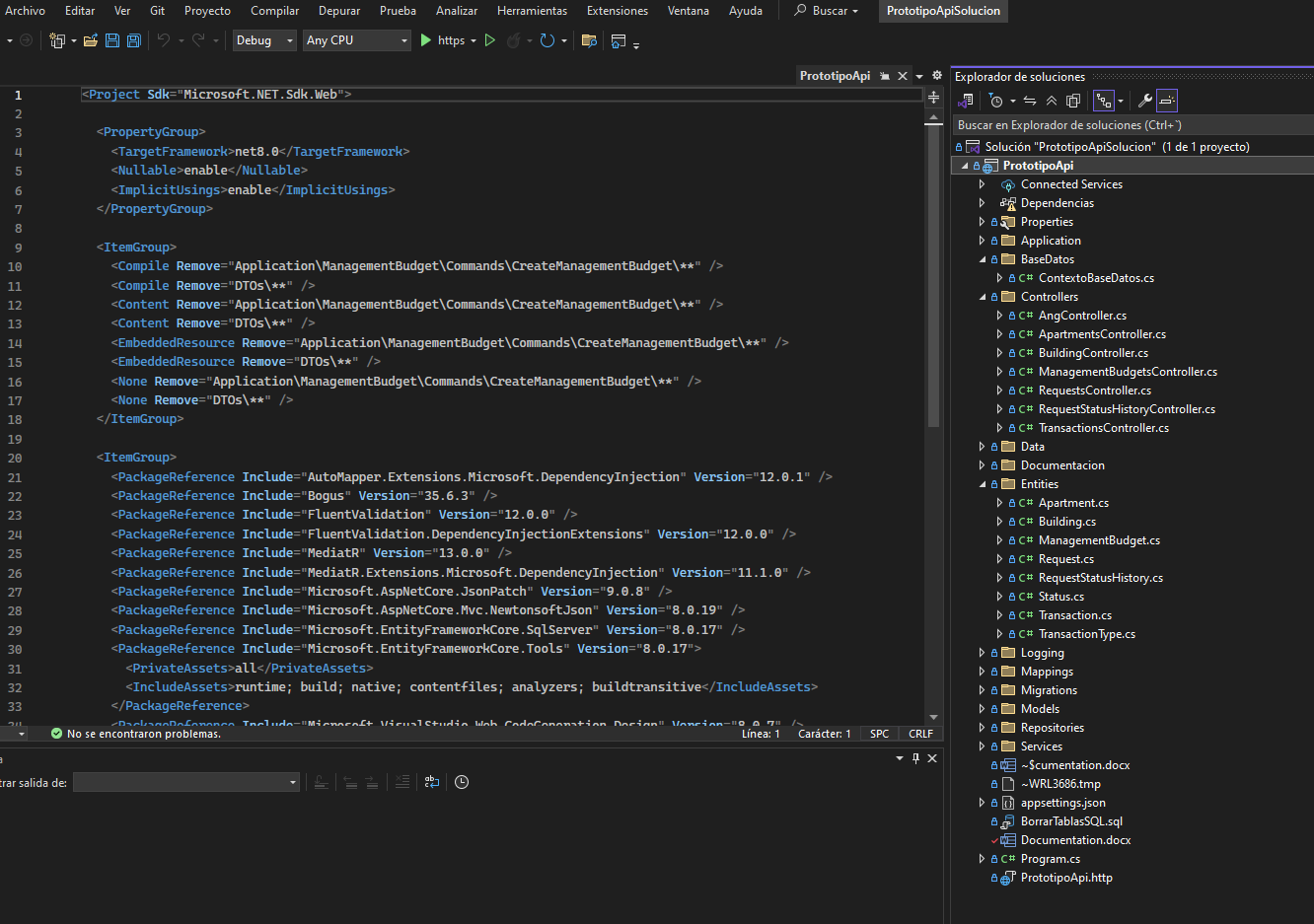
| Base de Datos |

+---------------------+

**6.2. Estructura de Carpetas del Backend**

El proyecto **PrototipoApi** está organizado de la siguiente manera:

* **Application**: Contiene la lógica de negocio principal, DTOs y mapeos.
* **BaseDatos**: Contexto de Entity Framework Core y migraciones.
* **Controllers**: Los controladores de la API que exponen los *endpoints*.
* **Data**: Clases para inicialización de datos (seeders) y ejemplos.
* **Documentacion**: Archivos de documentación interna (PRD, etc.).
* **Entities**: Definición de los modelos de base de datos.
* **Logging**: Implementación de la gestión de logs (Serilog).
* **Mappings**: Clases de mapeo entre entidades y DTOs (ej. AutoMapper).
* **Migrations**: Archivos generados por Entity Framework Core para la gestión del esquema de la base de datos.
* **Models**: DTOs (Data Transfer Objects) para las peticiones y respuestas de la API.
* **Repositories**: Interfaces e implementaciones para el acceso a datos.
* **Services**: Lógica de negocio específica y operaciones de alto nivel.

