PRECIO JUSTO

Aplicación de subastas de Inmuebles entre particulares nº 1 (la única que hay)

Jordi Mimbrera Rubio

Grado de Ingeniería Informática

Trabajo Final de Grado – Java EE

Consultor: Antoni Oller Arcas

19/01/2024

FICHA DEL TRABAJO FINAL

Título del trabajo	Precio Justo Portal de subastas entre particulares
Nombre del autor	Jordi Mimbrera Rubio
Nombre del consultor	Antoni Oller Arcas
Fecha de entrega (mm/aaaa)	01/2024
Área del Trabajo Final	Java EE
Titulación	Grado de Ingeniería Informática

Resumen del Trabajo (máximo 250 palabras):

En este Trabajo de Final de Grado, he diseñado y desarrollado la aplicación PRECIO JUSTO, una plataforma para la subasta de propiedades inmobiliarias entre particulares. Este sistema busca facilitar operaciones inmobiliarias de manera transparente y eficiente.

La aplicación permite a los usuarios agregar propiedades al sistema, crear y consultar subastas, ver pujas actuales, revisar historiales de subastas anteriores y comunicarse con vendedores a través de mensajería instantánea.

El backend de este proyecto se ha construido siguiendo una arquitectura de microservicios, proporcionando contenido a la aplicación multiplataforma mediante APIs REST. Un Gateway se encarga de gestionar el enrutamiento del tráfico y la validación de las solicitudes.

La duración de este proyecto ha sido de cuatro meses. Durante este tiempo, se estableció un Plan de Trabajo dividido en etapas. La fase de implementación se ha llevado a cabo siguiendo una metodología ágil, caracterizada por sprints iterativos y cortos que han permitido una mejora y adaptación continua del proyecto.

Abstract (in English, 250 words or less):

In this Final Degree Project, I have designed and developed the PRECIO JUSTO application, a platform for auctioning real estate properties among individuals. This system aims to facilitate real estate operations in a transparent and efficient manner.

The application allows users to add properties to the system, create and consult auctions, view current bids, review histories of previous auctions, and communicate with sellers through instant messaging.

The backend of this project has been built following a microservices architecture, providing content to the multi-platform application through REST APIs. A Gateway is responsible for managing traffic routing and validating requests.

The duration of this project has been four months. During this time, a Work Plan divided into stages was established. The implementation phase has been carried out following an agile methodology, characterized by iterative and short sprints that have allowed continuous improvement and adaptation of the project.

Índice

1. Introducción....7
  1.1 Contexto y justificación del Trabajo....7
  1.2 Objetivos del Trabajo....8
  1.3 Enfoque y metodología seguidos....10
  1.4 Planificación del Trabajo....11
  1.5 Breve sumario de productos obtenidos....13
  1.6 Breve descripción de los otros capítulos de la memoria....14
2- Requerimientos, análisis y diseño....16
  2.1 Actores y modelo de casos de uso....16
  2.2 Fichas de los casos de uso....19
  2.3 Modelo de pantallas (Prototipo)....29
  2.4 Diagrama de arquitectura....35
  2.5 Microservicios....39
  2.6 Operaciones y consultas....45
  2.7 Gateway....47
  2.8 Gestión del token, control de sesiones....48
  2.9 Diagrama de las clases principales....51
  2.10 Diagrama relacional de la base de datos....54
  2.11 Primeros pasos, creando la estructura y User microservicio....56
3. Implementación....64
  3.1 Entorno de desarrollo....64
  3.2 Cambios en los casos de uso y funcionalidades del proyecto....64
  3.3 Creación de la infraestructura....68
  3.4 API GATEWAY....76
  3.5 API REST y App....81
4. Mejoras y evolución del proyecto....92
5. Conclusiones....93
6.Glosario....94
7.Bibliografia....96
8. Anexo....96
  8.1 Instalación y ejecución....96
  8.2 Tests unitarios y pruebas de integración....98

Tabla de Ilustraciones

Ilustración 1 Planificación de la PEC 1 12
Ilustración 2 Planificación de la PEC 2 12
Ilustración 3Planificación de la PEC 3 13
Ilustración 4 Memoria y presentación 13
Ilustración 5 Buscador y pantalla principal....31
Ilustración 6 Inicio de sesión 31
Ilustración 7 Lista de inmuebles con foto....32
Ilustración 8 Agregar a favorito....32
Ilustración 9 Enviar mensaje....33
Ilustración 10 Mensaje enviado correctamente 34
Ilustración 11 Formulario para agregar un inmueble....34
Ilustración 12 Arquitectura del proyecto....36
Ilustración 13 Boceto del retorno del microservicio de User del token JWT 38
Ilustración 14 Esquema Microservicio User 41
Ilustración 15 Esquema Microservicio Communication 42
Ilustración 16 Esquema Microservicio Auction....43
Ilustración 17 Esquema Microservicio Property 44
Ilustración 18 Configuración Gateway 48
Ilustración 19 Pequeño esquema del login en el microservicio User 49
Ilustración 20 Clase que contiene los datos del token 49
Ilustración 21 Interfaz TokenService (puerto de salida en User microservicio)....50
Ilustración 22 Ejemplo de cabecera con token en una petición de la aplicación 51
Ilustración 23 Definición de las clases de cada microservicio 53
Ilustración 24 Definición de las tablas en base de datos.....55
Ilustración 25 Proyecto inicial de frontend en React Native 56
Ilustración 26 Backup en Github (me salvó varias veces al principio, gracias Tutor por el consejo)....57
Ilustración 27 Contenedores con las imágenes corriendo correctamente.....58
Ilustración 28 Varias capturas de la definición inicial de User....60
Ilustración 29 Definición de las Entities 60
Ilustración 30 Entidad Role....61
Ilustración 31 Inserts para pruebas....62
Ilustración 32 Llamada en Swagger al endpoint con todos los usuarios (y sus permisos) 62
Ilustración 33 Petición con usuario y contraseña correcta y token de salida 63
Ilustración 34 Llamada con usuario no valido con mensaje de error 63
Ilustración 35 Ilustración inmuebles usuario....65
Ilustración 36 Subasta creada con éxito 65
Ilustración 37 Imagen agregada a una Propeidad....66
Ilustración 38 Imagen de la organización de carpetas en Firebase Storage 67
Ilustración 39 Docker Desktop servicios desplegados....69
Ilustración 40 Clase Auction 70
Ilustración 41 Subastas activas con el objeto Auction 72

Ilustración 42 Entidad AuctionEntity a partir de la que se crea la tabla auctions 74
Ilustración 43 Scheme.sql con un Insert de una subasta 75
Ilustración 44 Tablas del Microservicio Auction 75
Ilustración 45 Filtro del Gateway para validar Token 76
Ilustración 46 La petición llega al Gateway y se resuelve satisfactoriamente 78
Ilustración 47 Vemos como el token se ha creado..78
Ilustración 48 Ejemplo de los métodos REST de Auction microservicio...................................81
Ilustración 49 Función de extracción de detalles de usuario de una petición...........................82
Ilustración 50 Clase AuctionRequest para crear una nueva subasta. 83
Ilustración 51 Login del usuario pablo 84
Ilustración 52 Subasta creada correctamente 85
Ilustración 53 Método crear subasta en la implementación del service..................................85
Ilustración 54 Guardado en la BBDD de la subasta 86
Ilustración 55 SpringDataAuctionRepository que extiende JPA con funciones CRUD 86
Ilustración 56 Funcionalidad de envio de topic "crear subasta" 87
Ilustración 57 Organización proyecto App..87
Ilustración 58 Subastas creadas 88
Ilustración 59 Propiedades del usuario 89
Ilustración 60 Vista de propiedades activas de un usuario 90
Ilustración 61 Creación del Modal de nueva subasta en la screen de Properties del user 91
Ilustración 62 Objeto NewAuctionModal, al cerrar ejecuta la función onAuctionSuccess().......91
Ilustración 63 Crear subasta sin token en la cabecera 92
Ilustración 64 Clases de dominio para informes...93

1. Introducción

1.1 Contexto y justificación del Trabajo

Lo primero que quiero hacer, es intentar explicar por qué elegí este modelo de negocio o esta temática para mi TFG, imagínate que pretendes comprar una vivienda en Barcelona o Madrid y te pasas días, semanas o incluso meses navegando por diversos portales inmobiliarios, buscando la casa de tus sueños o sencillamente una inversión interesante.

Después de un tiempo de búsqueda cualquiera puede darse cuenta de que, aunque hay muchas opciones (en algunos lugares escasean, pero las razones no serían el objeto que deseo abordar en esta TFG), lo cierto es que el precio que se refleja en ellas es simplemente el que desearía conseguir una de las partes (el vendedor) y la fuerza del comprador queda supeditada a lo que cada uno individualmente pueda negociar.

Esto es exactamente lo que me pasó a mí. No hay, que yo haya visto, un sistema unificado que facilite la puja o subasta de inmuebles, algo parecido a lo que hace eBay con otros productos. Tampoco existe un organismo que facilite las transacciones, garantizando una compraventa transparente y asegurando que el precio reflejado es el real de mercado, lo más parecido que he visto buscando e indagando (salvando mucho las distancias, porque se "fuerza" la venta) sería la página de subastas electrónicas judiciales y de la agencia tributaria (https://subastas.boe.es/index.php) y como indicador de precios algún informe que hacen los notarios anualmente, pero nada fácilmente accesible ni mucho menos cómodo de consultar (de hecho yo no he logrado encontrarlo, solo alguna nota de prensa).

Así, durante mi búsqueda de vivienda, identifiqué la ausencia de dos importantes pilares en el mercado de cualquier bien en el inmobiliario, el primero sería la ausencia de un lugar, físico o virtual, aplicación o portal que ofrezca subastas de inmuebles de forma segura y transparente, permitiendo no solamente un precio acorde a lo que desea el vendedor

(ya que se puede poner un precio mínimo) sino también garantizando la liquidez para los vendedores, esto no ocurre así en otros lugares del mundo, por ejemplo, en Canadá, la transparencia en las transacciones inmobiliarias es bastante alta (gracias a organismos específicos como MLS)

Y de esta necesidad personal combinado con la observación de mercado durante los meses de verano, aprovechando que debía preparar una propuesta interesante para presentar, nació la idea de mi Trabajo Final de Grado.

1.2 Objetivos del Trabajo

Con mi proyecto, me gustaría intentar crear una plataforma en la que, no solo se pueda visualizar y buscar inmuebles y contactar con el vendedor de una propiedad, como en un portal inmobiliario tradicional, sino que además se puedan cerrar un acuerdo y obtener liquidez de una forma rápida mediante el sistema de subastas.

Podemos enumerar los siguientes objetivos:

1. Crear una plataforma que dé respuesta a las necesidades que hemos considerado en el punto anterior y que ofrezca una aplicación con las siguientes funcionalidades:

- Un sistema de gestión, registro y acceso para los usuarios y permisos
- Un módulo de gestión y registro de inmuebles
- Un sistema de subastas y pujas
- Un sistema de alertas para notificar a los usuarios sobre cambios en las pujas, finalización de subastas o cualquier otro evento relevante.
- Un módulo donde los usuarios puedan ver historial de pujas, propiedades vendidas/compradas, y otros detalles relevantes.

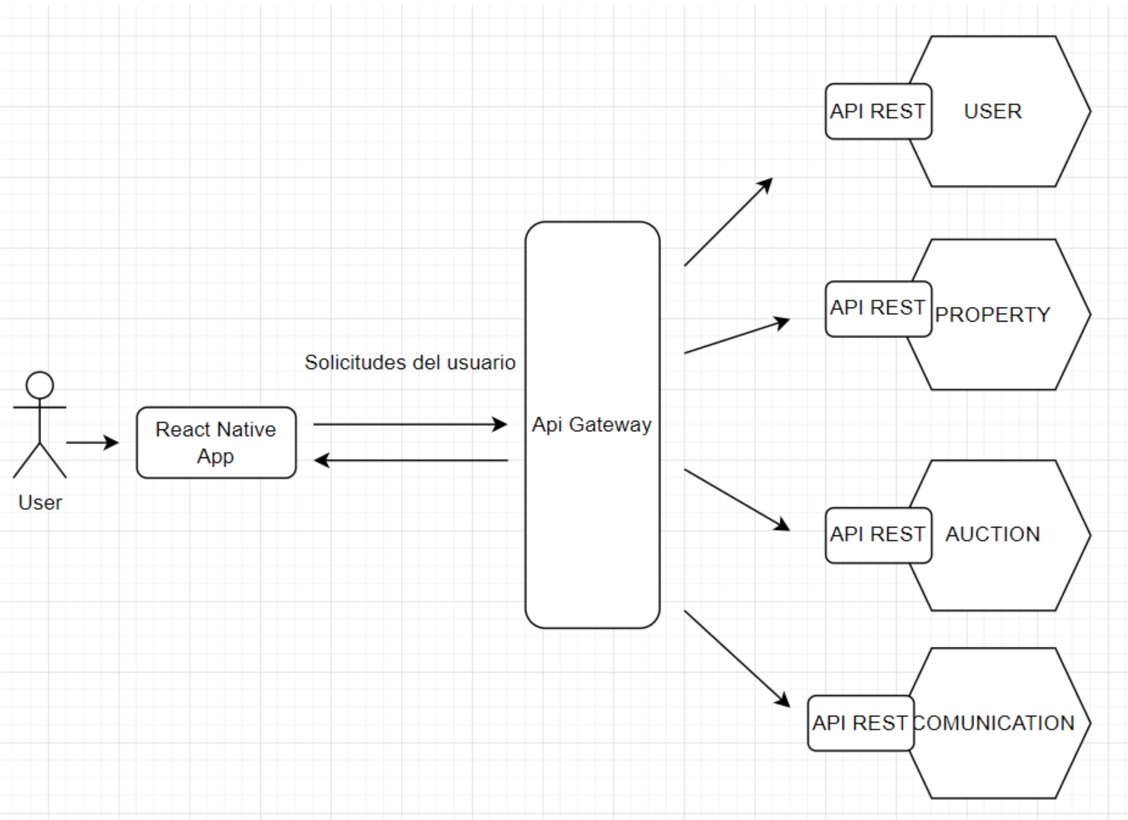
- Un sistema de "informes" para la consulta de valores históricos de ventas para el asesoramiento de los distintos usuarios y para ver informes en base a sus criterios de búsqueda o interés.
- Un módulo de comunicación: Un chat o sistema de mensajería para que compradores y vendedores puedan comunicarse directamente dentro de la plataforma.
- Un módulo de logs de la aplicación para llevar un registro de las acciones críticas/relevantes realizadas de los usuarios.
- Un módulo de validación de los inmuebles, para garantizar que las propiedades insertadas son legítimas, tal vez mediante la verificación de la nota simple del registro por el Administrador.

2. Montar una infraestructura basada en microservicios con una arquitectura hexagonal que facilitará a los clientes las funcionalidades a partir de llamadas a servicios API REST.

3. Crear un Gateway que actúe como un intermediario que gestione las solicitudes y respuestas entre el cliente y el backend que alojan la API REST, pretendo de alguna forma unificar las rutas con unas URL más "amigables" permitiendo además añadir algo de seguridad al "enmascarar" mis endpoints, además de la autenticación y control de validez de token en el acceso a según que recursos.

4. Crear una aplicación multiplataforma que actúe como frontend y que mediante el Gateway consuma los servicios API REST para consultar los distintos contenidos de la plataforma.

El esquema básico sería algo como lo que se ve a continuación:



1.3 Enfoque y metodología seguidos

La estrategia que he seguido es la de desarrollar las funcionalidades básicas de una plataforma de subastas, que pueda ser ampliada y mejorada posteriormente, aplicando los conocimientos adquiridos en las distintas asignaturas que he cursado en el grado.

Hay que destacar que la única experiencia que tengo en estas tecnologías es la que he aprendido durante este grado, profesionalmente no me dedico a ello y esto me ha supuesto un gran reto personal.

Por la parte del backend, específicamente, se han desarrollado los microservicios utilizando las tecnologías que se utilizaron en las asignaturas de *Ingeniería de Programación de Componentes y Sistemas Distribuidos* y *Proyecto de Desarrollo de Software*, básicamente serían, **Java** como lenguaje de programación, como IDE usaré **IntelliJ IDEA**

(Ultimate Edition), **Java Spring Boot** como framework y **Kafka** para la comunicación entre microservicios.

Para la capa de datos se ha optado por **PostgreSQL**.

Además, se ha empleado docker-compose para configurar y automatizar el arranque de todos los servicios necesarios.

En cuanto al frontend, se han destinado unos días durante la fase de "Plan de trabajo" para la investigación y se ha optado por utilizar, según consejo del tutor, **React Native**, ya que permite desarrollar aplicaciones mobile y web con un solo código.

Para el desarrollo de la plataforma, se ha adoptado una metodología ágil con un desarrollo iterativo, dividiéndome el tiempo en sprints con unas tareas claras y revisando si los he logrado al final de cada sprint.

Se han creado dos repositorios en github, uno para almacenar el backend y otro para el frontend.

1.4 Planificación del Trabajo

Definición a grandes rasgos de las tareas que han compuesto esta TFG hasta su resultado final y los tiempos que se han destinado a cada una de ellas.

La planificación del TFG del área de Java EE se organiza en cuatro fases, y basándome en las explicaciones en reuniones con el tutor, sabemos que la segunda y tercera fase son las más intensivas en tiempo de trabajo ya qué hay que realizar la memoria en paralelo.

En la primera fase el objetivo principal consiste en establecer una primera toma de contacto con el tutor, concretar la idea del TFG a desarrollar, con

las sugerencias que se puedan dar en la tutoría, y establecer un plan de trabajo del proyecto, como se contempla en la ilustración 1.

Además, se ha realizado un estudio de las tecnologías a implementar y se han establecen las herramientas y frameworks que se utilizarán.

+ Add · Expand all · Collapse all · Zoom in · Zoom out · Zoom to fit

ID	Name
1	1- PEC 1
2	1.1 - Estudio plan docente y ideas tema proye...
3	1.2 - Tutoría. Exposición ideas y solución dudas
4	1.3 - Definición calendario entregas
5	1.4 - Estudio tecnológias implementar
6	1.5 - Redacción entrega
7	1.6 - Definición de funcionalidades proyecto

Sep 24, 2023 · Oct 01, 2023 · Oct 08, 2023

Ilustración 1 Planificación de la PEC 1

En la segunda fase, ilustración 2, se han definido los casos de uso y su flujo en fichas, realizamos un modelo de pantallas o prototipo, se diseña la base de datos y se realiza el diagrama de clases y el de arquitectura.
Además, se empiezan a realizar pruebas con el IDE, acceso a la BBDD o pequeñas pruebas con algún endpoint básico.

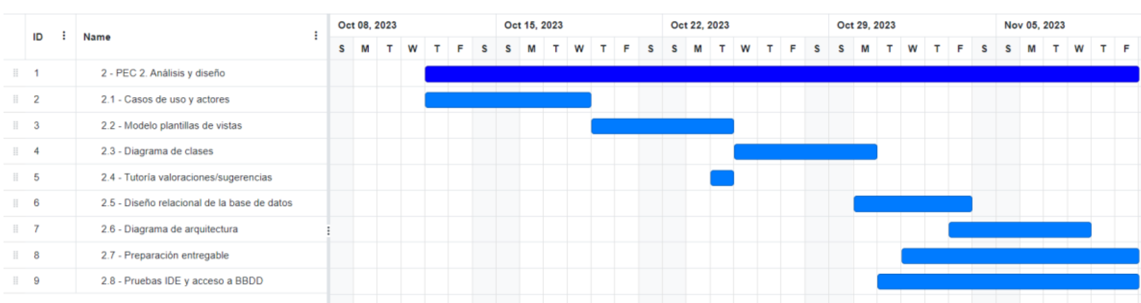


Ilustración 2 Planificación de la PEC 2

En la fase tres, la de implementación que se puede ver en la ilustración 3, se ha seguido una metodología ágil por iteraciones con sprints cortos. Primero desarrollando de una forma simple uno de los casos de uso para validar la arquitectura y posteriormente agregando complejidad a los

campos, usabilidad y manejos de errores. Al finalizar cada historia, las siguientes son más fácilmente implementables al tener ya la arquitectura creada y controlada.

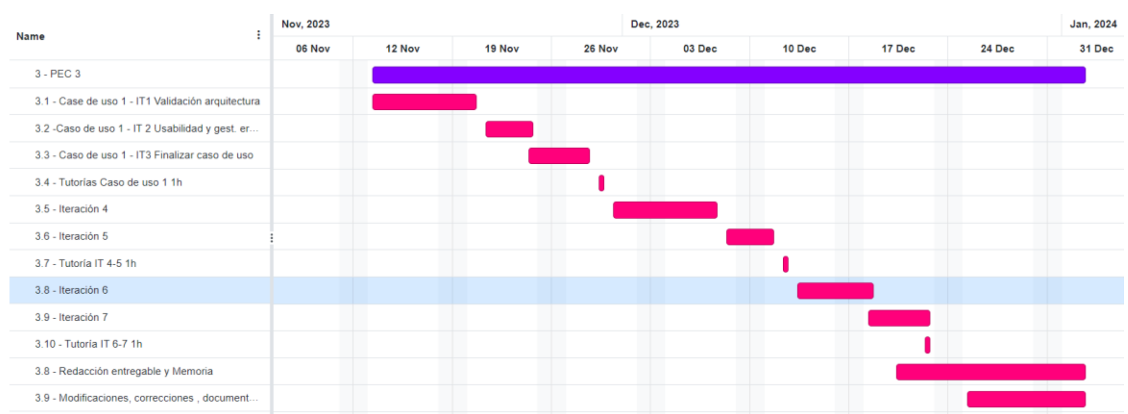


Ilustración 3Planificación de la PEC 3

En la cuarta fase, plasmada en la ilustración 4, se completará toda la documentación de la memoria, que se habrá ido creando en paralelo, durante todo el cuatrimestre, y también elaboraremos un video de presentación de nuestro TFG que incluirá una pequeña demostración de la App.

ID	Name
1	4 - Memoria y presentación
2	4.1 - Finalizar redacción Memoria
3	4.2 - Preparar presentación
4	4.3 -Grabar video de la presentación
5	4.4 - Tutoria
6	4.5 - Correcciones post-tutoria
7	4.6 - Entrega proyecto finalizado

Dec 31, 2023 · Jan 07, 2024 · Jan 14, 2024 · Jan 21, 2

Ilustración 4 Memoria y presentación

1.5 Breve sumario de productos obtenidos

El trabajo finalizado se compone de los siguientes productos:

- Documento con Memoria del proyecto
- Enlace a repositorio de github con el backend (incluye los microservicios, gateway y fichero de *docker-compose.yml* para despliegue de estos)
- Fichero comprimido con el código fuente desarrollado
- Documento con Presentación Virtual
- Video con la presentación virtual del trabajo
- Video con demo de la App.

1.6 Breve descripción de los otros capítulos de la memoria

La memoria del trabajo está conformada por los siguientes puntos:

1- Introducción

En el apartado actual se ha tratado el contexto que justifica el desarrollo de la aplicación.
Así como los objetivos que se persiguen y la necesidad que se presente cubrir, finalmente la planificación realizada.

2- Requerimientos, análisis y diseño

En el segundo punto se expone el análisis realizado, tanto de los requerimientos que da como resultado la elaboración de un pequeño prototipo, como a nivel de estudio y selección de las herramientas de software a utilizar para su implementación.

Este análisis y previsiones sobre el mismo se plasmaron en la entrega de la PEC2. Debido a la falta de experiencia en la mayoría de las tecnologías expuestas en la planificación, el diseño final ha podido sufrir variaciones, bien por inviabilidad de aplicación de algunas o en otras por la propia metodología ágil que se ha empleado, estos cambios se han ido documentando en esta memoria.

3- Implementación

Este apartado incluye detalles acerca de la implementación final del producto, las características incorporadas y su alcance.

4- Mejoras y evolución del proyecto

Este apartado reflejará aquellos retos que quedan por abordar para aportar un mayor valor al proyecto.

5- Conclusiones

Consideraciones personales sobre como se ha afrontado el proyecto y el resultado final.

6- Bibliografía

Distintas fuentes que se han ido consultando a lo largo de la implementación.

7- Glosario
Pequeño "diccionario" con el significado de algunos términos que se mencionan a lo largo de este TFG.

8- Anexo

En el anexo se han adjuntado las rutas para descargar los repos de Github, las instrucciones de instalación y capturas de pantalla de las pruebas unitarias y de integración.

2- Requerimientos, análisis y diseño

2.1 Actores y modelo de casos de uso

Se pueden identificar los siguientes actores en la aplicación:

Usuario comprador: Representa a los usuarios que están registrados y buscan adquirir propiedades. Tienen la capacidad de pujar por inmuebles, recibir alertas, comunicarse con vendedores, etc.

Usuario vendedor: Estos serían los usuarios ya registrados que ponen a la venta una propiedad. Pueden gestionar sus inmuebles y comunicarse con otros usuarios.

Administrador: Este actor tiene privilegios especiales en la plataforma. Puede gestionar usuarios y validar inmuebles entre otras tareas.

Sistema: Realizaría todas las acciones automáticamente en respuesta a eventos como pujas, notificaciones, etc.

Se quiere, además, añadir al **Visitante/Invitado** como actor para el caso de uso del registro o bien se mostraría sin un actor concreto, así que se pone a parte para algunos casos de uso donde también podrá usar la aplicación un usuario sin registrar.

Aunque se han puesto como actores distintos al usuario comprador y vendedor, podría considerarse que es el mismo actor con distintos roles o privilegios, la idea es que un mismo usuario pudiera agregar inmuebles si lo desea y ahí es donde tendría acceso a las opciones del actor vendedor, al menos es como se ha planteado para esta aplicación.

Ahora se identifican las necesidades de los distintos usuarios para plasmarlas como historias de usuario, que posteriormente serán los casos de uso, que son las que motivan el desarrollo de esta aplicación:

- **Como** usuario comprador, **quiero** pujar por un inmueble **para** intentar adquirirlo.

