UniLiving

- Sistemas y tecnologías web -

Jorge Leris Lacort (845647)
Luis Daniel Gómez Sevilla (819304)
Ahmed Karafy Mohib (850111)
Kamal Bouizy Ghazzal (838700)

Universidad Zaragoza
1542

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Índice

1. URLs de acceso

Acceso a la API del backend (Swagger):

- https://uniliving-backend.onrender.com/api/docs

  Importante: asegurarse que el servidor que se está usando sea el de desarrollo y no el de pruebas (por defecto está puesto el de desarrollo)

Acceso al frontend:

- https://uniliving-frontend.onrender.com/

2. Credenciales de acceso

Usuario:

- **Correo:** example2025@example.com | **Contraseña:** SisWeb2025

Administrador:

- **Nombre:** jfabra | **Contraseña:** SisWeb2025

También es posible registrarse para acceder como un usuario normal

3. Introducción

Introducción de producto
Este proyecto nace con el objetivo de facilitar la búsqueda de pisos de alquiler en Zaragoza, ofreciendo una plataforma completa y fácil de usar. Los usuarios pueden explorar pisos en alquiler filtrando por sus preferencias, conocer en profundidad los barrios mediante datos reales y comentarios de otros usuarios, y encontrar compañeros de piso compatibles mediante un sistema de emparejamiento. La plataforma también incluye herramientas interactivas como mapas y chats. Además, cuenta con un panel de administración con funciones de moderación, estadísticas de uso y gestión de reportes para garantizar una experiencia de calidad.

Introducción técnica
El proyecto ha sido desarrollado utilizando el stack MERN (MongoDB, Express, React, Node.js), con una arquitectura modular basada en el patrón MVC para el backend y documentación automática con Swagger. En el frontend, se ha empleado React con Bootstrap para el diseño responsive. Además, se integran mapas interactivos con Leaflet, gráficos con Recharts y múltiples librerías para mejorar la experiencia de usuario.

4. Implementación del backend

El backend ha sido desarrollado con una arquitectura modular que favorece la escalabilidad, el mantenimiento y la claridad del código. Para ello, se ha adoptado el patrón arquitectónico Modelo-Vista-Controlador (MVC), que separa las responsabilidades en módulos independientes: la gestión de los datos mediante los modelos, la lógica de negocio a través de los controladores y la definición de las rutas que exponen la API. Esta separación mejora significativamente la organización, facilita la reutilización de código y simplifica futuras ampliaciones o modificaciones.

El backend está desarrollado con Node.js, que permite crear aplicaciones servidoras rápidas y escalables. Para la gestión de datos se utiliza MongoDB, una base de datos NoSQL, junto con Mongoose, que facilita la definición de esquemas, validaciones y operaciones sobre la base de datos de forma sencilla y estructurada.

- **Modelos:** representan las entidades de la base de datos y definen su estructura y validaciones.

- **Controladores:** contienen la lógica de negocio y procesan las peticiones recibidas, comunicándose con los modelos.

- **Rutas:** definen los endpoints que recibe el servidor y redirigen cada petición al controlador correspondiente.

Módulos desarrollados

- **Config:** Centraliza la configuración del sistema, incluyendo la conexión a la base de datos MongoDB, la inicialización con datos de prueba, la configuración del sistema de logs con Winston, la integración con Passport para autenticación local y mediante Google, la carga de la documentación Swagger, y parámetros para el consumo de la API pública de Zaragoza.
- **Models:** Define mediante Mongoose los esquemas y modelos de datos, que representan las colecciones en MongoDB, asegurando la integridad y estructura de la información almacenada.
- **Controllers:** Contiene la lógica que procesa las peticiones, interactúa con los modelos para realizar operaciones sobre los datos y devuelve las respuestas adecuadas a las rutas.
- **Routes:** Organiza las rutas REST de la API por funcionalidad, conectando cada endpoint con su controlador correspondiente. La documentación Swagger facilita la consulta y prueba de estas rutas.
- **Utils:** Incluye funciones auxiliares reutilizables, como el algoritmo de emparejamiento entre usuarios.