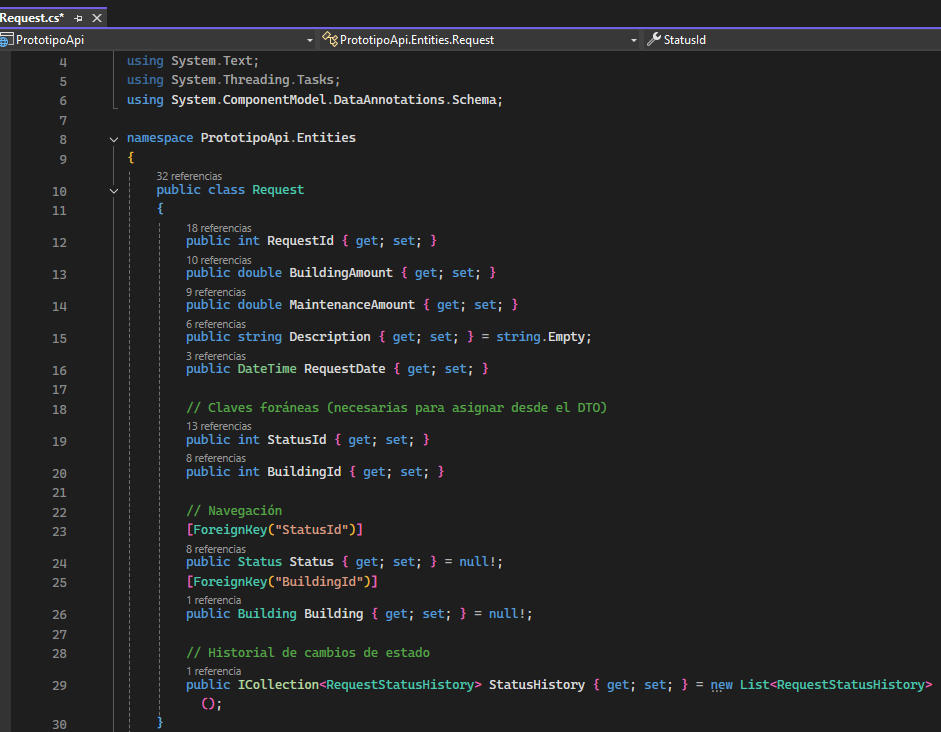


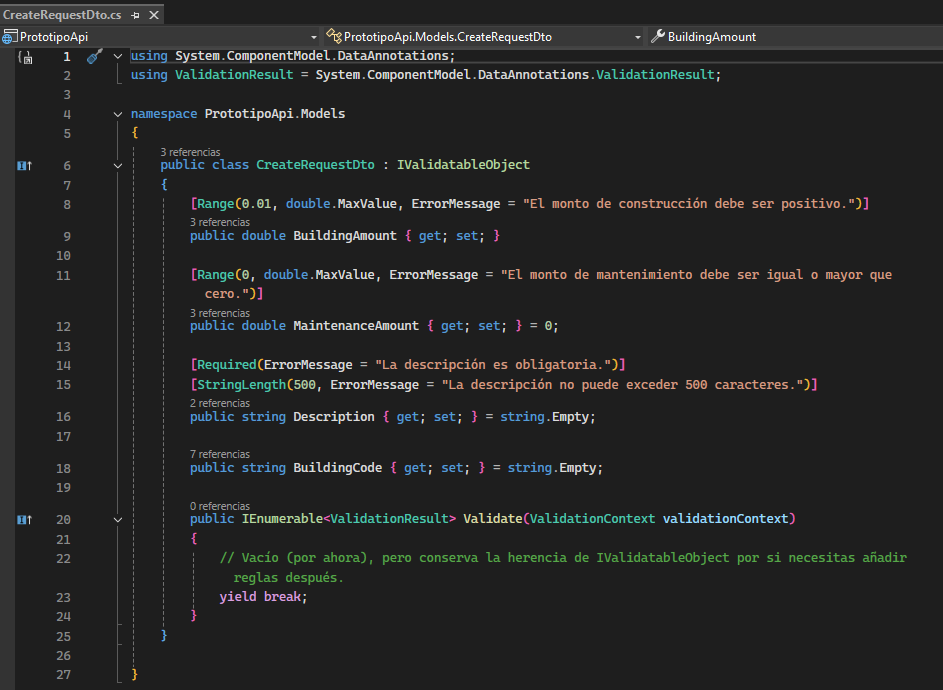
Captura 1: Estructura del proyecto PrototipoApi en el Explorador de soluciones.

**6.3. Entidades y Modelos de Datos (DTOs)**

Las **Entidades** (Entities) representan la estructura de los objetos en la base de datos. Los **DTOs** (Models) son versiones simplificadas o adaptadas de estas entidades, utilizadas para el envío y recepción de datos a través de la API.

* **Entities ejemplos:** Apartment.cs, Building.cs, Request.cs, Transaction.cs.
* **Models (DTOs) ejemplos:** CreateRequestDto.cs, UpdateRequestDto.cs, TransactionDto.cs.

.

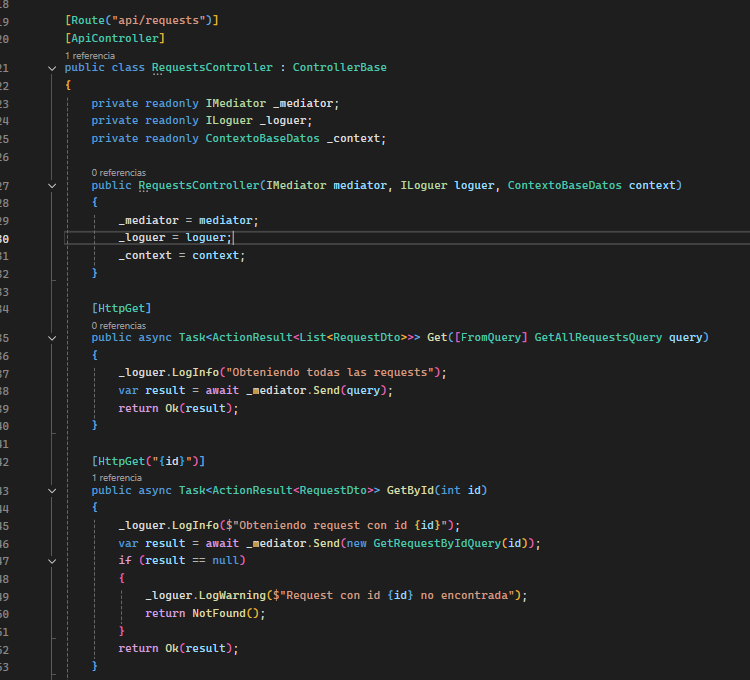


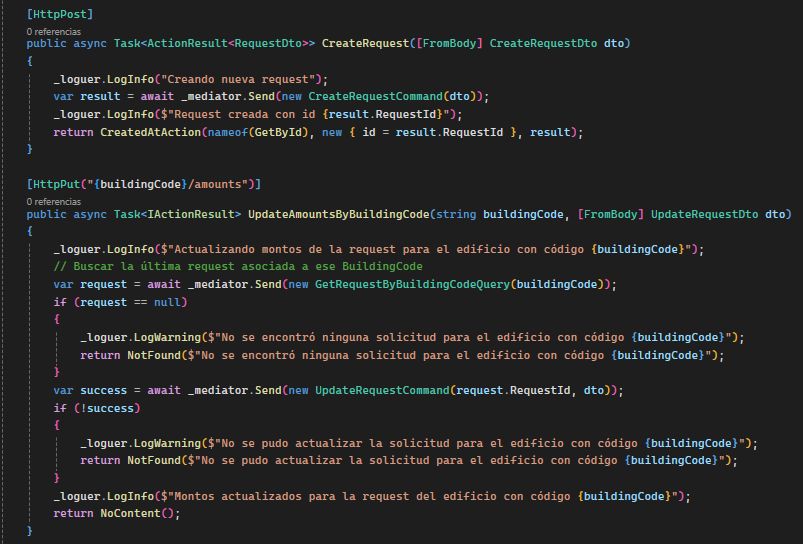
Captura 2: Ejemplo de la entidad Request y su DTO asociado CreateRequestDto.

**6.4. Controladores (API Endpoints)**

Los controladores son la interfaz del backend. Cada controlador agrupa un conjunto de *endpoints* relacionados.