- **Como** usuario comprador, **quiero** poder enviar un mensaje a los usuarios vendedores **para** resolver dudas o discutir detalles.

- **Como** usuario comprador **quiero** recibir avisos o notificaciones de otras pujas **para** mantenerme informado de su estado.

- **Como** usuario comprador, **quiero** tener un buscador o filtrado **para** visualizar las propiedades de una zona.

- **Como** usuario comprador, **quiero** visualizar mi historial de pujas **para** llevar un registro de mis actividades de compra.

- **Como** usuario comprador, **quiero** añadir un inmueble a favoritos **para** que me avisen de las pujas que tiene.

- **Como** usuario vendedor, **quiero** poder registrar uno o más inmuebles **para** ponerlo en venta.

- **Como** usuario vendedor, **quiero** poder iniciar la subasta de uno o más inmuebles **para** ponerlo venderlos.

- **Como** usuario vendedor, **quiero** poder editar o un inmueble **para** mantenerlo actualizado.

- **Como** usuario vendedor, **quiero** eliminar un inmueble **que** ya no deseo vender.

- **Como** usuario vendedor, **quiero** ver las pujas hechas a mis inmuebles **para** ver cómo va la venta o si no hay ofertas.

- **Como** usuario (indistinto rol) **quiero** poder consultar precios de ventas anteriores **para** saber si el inmueble que veo tiene un precio acorde.

- **Como** usuario vendedor, **quiero** comunicarme con los compradores interesados **para** negociar o aclarar detalles.

- **Como** administrador, **quiero** gestionar (bloquear) usuarios para asegurarme que todo va correctamente en la plataforma.

- **Como** administrador, **quiero** poder validar inmuebles **para** asegurar la validez del inmueble para pujas.

**Se agrega historia de usuario fruto de la necesidad vista durante la implementación para mantener la coherencia en el sistema, más adelante se detallará.*

A partir de los actores que se han nombrado y de las historias de usuario que se han expuesto, se extraen los siguientes casos de uso:

Id	Actor	Caso de uso
CU01.1	Visitante	Registrar-se en aplicación
CU01.2	Usuario comprador Usuario vendedor Administrador	Inicio de sesión de la aplicación
CU01.3		Cerrar sesión de la aplicación
CU02	Visitante Usuario (indistinto rol) Administrador	Buscar inmuebles
CU03	Visitante Usuario (indistinto rol) Administrador	Consultar inmueble
CU04	Usuario comprador	Pujar por un inmueble
CU05	Usuario (indistinto rol)	Enviar/recibir mensajes
CU06	Usuario (indistinto rol)	Consultar subastas anteriores
CU07	Usuario comprador	Añadir a favorito un inmueble
CU08	Usuario vendedor	Agregar/modificar inmueble
CU09	Usuario vendedor	Eliminar inmueble
CU10	Usuario vendedor	Ver su histórico de subastas
CU11	Administrador	Bloquear usuarios
CU12	Administrador	Validar inmuebles
CU13	Usuario vendedor	Inicializar subasta
CU14	Visitante Usuario (indistinto rol) Administrador	Consultar subastas actuales

*En gris los que se han añadido durante la implementación.

2.2 Fichas de los casos de uso

Las fichas para estos casos de uso se detallan a continuación. Para la gestión de inmuebles, con acciones múltiples (consultar, eliminar, agregar, etc.) solamente realizaremos una de las fichas.

Las modificaciones sufridas en la implementación se agregan subrayadas.

Id	**CU01.1**
Caso de uso	Registrarse en la aplicación
Actor	Visitante
Descripción	Un usuario introduce en el sistema, a través de un formulario, la información básica para poder crear su usuario y contraseña. Valida su email y pasa a estar registrado.
Precondiciones	EL visitante (o su email) no deben tener una cuenta previamente registrada en la aplicación
Postcondiciones	El usuario pasa a estar registrado y ahora es un “Usuario comprador”.
Flujo principal	- En la pantalla de registro, el visitante rellena los campos requeridos para el registro. - Hace “click” en el botón "registrarse". -El sistema envía un email al correo del usuario confirmando el registro correcto. - El sistema confirma el registro.
Flujo alternativo	Si los datos ingresados son incorrectos o ya existen, al hacer clic en "Registrarse", se

	permanecerá en la pantalla de registro y se informará del error.

Id	**CU01.2**
Caso de uso	Inicio de sesión de la aplicación
Actor	Usuario Comprador, Usuario Vendedor, Administrador
Descripción	Pantalla que muestra dos campos para que el usuario pueda introducir su nombre de usuario y contraseña y un botón que le permita iniciar sesión y acceder a la aplicación.
Precondiciones	El usuario tiene que estar previamente registrado en la aplicación.
Postcondiciones	Se crea un identificador para la sesión de usuario y accede a la pantalla principal de la aplicación, con sus opciones en función de los permisos o rol.
Flujo principal	- En la pantalla principal, el usuario rellena los campos "Usuario" y "Contraseña" con sus datos de acceso. - Se hace clic en el botón "acceder". - El sistema redirige a la pantalla principal que muestra las opciones que el usuario tiene disponibles para sus permisos.
Flujo alternativo	Si al apretar a "acceder" el nombre de usuario o la contraseña son incorrectos, se permanecerá en la pantalla principal y se informará del error.
Prototipo	Corresponde a la Captura núm. 2 de los modelos de pantalla.

Id	CU01.3
Actores	Usuario Comprador, Usuario Vendedor, Administrador
Caso de uso	Cerrar sesión de la aplicación
Descripción	Opción que permite al usuario cerrar su sesión y salir de la plataforma.
Precondiciones	El usuario debe haber iniciado sesión previamente.
Postcondiciones	La sesión del usuario se cierra y se redirige a la pantalla de inicio.
Flujo principal	- El usuario selecciona la opción “Cerrar Sesión” desde su perfil o menú de navegación. - El sistema cierra la sesión activa del usuario. - Se redirige al usuario a la pantalla de inicio.

Id	CU02
Actores	Visitante, Usuario (indistinto de rol), Administrador
Caso de uso	Buscar inmuebles
Descripción	Pantalla con un formulario o barra de búsqueda que permite ingresar criterios para encontrar inmuebles disponibles.
Precondiciones	Ninguna, ya que en principio un visitante puede buscar inmuebles.
Postcondiciones	Se muestra una lista de inmuebles que coinciden con los criterios de búsqueda.
Flujo principal	- El actor introduce criterios de búsqueda (todavía no definidos) como ubicación o

	dirección, precio, tipo de inmueble, etc. - Presiona en "Buscar". - El sistema muestra una lista de inmuebles que coinciden con los criterios ingresados.
Flujo alternativo	- Si no se encuentran coincidencias, el sistema muestra un mensaje indicando que no hay resultados para la búsqueda solicitada.
Prototipo	Corresponde a la 1era Captura de los modelos de pantalla. Y la captura 3era sería el resultado de la acción.

Id	CU03
Actores	Usuario comprador
Caso de uso	Pujar por un inmueble
Descripción	Funcionalidad que permite al usuario "comprador" realizar una puja por un inmueble en subasta.
Precondiciones	El usuario comprador debe estar registrado y haber iniciado sesión. El inmueble debe estar validado y en subasta actualmente.
Postcondiciones	La puja se registra y, en ese momento, el usuario comprador se convierte en el principal postor.
Flujo principal	- El usuario selecciona un inmueble en subasta. - Hace clic en el botón "Pujar - Introduce la cantidad deseada y confirma la puja.

	- El sistema registra la puja. - Si es la oferta más alta, el sistema notifica al usuario que es el principal postor. Si no es así, se actualizan las cantidades y se notifica que debe hacer una oferta mayor.
Flujo alternativo	- Si el tiempo de la subasta ya ha terminado o el inmueble ya ha sido vendido, el sistema informa al usuario y no permite realizar la puja.

*Se ha permitido introducir cualquier cantidad controlando los posibles errores

Id	**CU04**
Caso de uso	Enviar/recibir mensajes
Actores	Usuario (indistinto de rol)
Descripción	Funcionalidad que permite al usuario enviar y recibir mensajes directos a otros usuarios dentro de la plataforma**.**
Precondiciones	El usuario debe estar registrado y haber iniciado sesión.
Postcondiciones	El mensaje es enviado y el receptor recibe una notificación.
Flujo principal	- El usuario accede a la sección de mensajes o a través del botón “Contactar” del inmueble (irá a parar al propietario) al que desea enviar el mensaje. - Redacta su mensaje en el campo correspondiente. - Hace clic en el botón "Enviar". - El sistema envía el mensaje y notifica al receptor.

	-El sistema confirma al emisor con un mensaje de “enviado correctamente”
Flujo alternativo	- Si hay un error al enviar el mensaje, el sistema notifica al usuario.
Prototipo	Captura 5 – Enviar mensaje en las pantallas Captura 6 – Mensaje enviado correctamente

Id	CU05
Actores	Usuario (indistinto de rol)
Caso de uso	Consultar subastas anteriores
Descripción	Permite al usuario revisar las subastas en las que ha participado anteriormente.
Precondiciones	El usuario debe estar registrado y haber iniciado sesión.
Postcondiciones	Se muestra un historial de subastas anteriores.
Flujo principal	- El usuario accede a la sección "Subastas Finalizadas". - El sistema muestra un listado de subastas anteriores con detalles relevantes.

Id	CU06
Actores	Usuario comprador
Caso de uso	Añadir a favorito un inmueble

Descripción	Permite al usuario comprador marcar un inmueble de interés para fácil acceso en el futuro.
Precondiciones	El usuario debe estar registrado y haber iniciado sesión.
Postcondiciones	El inmueble se añade a la lista de favoritos del usuario.
Flujo principal	- El usuario visualiza el inmueble de interés. - Hace clic en el icono o botón "Favoritos". - El sistema añade el inmueble a la lista de favoritos del usuario y cambia el color y el texto del icono, por el texto “Añadido a favoritos”.
Prototipo	Captura 4 de los modelos de pantallas del siguiente apartado.

Id	**CU07**
Actores	Usuario vendedor
Caso de uso	Agregar inmueble
Descripción	Permite al usuario vendedor agregar un nuevo inmueble para subastar.
Precondiciones	El usuario debe estar registrado como vendedor y haber iniciado sesión.
Postcondiciones	El inmueble es añadido al sistema de subastas.
Flujo principal	-El usuario accede a "Agregar nuevo inmueble" -Rellena o edita los campos necesarios para insertar un nuevo inmueble y sube documentación. -Hace click en el botón "guardar".

	-El sistema guarda el inmueble y muestra una confirmación, pendiente de validación. - El administrador valida el inmueble y queda automáticamente guardado en el sistema para poder iniciar una subasta cuando el vendedor lo considere. - El sistema notifica al "usuario vendedor" de la validación del inmueble.
Flujo alternativo	- Si el inmueble no es validado por el administrador, se elimina del sistema. -Se notifica al vendedor.
Prototipo	Captura 7 – Formulario agregar un inmueble

Id	CU11
Actores	Administrador
Caso de uso	Validar inmuebles
Descripción	Funcionalidad que permite al administrador validar inmuebles para garantizar que son del propietario y legalmente registrados antes de que sean visibles en la plataforma.
Precondiciones	El usuario administrador debe haber iniciado sesión.
Postcondiciones	El inmueble validado es visible en la plataforma para todos los usuarios.
Flujo principal	El administrador accede a la lista de inmuebles pendientes de validación.

	Selecciona el inmueble que desea validar. Hace clic en "Validar". El sistema marca el inmueble como validado y este pasa a ser visible en la plataforma.
Flujo alternativo	Si el inmueble no está correctamente escriturado/registrado, el administrador puede rechazar la validación y notificar al vendedor. El sistema lo marca como DELETED

Nuevos casos de uso agregados durante la implementación:

- La razón por la que se agrega este caso de uso es porque si una subasta puede quedar desierta, tiene sentido que el vendedor pueda volver a subastar el inmueble más adelante y a precio distinto, de lo contrario quedaría en un “limbo” sin ninguna lógica real, esto me ha llevado a que, una vez esta validado el inmueble, el usuario decide cuando subastarlo, si la subasta queda desierta, puede volverlo a subastarlo.

Id	CU13
Actores	Usuario vendedor
Caso de uso	Inicializar subasta
Descripción	Funcionalidad que al propietario de un inmueble iniciar una subasta de este.
Precondiciones	El usuario vendedor debe haber iniciado sesión. El inmueble debe estar validado, en ningún otro estado se permite iniciar subasta.
Postcondiciones	El inmueble subastado es visible en la plataforma para todos los usuarios.

Flujo principal	El vendedor accede a sus inmuebles. Selecciona el inmueble que desea subastar. Introduce el importe por el que lo quiere subastar. El sistema inicializa una subasta que durará (predeterminado) 14 días por el importe seleccionado.
Flujo alternativo	Si el inmueble no tiene el estado validado no será posible inicializar una subasta por parte del vendedor.

Id	CU14
Actores	Usuario (indistinto de rol)
Caso de uso	Consultar subastas activas
Descripción	Permite al usuario revisar las subastas que actualmente están activas.
Precondiciones	Ninguna, un visitante puede ver las subastas activas.
Postcondiciones	Se muestran las subastas por las que se puede pujar actualmente.
Flujo principal	- El usuario accede a la sección "Subastas Activas". - El sistema muestra un listado de subastas con detalles relevantes y permite pujar (si se está logeado).

2.3 Modelo de pantallas (Prototipo)

El prototipo se ha realizado utilizando la herramienta que se empleó en la asignatura de Interacción persona ordenador, Figma.

En este diseño de las pantallas se ha intentado tener presentes los principios básicos de IPO, se describen brevemente a continuación algunos de ellos:

Affordance

Botones claros: En nuestra página los botones para las distintas acciones "Acceder", "Guardar", "Buscar" y otros tienen un diseño claro, indicando que son clickables y trataremos que tengan iconos que representan su función (por ejemplo, una estrella para "favoritos" o una lupa para “Buscar).

Barra de búsqueda: Se destaca en la parte central de la página, indicando que uno de los principales affordances de nuestra app es buscar propiedades.

Filtros: Los presentaríamos en forma de desplegable o que indiquen al usuario que puede concretar y depurar sus búsquedas según diferentes criterios.

Metáforas

Mapa interactivo: Muchas páginas de bienes raíces usan mapas para mostrar la ubicación de las propiedades y la idea sería que en nuestra aplicación lograra haberlos. Este es una metáfora de un mapa real, ayudando a los usuarios a ver dónde se localiza el inmueble y les permite explorar las ubicaciones para ver el barrio, etc.

Galería de fotos: Al ver propiedades se intentará lograr desarrollar una especie de álbum o galería con las fotos, esto permite recordar a un álbum de fotos real, como cuando uno va a una agencia inmobiliaria y le enseñan una ficha de un inmueble.

Iconografía: Los íconos que vamos a intentar poner, representan diferentes características de una propiedad (como baños, habitaciones) o de las

distintas acciones (lupa, etc…), estas no dejan de ser metáforas visuales que simplifican la información para el usuario.

Retroalimentación Visual

En cuanto a lo que se refiere a las señales visuales que un sistema proporciona en respuesta a las acciones del usuario, pretendo, por ejemplo, si un usuario hace clic en un botón para enviar un formulario, la aparición de un mensaje que diga "Enviado con éxito", esto proporciona retroalimentación visual.

Restricciones

Para la elección de los colores solamente he tenido en cuenta la no utilización del rojo para la web ya que por razones culturales no es un color adecuado, pues indica error o peligro. He usado una combinación que he visto que funcionaria por internet, y poco más en ese tema.

Se adjuntan imágenes de las distintas pantallas.

*Estas han sufrido cambios de diseño en el proyecto real, aunque todos los atributos se mantienen, los cambios son más de diseño de frontend por falta de tiempo se ha hecho más simple

Captura 1 – Buscador y pantalla principal

Se puede ver cómo será la pantalla inicial con el buscador de subastas en la siguiente ilustración:

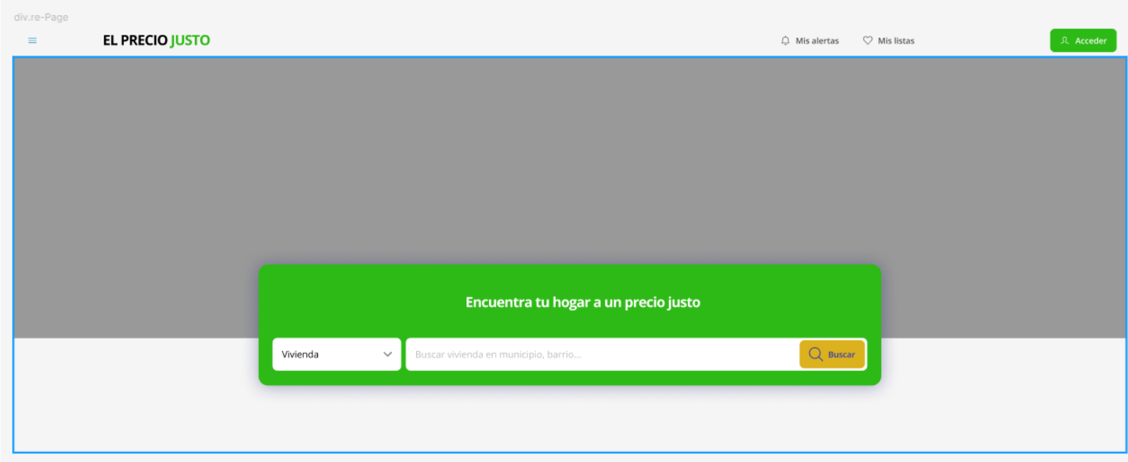


Ilustración 5 Buscador y pantalla principal

Captura 2 – Inicio de sesión

En la ilustración 6 se puede ver cómo será la pantalla de login, que constará de una ventana modal que permita insertar el email y la contraseña.

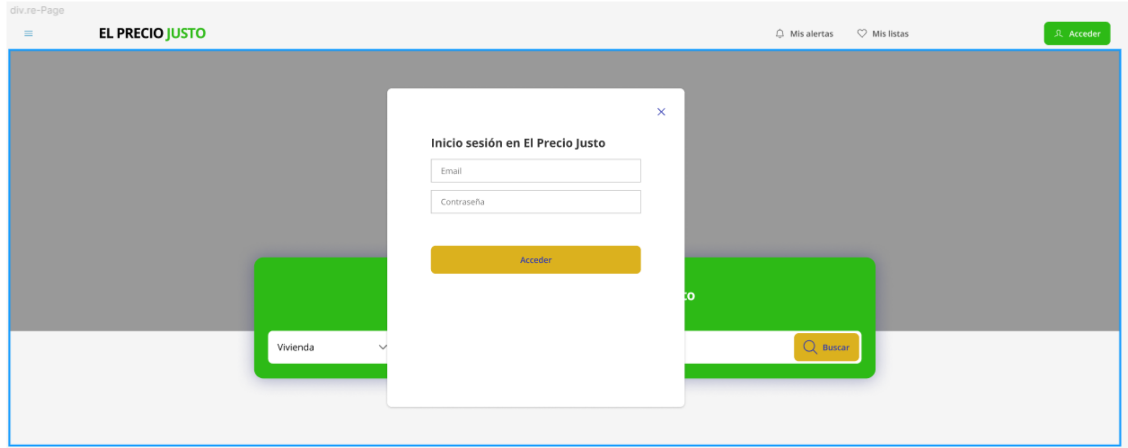


Ilustración 6 Inicio de sesión

Captura 3 - Lista de inmuebles

Una vez iniciada la sesión además de la lista de inmuebles, se podrá contactar con el propietario, agregar a favoritos y accediendo al inmueble, pujar por él (ilustración 7).

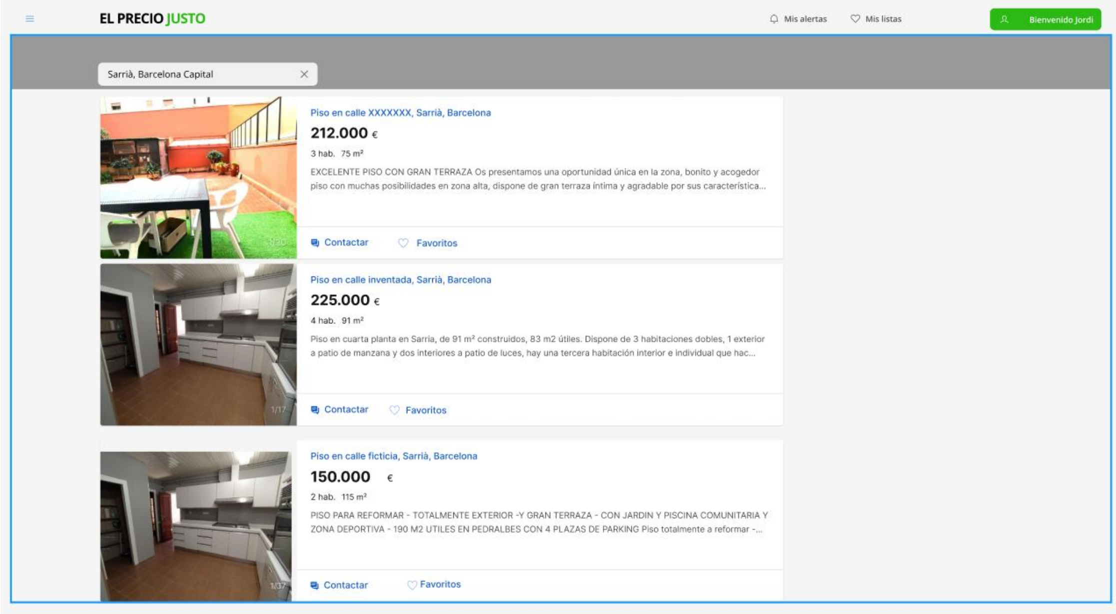


Ilustración 7 Lista de inmuebles con foto

Captura 4 – Inmueble añadido a favoritos

En la ilustración 8 se muestra cómo ve el usuario el inmueble añadido a favoritos después de clicar en el icono, estando "logeado".

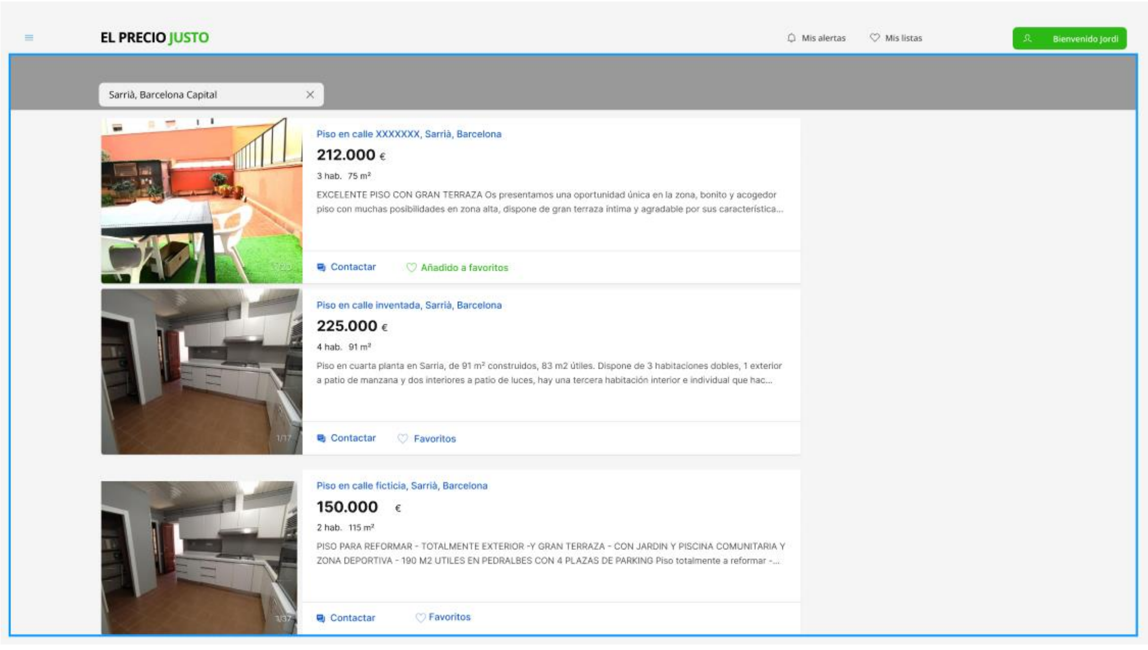


Ilustración 8 Agregar a favorito

Captura 5 – Enviar mensaje

Esta funcionalidad estará disponible desde el listado de inmuebles.

Se ha añadido durante la implementación un apartado de mensajes del usuario, donde puede agregar nuevos mensajes al hilo, a modo de “chat”, como muestra la ilustración 9, como funcionalidad no deja de ser enviar un mensaje igual que este ejemplo de inicio de hilo, pero anidándolo a un mensaje previo.