- **socket.js:** Gestiona la comunicación en tiempo real con Socket.io, habilitando funcionalidades como chats públicos y privados, y otras interacciones entre usuarios conectados simultáneamente.

Tecnologías y librerías externas

Para el desarrollo y la calidad del backend se han empleado diversas librerías que aportan funcionalidades clave y facilitan las buenas prácticas:

- **Express:** Framework web para gestionar rutas, middlewares y el flujo de peticiones HTTP de forma eficiente.
- **Mongoose y mongodb:** Librerías para definir esquemas, modelos y gestionar la base de datos NoSQL MongoDB.
- **Passport (local, JWT y Google OAuth 2.0):** Herramienta de autenticación segura que permite inicio de sesión tradicional y mediante proveedores externos como Google.
- **jsonwebtoken:** Permite la creación y validación de tokens JWT para autorización y sesiones sin estado.
- **express-rate-limit:** Middleware que limita la cantidad de peticiones de un usuario para proteger la API de abusos y ataques DoS.
- **winston:** Sistema avanzado de logging para registrar eventos y errores, facilitando el monitoreo y la depuración.
- **socket.io:** Biblioteca para comunicación en tiempo real mediante WebSockets, utilizada para chats y notificaciones.
- **swagger-jsdoc y swagger-ui-express:** Herramientas para generación y presentación automática de la documentación API.
- **dotenv:** Gestiona variables de entorno desde archivos .env, mejorando la configuración y seguridad.
- **axios:** Cliente HTTP para realizar peticiones a APIs externas, como la integración con Idealista.
- **bcrypt:** Para el hashing seguro de contraseñas, reforzando la seguridad del sistema de autenticación.
- **nodemailer:** Servicio para el envío de correos electrónicos, usado para enviar notificaciones de error y warning de Winston a nuestros correos.
- **eslint y prettier:** Herramientas para análisis estático y formato automático de código que garantizan calidad y consistencia.

Modelo de datos

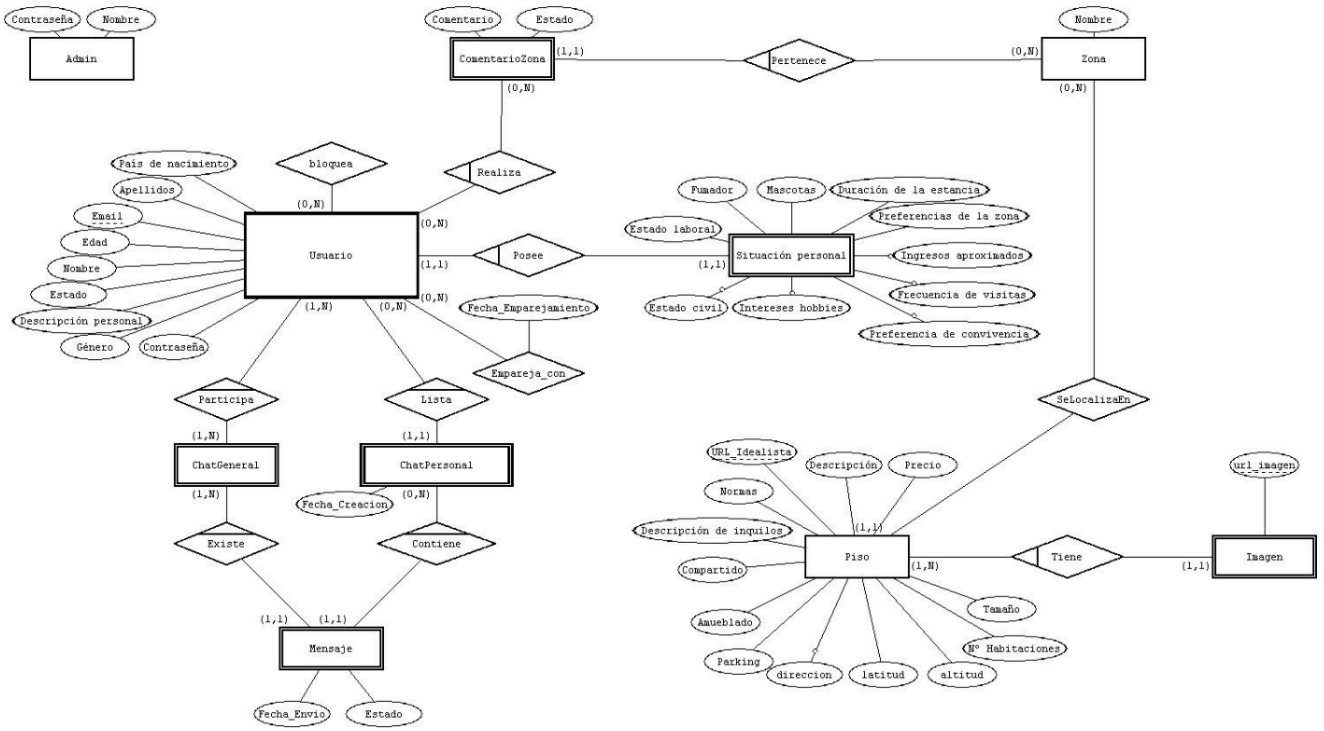
El diseño del modelo de datos comenzó con un esquema Entidad-Relación (E-R), que permitió identificar y definir las entidades principales, sus atributos y las relaciones entre ellas de forma clara y estructurada. Este enfoque facilitó una visión global y lógica de la información necesaria para la aplicación.