* **AngController.cs**: Expone varios endpoints relacionados con estadísticas y gestión de edificios, apartamentos y presupuesto. Utiliza MediatR para manejar consultas y Entity Framework Core para acceder a la base de datos. Sus endpoints permiten obtener resúmenes de solicitudes, gastos mensuales, conteos de edificios por distrito, edificios comprados, información de edificios/apartamentos aprobados y el monto actual del presupuesto de gestión.
* **ApartmentsController.cs**: Gestiona las operaciones CRUD para apartamentos.
* **BuildingController.cs**: Gestiona las operaciones CRUD para edificios.
* **RequestsController.cs**: Administra el ciclo de vida de las solicitudes (creación, actualización de estado, consulta).
* **TransactionsController.cs**: Maneja las transacciones financieras asociadas a las propiedades.





Captura 3: Extracto de código del RequestsController, mostrando endpoints de ejemplo.

**Ejemplo de Endpoint: GET /api/requests**

* **Descripción:** Recupera una lista paginada de solicitudes con posibles filtros.
* **Método:** GET
* **Parámetros de Query (Opcional):** page=1, pageSize=10, status=Pendiente
* **Respuesta Exitosa (Código 200 OK):**

JSON

[

{ "id": 1, "description": "Fuga en el baño", "status": "Pendiente", "buildingId": 101, "dateCreated": "2024-05-20T10:00:00Z" },

{ "id": 2, "description": "Revisión de electricidad", "status": "Aprobada", "buildingId": 102, "dateCreated": "2024-05-18T14:30:00Z" }

]

* **Respuesta de Error (Código 500 Internal Server Error):**

JSON

{ "error": "No se pudieron recuperar las solicitudes." }

**6.5. Servicios y Lógica de Negocio**

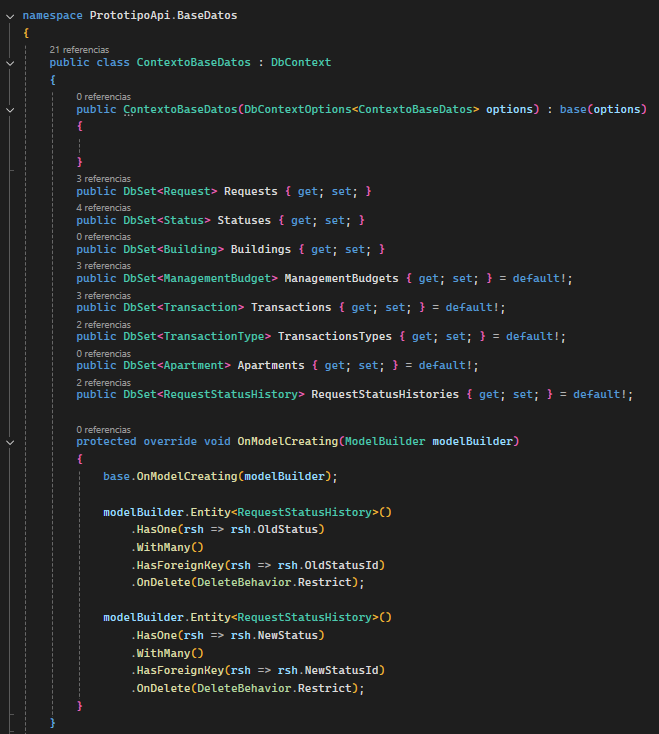
Los servicios (Services) contienen la lógica de negocio principal. Separan la lógica de la interacción directa con los controladores y los repositorios, facilitando la reutilización y el testeo.

**•** ExternalApartmentService.cs / ExternalBuildingService.cs: Estos servicios están diseñados para futuras integraciones, permitiendo la conexión con aplicaciones de otros departamentos y facilitando el intercambio de información sobre apartamentos y edificios a través de APIs externas.

**6.6. Persistencia de Datos (Entity Framework Core)**

Utilizamos **Entity Framework Core** como ORM para la interacción con la base de datos.

* **ContextoBaseDatos.cs**: Define el contexto de la base de datos y los DbSet para cada entidad.
* **Migraciones:** Se gestionan automáticamente al iniciar la aplicación (MigrateDatabase()).



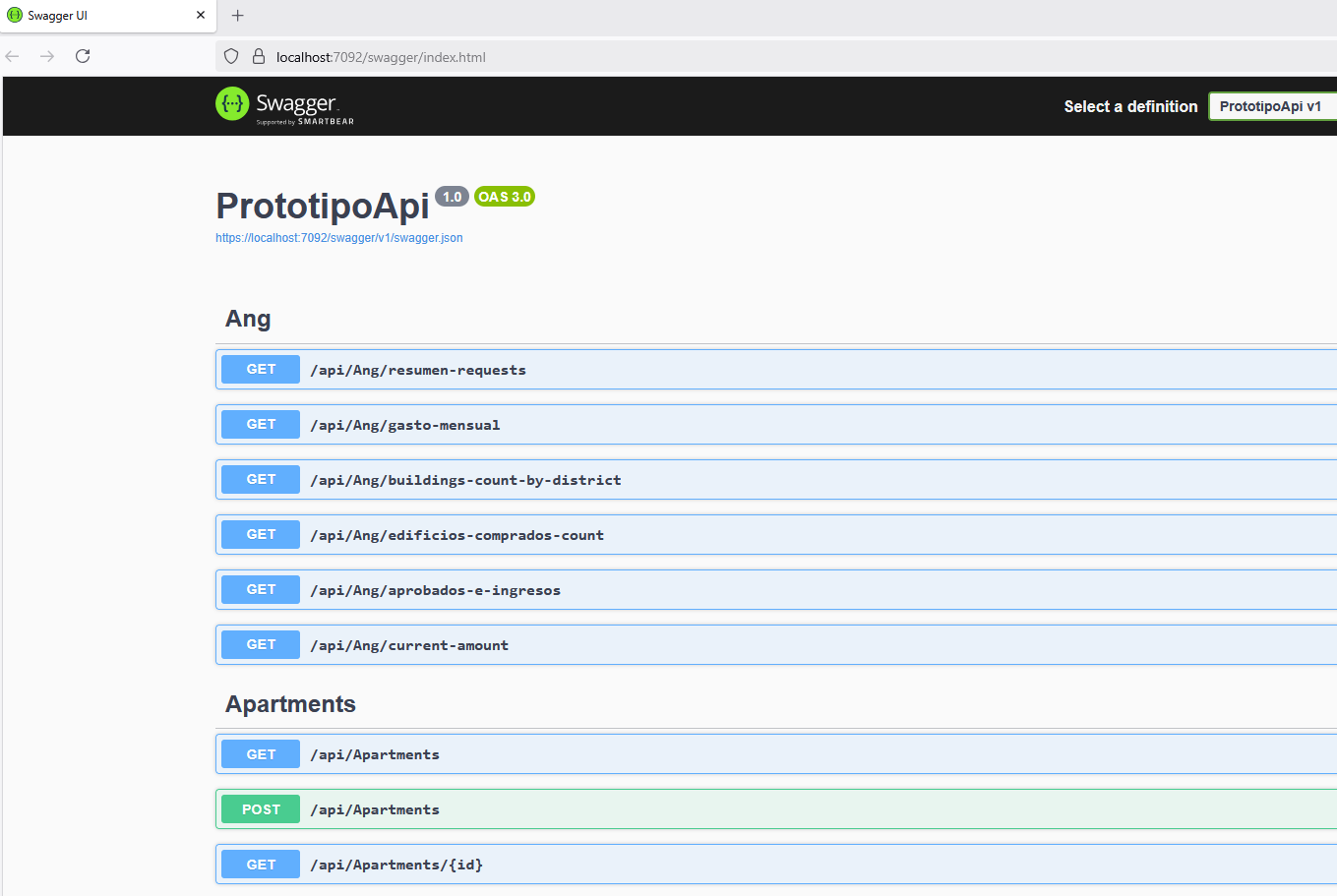
Captura 4: Definición del Contexto de la Base de Datos con DbSet para las entidades.