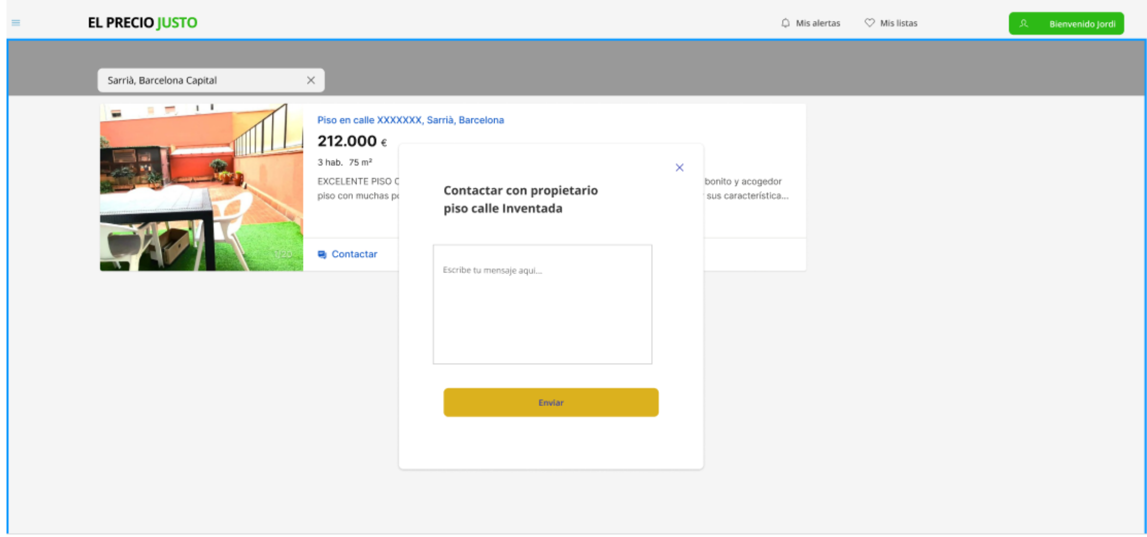


Ilustración 9 Enviar mensaje

Captura 6 – Mensaje enviado correctamente

Esta captura muestra el mensaje enviado correctamente (ilustración 10), si hubiera algún problema el mensaje se mostraría en rojo debajo del texto escrito, por ejemplo, el sistema de avisos seria idéntico para otras acciones como el pujar.

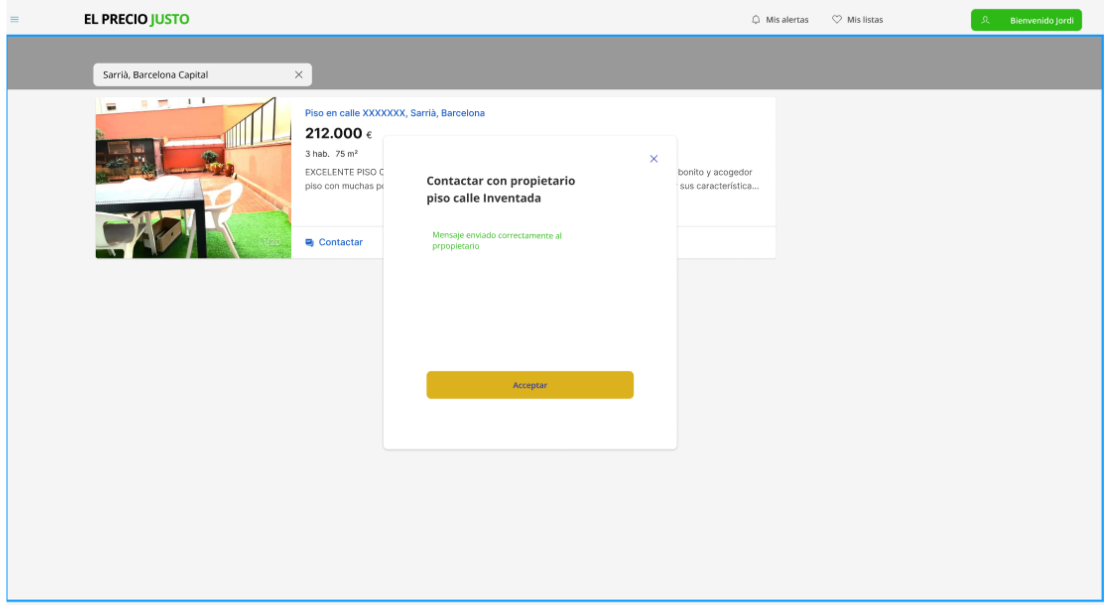


Ilustración 10 Mensaje enviado correctamente

Captura 7 – Formulario para agregar un inmueble

Se puede ver el modelo de formulario para agregar un nuevo inmueble a las propiedades del usuario en la siguiente ilustración:

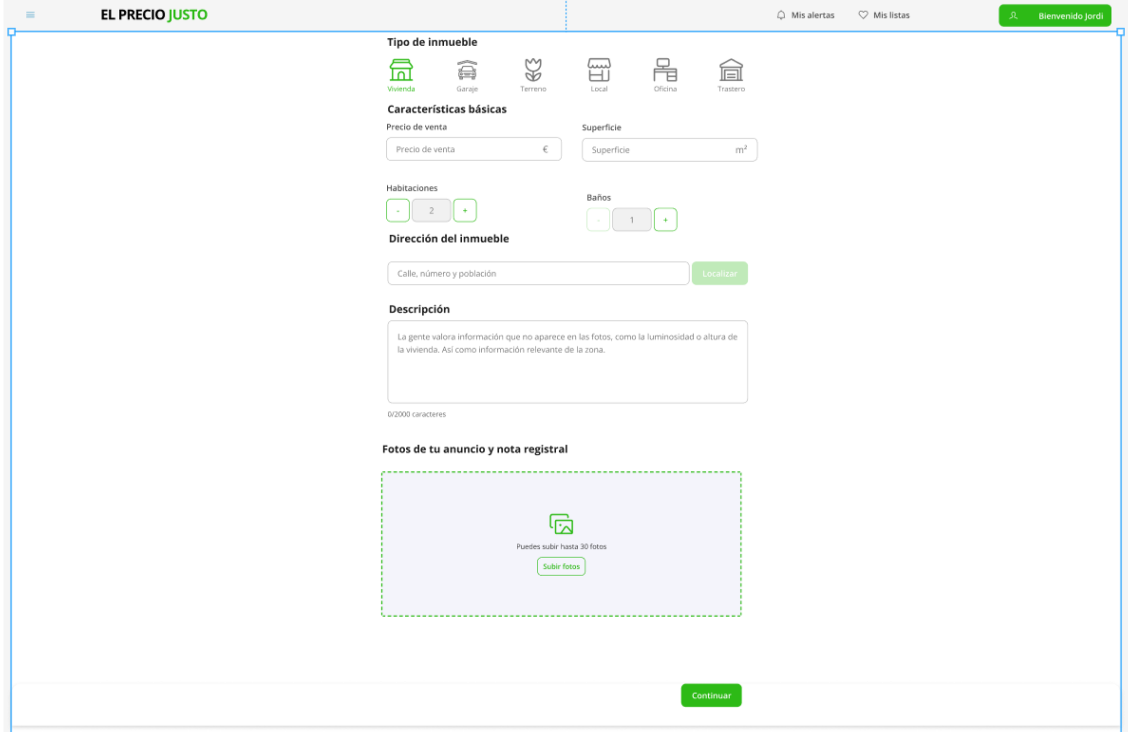


Ilustración 11 Formulario para agregar un inmueble

Se debe comentar que, una vez agregado el inmueble, se esperará a la validación del inmueble por parte del administrador y cuando sea validado, tendrá el estado de “Validated” y se dejará al usuario decidir el precio de la subasta. La duración de la subasta es de 14 días por defecto, para simplificar, pero sería fácil cambiarlo en una expansión.

El estado pasaría a “In_auction” al crearla, al terminar los días establecidos si no ha tenido pujas, pasa de nuevo a “Validated” y si ha tenido pujas pasa a “Sold”.

Esa sería la idea inicial, del proceso de cómo controlar la subasta del inmueble propiamente dicha.

El acta o nota registral de la propiedad será agregada en un apartado distinto y almacenada en la BBDD para una posterior validación por parte del administrador del sistema, (y confirmar el cambio de la propiedad del inmueble de usuario si esto fuera necesario, ya que, puede ocurrir que otra persona Y sea la propietaria actual en nuestro sistema de ese inmueble con X referencia catastral, registrando en la tabla intermedia las fechas en las que el usuario Y fue propietario), pero para hacerse una idea a nivel de Modelo de Pantalla, se ha agrupado en un mismo “drag and drop” de imágenes.

Para intentar hacer el sistema más “realista” se tiene en cuenta que no puede haber dos inmuebles en la plataforma con la misma referencia catastral, más adelante se explicará cómo se puede “cambiar de manos” el inmueble y qué condiciones se tienen que dar y como se ha solucionado esa parte lógica.

2.4 Diagrama de arquitectura

La arquitectura de la aplicación consta de cuatro microservicios con servicios API REST a los que el usuario tendrá acceso desde su dispositivo

mediante una App (con React Native) y a través de un Gateway, ver ilustración 12.

Se utilizará Docker Compose que permitirá ejecutar contenedores con las bases de datos y los microservicios (si los consigo configurar correctamente) a través del fichero YAML *docker-compose.yml.*

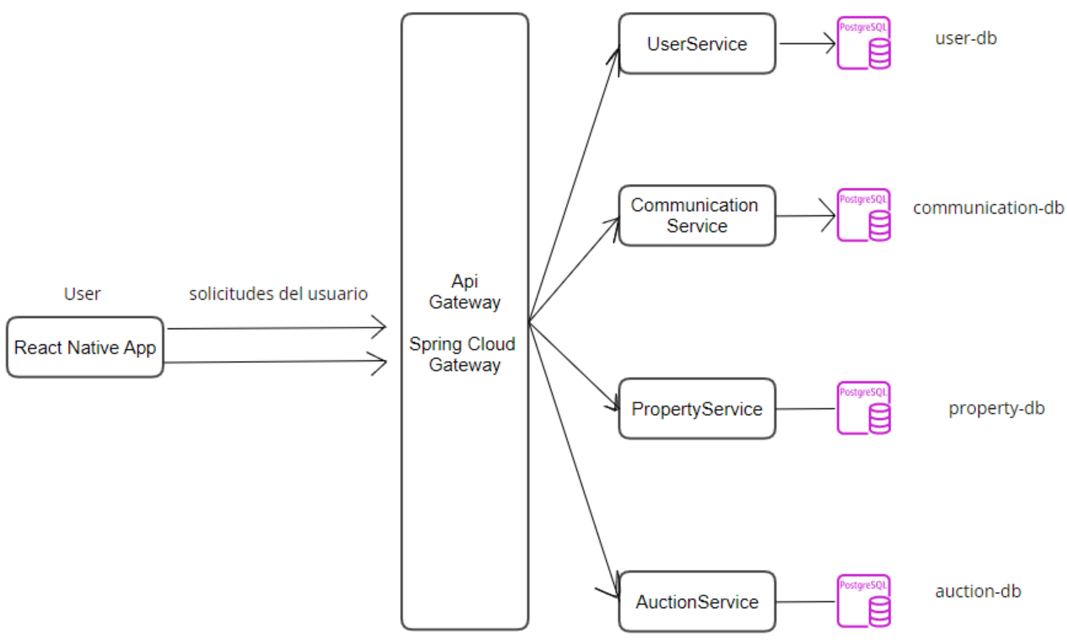


Ilustración 12 Arquitectura del proyecto

La idea que se intentará plasmar, de momento todo a nivel teórico, luego a ver si es posible implementarla, es usar una capa intermedia con **Spring Cloud Gateway**, que de alguna forma es el enrutador de las peticiones del usuario hacia los microservicios, para no llamar desde **React** directamente a los **endpoints** de los microservicios, si no se consiguiera, se usarían directamente los endpoints.

Al ser una arquitectura pensada en microservicios, es deseable por principios de diseño tener su base de datos independiente para cada uno (ilustración 12), por un tema de desacoplamiento, escalabilidad y seguridad, con la parte necesaria para su funcionamiento independiente, equivalente a las entidades y tablas que ya luego se comentará con más detalle.

Así, de una forma general, la arquitectura de la aplicación se podría dividir en dos partes:

- **Front-end que será una aplicación con React Native**

  Esta aplicación actuará como cliente y básicamente intentare comunicarme satisfactoriamente con los microservicios de nuestra API y a partir de estos montar las pantallas o vistas necesarias, empezaré con una estructura simple y básica y si posteriormente, se ve que da tiempo, ir añadiendo o bien funcionalidades, o bien mejorando las que tenga, por ejemplo el registro de propiedades, se empezará con datos básicos para posteriormente implementar imágenes que exigen de un servicio de almacenamiento externo como **Firebase storage** y el uso de **Nominatim** para la búsqueda de direcciones.

- **Back-end, Microservicios con servicios Api Rest con arquitectura hexagonal**:

  Aquí se configuraría el Api Gateway con Spring Cloud Gateway que actuaría como un punto de entrada unificado y poder controlar el enrutamiento, y autenticación de los usuarios, como se puede ver en las ilustraciones 12 y 13.

  Se utilizará InteliJ y **SpringBoot** para desarrollar la parte del back-end de los microservicios y APIs REST y además se configurará un **Spring Cloud Gateway** como otro "microservicio" que haga de Gateway, y a través de **Spring Security** se gestionarán los permisos para los distintos endpoints de los microservicios y autenticación en el Gateway y microservicios.

  Se usará **Dockers** para los contenedores y el despliegue de la aplicación (el backend) y **Docker Compose** que permite definir como

ha de ser la ejecución de varios contenedores, de los servicios y las bases de datos que se ven en el diagrama.

Se definen también en ese fichero YAML, llamado docker-compose.yml los servicios de **Kafka**, **Zookeeper** y **Adminer** que se encargan de la gestión de Kafka y el segundo de la gestión web de las bases de datos de los microservicios.

También se utilizará **Spring Boot** y varias de sus funcionalidades y starters, según necesidades que hemos comentado en la PEC1 y en esta misma PEC2 (y otras que surjan). También hay que comentar que, en base a lo utilizado en las asignaturas de programación realizadas en otros años, usaremos **Lombok** que permite ahorro al codificar métodos y procesos mediante anotaciones que puedes agregar a tus clases, las cuales se procesan en tiempo de compilación para generar automáticamente el código correspondiente.

También se usará **Swagger** y **Log4j2**, para facilitar el desarrollo de los microservicios y tener una forma más visual de acceder a ellos y ver más fácilmente cómo funciona nuestra API Rest,

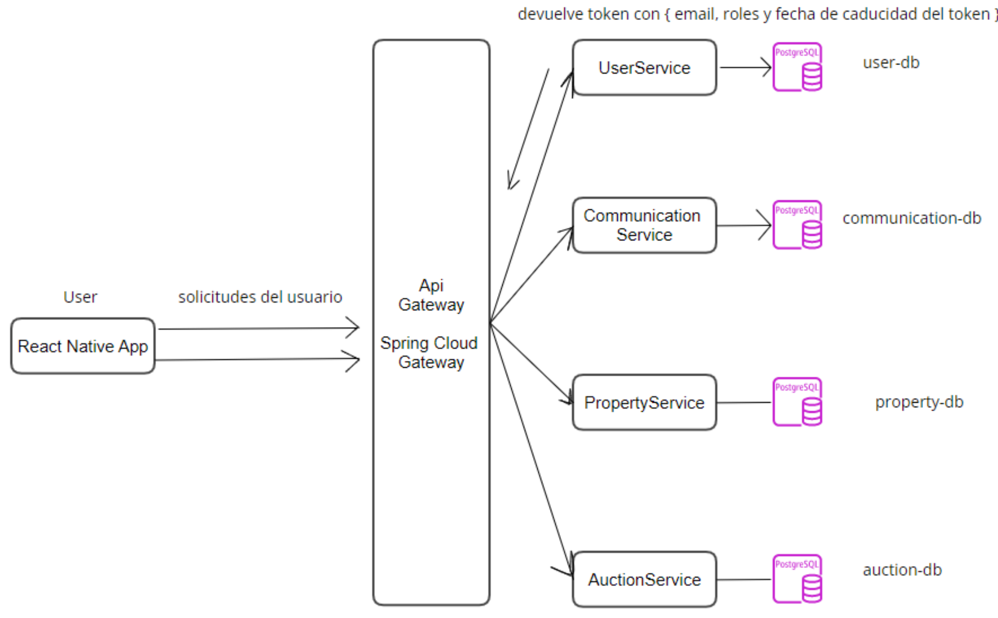


Ilustración 13 Boceto del retorno del microservicio de User del token JWT

En cuanto a la gestión de permisos para consultar la API y sus distintas funcionalidades, como se puede ver en la ilustración 13 , se puede ver como es el microservicio UserService a través del endpoint de login el que valida si el usuario existe y devuelve un **token JWT,** cuyo contenido será:

- el email del usuario, el identificador, el nombre y los permisos que tiene este usuario y una fecha de expiración del propio token(se define en 1 día por defecto)

De forma que, una vez devuelto ese token al cliente, el navegador lo almacene con *AsyncStorage* para agregarlo en las cabeceras de las peticiones subsiguientes.

Concretamente en la cabecera de la autorización de cada petición (Bearer) se pondrá el token que gestionará el Gateway para validarlo (sino lo harían los microservicios) en un adapter de entrada que haría como filtro para validar los tokens.

Al hacer logout, lo que se haría seria destruir el token con AsyncStorage de la aplicación cliente y llamar a UserService que registrará el logout con un log por seguridad o lo que quisiera implementarse en el futuro.

2.5 Microservicios

Los microservicios usaran las tecnologías comentas anteriormente:

Lenguaje desarrollo	Java 17
Herramienta de gestión del proyecto	Maven
manipulación de datos	Spring JPA repository
Base de datos	PostgreSql
Productor / consumidor mensajes	Kafka
Automatización configuración	Spring boot
Librería facilidad generar código	Lombok
Seguridad / Gestión permisos	Spring Security

El diseño de los microservicios se realizará aplicando un patrón de arquitectura hexagonal quedando establecido de la siguiente forma:

UserService

De este microservicio, ver ilustración 14, lo más importante es la validación (login) y registro de usuarios, el cual debe notificar al usuario enviando un mensaje a un topic de Kafka que el microservicio ComunicationService se encargará de consumir y proceder al envío del email de confirmación de registro.

Finalmente, durante la implementación, se ha agregado un consumidor de Kafka en el microservicio de User para poder añadir el rol de “SELLER” cuando el Administrador valida un inmueble del usuario (y así poder tener acceso a las opciones de menú de ese rol).

Además del Producer para agregar un mensaje al topic de agregar usuario cada vez que se registra un usuario como ya se tenía previsto, para que el microservicio de communication envíe el email de “registro correcto”, se ha incorporado un TokenService, que básicamente gestiona, genera/resuelve un token JWT a partir de los datos del usuario validado al hacer “login”, por lo tanto realmente lo único que se verifica en UserService al logearse es si *findUserByEmail* devuelve un usuario (es decir está en nuestra base de datos registrado) y si es válido se devuelve el token con sus datos con una validez de 24 horas, este token es el que se usa tanto en el *frontend* como en todos los microservicios para identificar al usuario.

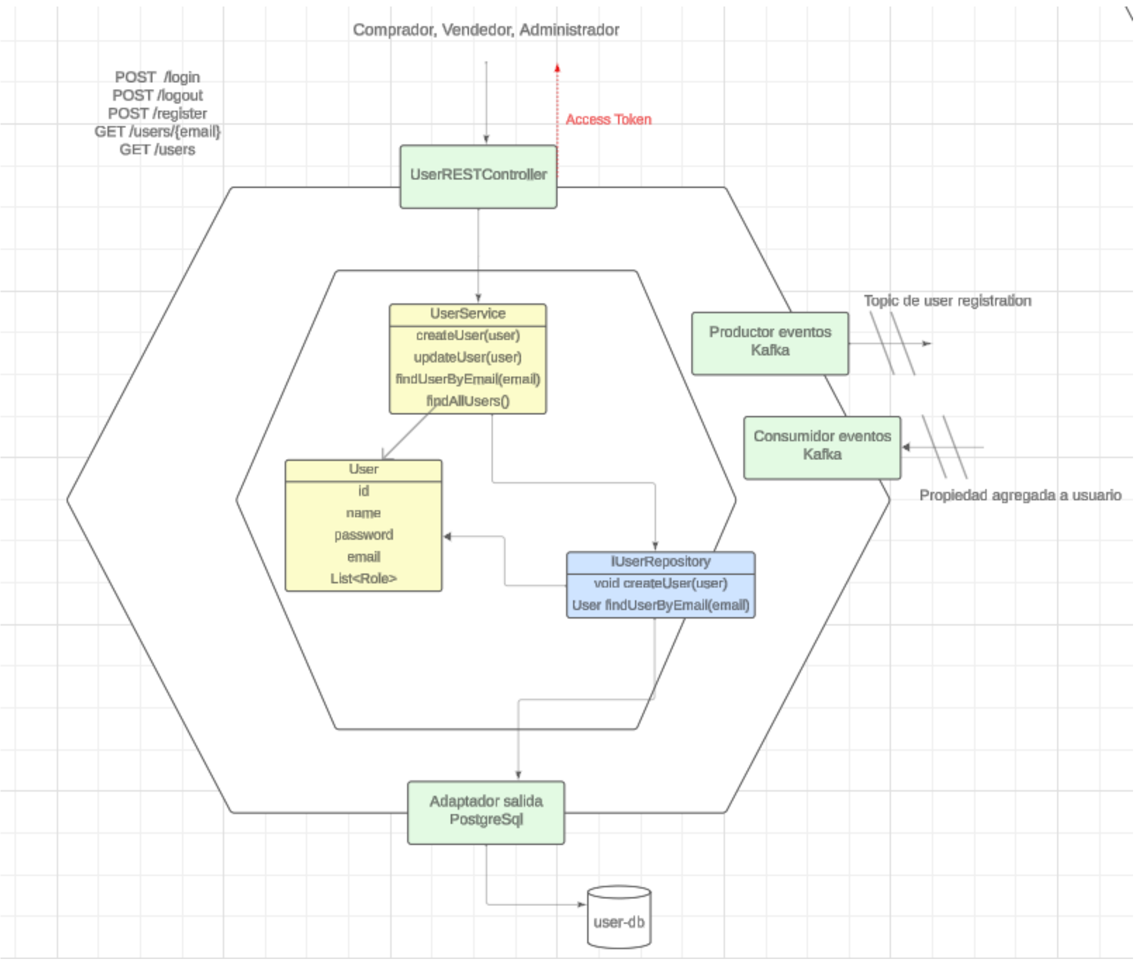


Ilustración 14 Esquema Microservicio User

CommunicationService

En un principio seria consumidor del topic anteriormente comentado en UserService y a su vez enviaría notificaciones fuera del sistema mediante email y guardaría mensajes de chat entre usuarios en el sistema, ver ilustración 15.

Además, consume mensajes de los topic del microservicio de Property, que serían:

Validar, invalidar (eliminar) una propiedad y solicitud de cambio de propietario (esto ocurre cuando alguien quiere registrar un inmueble que actualmente está en posesión de otro usuario en estado activo, se le envía email al propietario actual por si quiere eliminar el inmueble porque ya no es suyo).

Este microservicio también es el responsable de guardar los mensajes entre usuarios y la posterior recuperación de estos desde el perfil de cada usuario.

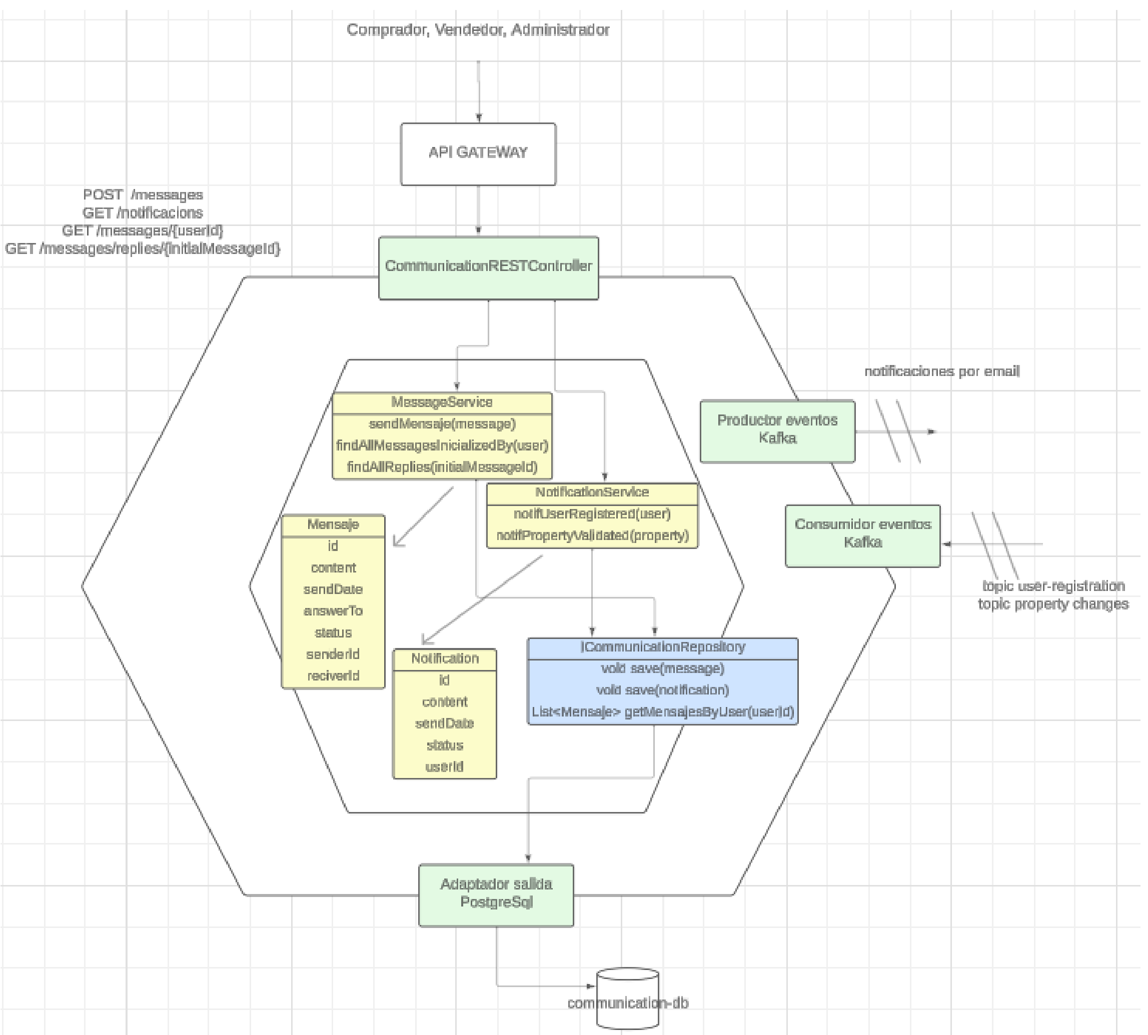


Ilustración 15 Esquema Microservicio Communication

AuctionService

Este microservicio es el responsable de crear las subastas y de registrar las pujas de los usuarios, se definen algunas de las funcionalidades en el diagrama de la ilustración 16, el resto se especifican más adelante.