A partir de este modelo conceptual, se realizó la transformación al modelo físico utilizando Mongoose, la biblioteca de modelado para MongoDB. Se aplicaron las técnicas específicas de Mongoose para representar las entidades como esquemas, que definen la estructura de los documentos, las propiedades, los tipos de datos y las validaciones necesarias para garantizar la integridad de la información.

Además, se gestionaron las relaciones entre colecciones utilizando referencias (ref) cuando fue adecuado, para mantener la normalización y facilitar las consultas entre documentos relacionados. En otros casos, se optó por la incrustación (embedding) para optimizar el acceso a datos altamente relacionados y mejorar el rendimiento.

Gracias a esta adaptación cuidadosa del modelo E-R a las características de MongoDB y Mongoose, se consiguió un modelo de datos eficiente, flexible y alineado con los requisitos funcionales de la aplicación.

El modelo E-R se puede observar a continuación:



Fuentes de datos utilizadas

Para dotar al sistema de información relevante y actualizada sobre los pisos y su entorno, se han integrado diversas fuentes de datos externas:

- **API de Idealista**: Se ha utilizado esta API como fuente principal de información sobre los inmuebles disponibles. A través de ella se obtiene información detallada y estructurada de cada piso, como su ubicación, superficie, precio, características específicas y disponibilidad. No obstante, debido a las limitaciones en el número de peticiones permitidas por la API, se ha implementado un sistema de almacenamiento en base de datos (MongoDB) para cachear los resultados. Esta solución permite persistir los datos obtenidos y reducir significativamente el número de consultas directas a la API, respetando así sus restricciones de uso y mejorando el rendimiento general del sistema. Gracias a esta integración, se ofrece a los usuarios una base de datos rica, actualizada y optimizada de viviendas reales.

- **Datos Abiertos del Ayuntamiento de Zaragoza**: Se ha accedido a la plataforma municipal de datos públicos para enriquecer la información contextual de cada barrio. Entre los datos utilizados se incluyen:

    - Estadísticas demográficas (población joven, densidad de habitantes, etc.)
    - Indicadores económicos (salario medio, índice de actividad)
    - Indicadores sociales (índices de natalidad y maternidad)

- Estos datos han sido fundamentales para proporcionar a los usuarios un análisis más completo del entorno de cada vivienda, lo que facilita una toma de decisiones más informada.