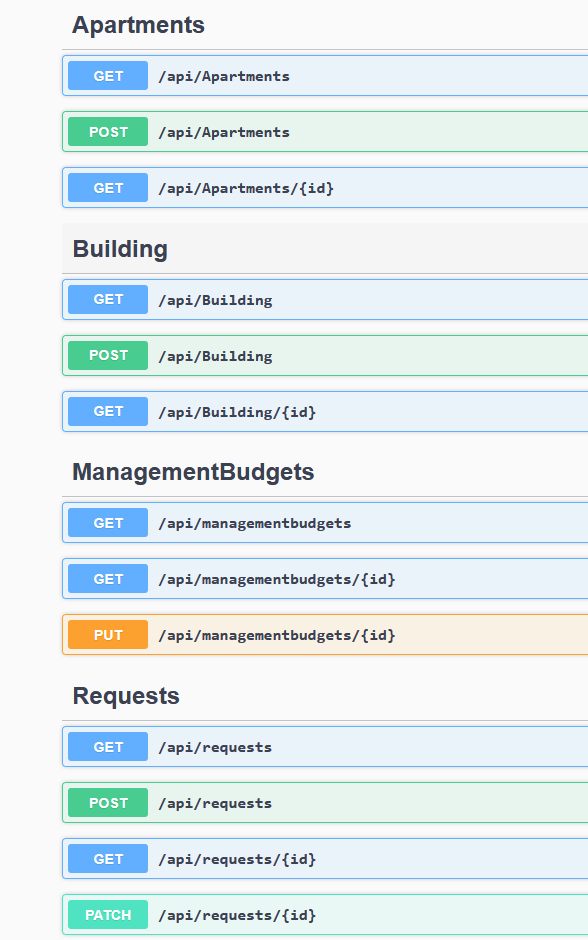
**6.7. Autenticación y Seguridad**

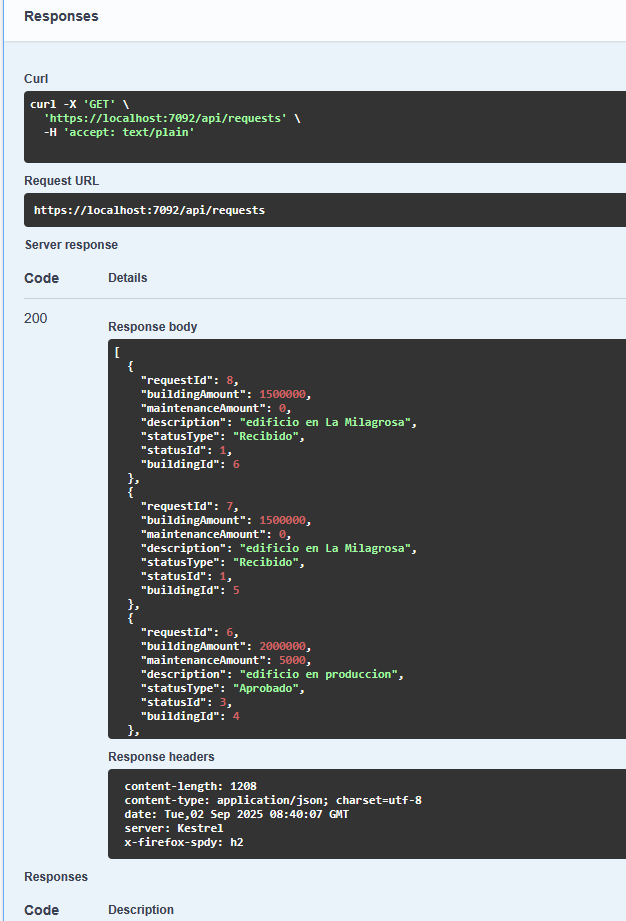
Actualmente, la autenticación y autorización en la API no están implementadas. Se contempla como una mejora futura, donde se integrarán mecanismos como JWT o Identity Core para gestionar el acceso seguro, la asignación de roles y permisos, y proteger los endpoints sensibles del sistema.

**6.8. Documentación de API (Swagger)**

La API está auto-documentada usando **Swagger/OpenAPI**, lo que permite explorar, entender y probar los *endpoints* de manera interactiva.







Captura 5: Interfaz de Swagger UI mostrando la documentación interactiva de la API.

**7. Documentación del Frontend (Angular)**

El frontend de **SpecuLabTeam** es una aplicación web interactiva desarrollada con **Angular**. Proporciona la interfaz de usuario para que los gestores de propiedades interactúen con el sistema.