Al iniciar una subasta, es cuando se marcará el precio inicial de la misma. A través de Kafka notifica el inicio de una subasta y el final de una subasta.

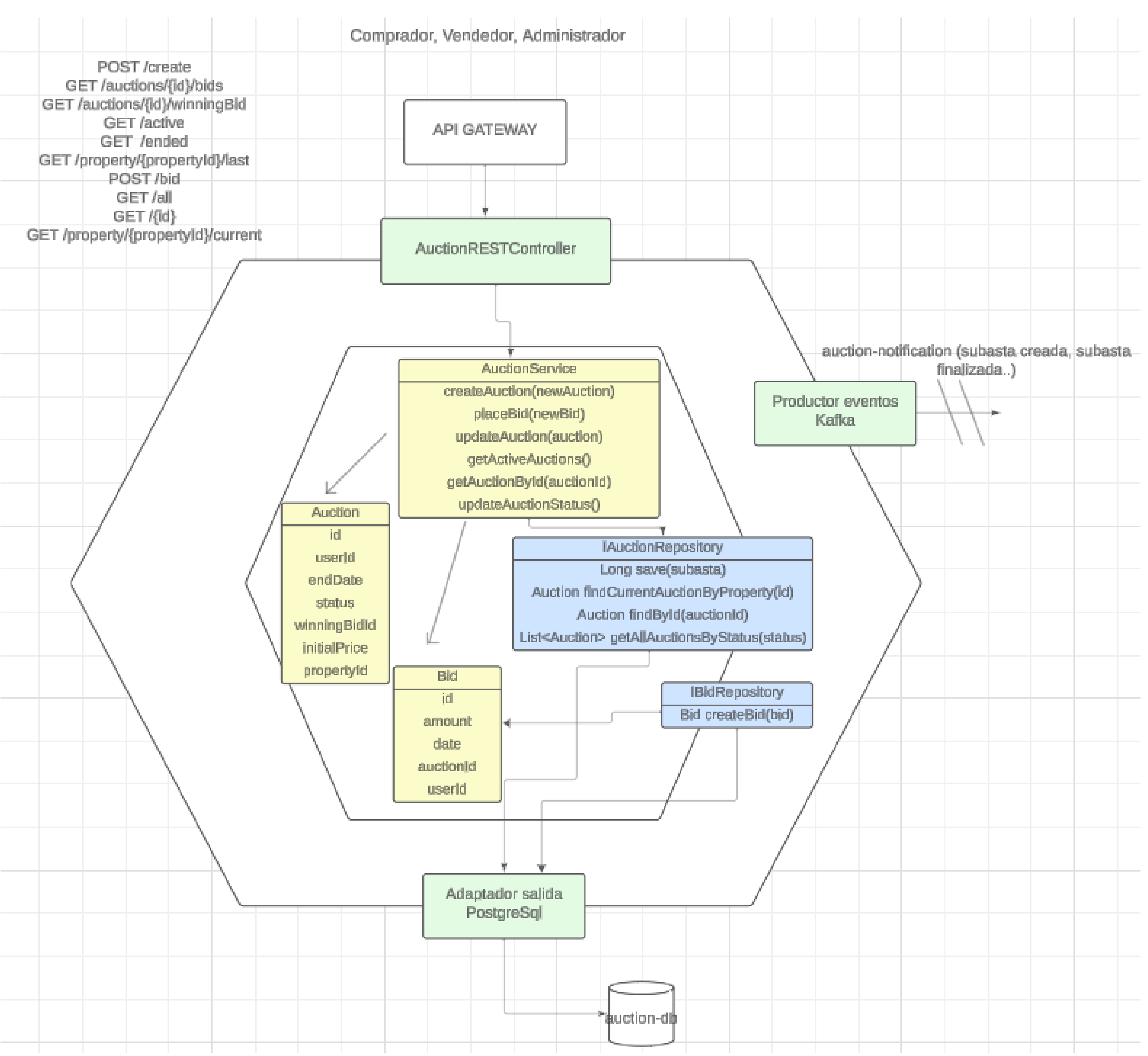


Ilustración 16 Esquema Microservicio Auction

*Nota, también el visitante seria parte de los consumidores del servicio, teóricamente no hace falta un rol para acceder a algunas partes, como ver las subastas actuales (no así pujar en ellas).

PropertyService

Este microservicio es el encargado de registrar una propiedad y sus imágenes, véase la ilustración 17, así como registrar los propietarios de estas en el tiempo.

También, a través del AdressService agrega el país, ciudad, región y código postal de las propiedades que hay en el sistema (para futuras mejoras).

Además, modifica el estado de la propiedad en función de las subastas gracias al *consumer* de Kafka de los *topic* de "subasta creada" y "subasta finalizada".

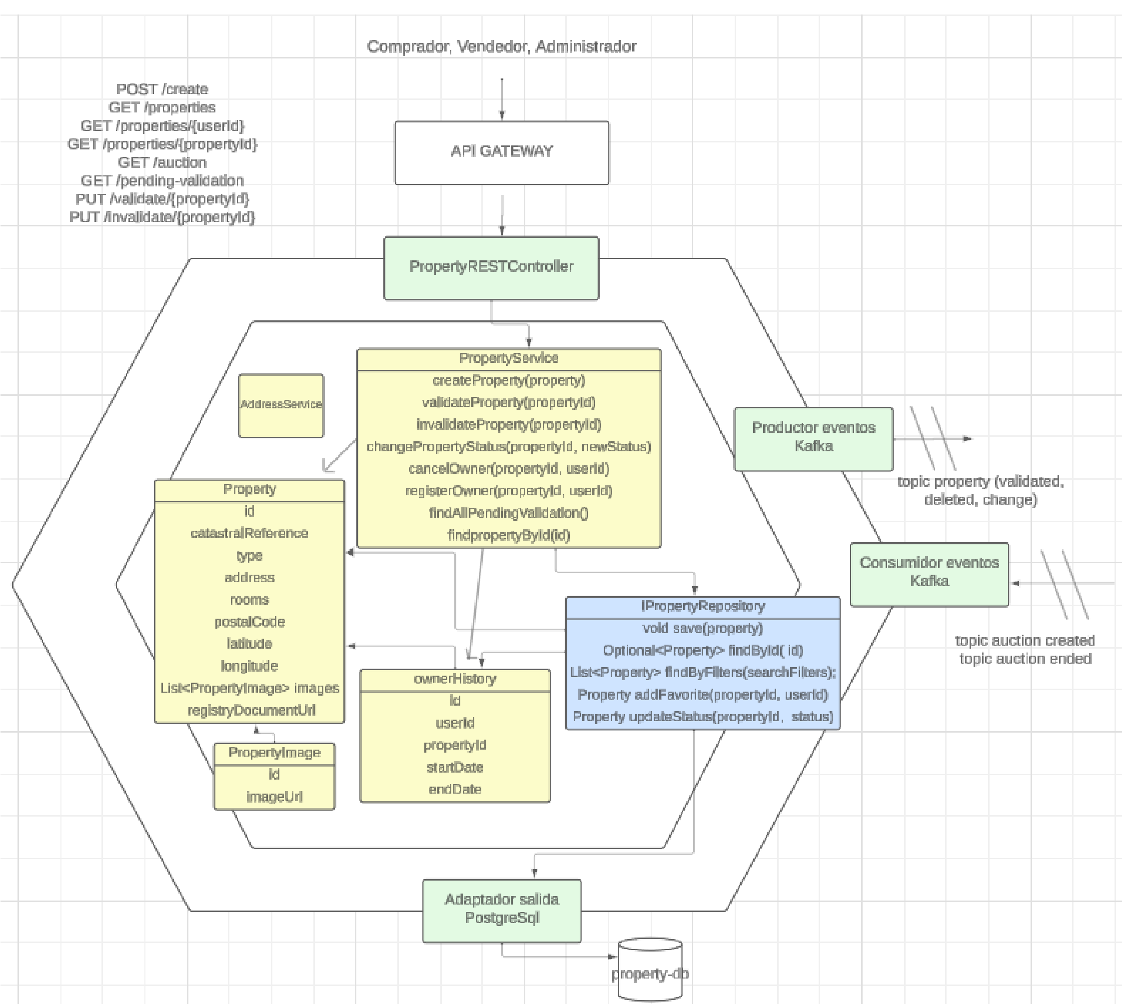


Ilustración 17 Esquema Microservicio Property

*Nota, también el visitante seria parte de los consumidores del servicio, teóricamente no hace falta un rol para acceder a algunos recursos, como ver propiedades.

2.6 Operaciones y consultas

Las operaciones de las que consta el proyecto y, por lo tanto los distintos microservicios, son las siguientes:

- Microservicio User

```
List<User> findAllUsers();

Optional<User> findUserById(Long id);

Optional<User> findUserByMail(String email);

User createUser(User user);
void updateUser(User user);
void deleteUser(Long id);
```

- Microservicio Communication

```
boolean notifyUserRegistered(User user);
List<Notification> findAllNotifications();

boolean notifyPropertyValidated(Property property);
boolean notifyPropertyDeleted(Property property);
boolean notifyChangeRequest(Property property);
```

```
boolean sendMessage(Message message);
List<Message> findAllMessagesInicializedBy(Long userId);
List<Message> findAllReplies(Long initialMessageId)
```

- Microservicio Property

```
Property createProperty(Property property);
Property validateProperty(Long propertyId);
Property invalidateProperty(Long propertyId);
```

```
Property deleteProperty(Long propertyId, Long petitionUserId);
Property changePropertyStatus(Long propertyId, PropertyStatus status);
boolean cancelOwner(Long propertyId, Long userId);
Long registerOwner(Long propertyId, Long userId);

boolean sendChangePropertyRequest(Property property, String
userRequestEmail);
boolean isUserActualOwner(Long propertyId, Long userId);
List<Property> findAllProperties();
List<Property> findAllPendingValidation();
List<Property> findAllPropertiesInAuction();
List<Property> findActivePropertiesByOwner(Long userId);
List<Property> findPropertiesByUserExcludingDeleted(Long userId);
Optional<Property> findPropertyById(Long id);
Optional<Property> findPropertyByCatastralReference(String reference);

List<OwnerHistory> findAllOwnersHistory();
```

```
Country addCountry(Country country);
City addCity(City city);
PostalCode addPostalCode(PostalCode postalCode);
Region addRegion(Region region);

PostalCode saveCompleteAddress(String countryName, String regionName,
String cityName, String postalCode);
Optional<Country> getCountry(String name);
Optional<City> getCity(String name);
Optional<Region> getRegion(String name);
Optional<PostalCode> getPostalCode(String name);
```

- Microserviocio Auction

```
Auction createAuction(Auction newAuction);
Auction updateAuction(Auction updatedAuction);

Optional<Auction> getAuctionById(Long auctionId);
Auction getCurrentAuctionByPropertyId(Long propertyId);

Auction getLastAuctionByPropertyId(Long propertyId);

List<Auction> getAllAuctions();
List<Auction> getAllActiveAuctions();
List<Auction> getLastAuctions();
void updateAuctionStatuses();

BigDecimal placeBid(Long auctionId, BigDecimal amount, Long userId);
```

Las siguientes operaciones son comunes a todos los microservicios (cada uno las tiene integradas en un TokenService, independientemente de los otros microservicios) a excepción de createToken que solamente está en el microservicio de User.

```
String createToken(UserSession session);
JSONObject solveToken(String token);
UserDetails extractUserDetails(String jwtToken);
```

2.7 Gateway

Para el desarrollo de esta especie de “microservicio” se ha utilizado Spring Cloud Gateway, actúa como API Gateway y es un intermediario entre el cliente y los microservicios que se encarga de enrutar las peticiones entrantes hacia el microservicio correspondiente.

Una ventaja que ofrece es que a la hora de desarrollar el frontend, usa una única puerta de entrada.

Además, en este proyecto se ha agregado una validación a través de Spring Security que consiste en comprobar si la petición tiene el token necesario (si es el caso) y si este token ha expirado o no (si tiene los roles necesarios la petición los valido en cada microservicio).

Se le pueden añadir funciones de balanceo entre otras, pero en este proyecto no se ha realizado.

Para la configuración del Gateway se le define la ruta del microservicio final y la ruta o path con la que pretenden aceptar las peticiones desde el Gateway, véase la ilustración 18, esto hace que las peticiones sean más amigables o simples ya que todas van por la misma ruta.

En la siguiente captura, podemos ver como, por ejemplo, todas las solicitudes que empiecen por */user* van a enrutarse hacia el microservicio de usuarios que en el apartado anterior he esquematizado.

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-origins=*
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-methods=*
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-headers=*

spring.cloud.gateway.routes[0].id=user-service
spring.cloud.gateway.routes[0].uri=http://localhost:18084
spring.cloud.gateway.routes[0].predicates[0]=Path=/user/**
spring.cloud.gateway.routes[0].filters[0]=RewritePath=/user/(?&lt;segment&gt;.*), /$\{segment}

spring.cloud.gateway.routes[1].id=communication-service
spring.cloud.gateway.routes[1].uri=http://localhost:18082
spring.cloud.gateway.routes[1].predicates[0]=Path=/communication/**
spring.cloud.gateway.routes[1].filters[0]=RewritePath=/communication/(?&lt;segment&gt;.*), /$\{segment}

spring.cloud.gateway.routes[2].id=property-service
spring.cloud.gateway.routes[2].uri=http://localhost:18083
spring.cloud.gateway.routes[2].predicates[0]=Path=/property/**
spring.cloud.gateway.routes[2].filters[0]=RewritePath=/property/(?&lt;segment&gt;.*), /$\{segment}

spring.cloud.gateway.routes[3].id=auction-service
spring.cloud.gateway.routes[3].uri=http://localhost:18085
spring.cloud.gateway.routes[3].predicates[0]=Path=/auction/**
spring.cloud.gateway.routes[3].filters[0]=RewritePath=/auction/(?&lt;segment&gt;.*), /$\{segment}
```

Ilustración 18 Configuración Gateway

2.8 Gestión del token, control de sesiones

En este desarrollo se ha optado por generar un token JWT en el microservicio de User que se almacenará en la aplicación cliente y posteriormente se agregará a todas las peticiones a los distintos microservicios, sirviendo de 2 cosas:

- Para identificar al usuario
- Para comprobar si tiene permisos para acceder al recurso solicitado.

La secuencia de acceso, ilustración 19, seria la siguiente:

En un primer momento, un User ya registrado quiere identificarse en la aplicación, lo hace a través del API REST del microservicio de User por su endpoint de login (este es uno de los casos en los que el API Gateway enruta la petición sin validar si tiene un token previo),
Una vez validado en la aplicación se le devuelve un token de acceso, que servirá para todo el sistema durante 24 horas.

Comprador, Vendedor, Administrador

POST /login
POST /logout

Access Token

UserRESTController

Ilustración 19 Pequeño esquema del login en el microservicio User

Esto se realiza a través de una clase, UserSession, que encapsula un objeto User y un LocalDateTime que representa la fecha de expiración del token.

A través del método getString, ver ilustración 20, se devuelven los datos “necesarios” evitando enviar información sensible en el payload del token.

```
public class UserSession implements Serializable{


    private User user;

    private LocalDateTime expireDate;


    public String getString() {
        return "{" +
                "\"id\":\"" + user.getId() +
                "\"mail\":\"" + user.getEmail() +
                "\", \"roles\":\"" + user.getRoles() +
                "\", \"expire\":\"" + expireDate.format(DateTimeFormatter.ISO_DATE_TIME)+
                "\"}";
    }
}
```

Ilustración 20 Clase que contiene los datos del token

El proceso comienza tras validar que el usuario y la contraseña proporcionados coinciden con los datos almacenados en la base de datos. En este punto, se invoca la función createToken del puerto de salida TokenService, ver ilustración 21.

Esta funcionalidad se implementa utilizando la librería JWT y el algoritmo HMAC256 para la firma del token.

La clave secreta para la firma, inyectada a través de la anotación @Value("${jwt.secret}"), y la clave se almacenan en la configuración para hacerlo más práctico, en un entorno real se recomienda adoptar métodos más seguros para manejar la clave secreta.

```
public interface TokenService {

    String createToken(UserSession session);

    JSONObject solveToken(String token);