- **Catálogo de equipamientos y servicios públicos**: También a través del portal de datos abiertos de Zaragoza, se ha utilizado información geográfica de interés para los usuarios, incluyendo:

    - Centros educativos (colegios, institutos)
    - Supermercados
    - Instalaciones deportivas y culturales
    - Otros servicios de proximidad (hospitales, etc)

Esta combinación de fuentes permite ofrecer una visión global del entorno de cada piso, más allá de las características físicas de la vivienda, incorporando elementos clave del contexto urbano y social.

5. Implementación del frontend

Tecnología utilizada

Para el desarrollo del frontend, se ha utilizado React como biblioteca principal para la construcción de interfaces de usuario. La principal utilidad que nos da React es crear sitios dinámicos y modulares gracias a su enfoque basado en componentes. Como apoyo al diseño visual y a la maquetación se ha integrado Bootstrap, en conjunto con su versión adaptada para React (react-bootstrap), permitiendo una rápida construcción de interfaces responsive.

Además, se han utilizado múltiples librerías complementarias que enriquecen la funcionalidad y la experiencia del usuario, como sistemas de mapas interactivos, gestión de rutas, gráficos estadísticos, etc.

Estructura del proyecto

La estructura del proyecto frontend sigue la convención básica de una aplicación React creada con Create React App. Se organiza de la siguiente manera:

- public/: Contiene el archivo index.html y recursos estáticos.
- src/: Carpeta principal de desarrollo, donde se agrupan:
    - components/: Componentes reutilizables como cabeceras, formularios, paginación, modales, chat, etc.
    - pages/: Páginas principales de la aplicación (Inicio, Perfil, etc.)
    - assets/: Recursos como imágenes o iconos personalizados.
    - index.js: Punto de entrada de la aplicación.

El enrutamiento está implementado utilizando react-router-dom, que permite la navegación entre páginas mediante rutas definidas (en indexRoutes). Este enrutador se encuentra encapsulado dentro de un AuthProvider, lo cual permite proteger rutas privadas y restringir el acceso a ciertas secciones de la plataforma según los permisos del usuario.

Módulos utilizados

A continuación, se muestran los principales módulos utilizados en la implementación del frontend:

- **React (react, react-dom):** Biblioteca principal para la construcción del frontend. Permite crear componentes reutilizables y manejar el estado.
- **React Router DOM (react-router-dom):** Gestiona la navegación entre páginas mediante rutas declarativas. Soporta rutas protegidas, navegación anidada y gestión de parámetros en la URL.
- **Axios:** Cliente HTTP para realizar peticiones al backend de forma sencilla, utilizado para obtener y modificar datos de la API.
- **Bootstrap y React-Bootstrap:** Sistema de diseño y componentes estilizados que garantizan una interfaz responsive. Se combina la potencia de Bootstrap con la flexibilidad de componentes React.
- **React Icons, Lucide React y FontAwesome:** Proveen iconografía variada para mejorar la experiencia visual y la navegación.
- **React Leaflet, Leaflet, y Leaflet.MarkerCluster:** Permiten la visualización de mapas interactivos y la representación de pisos en un mapa. Se utiliza marker clustering para agrupar los pisos en zonas densas y mejorar la usabilidad.
- **Recharts:** Biblioteca para visualización de datos mediante gráficos. Se usa principalmente para mostrar estadísticas de zonas y datos analíticos para los administradores.
- **Socket.io-client:** Permite la comunicación en tiempo real con el backend para implementar el chat privado y el chat general entre usuarios.
- **React infinite scroll:** Permite cargar más contenido de forma automática al hacer scroll en el chat, evitando paginación tradicional. Mejora la experiencia de usuario al navegar por muchos mensajes.
- **React Google ReCAPTCHA:** Integrado en formularios de registro para verificar que el usuario no es un bot, aumentando la seguridad.
- **React Image Gallery:** Utilizada para mostrar galerías de imágenes de cada piso en una vista tipo carrusel, con opciones de visualización ampliada. *(Aunque idealista solo nos devuelve una imagen, se tiene así para poder escalar)*
- **RC Slider Componente:** Visual que permite seleccionar rangos (por ejemplo, filtro de precio) de forma interactiva.