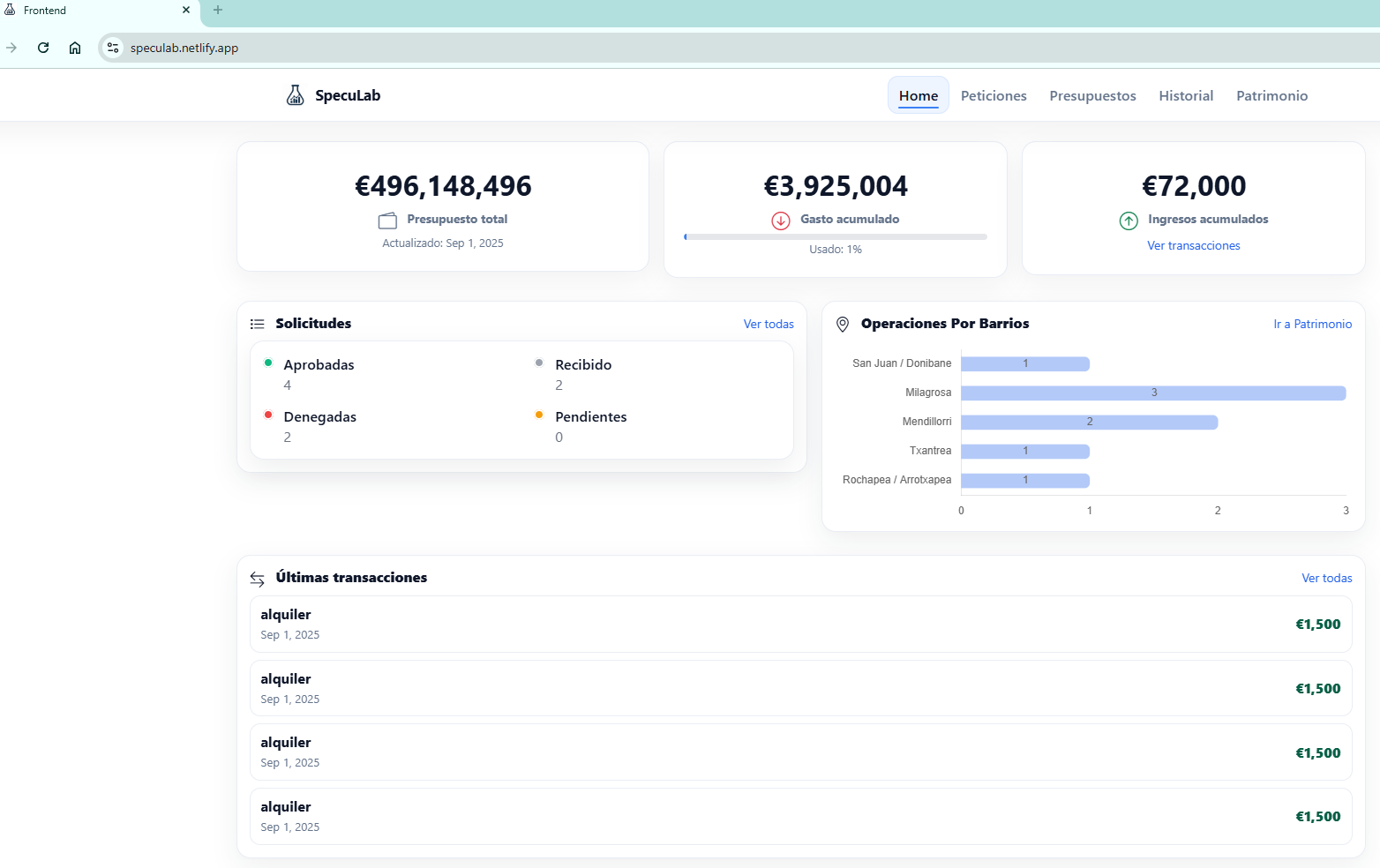
**7.1. Arquitectura y Tecnologías**

* **Framework:** Angular 20.1
* **Lenguaje:** TypeScript
* **Gestión de Estado:** Actualmente, la gestión de estado en el frontend se realiza principalmente mediante RxJS para el manejo asíncrono de datos y la comunicación entre componentes. Se contempla la posible integración de librerías como Ngrx o Akita en futuras versiones para una gestión de estado más robusta y escalable.
* **Librerías de UI:** Angular Material, Bootstrap, CSS personalizado.
* **Routing:** Modularizado y protegido mediante [Ejemplo: AuthGuard] para las rutas que requieren autenticación.

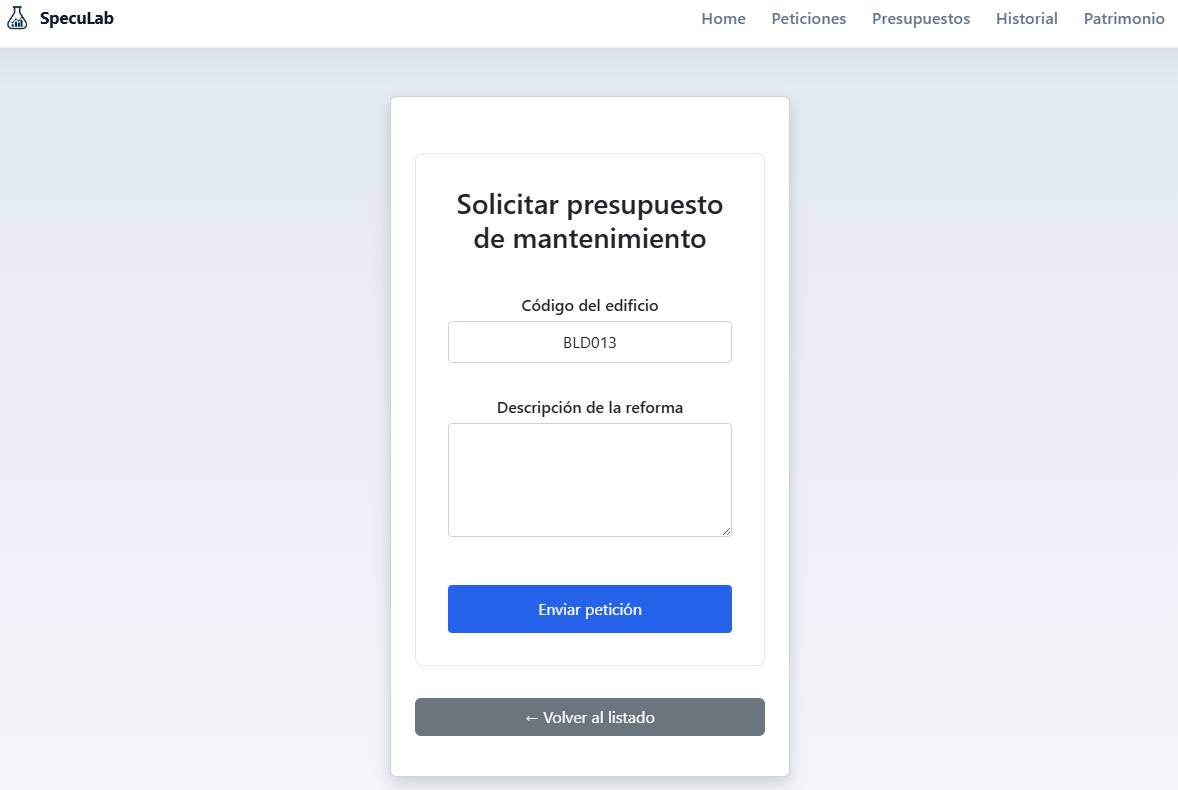
**7.2. Flujos de Usuario y Componentes Clave**

Aquí se detallan los principales flujos de interacción del usuario con la aplicación, acompañados de capturas de pantalla para una mejor comprensión.