    UserDetails extractUserDetails(String jwtToken);
}
```

Ilustración 21 Interfaz TokenService (puerto de salida en User microservicio)

Los demás microservicios, por su parte, implementan el mismo TokenService en sus puertos de entrada, aunque sin la capacidad de crear tokens.
Con Spring Security se realiza la autorización de roles en los “endpoints” de cada microservicio, esto se consigue extrayendo los datos del token en cada *Request* y agregando los roles a una clase propia de Spring Security, *userDetails,* que luego se valida en cada endpoint del API REST (en este proyecto la clase es “customizada” importando userDetails ya que se ha añadido el atributo userId (los microservicios identifican/guardan a los usuarios por el id de usuario y no por email) quizás habría sido mejor de otra forma, aprovechando el email, pero ya estaba muy avanzado el proyecto y se ha optado por seguir guardando el userId como identificador del usuario donde es necesario.

Los endpoints de los microservicios validan si el peticionario tiene los privilegios o roles necesarios para acceder al contenido que solicita.

Las peticiones incluyen el token en su cabecera Bearer, siempre que sea necesario para la autorización y autenticación del usuario, como se muestra en la ilustración 22.

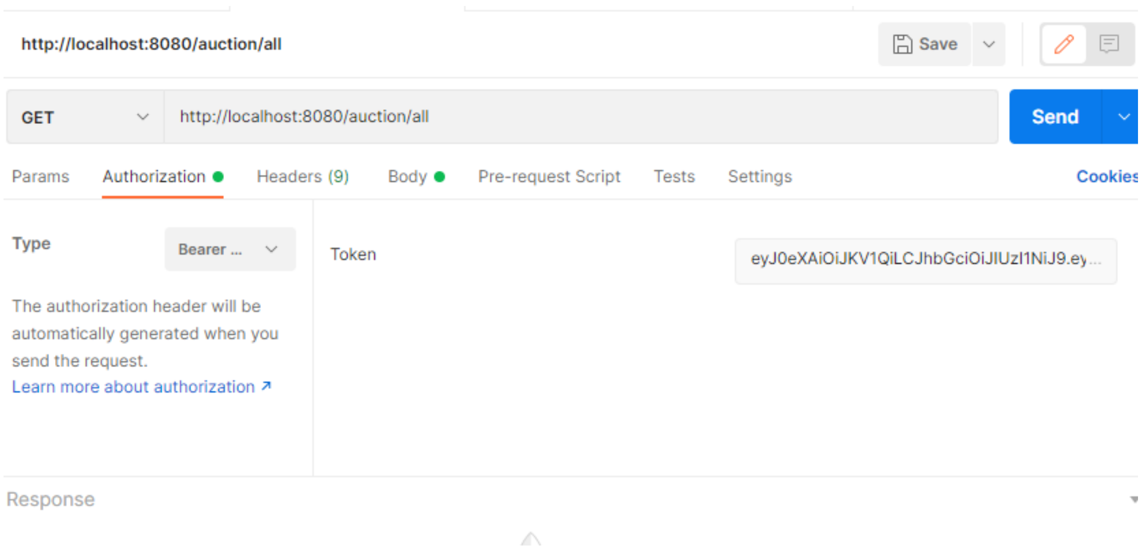


Ilustración 22 Ejemplo de cabecera con token en una petición de la aplicación

Y a nivel de frontend lo que se hace es recuperar el token cuando se registra el usuario y guardarlo durante toda la sesión en un lugar seguro con AsyncStorage, que básicamente es un sistema de almacenamiento local y asincrónico que permite guardar datos de manera persistente en el dispositivo.

Cuando el usuario hace logout, simplemente la aplicación creada con React Native elimina del almacenamiento con AsyncStorage el token guardado y llama al endpoint para dejar constancia de este hecho, en el futuro se puede implementar alguna blacklist o algún sistema de control para los tokens “deslogados”, que dure 24 horas en la lista, hasta que ya caduquen, por poner un ejemplo de otras cosas que se pueden implementar al hacer logout pero no es el caso.

2.9 Diagrama de las clases principales

La idea principal sería usar una combinación de arquitectura de microservicios con una arquitectura hexagonal dentro de cada

microservicio, primero porque es lo que un poco hemos aprendido en las asignaturas de Java durante la UOC y porque es una buena forma de hacer que el sistema sea escalable y, personalmente, me gustó mucho como se podía organizar, aunque no toqué todas las partes del proyecto cuando hice el desarrollo en grupo.

En base a esto, podría tener la siguiente estructura:

Microservicio de Autenticación y Usuarios (**UserService**)

Se encargaría de la gestión de usuarios, roles, autenticación y autorización.

El dominio del problema que se encargaría de resolver este microservicio serían Usuario y Role en la base de datos que a continuación se mostrará.

Los casos de uso que se han definido tendrían que ver con el registro, login, logout, modificación de perfil (bloqueo, etc..), asignación de roles.

Microservicio de Comunicación (**CommunicationService**)

Este microservicio sería el encargado de la gestión de mensajes entre usuarios y notificaciones.

Su dominio o subdominio serían las entidades Message y Notificación y debe tener acceso a los Usuarios mediante el token de autorización para poder registrar el envío de mensajes de la aplicación y enviar notificaciones.

*Todos los microservicios reciben el token una vez logeado el usuario, de esta forma saben si tiene permisos para el recurso que está solicitando.

Microservicio de Propiedades (**PropertyService**)

Su responsabilidad se centraría en la gestión de propiedades y las ubicaciones de estas para posteriores estadísticas (informes, etc..), así como el registro de los propietarios en el tiempo.

Abarca las entidades Property, PropertyImage, OwnerHistory, PostalCode, City, Region y Country.

Algunas de las fichas de casos de uso que explicadas que se gestionarían en este microservicio serían las listar propiedades, añadir propiedad, modificar propiedad, eliminar propiedad, validar o invalidar propiedad.

Microservicio de Subastas y Pujas (**AuctionService**)

Se encargaría de la gestión de subastas, pujas y estados de las subastas. Su alcance seria las clases Auction y Bid.

Algunos casos de uso podrían ser: Iniciar subasta o crearla, pujar en subasta, finalizar subasta, listar subastas activas, listar subastas finalziadas.

La siguiente imagen muestra los microservicios y las entidades involucradas de una forma más visual:

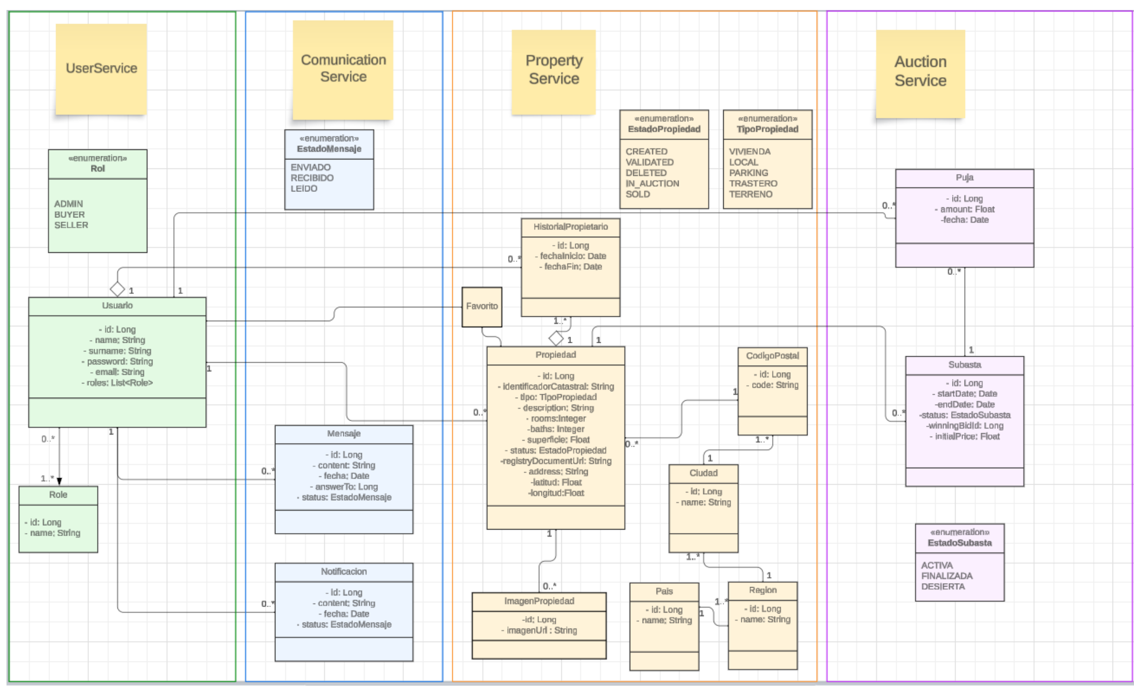


Ilustración 23 Definición de las clases de cada microservicio

* Cambios más relevantes que se han hecho en la implementación:

- Agregar la latitud y longitud en un inmueble
- El cambio de nombres al inglés por homogeneidad

- En el microservicio de Auction se ha agregado el userId (para futuras expansiones en informes)

Finalmente se almacenan las notificaciones enviadas al usuario por email (usuario registrado correctamente, inmueble agregado/eliminado a tu lista) aunque no se hace ningún uso posterior de las mismas.

2.10 Diagrama relacional de la base de datos

Aunque ya se ha establecido anteriormente que la base de datos, al ser una arquitectura hexagonal, cada microservicio solo tendrá acceso a las entidades o tablas que le corresponda, se ha hecho un pequeño boceto de la base de datos en global y las consideraciones que se han tenido al diseñar el sistema, ilustración 24.

Finalmente, durante la implementación, se ha optado por guardar los datos que equivaldrían a ENUM como String en la base de datos, y controlar a través del lenguaje de programación que los datos que se inserten sean los esperados por los Enum, esto ha sido más práctico que transformar posteriormente los datos para insertarlos con JPA, además, teniendo en cuenta que los estados son propios del dominio, si se modifica o agrega alguno posteriormente, en principio modificándolo en el dominio, ya se guardaría en la bbdd independientemente de si se sigue usando PostgreSQL u otra.

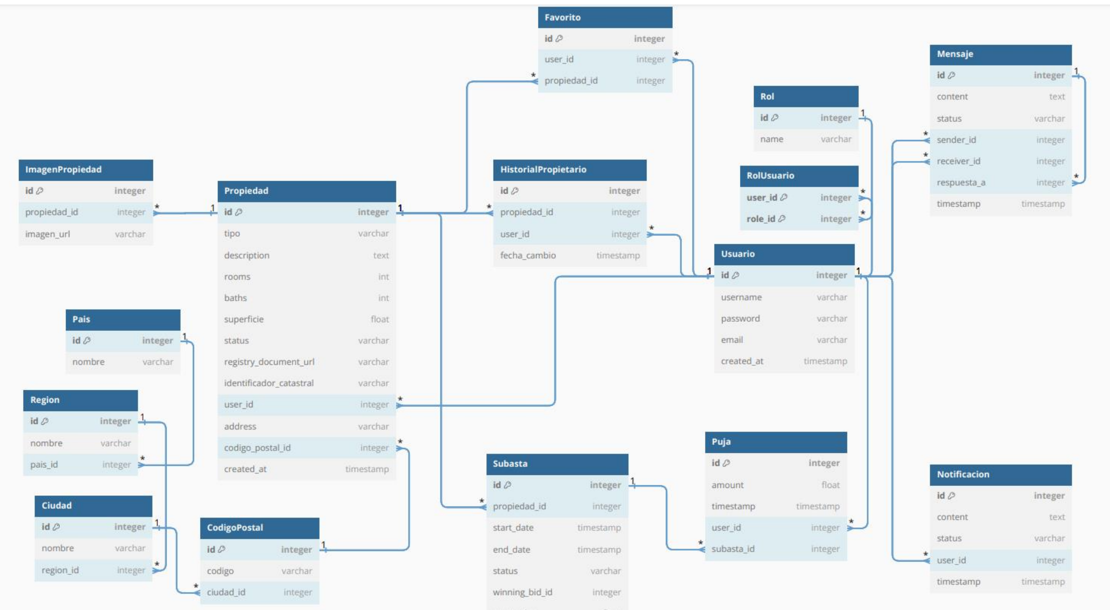


Ilustración 24 Definición de las tablas en base de datos.

A nivel de base de datos, se parten de varias premisas:

La principal es que, una propiedad no puede ser creada más que una sola vez y por lo tanto lo que ocurrirá es que puede cambiar de propietario, si presentan la documentación y el validador (Admin) considera demostrado que el titular es otra persona.

La segunda, que una propiedad puede ser subastada varias veces, si bien en un primer momento se valoró poner solo dos campos en la tabla de Propiedad, subasta_inicio y subasta_final ya que:

“cuando no haya tenido pujas en la subasta, si el usuario decidiera volver a ponerla a subastar, se actualizarían las fechas de subasta_inicio y subasta_final”.

Pero esto haría perder qué subastas han quedado desiertas y a la hora de hacer informes posteriormente puede ser una información útil, por el precio al que se pusieron a la venta, etc...

Así que, al final, se ha optado por usar una tabla o entidad Auction que puede estar activa, finalizada o desierta, por ejemplo, para consultas posteriores.

Sobre los Mensaje, va a ser una especie de “chat”, donde se anidan mensajes en respuesta a otros, partiendo de uno inicial que no tiene es respuesta a ningún otro.

Se ha plasmado en las tablas como un campo “respuesta_a” o “answer_to” que sería la id del mensaje “padre”.

La solución de una tabla intermedia para los roles es porque la intención es que haya usuarios que puedan tener varios roles al mismo tiempo, en este caso el de comprador y vendedor (incluso a la vez administrador), como hemos comentado en la anterior PEC.

2.11 Primeros pasos, creando la estructura y User microservicio

Para empezar las pruebas se ha generado con Visual Code Studio y Expo el proyecto de frontend de React Native, ver ilustración 25, solamente se ha creado un login para conectar con el endpoint del backend de UserService.

Lo primero que se ha hecho ha sido crearlo y guardarlo en GitHub como backup, ilustración 26.

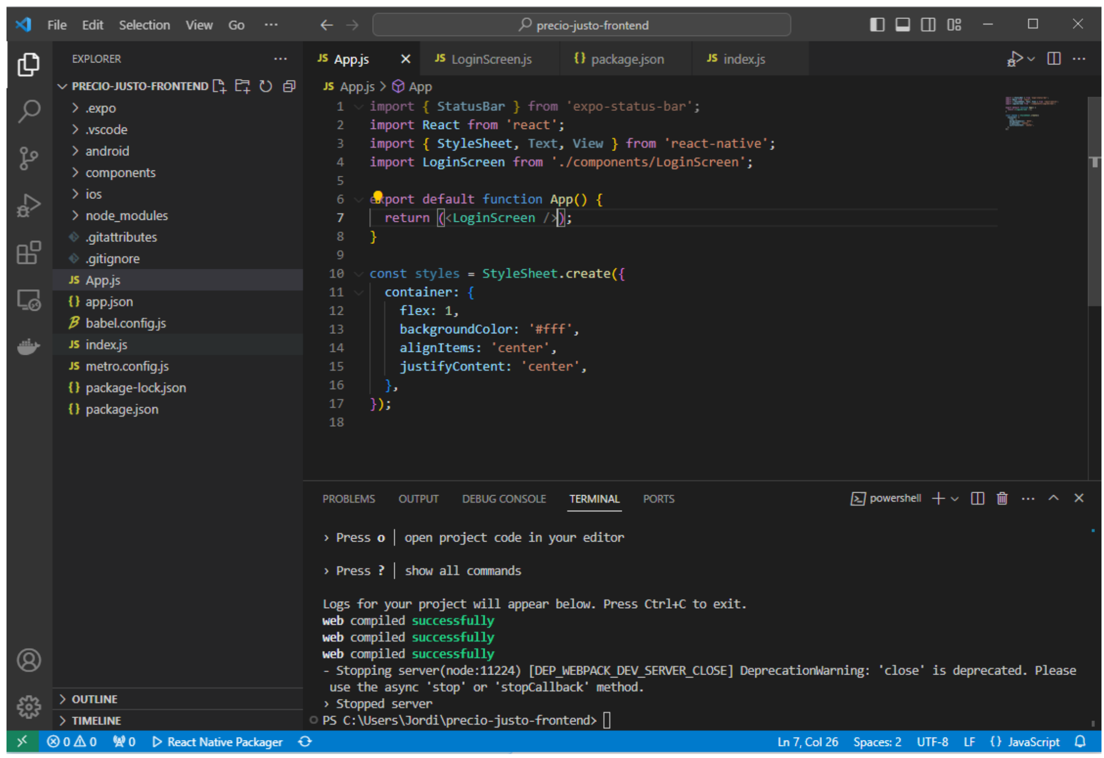


Ilustración 25 Proyecto inicial de frontend en React Native

Creación satisfactoria en web, con el comando npm run web, para comprobar si funcionaba a pesar de ser un login básico con una llamada al endpoint del backend.

Y a continuación se ha guardado en un repositorio privado para tener un backup en caso de tener algún problema posteriormente, como vemos en la imagen siguiente.

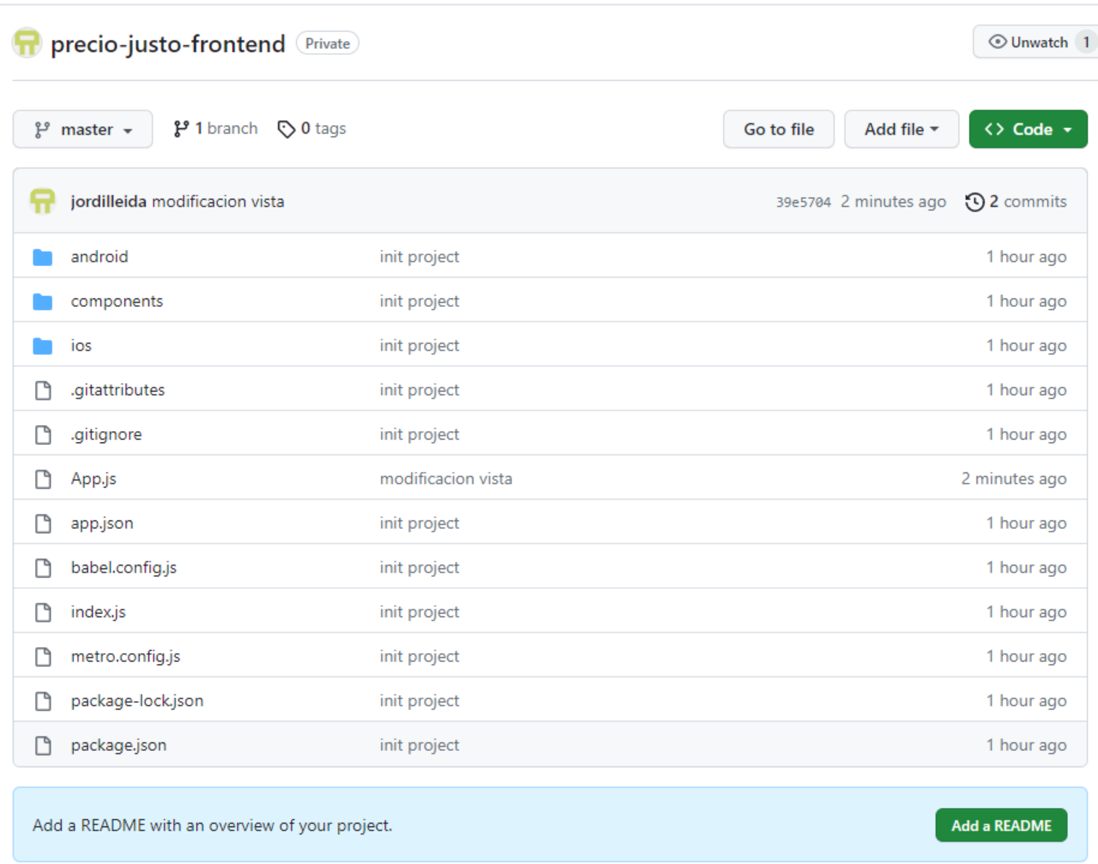


Ilustración 26 Backup en Github (me salvó varias veces al principio, gracias Tutor por el consejo)

Se crea el proyecto empezando por el microservicio de userService y se intenta crear manualmente los usuarios, vía sentencias SQL, esperando que genere los usuarios con los Roles que se han comentado anteriormente, con la posibilidad de tener más de un rol, para luego poderlo gestionar en el cliente en los distintos microservicios.

Se monta el servicio y la estructura comentados, básicamente, usando la configuración de base de la asignatura “Ingeniería y programación de sistemas distribuidos”, se logra configurar el proyecto y con el Docker-

compose.yml y el comando `**docker-compose up**` en principio se muestran los contenedores en Dockers Desktop, ver imagen 27.

A nivel personal quisiera comentar que IntelliJ Ultimate facilita mucho las tareas porque detecta y descarga fácilmente las dependencias.

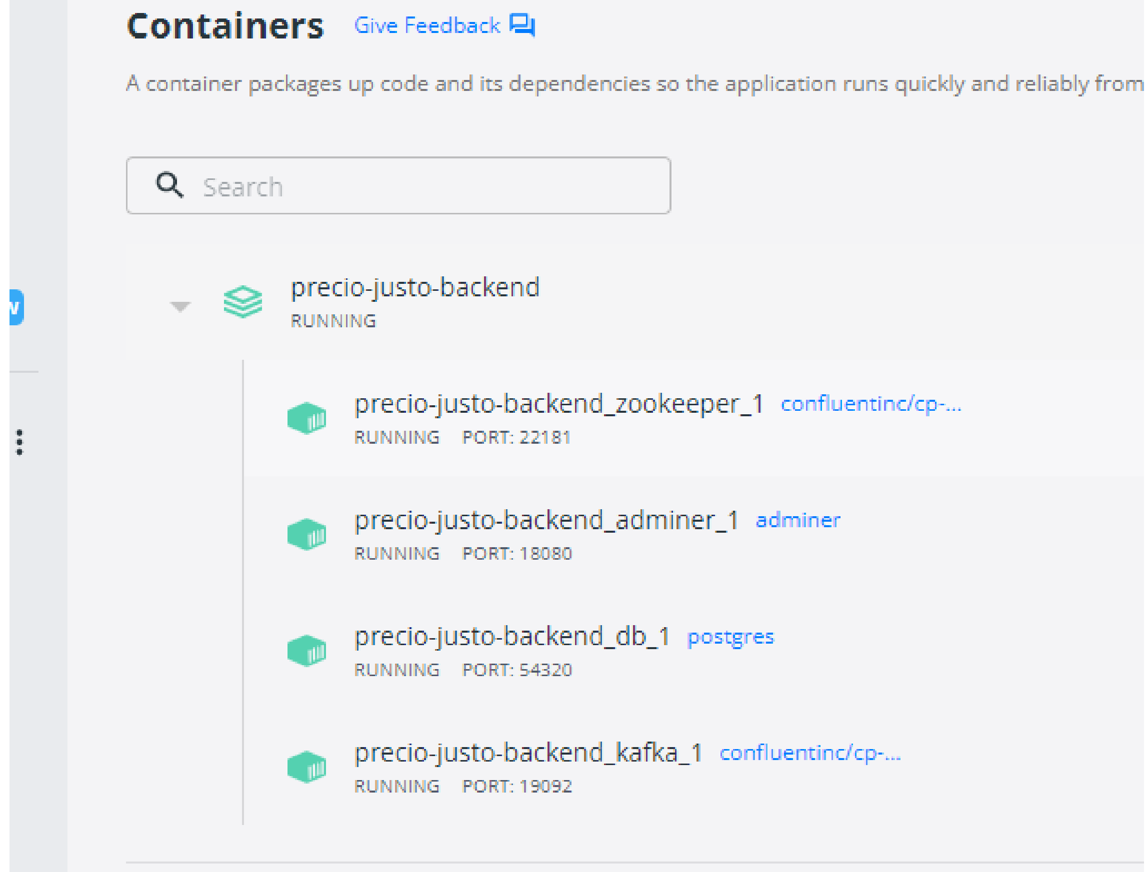


Ilustración 27 Contenedores con las imágenes corriendo correctamente.

Lo que se ha hecho es intentar crear las entidades que vamos a usar ya en nuestro UserService, que básicamente son Role y User, ha costado bastante poder crear las tablas y la relación de estas, pero insertando los datos a nivel SQL se guardan correctamente. Posteriormente cuando se registre un usuario será más problemático, pero no es el propósito de las primeras pruebas sino lograr que funcione como se espera de forma manual.

En el dominio se crean las clases como he comentado y el enum, como se ve a continuación:

```
schema.sql × | Rol.java × | application.pro
1  package edu.uoc.epcsd.user.doma
2
3  public enum Rol {
4      BUYER, SELLER, ADMIN
5  }
6
```

```
schema.sql × | application.properties × | Role.java ×
1  package edu.uoc.epcsd.user.domain;
2
3  import lombok.*;
4
5  @ToString
6  @Getter
7  @Setter
8  @EqualsAndHashCode
9  @Builder
10 @NoArgsConstructor
11 @AllArgsConstructor
12 public class Role {
13     private Long id;
14     private Rol name;
15 }
16
```

```
schema.sql   application.properties   Role.java   User.java
12  @NoArgsConstructor
13  @AllArgsConstructor
14  public class User {
15
16      private Long id;
17
18      private String name;
19
20      private String surname;
21
22      private String email;
23
24      private String password;
25
26      private Set<Role> roles;
27
28      private Boolean disabled;
29  }
30
```

Ilustración 28 Varias capturas de la definición inicial de User

Y se hace lo mismo en el repository para la creación de las tablas en postgreSql con JPA, como se ve en la ilustración 29 y 30.

Se ha creado una clase para mapear los permisos de usuario en el data transfer de los objetos de una forma más centralizada.

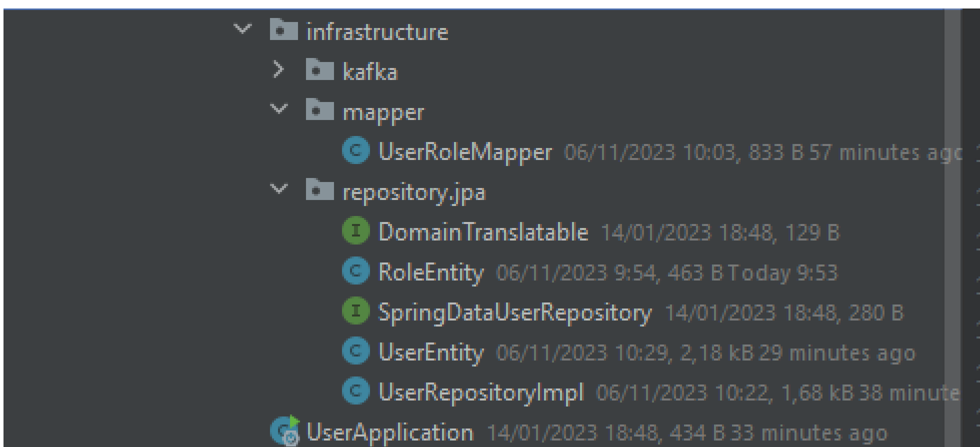


Ilustración 29 Definición de las Entities

```
schema.sql   Rol.java   RoleEntity.java   application.properties   Role
6  import javax.persistence.*;

7  @Entity
8  @ToString
9  @Getter
10 @Setter
11 @EqualsAndHashCode
12 @Builder
13 @NoArgsConstructor
14 @AllArgsConstructor
15 @Table(name = "roles")
16 public class RoleEntity {
17

18     @Id
19     @GeneratedValue(strategy = GenerationType.IDENTITY)
20     private Long id;
21

22     @Enumerated(EnumType.STRING)
23     private Rol name;
24
25 }
```

Ilustración 30 Entidad Role

Tras bastantes horas probando, se crean aparentemente bien los usuarios y roles, como se ha comentado anteriormente (teóricamente ya no añadiríamos más Roles así que al arrancar nuestro proyecto ya los tendríamos todos definidos como Buyer = 1L, Seller = 2L, y Admin = 3L), como vemos en la imagen 31,

* En la implementación se han logrado incorporar distintos Roles en un mismo usuario, por defecto el User tendrá, cuando se registre, el rol de BUYER, en el momento que incorpora 1 propiedad y es validada por el ADMIN, pasa a tener el rol de SELLER, que le permitirá acceder a un menú específico para este rol.

El rol ADMIN tiene su propio menú, donde valida inmuebles y lista usuarios solamente.

Más o menos es lo que se había pensado inicialmente ya que no se quería tener que acceder con distintos usuarios para ser SELLER y para ser BUYER.

```sql
INSERT INTO users (id, name, surname, email, password, disabled)
VALUES  (0, 'Jordi', 'Prueba', 'jordi@uoc.edu', '123', false),
        (1, 'Pepe', 'Prueba', 'pepe@uoc.edu', '123', false),
        (2, 'Juan', 'Prueba', 'juan@uoc.edu', '123', false);

INSERT INTO roles (name) VALUES ('BUYER'), ('SELLER') , ('ADMIN');