Destacamos también el uso de de un fichero geojson, el cual permite mostrar los diferentes barrios de Zaragoza de forma visual y poder seleccionar cada uno de ellos.

6. Validación y pruebas realizadas

Para garantizar la calidad, robustez y correcto funcionamiento del backend, se ha utilizado Jest como framework principal de pruebas automatizadas.

Se han desarrollado pruebas unitarias y de integración que abarcan:

- Validación de la lógica en los controladores
- Comprobación de rutas y sus respuestas
- Verificación del correcto funcionamiento de los servicios y modelos

Estas pruebas permiten detectar errores de forma temprana durante el desarrollo y asegurar que los cambios futuros no introduzcan regresiones. Además, la ejecución automatizada de las pruebas forma parte del flujo de integración continua configurado, asegurando que cada versión del backend mantiene un nivel de calidad adecuado antes de ser desplegada.

Respecto al frontend no se han realizado pruebas automatizadas, por lo que todas las validaciones que han sido necesarias se han realizado manualmente.

7. Mejoras implementadas

A lo largo del desarrollo del proyecto se han incorporado varias mejoras técnicas con el objetivo de reforzar la seguridad, aumentar la calidad del código y mejorar el flujo de trabajo en equipo. A continuación, se detallan las más relevantes:

Protección del formulario de registro con CAPTCHA

Se ha incorporado **Google reCAPTCHA v2** en el formulario de registro con el objetivo de prevenir registros automatizados por bots. Esta solución fue seleccionada tras analizar diferentes alternativas y se optó por la versión con casilla de verificación ("No soy un robot") por las siguientes razones:

- Es una de las soluciones más reconocidas y utilizadas en el mercado, lo que garantiza fiabilidad y seguridad.
- La interfaz es intuitiva y accesible, ya que solo requiere una acción mínima por parte del usuario, lo que mejora la experiencia de uso.
- Ofrece alternativas de verificación por audio, permitiendo su uso por parte de personas con discapacidad visual, cumpliendo así con criterios de accesibilidad.
- Cuenta con un componente oficial para React (`react-google-recaptcha`), lo que facilita su integración en aplicaciones desarrolladas con este framework.
- Google actualiza de forma constante el servicio, asegurando su efectividad frente a nuevas técnicas de evasión por parte de bots.
- Es menos intrusivo que otros métodos de verificación más complejos, sin comprometer la seguridad.
- Dispone de un plan gratuito que permite hasta 10.000 verificaciones mensuales, lo cual es más que suficiente para las necesidades previstas del sistema.
- Incluye un panel de administración con analíticas y configuración accesible, lo que facilita su supervisión y ajuste.

Adicionalmente, se ha implementado un sistema de control de intentos fallidos en el proceso de registro. En caso de múltiples intentos inválidos, el acceso al formulario queda temporalmente bloqueado, reforzando así la protección contra ataques de fuerza bruta.

Alta cobertura de pruebas en backend

Se ha logrado una **cobertura de código superior al 75%** en el backend mediante pruebas automáticas utilizando **Jest**. Estas pruebas abarcan los principales componentes del sistema:

- Controladores
- Rutas
- Servicios
- Modelos

La cobertura ha sido verificada mediante el uso del parámetro `--coverage` de Jest y mediante el análisis en **SonarCloud**, lo que proporciona una validación objetiva y centralizada del nivel de cobertura alcanzado.