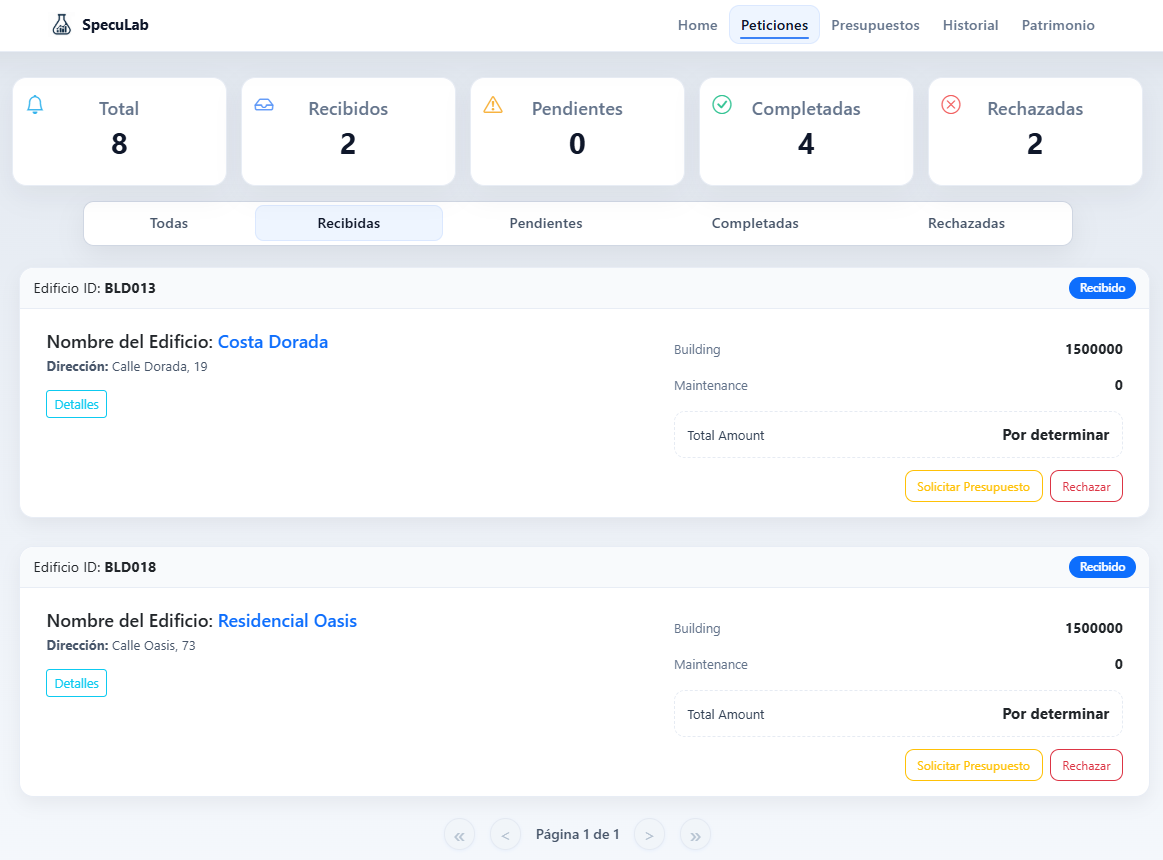
* **7.2.1. Dashboard Principal de Propiedades/Solicitudes**
  + **Descripción:** Tras iniciar sesión, el usuario accede al dashboard principal, que ofrece una visión general de las propiedades activas, las solicitudes pendientes y un resumen de las métricas clave del sistema, facilitando la gestión y el seguimiento de las operaciones más relevantes.



* + *Captura 6: Dashboard Principal de SpecuLabTeam.*
* **7.2.2. Gestión de Solicitudes (Creación, Visualización, Actualización)**
  + **Descripción:** Los usuarios pueden crear nuevas solicitudes, ver los detalles de las existentes y actualizar su estado.



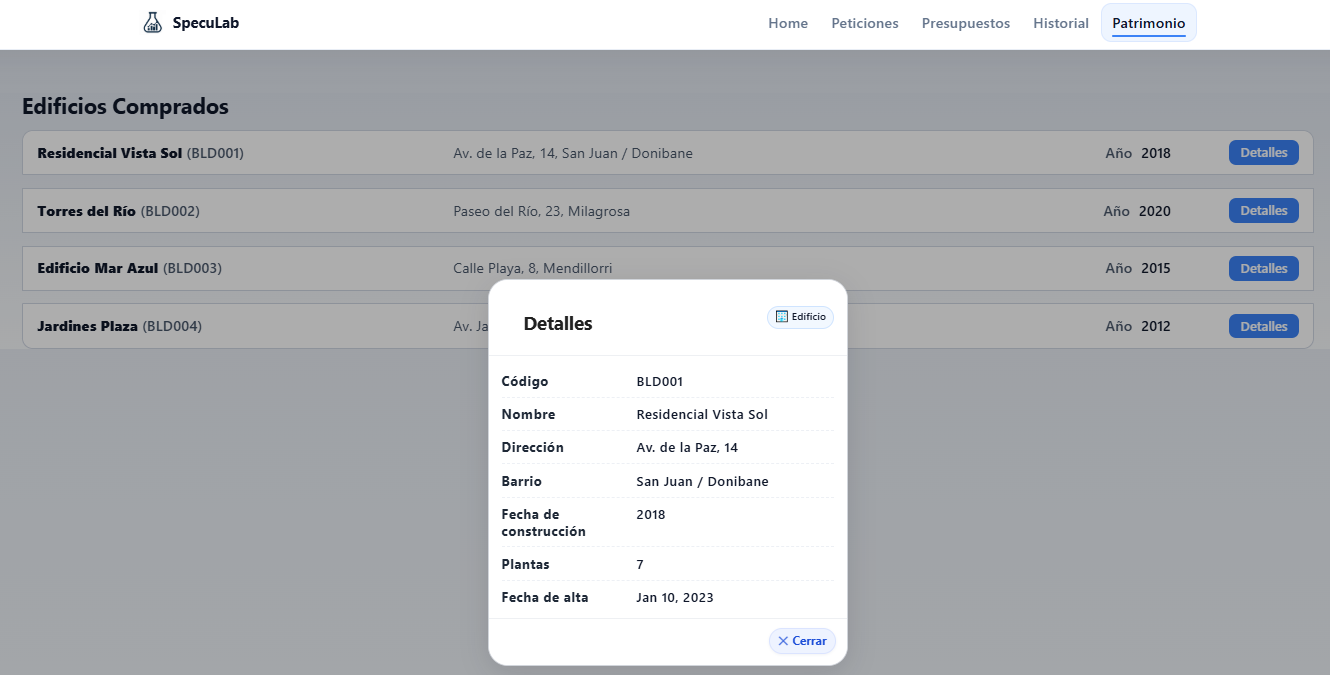
* + *Captura 7: Formulario para crear una nueva solicitud.*



* + *Captura 8: Listado de Solicitudes con opciones de filtro.*
* **7.2.3. Historial de Estados de Solicitudes**
  + **Descripción:** Para cada solicitud, se puede acceder a un historial detallado de los cambios de estado (Ej. Recibido -> Pendiente -> Aprobado).