INSERT INTO user_roles (user_id, role_id)
VALUES  (1, 3), -- Jordi es ADMIN
        (2, 1), -- Pepe es SELLER
        (0, 1), -- Empresario es BUYER
        (0, 2); -- Empresario también es SELLER
```

Ilustración 31 Inserts para pruebas

Se pueden comprobar a través de Swagger que los inserts se han realizado con éxito, y se muestran los roles de los usuarios al listarlos, imagen 32.

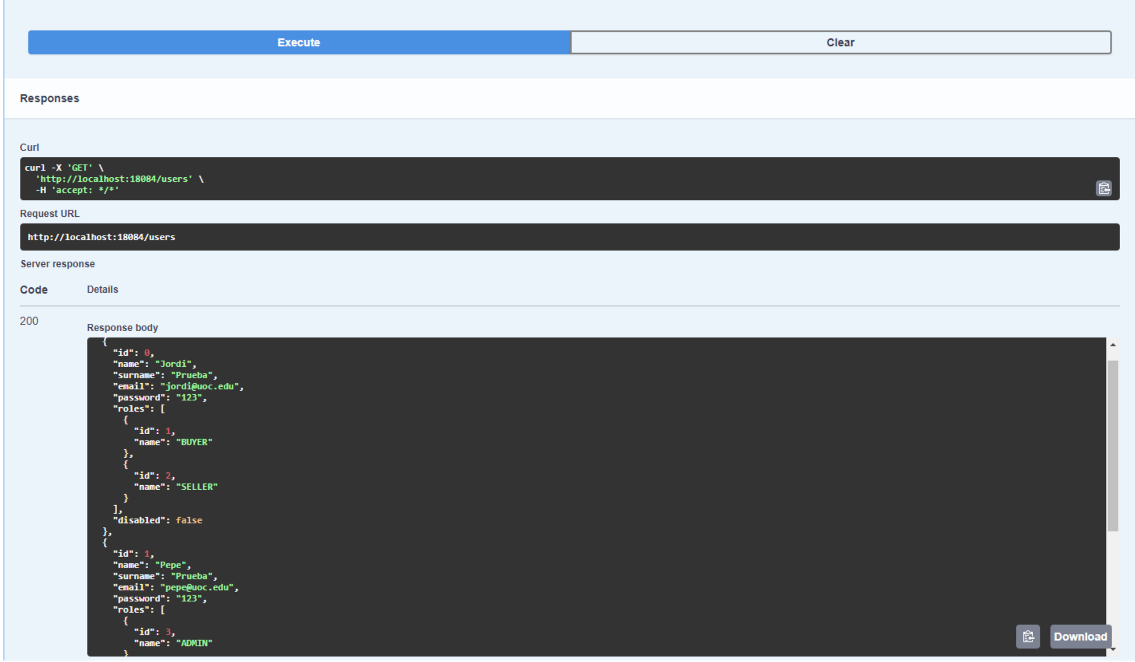


Ilustración 32 Llamada en Swagger al endpoint con todos los usuarios (y sus permisos)

Esto es importante para la seguridad, habrá métodos a los que solo puedan acceder los ADMIN (invalidar o validar propiedad, consultar usuarios, etc..), y también para habilitar las acciones que podrá realizar cada usuario en la App.

Por último, se intenta arrancar el servicio de UserService e iniciar sesión desde React Native, se ve como devuelve un token con la contraseña correcta y que devuelve cuando no es válida, ilustración 33 y 34.

Usuario y password válidos

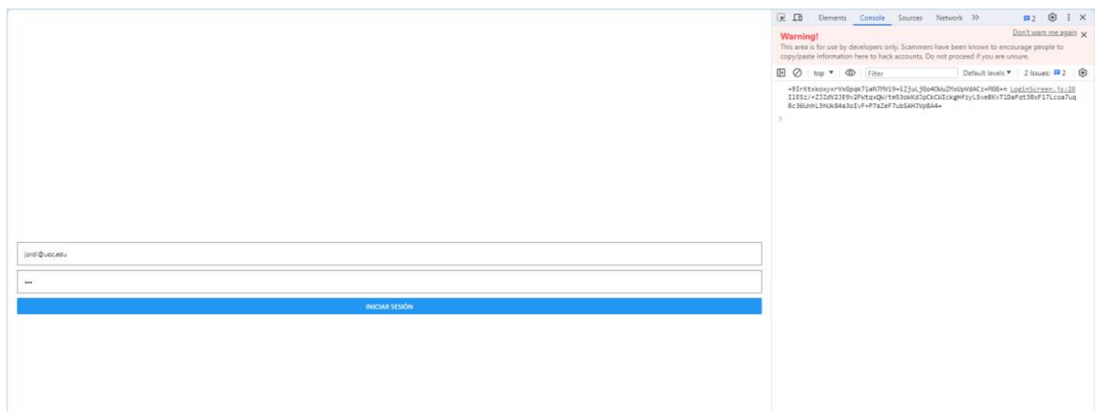


Ilustración 33 Petición con usuario y contraseña correcta y token de salida

Usuario no válido

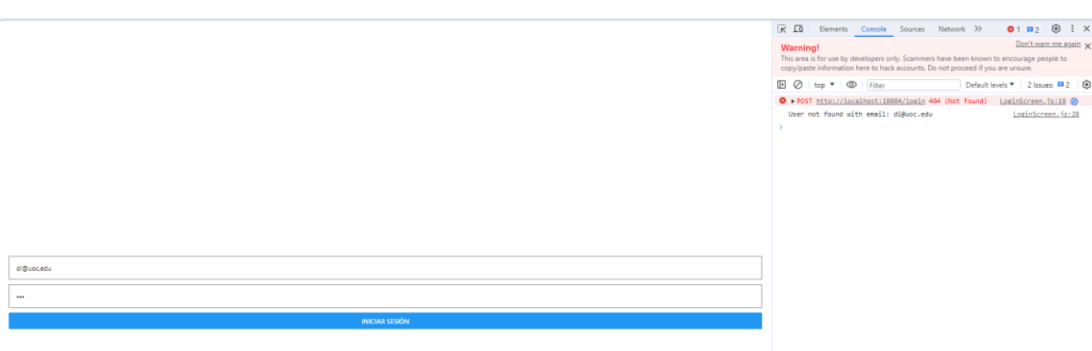


Ilustración 34 Llamada con usuario no valido con mensaje de error

Aunque la respuesta del login es un String, en principio en base a si retorna un 400, 401 , 404 o 200 se debería poder manejar la respuesta en cliente.

* Finalmente se ha utilizado otro tipo de token que cumpliera los “standards” JWT para poder manejar más fácilmente la resolución de

estos en los microservicios y el frontend, pero la idea sigue siendo la misma.

Se sube una copia a GitHub, como se ha hecho con el front-end, para ir periódicamente guardando los cambios como backup .

3. Implementación

3.1 Entorno de desarrollo

Se ha utilizado principalmente dos entornos de desarrollo (IDE) para la implementación del proyecto:

- Aplicación (web y mobile) desarrollado con React Native utilizando Visual Studio Code
- API Gateway y Microservicios desarrollados con Java EE y SQL con IntelliJ IDEA Ultimate

3.2 Cambios en los casos de uso y funcionalidades del proyecto

1.

Se ha agregado la funcionalidad de “crear subasta”, para poder volver a subastar varias veces un mismo inmueble si no se llega a tener una puja ganadora.

Vemos en la Ilustración 35 como, una vez el inmueble tiene estado VALIDATED (que el administrador ha validado esta propiedad) el propietario puede ponerla en subasta (IN_AUCTION).

Si una subasta queda desierta, el inmueble pasaría a estado VALIDATED de nuevo y podría iniciar una nueva subasta.

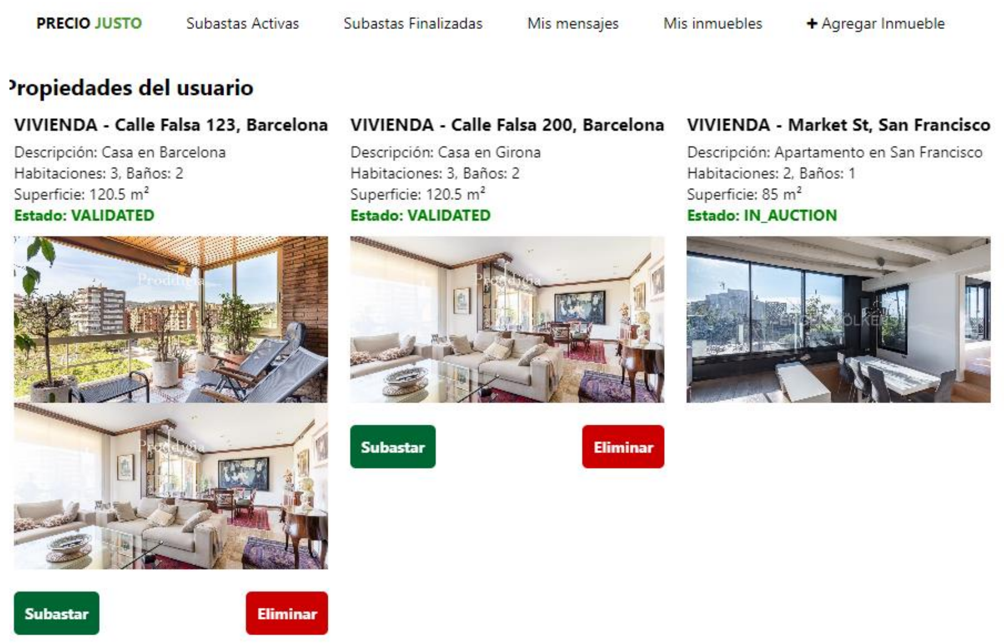


Ilustración 35 Ilustración inmuebles usuario

Al pulsar el botón "SUBASTAR", el usuario deberá añadir el importe y desde ese mismo momento la subasta se inicializará con una duración predefinida de 14 días (para simplificar), ilustración 36.

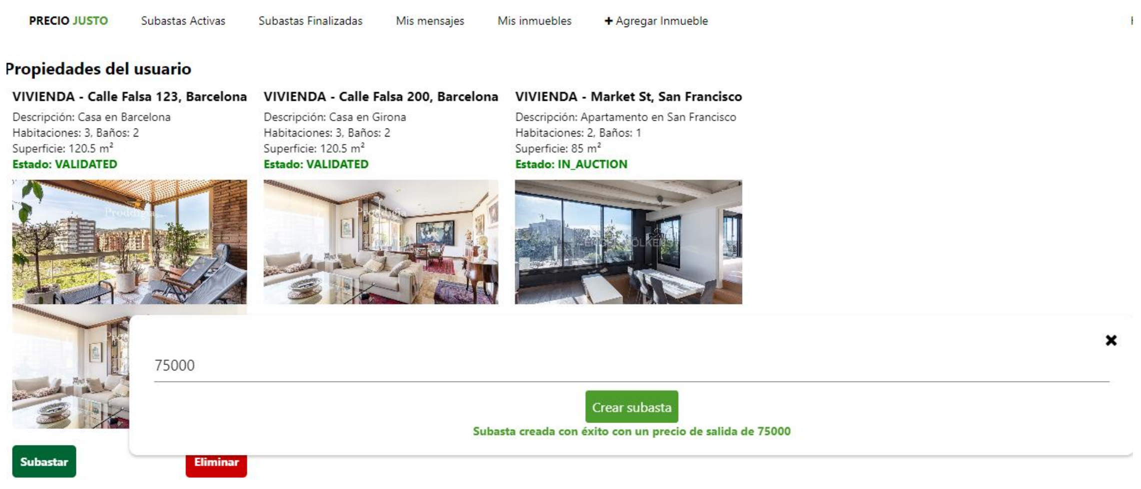


Ilustración 36 Subasta creada con éxito

2.

Para insertar imágenes de un inmueble y la documentación registral que pueda justificar que se es propietario del inmueble se ha utilizado Firebase Storage, en un principio se habia valorado Github Pages pero no era posible, este proceso se hace completamente desde la App y se envía al Microservicio de Property las url creadas como una cadena de texto o una lista si son varias imágenes junto al resto de datos del inmueble a crear.

Vemos como a medida que se seleccionan imágenes en el registro de una nueva propiedad se van incorporando y se guardan en Firebase Storage, ilustración 37. (Faltaría eliminarlas si finalmente no se crea el inmueble)

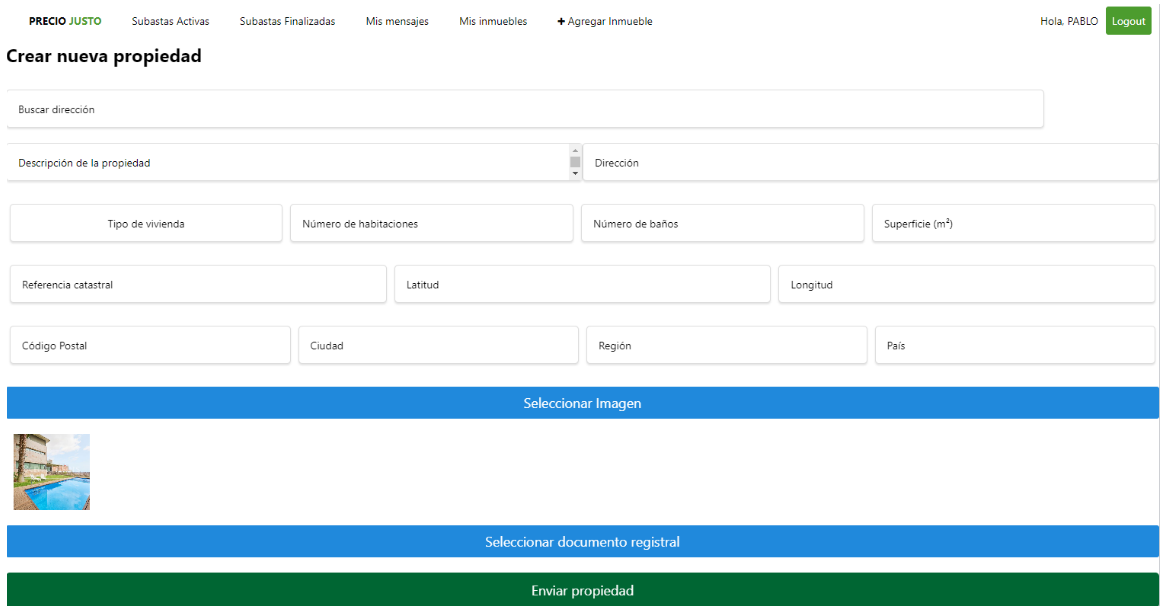


Ilustración 37 Imagen agregada a una Propeidad

Se guarda en distintas carpetas, por un lado, las imágenes y por otro el documento registral (único), como en la ilustración 38 vemos.

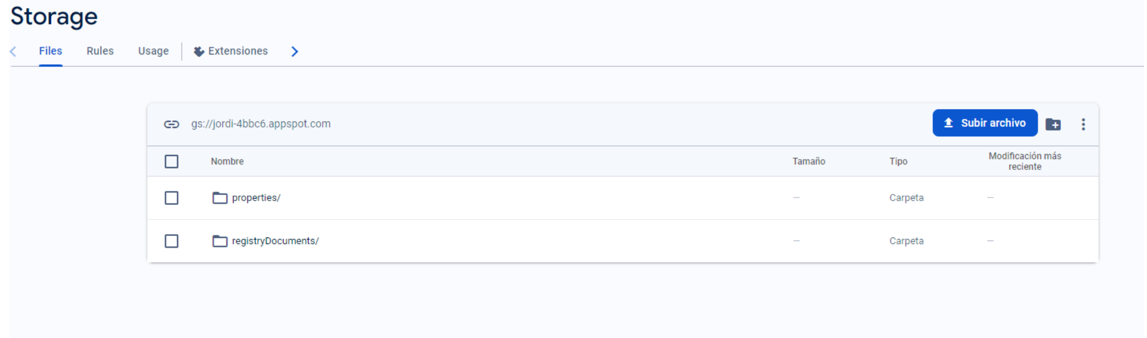


Ilustración 38 Imagen de la organización de carpetas en Firebase Storage

Y aquí vemos la imagen recién subida:

gs://jordi-4bbc6.appspot.com > properties — Subir archivo

Nombre	Tamaño	Tipo	Modificación más reciente
1703937170528.jpg	96.61 KB	image/jpeg	30 dic 2023
1704969216652.jpg	96.61 KB	image/jpeg	11 ene 2024

y su URL para visualizarla:

https://firebasestorage.googleapis.com/v0/b/jordi-4bbc6.appspot.com/o/properties%2F1704969216652.jpg?alt=media&token=d11f423b-dcd1-452d-bdf2-1970708d3991

3.

Debido a que una propiedad puede cambiar de manos, se establece que los datos y fotos de un inmueble solo pueden ser modificados por su actual propietario, de forma que, si alguien registra un inmueble con una referencia catastral pre-existente, solamente se llevará a cabo el cambio de propietario si el estado actual del inmueble es SOLD o DELETED (que equivale a que ha sido vendida en una subasta o eliminada por su último propietario).

Si el estado no es uno de esos dos y un usuario lo intenta registrar, se enviará un email al actual propietario para pedirle que si ya no es suyo lo

ponga en DELETED, lo elimine de sus inmuebles, para poder entonces sí, ser registrado por el otro usuario.

Se ha utilizado esta simplificación por el poco tiempo del que se dispone para desarrollar el proyecto y no abordar problemas de mayor calado. Se podría haber optado por revertir cambios, o tener una tabla intermedia con las propuestas nueva propiedad.

3.3 Creación de la infraestructura

Una vez montado el esqueleto del proyecto, con las funciones básicas de login y su Microservicio User, se ha creado el resto de la infraestructura y desplegado los servicios, ver ilustración 39.

Para este fin, se ha definido el fichero docker-compose.yml donde además de los servicios y sus respectivas db's, se encuentran definidos algunos servicios auxiliares necesarios como Kafka, Zookeeper y Adminer.

Zookeeper y Kafka forman la base para la gestión de colas de mensajes entre diferentes microservicios, ya comentados anteriormente.

Desplegamos la infraestructura con el comando ` docker-compose up –-build -d`

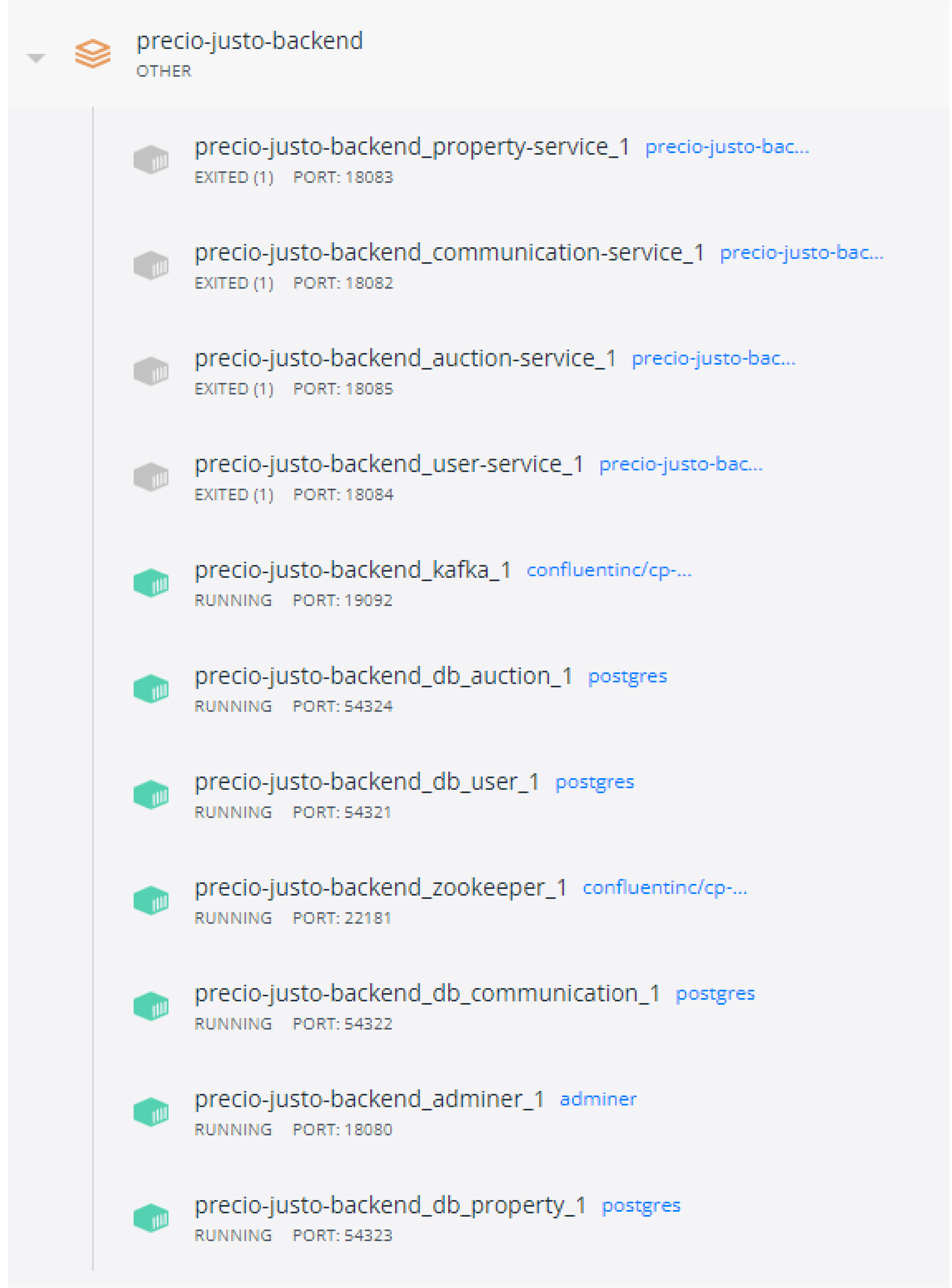


Ilustración 39 Docker Desktop servicios desplegados

*Nota: en mi caso los servicios y el Gateway los arranco en local desde IntelliJ porque tengo algunos problemas en la configuración de los microservicios al conectar con sus BBDD y no he podido solucionarlo por ahora.

Errores como la conexión de la base de datos refused:

```
org.postgresql.util.PSQLException: Connection to localhost:54323 refused. Check t
```

Durante esta etapa, se han creado en el dominio las clases que representan los objetos que se han ido detallando en el diseño de la aplicación.

Por comentar alguna de ellas, podemos ver Auction en la siguiente ilustración:

```java
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Auction {

    private Long id;

    private Long propertyId;


    private Long userId;

    private LocalDateTime endDate;

    private BigDecimal initialPrice;


    private AuctionStatus status;

    private Long winningBidId;

    private List<Bid> bids;
}
```

Ilustración 40 Clase Auction

En esta clase, ilustración 40, se han definido los atributos endDate (se inicia automáticamente al crearla por lo que no es necesaria una startDate), se establece el initialPrice, la propiedad a la que hace referencia con propertId y el propietario con userId (con esto se podría evitar que el propietario puje en su propia subasta) y el status de la subasta.

Además, contiene la “List<Bid> bids” que son las pujas sobre la subasta y para un rápido acceso, tiene el winningBidId que indica cual es la puja ganadora finalmente.

Con la anotación de Lombok @Data se establecen los getters y los setters además de toString, entre otros.

En la ilustración 41 se puede ver este objeto en las subastas activas, con la puja ganadora aun a *null* y sin pujas porque es una subasta recientemente iniciada.

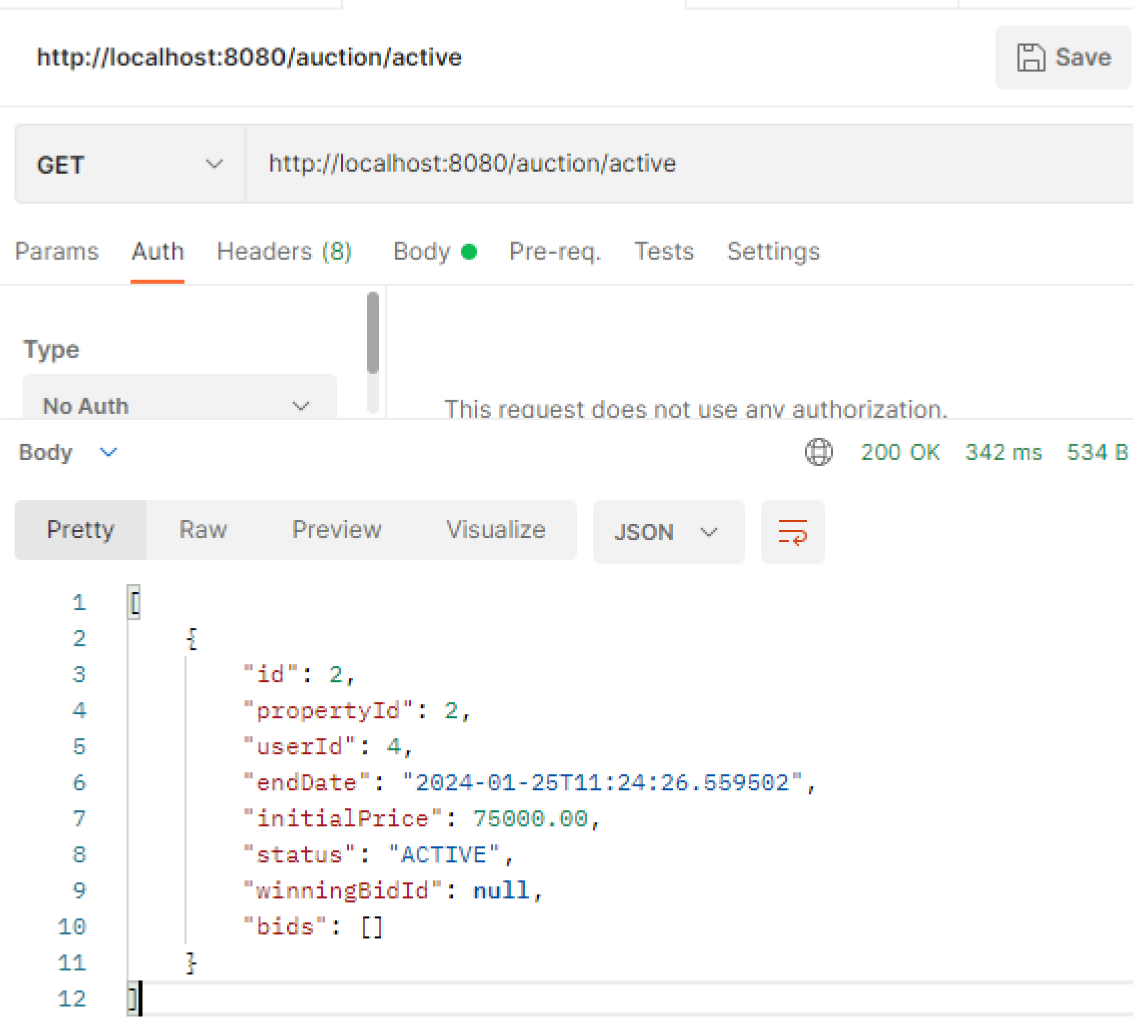


Ilustración 41 Subastas activas con el objeto Auction

También se ha definido dentro de la infraestructura de los microservicios, el repository.jpa con la especificación JPA que facilita el mapeo de las clases de dominio a la base de datos, en este caso vemos el contenido de AuctionEntity.java en la ilustración 42.

```
@Entity
@ToString(exclude = "bids")
@Getter
@Setter
@EqualsAndHashCode(exclude = "bids")
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "auctions")
public class AuctionEntity implements DomainTranslatable<Auction> {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;


    @Column(name = "property_id")
    @NotNull
    private Long propertyId;


    @Column(name = "user_id")
    @NotNull
    private Long userId;

    @Column(name = "start_date")
    @Builder.Default
    private LocalDateTime startDate = LocalDateTime.now();


    @Column(name = "end_date")
    private LocalDateTime endDate;


    @Column(name = "initial_price")
    @NotNull
    private BigDecimal initialPrice;
```

```
    private String status;


    @Column(name = "winning_bid")
    private Long winningBidId;


    @OneToMany(cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.EAGER)
    @JoinColumn(name = "auction_id")
    private Set<BidEntity> bids = new HashSet<>();


    @Override
    public Auction toDomain() {
        return Auction.builder()
                .id(this.id)
                .propertyId(this.propertyId)
                .userId(this.userId)
                .endDate(this.endDate)
                .initialPrice(this.initialPrice)
                .status(AuctionStatus.valueOf(this.status))
                .winningBidId(this.winningBidId)
                .bids(this.bids != null ? this.bids.stream() Stream<BidEntity>
                        .map(BidEntity::toDomain) Stream<Bid>
                        .collect(Collectors.toList())
                        : null)
                .build();
    }
```

```
    public static AuctionEntity fromDomain(Auction auction) {
        Set<BidEntity> bidEntities = null;
        if (auction.getBids() != null) {
            bidEntities = auction.getBids().stream() Stream<Bid>
                    .map(BidEntity::fromDomain) Stream<BidEntity>
                    .collect(Collectors.toSet());
        }

        return AuctionEntity.builder()
                .id(auction.getId())
                .propertyId(auction.getPropertyId())
                .userId(auction.getUserId())
                .endDate(auction.getEndDate())
                .initialPrice(auction.getInitialPrice())
                .status(auction.getStatus().toString())
                .winningBidId(auction.getWinningBidId())
                .bids(bidEntities)
                .build();
    }
}
```

Ilustración 42 Entidad AuctionEntity a partir de la que se crea la tabla auctions

La etiqueta @Entity es la que indica que esta entidad se debe mapear en la base de datos en una tabla.

La etiqueta @Table permite darle el nombre deseado a la tabla.

La etiqueta @Column no solo indica que es un atributo a incluir en la tabla, sino que se puede definir el nombre con el que se guarde.

Otra anotación a comentar en este apartado es el @OneToMany que determina el tipo de relación entre el objeto AuctionEntity y BidEntity, en este caso indica que es una a muchos, una subasta puede tener muchas pujas.

Cascade.ALL implica que, si se elimina una subasta, sus pujas también deberían eliminarse, pues carecen de sentido sin la subasta.

FetchType.Eager hace que cuando se recupera una AuctionEntity, se recuperen a la vez las pujas de la misma de forma “ansiosa”, que significa que se recuperan al mismo tiempo.

La anotación @JoinColumn se establece al definir el @OneToMany para indicar el campo con el que vamos a realizar la unión en BidEntity.

@Id y @GenerateValue simplemente indican cual es la clave de la tabla y que esta se genere automáticamente con autoincremento.

Gracias a la anotación @Builder de Lombok , junto a los métodos fromDomain y toDomain, se puede hacer una conversión o mapeo más cómodo y simple entre la entidad AuctionEntity y los objeto de dominio Auction.

Y para poder trabajar con datos reales al inicializar la aplicación y los microservicios, se han generado en cada uno un archivo los inserts *scheme.sql como muestra la imagen 43.*

Property.java | AuctionEntity.java | DomainTranslatable.java | schema.sql | BidEntity.java | PropertyEntity.java | docker-compose

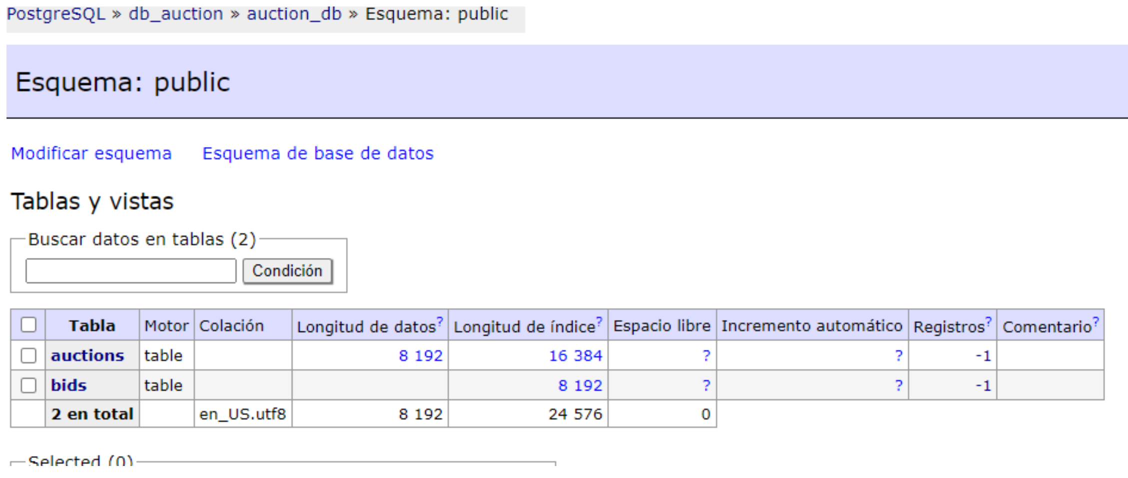
No data sources are configured to run this SQL and provide advanced code assistance.

SQL dialect is not configured.