* + *Captura 9: Historial de estados de una solicitud específica.*
* **7.2.4. Gestión de Edificios/Apartamentos**
  + **Descripción:** Módulos para añadir, editar o visualizar información detallada sobre edificios y apartamentos.



* + *Captura 10: Listado y detalles de Edificios/Apartamentos.*

**8. Guía de Instalación y Despliegue**

**8.1. Instalación Local para Desarrollo**

Para configurar y ejecutar el proyecto **SpecuLabTeam** en un entorno de desarrollo local, sigue estos pasos:

1. **Clonar el Repositorio de GitHub:**

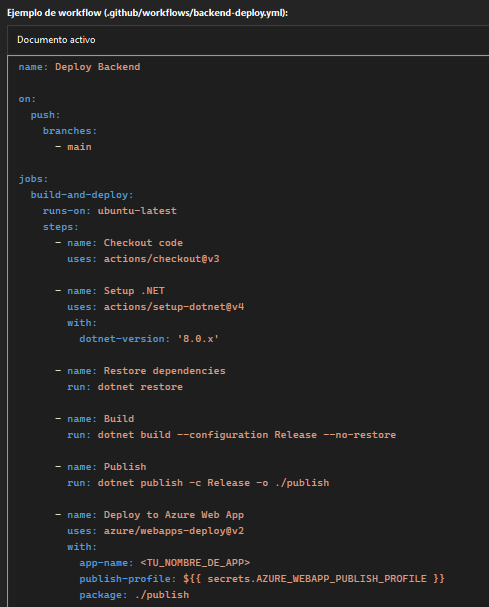
* Bash
* git clone <https://github.com/alexgn12/SpecuLabTeam.git>
* cd SpecuLabTeam

1. **Configuración del Backend (.NET API):**
   * **Requisitos:** .NET SDK [Versión, ej. 6.0 o 8.0], SQL Server [Versión, si aplica].
   * Navega a la carpeta PrototipoApiSolucion/PrototipoApi.
   * Restaura las dependencias: dotnet restore
   * Configura la cadena de conexión a tu base de datos en appsettings.json.
   * Aplica las migraciones de la base de datos: dotnet ef database update
   * Ejecuta el backend: dotnet run (o inicia desde Visual Studio).
   * La API estará disponible en https://localhost:[puerto] (revisar launchSettings.json).
2. **Configuración del Frontend (Angular):**
   * **Requisitos:** Node.js [Versión], npm/yarn, Angular CLI.
   * Navega a la carpeta del frontend (si está en el mismo repo, ej. SpecuLabTeam/frontend).
   * Instala las dependencias: npm install (o yarn install)
   * Configura la URL de la API del backend en el archivo de entorno (src/environments/environment.ts).
   * Inicia el servidor de desarrollo: ng serve --open
   * El frontend estará disponible en http://localhost:4200.

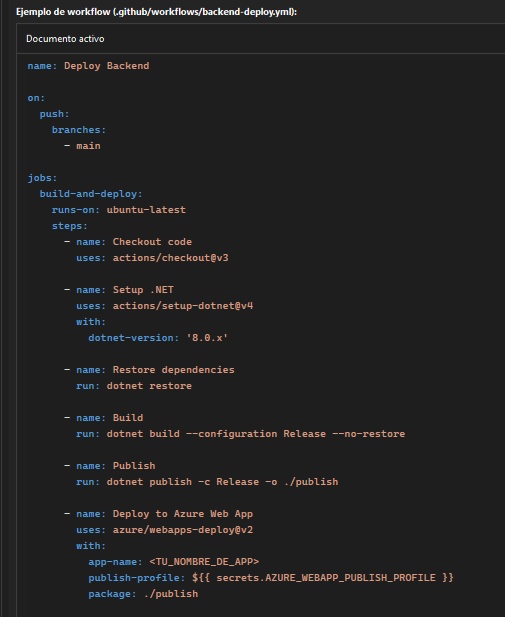
**8.2. Despliegue (CI/CD)**

Para este proyecto se utilizan GitHub Actions como sistema de CI/CD, permitiendo automatizar la construcción y despliegue tanto del Frontend como del Backend cada vez que se actualiza la rama principal. **GitHub Actions para el Frontend:**

* + **Descripción:** Un workflow automatizado construye la aplicación Angular y la despliega en Azure Static Web Apps cada vez que se detectan cambios en la rama main.



* + *Captura 11: Workflow de GitHub Actions para el despliegue del Frontend.*
* **GitHub Actions para el Backend:**
  + **Descripción:** Otro workflow se encarga de construir la API de .NET 8 y desplegarla en Azure App Service tras cada actualización en la rama principal.



* + *Captura 12: Workflow de GitHub Actions para el despliegue del Backend.*

**9. Login y Autenticación de Usuarios**

**9.1. Autenticación y Autorización**

1. El backend utiliza ASP.NET Core Identity junto con JWT (JSON Web Tokens) para la autenticación y autorización de personas usuarias.
2. Los registros se almacenan en la base de datos mediante las tablas estándar de Identity (*AspNetUsers*, *AspNetRoles*, etc.), con soporte para roles y claims personalizados.

**9.2. Flujo de Login**

1. **9.2.1 Solicitud de Login:** La persona usuaria envía su email y contraseña al endpoint */api/Auth/login* mediante una petición POST.
2. **9.2.2 Validación:** El backend valida las credenciales usando Identity. Si son correctas, se generan dos tokens:
3. **Access Token (JWT):** Incluye los claims y roles, con expiración corta (minutos).
4. **Refresh Token:** Token seguro y de larga duración, almacenado en la base de datos y enviado como cookie segura.
5. **9.2.3 Respuesta:** El access token se devuelve al frontend, que lo almacena localmente y lo adjunta en el header Authorization para futuras peticiones.

**9.3. Seguridad**

1. El access token se firma con una clave secreta y contiene información mínima (ID, email, roles).
2. El refresh token permite renovar el access token sin necesidad de reingresar credenciales, mejorando la experiencia de las personas usuarias y la seguridad.
3. Las contraseñas se almacenan hasheadas y nunca se exponen.
4. Se aplican políticas de contraseña segura y bloqueo tras múltiples intentos fallidos.

**9.4. Frontend**

1. El frontend Angular gestiona el ciclo de vida del token:
2. Al iniciar sesión, almacena el access token y su expiración.
3. Usa un interceptor HTTP para añadir automáticamente el token a cada petición.
4. Si el access token expira, intenta renovarlo usando el refresh token.
5. Permite cerrar sesión limpiando los tokens y notificando al backend.

**9.5. Configuración y Extensibilidad**

1. La configuración de JWT (clave, emisor, audiencia, tiempos de expiración) se gestiona por variables de entorno.
2. El sistema soporta roles y claims, permitiendo autorización granular en los endpoints mediante atributos *[Authorize]*.

**9.6. Ventajas**

1. Seguridad robusta y escalable.
2. Experiencia de persona usuaria fluida gracias al uso de refresh tokens.
3. Fácil integración con frontend moderno y soporte para autorización basada en roles.

**10. Manejo de Excepciones y Errores**

**10.1 Middleware Global de Excepciones**

1. El backend implementa un middleware global para capturar y gestionar todas las excepciones no controladas que ocurren durante el procesamiento de las solicitudes HTTP.
2. Este middleware se configura en el pipeline de la aplicación en *Program.cs*, antes de los controladores, asegurando que cualquier error sea interceptado y tratado de forma centralizada.

**10.2 Tipos de Excepciones Gestionadas**

1. **10.2.1 Excepciones de Validación:** Cuando ocurre una excepción de validación (por ejemplo, mediante FluentValidation), el middleware captura el error y devuelve una respuesta HTTP 400 (Bad Request) con detalles estructurados sobre los campos inválidos.
2. **10.2.2 Errores No Controlados:** Cualquier otra excepción no prevista es registrada y se responde con un mensaje genérico y un código HTTP 500 (Internal Server Error), evitando exponer detalles sensibles al cliente.

**10.3 Registro y Monitorización**

1. Se utiliza Serilog para registrar todas las excepciones, tanto de validación como generales, incluyendo información contextual útil para el diagnóstico y la monitorización.
2. Los logs se almacenan en consola y en archivos rotativos, facilitando el seguimiento de incidencias en producción y desarrollo.

**10.4 Respuestas al Cliente**

1. Las respuestas de error siguen un formato consistente y estructurado, facilitando su manejo desde el frontend.
2. Para errores de validación, se utiliza el estándar *ProblemDetails* de ASP.NET, permitiendo al cliente identificar fácilmente los campos y mensajes asociados.

**10.5 Ventajas de la Implementación**

1. **10.5.1 Centralización:** Todo el manejo de errores se realiza en un único punto, simplificando el mantenimiento y la evolución del sistema.
2. **10.5.2 Seguridad:** Se evita la exposición de información sensible en los mensajes de error.
3. **10.5.3 Experiencia de persona usuaria:** El frontend recibe mensajes claros y estructurados, mejorando la interacción y la gestión de errores en la interfaz.

**11. Implementación de SignalR**

**11.1 ¿Qué es SignalR y para qué se usa en el proyecto?**

1. SignalR es una biblioteca de ASP.NET que permite la comunicación en tiempo real entre el servidor y los clientes (por ejemplo, aplicaciones web Angular). En este proyecto, SignalR se utiliza para enviar notificaciones en tiempo real a las personas usuarias, como actualizaciones de datos, cambios de estado o eventos relevantes sin necesidad de que la persona usuaria refresque la página.

**11.2 Configuración y Arquitectura de SignalR**

1. La integración de SignalR en el proyecto comienza mediante su registro en el pipeline de la aplicación. Para ello, en el archivo *Program.cs*, se añade el servicio SignalR al contenedor de dependencias con *builder.Services.AddSignalR()*.
2. Posteriormente, se define al menos un Hub, que actúa como el punto central de la comunicación en tiempo real entre el backend y las personas usuarias. Un ejemplo típico es el *LiveHub*, donde se gestionan las conexiones y los mensajes enviados y recibidos.
3. El endpoint correspondiente al Hub se expone a través de una ruta específica, como */hubs/live*, utilizando *app.MapHub("/hubs/live")*. Esto permite que los clientes puedan conectarse fácilmente desde sus aplicaciones frontend.
4. Para facilitar el envío de notificaciones, se implementa un servicio dedicado, como *IRealTimeNotifier*, que abstrae la lógica de comunicación y permite enviar mensajes a todas las personas conectadas o a grupos específicos dentro del Hub.
5. SignalR se configura en el archivo *Program.cs* mediante la línea:
6. builder.Services.AddSignalR();
7. Se define un Hub (por ejemplo, *LiveHub*) que actúa como punto central de comunicación entre el servidor y las personas usuarias.
8. El endpoint del hub se expone en la ruta /hubs/live:
9. app.MapHub("/hubs/live");
10. Se registra un servicio de notificaciones en tiempo real (*IRealTimeNotifier*), que permite a otras partes del backend enviar mensajes a las personas conectadas.

**11.3 Seguridad y gestión de conexiones en SignalR**

1. Para garantizar una comunicación segura, SignalR se integra con los mecanismos de autenticación y autorización del proyecto. De esta manera, solo personas autenticadas pueden establecer conexión con el Hub, y es posible restringir el acceso a determinados grupos o canales según el rol o los permisos de cada persona usuaria.
2. La gestión eficiente de conexiones es fundamental en escenarios con múltiples clientes. SignalR administra automáticamente la reconexión y el rastreo de sesiones, permitiendo identificar a cada persona conectada mediante un identificador único (*ConnectionId*). Esto facilita enviar mensajes dirigidos a individuos o grupos y mantener la consistencia de las notificaciones incluso si una persona cambia de dispositivo o recarga la página.
3. Para reforzar la seguridad y el rendimiento, se recomienda configurar límites de tiempo de conexión, manejo de desconexiones inesperadas y políticas de reintentos. Además, es posible registrar eventos clave como conexiones, desconexiones y errores dentro del Hub, lo que contribuye a la monitorización proactiva y la detección de posibles incidencias en tiempo real.
4. Cuando ocurre un evento relevante en el backend (por ejemplo, cambio de estado de una solicitud), el servicio de notificaciones utiliza el hub de SignalR para enviar un mensaje a todas las personas suscritas o a un grupo específico.
5. Las personas usuarias (por ejemplo, desde el frontend Angular) se conectan al hub y reciben estos mensajes instantáneamente, permitiendo actualizar la interfaz de usuario en tiempo real.

**11.4 Ventajas de utilizar SignalR**

1. Interactividad: Las personas usuarias ven los cambios en tiempo real sin recargar la página.
2. Escalabilidad: SignalR soporta múltiples clientes y puede integrarse con servicios de mensajería distribuidos si el proyecto crece.
3. Simplicidad: La integración es directa tanto en backend como en frontend.
4. Interactividad: Las personas usuarias ven los cambios en tiempo real sin recargar la página.
5. Escalabilidad: SignalR soporta múltiples clientes y puede integrarse con servicios de mensajería distribuidos si el proyecto crece.
6. Simplicidad: La integración es directa tanto en backend como en frontend.