```
1  INSERT INTO auctions (property_id, user_id, start_date, end_date, initial_price, status, winning_bid)
2  VALUES (4, 4, '2023-12-01T00:00:00',CURRENT_TIMESTAMP + INTERVAL '10 minutes', 100000.00, 'ACTIVE', NULL);
3
```

Ilustración 43 Scheme.sql con un Insert de una subasta

Finalmente, se puede ver en la imagen 44, con Adminer las tablas creadas, es este caso, del microservicio de Auction, las tablas auctions y bids.



PostgreSQL » db_auction » auction_db » Esquema: public

Esquema: public

Modificar esquema Esquema de base de datos

Tablas y vistas

Buscar datos en tablas (2)

Condición

Tabla	Motor	Colación	Longitud de datos?	Longitud de índice?	Espacio libre	Incremento automático	Registros?	Comentario?
auctions	table		8 192	16 384	?	?	-1	
bids	table			8 192	?	?	-1	
2 en total		en_US.utf8	8 192	24 576	0			

Selected (0)

Ilustración 44 Tablas del Microservicio Auction

3.4 API GATEWAY

Por comentar algo que no se haya comentado durante las anteriores fases, en la ilustración 45 se puede ver como se ha implementado un filtro global para el Gateway, por donde pasan todas las peticiones.

Su función es la de extraer y validar si no ha expirado el token que llega en la cabecera Bearer the Authorization.

```
@Log4j2
@Component
public class JwtAuthenticationFilter implements GlobalFilter, Ordered {


    @Autowired
    private TokenService tokenService;

    private static final Set<String> EXEMPT_ROUTES = Set.of("/user/login", "/user/register",
                                                            "/property/auction", "/auction/active");


    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        String path = exchange.getRequest().getPath().toString();

        if (isPublicRoute(path)) {
            return chain.filter(exchange);
        }

        String token = extractToken(exchange);

        if (token == null || !validateToken(token)) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }
        return chain.filter(exchange);
    }


    private boolean isPublicRoute(String path) { return EXEMPT_ROUTES.contains(path); }


    private String extractToken(ServerWebExchange exchange) {
        List<String> headers = exchange.getRequest().getHeaders().get("Authorization");
        if (headers == null || headers.isEmpty()) {
```

Ilustración 45 Filtro del Gateway para validar Token

```java
private boolean validateToken(String token) {
    try {
        JSONObject tokenData = tokenService.solveToken(token);
        Instant expiresAt = getInstantFromJson(tokenData, "expiresAt");

        if (expiresAt != null && expiresAt.isBefore(Instant.now())) {
            return false;
        }
        return true;

    } catch (RuntimeException e) {

        return false;
    }
}

public Instant getInstantFromJson(JSONObject jsonObject, String key) {
    try {
        String dateTimeString = jsonObject.optString(key);
        if (dateTimeString != null && !dateTimeString.isEmpty()) {
            DateTimeFormatter formatter = DateTimeFormatter.ofPattern("EEE MMM dd HH:mm:ss zzz yyyy", Locale.ENGLISH);
            ZonedDateTime zonedDateTime = ZonedDateTime.parse(dateTimeString, formatter);
            return zonedDateTime.toInstant();
        }
    } catch (Exception e) {
        log.error("Error parsing date time: ", e);
        return null;
    }
    return null;
}

@Override
public int getOrder() { return -1; }
```

Se han establecido una serie de rutas que no necesitan validación y si es una de esas rutas lo que ocurre es que se permite continuar el flujo normalmente con `return chain.filter(exchange)`.

Si no es una de esas rutas, simplemente se extrae el token y se valida que la fecha de expiración `expiresAt`, sea anterior a la actual.

Si eso no ocurre, se retorna un 401 UNAUTHORIZED.

Si todo es correcto se permite continuar a la petición y el Gateway la reenvía al microservicio correspondiente.

La implementación del Gateway ha traído varios problemas, sobre todo con la configuración (otro fue con Spring Security), a continuación, se refleja uno de los tantos ya que este en concreto se documentó un poco. Puede ser que hubiera otras soluciones, pero esta es la que se encontró.

Ejemplo de problema, parece ser con la versión de Spring Cloud Gateway y Spring Boot.

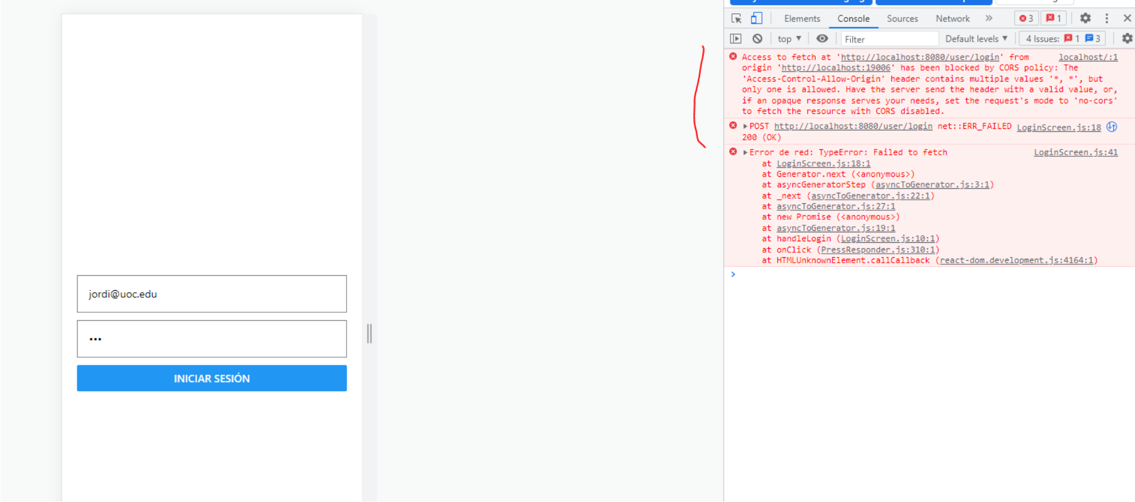
https://stackoverflow.com/questions/72617854/how-to-fix-multiple-values-of-access-control-allow-origin-in-spring-boot

Como se ilustra en la imagen 46 y 47, el problema que se tiene es idéntico al del comentario de stackoverflow, en backend se resuelve correctamente el login y se devuelve el token

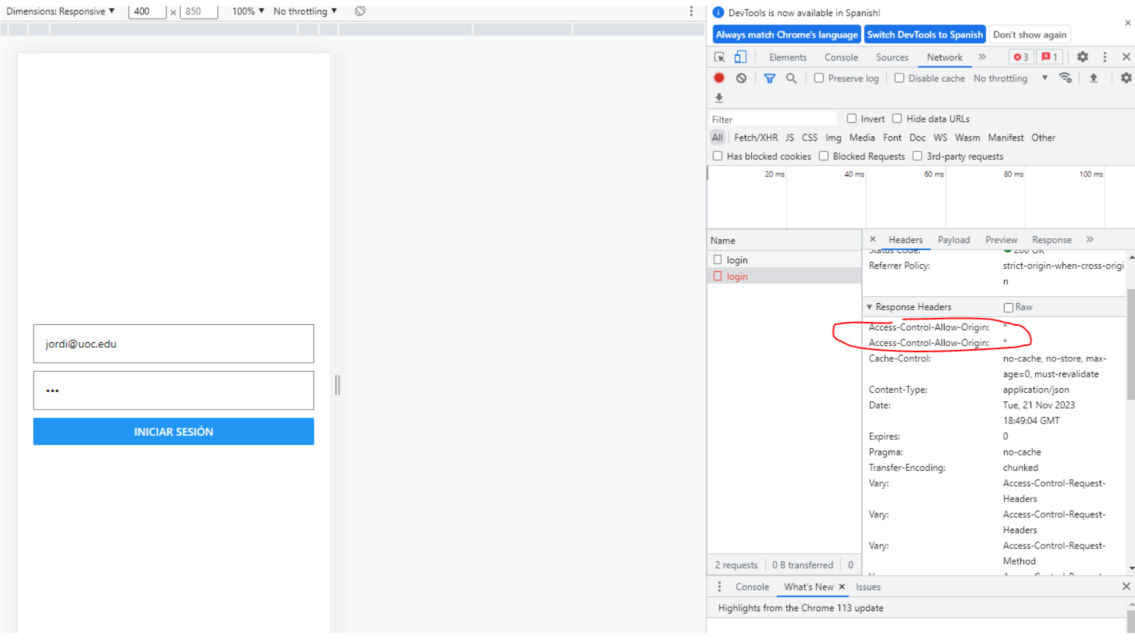
Ilustración 46 La petición llega al Gateway y se resuelve satisfactoriamente

Ilustración 47 Vemos como el token se ha creado

Pero al volver a cliente, entiendo que indica que hay multiple values en el header de Access-Control-Allow-Origin



Y efectivamente ahí están los múltiples valores, como en el comentario de stackoverflow

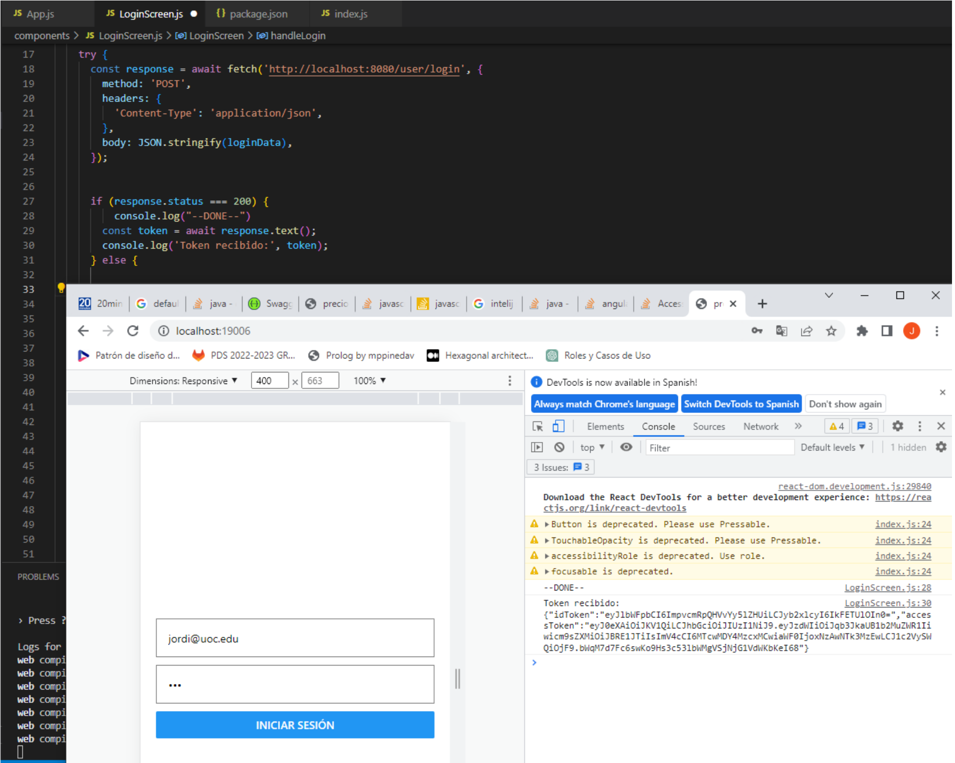


Parece que es un tema de incompatibilidades y problemas que no se han resuelto, por lo que he sabido leer, desde mi desconocimiento de la materia.

Así que se ha optado por la solución que proponen en ese hilo, que por suerte he podido llegar a encontrar, y he aplicado las versiones de Spring Boot y Spring Cloud propuestas al Gateway

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.o
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.6.6</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>tfg-spring-gateway</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>tfg-spring-gateway</name>
    <description>tfg-spring-gateway</description>
    <properties>
        <java.version>17</java.version>
        <spring-cloud.version>2021.0.1</spring-cloud.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-gateway</artifactId>
        </dependency>
```

Y al volver a probar, al hacer login con el endpoint del Gateway ya devuelve el resultado esperado



3.5 API REST y App

En esta última fase de la implementación, una de las labores más complicadas ha sido la creación de los métodos definidos en las anteriores fases, esto ha implicado completar todas las operaciones necesarias y, además, crear las vistas asociadas para cada caso de uso que se iba completando.

Al final del proyecto tenemos, para cada microservicio su API REST que son: AuctionRESTController, ilustración 48, CommunicationController, UserRESTController y PropertyRESTController.

Además de los servicios con las operaciones que ya hemos esquematizado en los diagramas de arquitectura hexagonal del apartado 2.5.

```
@Log4j2
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
@RestController
@CrossOrigin
public class AuctionRESTController {


    private final AuctionService auctionService;

    private final UserService userService;


    @PostMapping("/create")
    @PreAuthorize("hasRole('ROLE_SELLER')")
    public ResponseEntity<String> createAuction(@Valid @RequestBody AuctionRequest auctionRequest) {

        Optional<CustomUserDetails> userDetails = userService.getAuthenticatedUser();

        if (!userDetails.isPresent())
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED).build();

        Long authenticatedUserId = userDetails.get().getUserId();

        Auction newAuction = Auction.builder()
                .propertyId(auctionRequest.getPropertyId())
                .userId(authenticatedUserId)
                .initialPrice(auctionRequest.getInitialPrice())
                .status(AuctionStatus.ACTIVE)
                .endDate(LocalDateTime.now().plusDays(AuctionConstants.AUCTION_DURATION_DAYS))
                .build();

        Auction createdAuction = auctionService.createAuction(newAuction);

        if(createdAuction == null)
            return ResponseEntity.unprocessableEntity().body("No se ha podido crear la subasta");

        return ResponseEntity.ok( body: "Subasta creada con éxito con un precio de salida de " + createdAuction.getInitialPrice());
    }
```

Ilustración 48 Ejemplo de los métodos REST de Auction microservicio

En todos se les agrega la anotación Spring @RestController que sirve para indicar que es un servicio REST.

Además, tienen la anotación @Log4j2 para que quede registro de la actividad en distintos puntos de la aplicación, así como la que el propio programa al arrancar guarda.

Gracias a la anotación @CrossOrigin de Spring Boot se permite acceder a los recursos desde distintos orígenes (solicitudes de origen cruzado)

Los métodos tienen las anotaciones de Spring @PostMapping, @GetMapping o PutMapping, estas hacen referencia al verbo que se acepta en la solicitud, seguido de la url para llegar a ese endpoint, ejemplo @PostMapping("direccion")

También se utiliza la anotación de Spring Security @PreAuthorize, que define una precondición para acceder al recurso, en este caso se verifica si el usuario que se ha extraído de la petición en el token JWT tiene un role concreto.

Concretamente en Spring Security se utiliza la clase UserDetails para representar la información de un usuario, en este proyecto se ha extendido (para agregar userId como se ha comentado anteriormente) en la clase CustomUserDetails.

A esa clase se le agregan los "permisos" que tiene en una lista de objetos SimpleGrantedAuthority, ver ilustración 49, y estos son lo que verifica en la etiqueta @PreAuthorize.

```
public UserDetails extractUserDetails(String jwtToken) {
    JSONObject tokenData = solveToken(jwtToken);
    String email = tokenData.getString( key: "email");
    Long userId = tokenData.getLong( key: "userId");
    String rolesString = tokenData.getString( key: "roles");
    List<SimpleGrantedAuthority> authorities = Arrays.stream(rolesString.split( regex: ", ")) Stream<String>
            .map(role -> new SimpleGrantedAuthority( role: "ROLE_" + role.trim().toUpperCase())) Stream<SimpleGrantedAuthority>
            .collect(Collectors.toList());

    return new CustomUserDetails(email,  password: "", authorities, userId);
}
```

Ilustración 49 Función de extracción de detalles de usuario de una petición.

También se han agrupado los parámetros de entrada, en algunos casos, en objetos request, cuando se necesitan más de dos parámetros o establecer condiciones de validación, estos objetos se han agregado en application/request y con la anotación @RequestBody Spring convierte automáticamente el body de la petición en un objeto Java, lo mapea automáticamente a la clase Request, en este caso ActionRequest.

@Valid lo que hace es, una vez mapeado el objeto, verifica que se cumplan todas las restricciones que se hayan podido establecer en la clase, por ejemplo @NotNull, etc. si no se cumple devolvería una excepción, según se indique en cada anotación, ver ilustración 50.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AuctionRequest {

    @NotNull(message = "Falta la propiedad sobre la que se iniciará la subasta")
    private Long propertyId;

    @NotNull(message = "El precio inicial es obligatorio")
    @DecimalMin(value = "0.01", message = "El precio inicial debe ser mayor que cero")
    private BigDecimal initialPrice;
}
```

Ilustración 50 Clase AuctionRequest para crear una nueva subasta.

En la ilustración 50 se puede ver que los campos que espera el endpoint para crear una subasta serian, la id de la propiedad para subastar **propertyId** y un precio inicial **initialPrice**.

Se procede ahora a hacer login (para conseguir el token de acceso) y posteriormente crear una nueva subasta de un inmueble del usuario, ilustración 51, donde se ve que se retorna un accessToken.

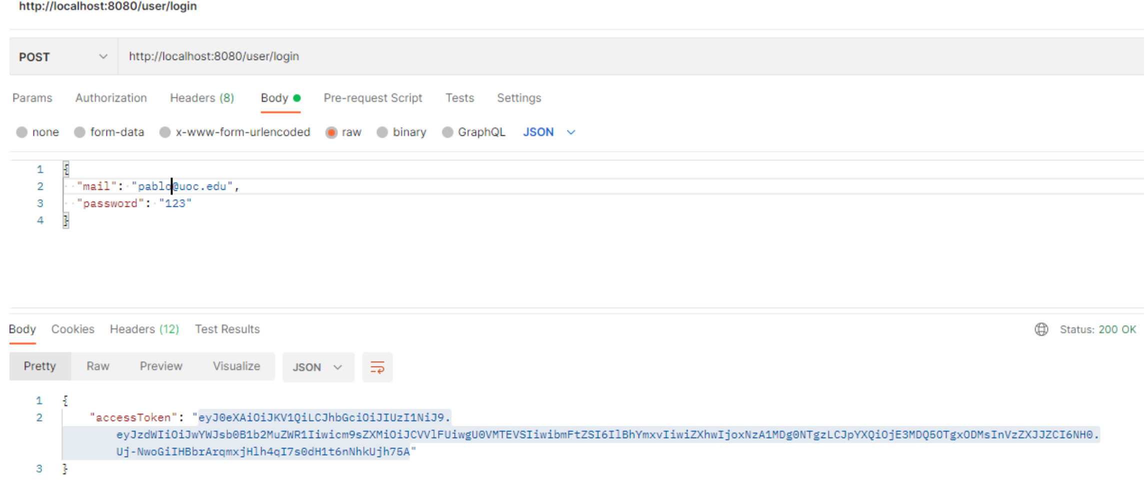
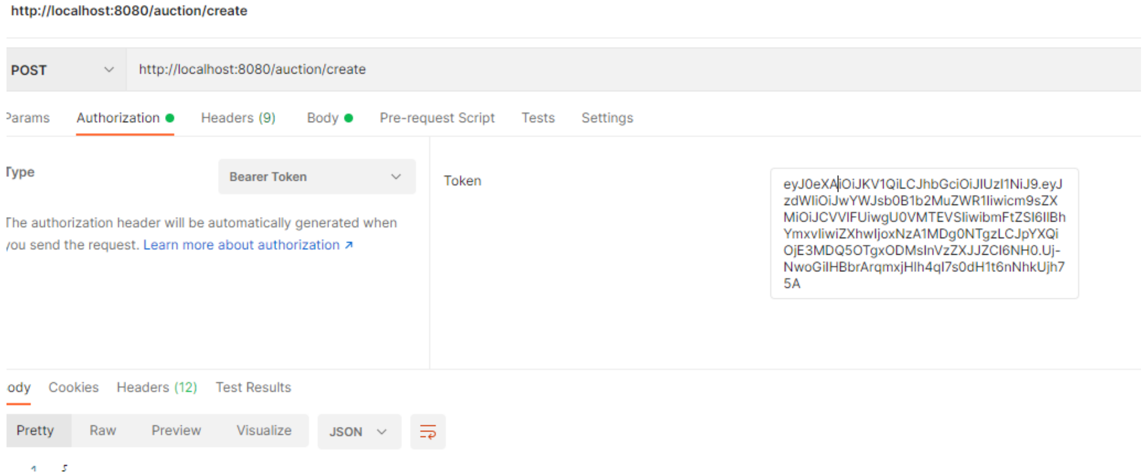


Ilustración 51 Login del usuario pablo

Ahora, se agrega el token a la cabecera Bearer en Authorization, como se ve en la ilustración 52.



Y se puede ver en la ilustración 52 como, a partir de los parámetros de la request en el body de la petición por POSTMAN, se ha validado correctamente en el endpoint. Se puede observar además que la petición se realiza a través del API Gateway, que ha redirigido correctamente la petición al microservicio.

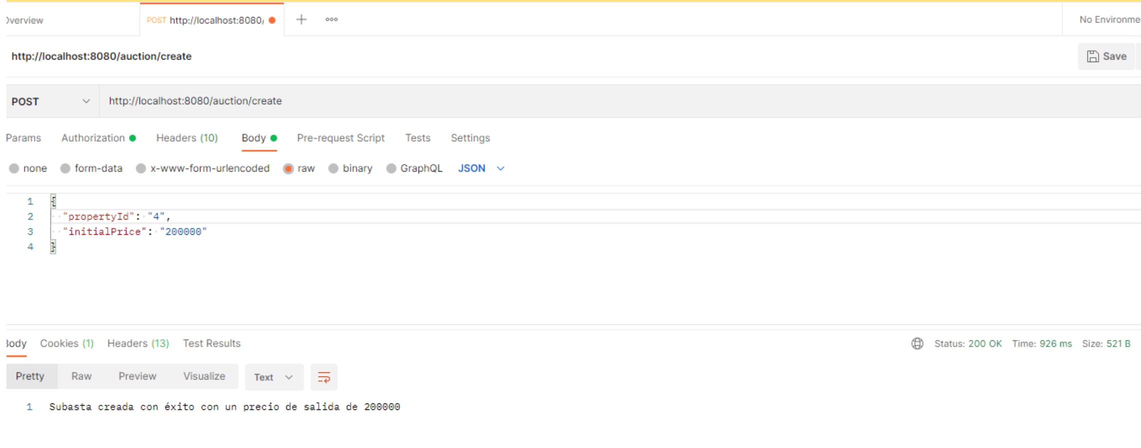


Ilustración 52 Subasta creada correctamente

En el fichero domain/service se encuentran todas las operaciones de cada servicio, ya se han mostrado previamente en los diagramas de cada microservicio.
Dentro de la carpeta se puede encontrar, por ejemplo, *AuctionServiceImpl* , que es la clase que tiene el método concreto para la creación de una subasta, como se ve en la ilustración 53, implementado el interfaz del puerto de entrada AuctionService.

```
@Override
public Auction createAuction(Auction newAuction){

    //Si hay una subasta actualmente activa, no se puede crear otra, devuelve la actual
    Auction currentAuction = getCurrentAuctionByPropertyId(newAuction.getPropertyId());

    if(currentAuction != null){
        return currentAuction;
    }

    Auction createdAuction = auctionRepository.editOrCreate(newAuction);

    kafkaAuctionMessagingService.sendCreatedMessage(createdAuction);

    return createdAuction;
}
```

Ilustración 53 Método crear subasta en la implementación del service

Tras una validación para no poder crear múltiples subastas sobre la misma propiedad mientras haya una activa, se crea la subasta nueva a través del adaptador de salida auctionRepository, como se ve en la imagen 54.

```java
@Component
@Log4j2
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class AuctionRepositoryImpl implements AuctionRepository {

    @Autowired
    private SpringDataAuctionRepository auctionRepository;


    @Override
    public Auction editOrCreate(Auction auction) {

            AuctionEntity auctionEntity = AuctionEntity.fromDomain(auction);
            AuctionEntity savedEntity = auctionRepository.save(auctionEntity);

        return savedEntity.toDomain();
    }
```

Ilustración 54 Guardado en la BBDD de la subasta

En la carpeta infrastructure/repository/jpa se encuentra la implementación del AuctionRepository con las funciones definidas en el puerto de salida, así como algunas clases SpringData{Entidad}Repository, como ejemplo se puede ver la ilustración 55, que implementan operaciones, además de extender JPARepository para las CRUD como save, findById, etc...

```java
public interface SpringDataAuctionRepository extends JpaRepository<AuctionEntity, Long> {
```

Ilustración 55 SpringDataAuctionRepository que extiende JPA con funciones CRUD

También se envía un mensaje a un topic de Kafka, ver ilustración 56, la razón de esto es que, cuando se crea una nueva subasta se pone la propiedad en estado IN_AUCTION y el microservicio de Property está escuchando para poder realizar este cambio.

Esto es así para impedir que alguien pueda eliminar o cambiar de propietario un inmueble mientras está en subasta.

```
@Override
public void sendCreatedMessage(Auction auction) {
    String topicName = KafkaConstants.PROPERTY_TOPIC + KafkaConstants.SEPARATOR + KafkaConstants.COMMAND_CREATED;
    kafkaTemplate.send(topicName, auction);
    log.info("Sent kafka message to topic " + topicName + ": created auction " + auction.getId());
}
```

Ilustración 56 Funcionalidad de envio de topic "crear subasta"

Ahora se procede a visualizar en la parte de la App como se ha creado esta subasta que se ha agregado vía POSTMAN, para que se vean las distintas partes del desarrollo de la Aplicación cliente.

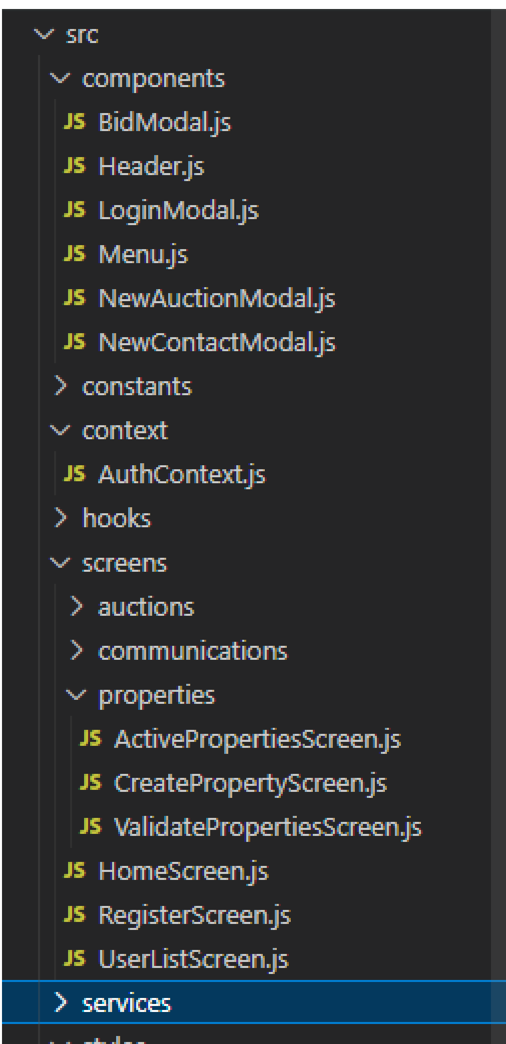


Ilustración 57 Organización proyecto App

En la ilustración 57 se puede ver como en la carpeta src, se ha creado una subcarpeta screens y otra de components donde van todos los modals

además de partes comunes como el Header y el Menu.

En este caso concreto de las subastas creadas, a través de la pantalla de ActivePropertiesScreen.js (que muestra las propiedades de un usuario registrado), al pulsar en Subastar (se ha mostrado en el apartado 3.2) se abre el modal NewAuctionModal.js y se realiza la misma operación que se ha materializado a través de POSTMAN anteriormente, volviendo a crear una segunda subasta.

El resultado es, ver ilustración 58, que se ha agregado la propiedad a subasta por 200.000 euros, además de la anterior de 75.000 que ya se había creado sobre otro inmueble en la explicación anterior.

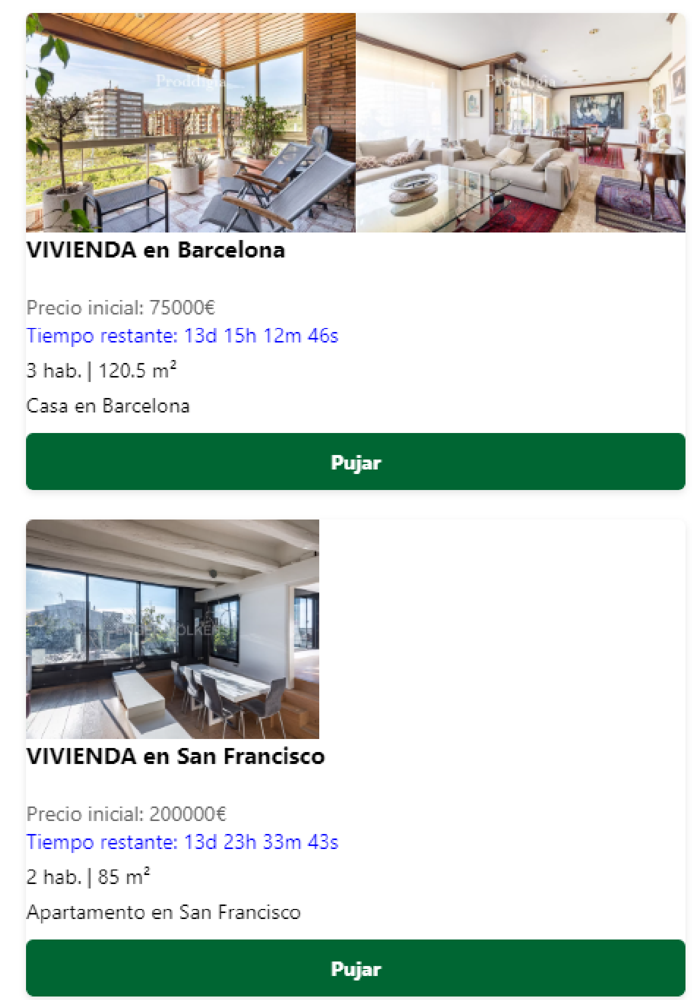


Ilustración 58 Subastas creadas

Y actualmente, en la vista de sus propiedades, el usuario tiene 2 inmuebles en subasta "IN_AUCTION", como se puede ver en la ilustración 59.

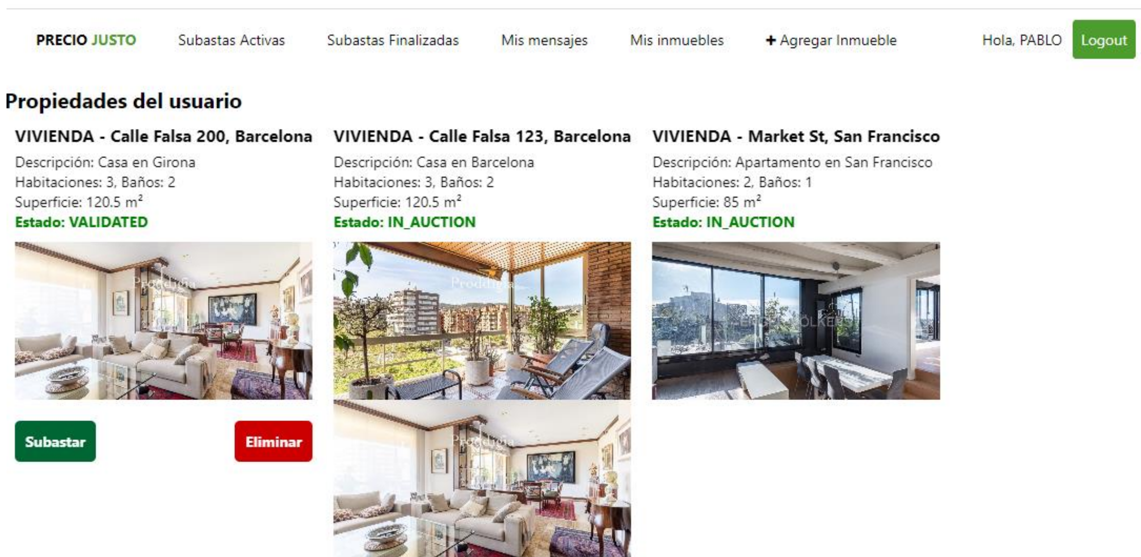


Ilustración 59 Propiedades del usuario

A nivel de aplicación React Native el lenguaje que se usa para programarlo es Javascript, que es el que interpreta finalmente la máquina, como se puede ver en la ilustración 60, al cargar la screen de ActiveProperties se ejecuta la llamada al endpoint para recuperar todas sus propiedades gracias a fetchProperties(), con el user.id del propietario, que previamente se almacena en el AsyncStorage al logearse, y en el proyecto existo un

objeto especifico almacenado para importar el token en las vistas que sea necesario, a través de la función useAuth.

```
import { makeRequest } from '../../utils/networkServices';
import NewAuctionModal from '../../components/NewAuctionModal';

const ActivePropertiesScreen = () => {
    const { user } = useAuth();
    const [properties, setProperties] = useState([]);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);
    const [shouldReload, setShouldReload] = useState(false);
    const [isNewAuctionModalVisible, setIsNewAuctionModalVisible] = useState(false);
    const [selectedPropertyId, setSelectedPropertyId] = useState(null);

    const fetchProperties = async () => {
        try {
            setLoading(true);
            setError(null);
            const response = await makeRequest(`${ApiConstants.BASE_URL}${ApiConstants.PROPERTY_URL}${ApiConstants.PROPERTIES_ENDPOINT}/${user.id}`);
            if (!response.ok) {
                throw new Error('Error en la respuesta del servidor');
            }
            const data = await response.json();
            setProperties(data);
        } catch (err) {
            setError(err);
        } finally {
            setLoading(false);
        }
    };

    useEffect(() => {
        fetchProperties();
    }, [shouldReload]);
```

Ilustración 60 Vista de propiedades activas de un usuario

Además, cuando se crea una subasta desde frontend, se vuelve a llamar y recargar las propiedades, para actualizar la lista de propiedades del usuario y sus estados, esto se consigue gracias a los estados de React y a la función que he definido como onAuctionSuccess() que modifica el setShouldReload, ver en la imagen a continuación:

```
const onAuctionSuccess = () => {
    setIsNewAuctionModalVisible(false);
    setShouldReload(prev => !prev);
};
```

Esta se ejecuta cuando se cierra el modal de nueva subasta, provocando que se refresquen los inmuebles, seguramente no será la mejor forma, pero es la que se ha aplicado para hacerlo de una forma simple y

funcional, aunque obviamente no será optima porque cerrar el modal no implica que se haya creado la subasta.

En las imágenes 61 y 62 se puede ver lo que se ha explicado anteriormente en código.

```
                )}
            />
            <NewAuctionModal
                isVisible={isNewAuctionModalVisible}
                onClose={() => setIsNewAuctionModalVisible(false)}
                propertyId={selectedPropertyId}
                onAuctionSuccess={onAuctionSuccess}
            />
        </View>
```

Ilustración 61 Creación del Modal de nueva subasta en la screen de Properties del user

```
const NewAuctionModal = ({ isVisible, onClose, propertyId, onAuctionSuccess }) => {
    const [amount, setAmount] = useState('');
    const { doPost, response, error, loading } = usePost(ApiConstants.BASE_URL + ApiConstants.AUCTION_URL + ApiConstants.CREATE_ENDPOINT);

    const handleAuction = async () => {
        const auctionData = {
            propertyId: propertyId,
            initialPrice: parseFloat(amount),
        };

        await doPost(auctionData);
    };

    const handleClose = () => {
        onAuctionSuccess();
        onClose();
    };
```

Ilustración 62 Objeto NewAuctionModal, al cerrar ejecuta la función onAuctionSuccess()

Además de implementar la lógica y del tema de la seguridad que ya se ha comentado anteriormente, se procede a verificar que las API REST devuelven un 401 UNAUTHORIZED cuando el usuario no tiene permisos para un recurso concreto.

Se procede a intentar crear una subasta sobre un inmueble sin pasar el Bearer en la cabecera, de forma que no debería permitir porque ya se ha visto que es necesario tener el rol de “SELLER”.

Como se ve en la ilustración 63, efectivamente retorna un 401 y se puede observar como no está activada la cabecera Authorization.

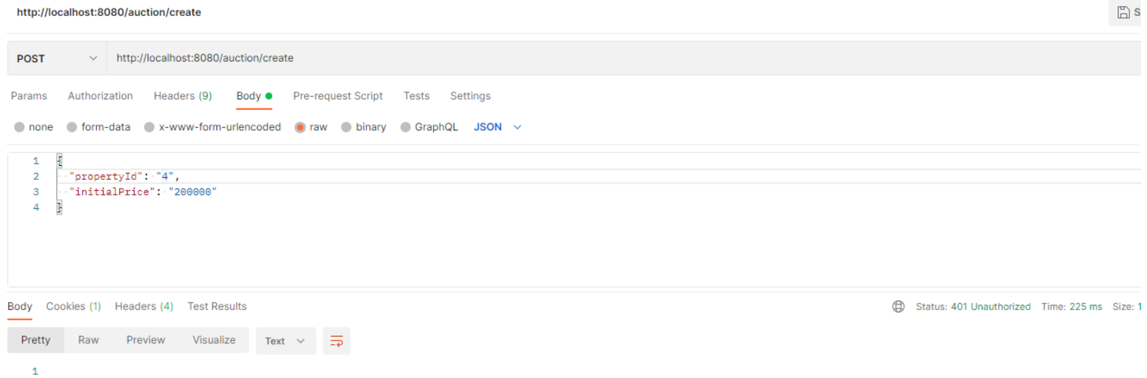


Ilustración 63 Crear subasta sin token en la cabecera

4. Mejoras y evolución del proyecto

Como se ha comentado durante la fase de Requerimientos y Análisis, se proponen para futuras evoluciones la implementación del caso de uso “Búsqueda de inmueble” utilizando el tipo de dato geo_point de elasticsearch, que sería relativamente fácil de implementar ya que actualmente ya se almacenan la latitud y la longitud en la clase del dominio Property, en el microservicio PropertyService.

Posteriormente simplemente se debería desarrollar el endpoint para las búsquedas y utilizar Nominatim (que actualmente ya está implementado en el “Registro de propiedades”).

Otra de las mejoras que se proponen es la de completar el caso de uso de “Favoritos”.

Además, para una futura evolución se propone la generación de informes con precios medios por ciudad, región o país, aprovechando que ya tenemos desarrolladas las clases de dominio, como podemos ver en la ilustración 64, que almacenan esta información de cada propiedad (sobre la que se hacen las subastas).

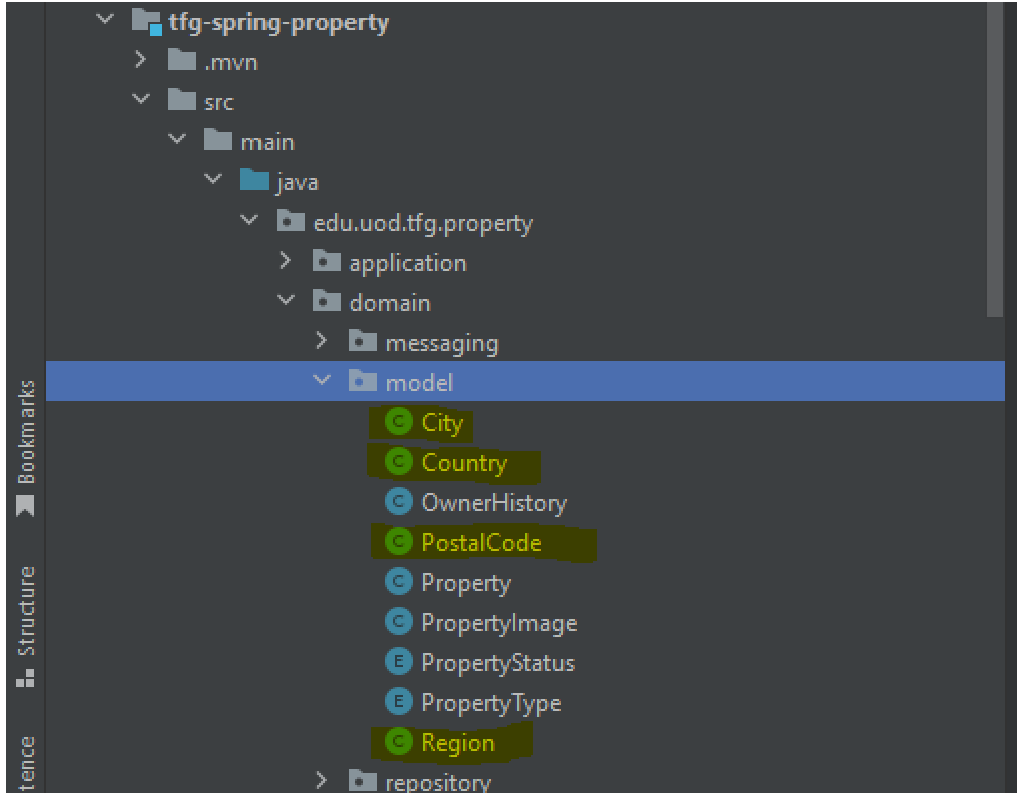


Ilustración 64 Clases de dominio para informes

También se plantea implementar un "blacklist" para los tokens ya caducados o para aquellos usuarios que hacen "logout", ya que actualmente el token continua siendo válido hasta 24 horas y esto podría suponer problemas de seguridad.

5. Conclusiones

La primera conclusión que saco es que ha sido una gran experiencia y un gran reto lograr desarrollar este proyecto desde cero. El balance de estos meses es positivo porque, en general, creo que he cumplido los objetivos que me había marcado en la primera entrega, donde establecimos el marco del proyecto y las tecnologías para desarrollarlo.

En la primera etapa, una de las grandes claves han sido las directrices del tutor, ya que la realidad es que llevaba solamente una idea (que no es poco pero tampoco suficiente), fruto de una necesidad personal, con las ideas básicas del negocio, pero sin una ruta clara para llegar a plasmarlo en un documento para desarrollar el proyecto.

Mi única experiencia previa con el desarrollo de proyectos y con el marco de trabajo ágil es la que he tenido en la asignatura de “Proyecto de desarrollo de software” en grupo (no creía que fuera a servirme tanto después de más de 1 año de haberla cursado, pero sí).

Creo que ha sido gratificante ir cumpliendo objetivos y viendo como cogía forma el producto final y quisiera destacar que, en mi opinión, son especialmente útiles las entregas por fases para centrar los objetivos.

En mi opinión, la segunda fase ha sido la clave de este TFG, es el punto de inflexión porque debes centrar el trabajo en los casos de uso, que posteriormente se convertirán en las funcionalidades de la aplicación y plasmar el esquema de tu desarrollo.

También se presenta un boceto de las pantallas de la App y esto te obliga a pensar muy bien cuales van a ser las necesidades reales (que útil fue la asignatura de “Integración persona ordenador”)

En cuanto al desarrollo, ha sido muy interesante utilizar las metodologías ágiles, aunque es realmente duro en las primeras iteraciones con esos primeros casos de uso donde todo está por hacer y ves el tiempo pasando, gracias a Toni Oller por ayudarme a centrar el foco en lo prioritario en esos momentos.

En cuanto a la implementación, me ha sorprendido que he podido desarrollar, lo que yo veo como, algo decente para las expectativas que tenía cuando empecé con solo una idea, he cubierto muchas de las funcionalidades y las que no, he dejado la base para hacerlo, con las entidades definidas y los objetos listos.

Se pretende desarrollar, en trabajos futuros, los casos de uso “Buscar inmuebles” y “Agregar a favorito”, además de implementar un sistema de informes/estadísticas que sirvan para orientar a los usuarios.

He aplicado muchas de las cosas que aprendí en la asignatura de “Ingeniería del software de componentes y sistemas distribuidos”, gracias a Pau Pineda por la paciencia en su momento.

6.Glosario

API REST: Un enfoque arquitectónico centrado en el desarrollo de servicios web que se basa en un modelo cliente-servidor.

Backend: La parte del desarrollo que se enfoca en la capa de negocios de una aplicación, proporcionando datos y lógica para el frontend.

Framework: Un entorno de trabajo que sigue una estructura estándar, proporcionando un conjunto de módulos, un lenguaje de programación y librerías para facilitar la creación de software.

Frontend: La parte del desarrollo que se concentra en la interfaz con la que los usuarios interactúan.

IDE: Un Entorno de Desarrollo Integrado.

IntelliJ IDEA: Un IDE desarrollado por JetBrains, diseñado para la programación en Java y otros lenguajes.

Java: Un lenguaje de programación orientado a objetos que se ejecuta en una máquina virtual llamada JVM (Java Virtual Machine).

Javascript: Un lenguaje de programación de scripting utilizado en el desarrollo web para operaciones en el lado del cliente.

Metodología ágil: Un enfoque de desarrollo de software que se basa en la flexibilidad y la iteración para adaptarse a los cambios rápidamente.

Microservicio: Un estilo arquitectónico que divide las aplicaciones en módulos independientes para lograr flexibilidad y escalabilidad.

Postman: Un software de pruebas para API REST desarrollado por Postman Inc.

REST: REpresentational State Transfer, un estilo arquitectónico que utiliza **HTTP** para la comunicación en aplicaciones web.

Spring Boot: Un framework de desarrollo para Java que simplifica tareas y se utiliza ampliamente en el desarrollo de microservicios.

Visual Studio Code: Un editor de código desarrollado por Microsoft que es compatible con varios lenguajes y se puede extender con extensiones.

Gateway: Un componente de software que actúa como intermediario entre diferentes sistemas o servicios, gestionando el tráfico y la comunicación entre ellos.

7.Bibliografia

1. PostgreSQL. https://www.postgresql.org/
   (Última vez visitado el 10/12/2023)
2. Apache Kafka. https://kafka.apache.org/090/documentation.html
   (Última vez visitado el 20/12/2023)
3. Spring Boot. https://spring.io/
   (Última visita el 05/01/2024)
4. JavaTpoint- Spring Boot https://www.javatpoint.com/restful-web-services-spring-boot
   (Última visita el 04/01/2024)
5. Spring Gateway https://spring.io/projects/spring-cloud-gateway/
   (Última visita 08/12/2023)
6. Spring Security https://spring.io/projects/spring-security/
   (Última visita 05/12/2023)
7. Project Lombok. https://projectlombok.org/features/
   (Consultado última vez 05/01/2024)
8. JavaTPoint – React Native https://www.javatpoint.com/react-native-tutorial
   (Última visita el 06/01/2024)
9. Nominatim https://nominatim.org/release-docs/develop/api/Overview/
   (Última visita el 22/12/2023)
10. Firebase Storage https://firebase.google.com/docs/storage/web/start?hl=es
    (Última visita el 04/01/2024)
11. Design-toolkit http://design-toolkit.recursos.uoc.edu/es/
    (Ultima visita el 09/11/2023)

8. Anexo

8.1 Instalación y ejecución

Se debe descomprimir el proyecto adjunto **precio-justo-backend.zip** y abrir el proyecto en el IDE (IntelliJ IDEA presentemente).

También se pueden descargar los proyectos desde el repositorio de Github:

Backend: https://github.com/jordilleida/precio-justo-backend
Frontend: https://github.com/jordilleida/precio-justo-frontend

Una vez dentro, se ve el siguiente esquema con los 4 microservicios y el Gateway (se deben agregar a Maven como módulos si no los detecta) y quedaría como en la siguiente captura.

El proyecto cuenta con un archivo `docker-compose.yml` donde están definidos los servicios, Adminer, Kafka, y otros servicios externos. Usando el comando siguiente se iniciarán y ejecutarán los servicios e imágenes si fuera necesario.

`docker-compose up –-build -d`

Luego solamente hay que ejecutar los servicios desde el IDE (debido a que no he podido configurar correctamente los microservicios en dockers)

O bien uno a uno con el botón Run y seleccionándolos, o llendo a View > Tools Window > Services y presionando el icono de play desde Spring Boot, como se ve en la siguiente imagen.

En el caso de la aplicación cliente, se debe extraer el precio-justo-frontend.zip en una carpeta y desde un terminal, en mi caso uso Visual Studio Code, se ejecuta npm run web (si se quiere ver ejecutar desde una web) o puede hacerse con

`npm run web` o bien con ` npx expo start --web` ya que para facilitar el desarrollo he agregado la dependencia a este framework, debe tenerse en cuenta que expo asigna el puerto automáticamente (del 19000 en adelante) cuando se arranca la aplicación,

Como se ve en la ilustración bajo este texto, el puerto en mi caso es 19006

```
PS C:\Users\Jordi\precio-justo-frontend> npm run web

> precio-justo-frontend@1.0.0 web
> expo start --web

Starting project at C:\Users\Jordi\precio-justo-frontend
Starting Metro Bundler
Starting Webpack on port 19006 in development mode.
(node:9964) [DEP_WEBPACK_DEV_SERVER_CONSTRUCTOR] Deprecatio
(Use `node --trace-deprecation ...` to show where the warni
(node:9964) [DEP_WEBPACK_DEV_SERVER_LISTEN] DeprecationWarn
```

8.2 Tests unitarios y pruebas de integración

En el proyecto de backend se han creado tests unitarios y pruebas de integración, cada microservicio tiene sus propios tests para validar que funciona correctamente.

Como se puede ver en la ilustración 65, se han realizado pruebas de integración a partir de la biblioteca RestAssured para verificar que los endpoints devuelven las respuestas esperadas y su funcionamiento por un lado, y por el otro lado tests unitarios con Junit y con Mockito para simular las dependencias tanto del RestController como de los Servicios.

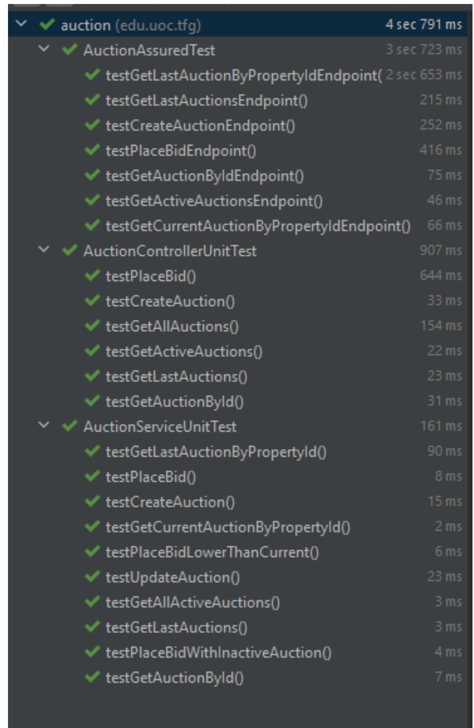


Ilustración 65 Tests del microservicio de Auction