Gracias a esta estrategia de pruebas y análisis continuo, se garantiza que la mayor parte del sistema ha sido evaluada y comprobada automáticamente, lo que contribuye significativamente a reducir el riesgo de errores en producción y a mejorar la mantenibilidad del proyecto.

Análisis estático de código

Se han empleado diversas herramientas de análisis estático para garantizar la calidad y consistencia del código fuente en todo el proyecto:

- En el **backend**, se ha utilizado una combinación de **ESLint**, **Prettier** y **SonarCloud**. Estas herramientas han permitido detectar y corregir:
    - Malos olores de código (*code smells*)
    - Errores comunes de lógica y sintaxis
    - Dependencias innecesarias u obsoletas
    - Inconsistencias en el estilo de codificación
- En el **frontend**, aunque inicialmente se utilizaron ESLint y Prettier, se optó finalmente por **Codacy** como herramienta complementaria. Esto se debe a que ciertas reglas impuestas por las configuraciones estándar de ESLint resultaban poco adecuadas para la estructura y el enfoque práctico del frontend, lo que dificultaba el desarrollo. Codacy ofreció un análisis más flexible y ajustado a las necesidades del proyecto, manteniendo un control de calidad efectivo sin interferir negativamente en la productividad.

El uso combinado de estas herramientas ha contribuido significativamente a mejorar la calidad, mantenibilidad y legibilidad del código, facilitando su evolución futura.

Integración continua con GitHub Actions

Se ha configurado una **pipeline de integración continua (CI)** usando **GitHub Actions**, que se ejecuta automáticamente en cada *push* y *pull request* al repositorio. Esta pipeline incluye:

- Instalación de dependencias
- Análisis de código con ESLint
- Ejecución automática de tests con Jest (incluyendo cobertura)

Gracias a esta integración, se asegura que el código que llega a la rama principal está **verificado, probado y libre de errores evidentes**, facilitando un desarrollo colaborativo eficiente y profesional

8. Valoración global del proyecto

El resultado final del proyecto es altamente satisfactorio. Se ha conseguido desarrollar una plataforma web completa, funcional y visualmente atractiva, que cumple con los objetivos planteados desde el inicio. A pesar de algunos inconvenientes y limitaciones durante el desarrollo, hemos logrado implementar un backend robusto y bien estructurado, junto con un frontend profesional.

La integración de funcionalidades clave como el sistema de búsqueda avanzada, los mapas interactivos, el emparejamiento de compañeros de piso y la moderación por parte de administradores, demuestra un alto nivel de desarrollo técnico, los cuales nos han supuesto diferentes tipos de desafíos. En conjunto, se ha conseguido una plataforma sólida, escalable y con potencial real de uso en un entorno productivo.

En la entrega actual por los impedimentos surgidos, lo único que está a medias es la integración del chat individual en el frontend (totalmente funcional en el backend)

9. Mejoras propuestas

Si volviéramos a desarrollar este proyecto desde cero, aplicaríamos varias mejoras tanto en el backend como en el frontend:

- En primer lugar, buscaríamos ampliar el acceso a la API de Idealista para obtener una mayor cantidad de datos, como por ejemplo más imágenes por piso, ya que actualmente solo disponemos de una imagen por inmueble y no podemos mostrar la galería.
- Además, optimizaríamos el sistema de cacheo de datos provenientes de Idealista, ya que, al contar con un número limitado de peticiones, los datos tienden a desactualizarse con rapidez.
- En el frontend, mejoraríamos la reutilización y organización de componentes para favorecer un desarrollo más limpio y mantenible. Una mayor separación de responsabilidades permitirían reducir la complejidad en componentes más grandes. Esto porque en el desarrollo actual se tienen páginas con muchas líneas y hay ciertas partes repetidas que se podrían haber separado en componentes.
- También sería útil en otros desarrollos o en continuación de este aplicar validación E2E (End-to-End) en el frontend para verificar el correcto funcionamiento de todo con el transcurso de las actualizaciones, ya que actualmente todas las verificaciones en el frontend se han hecho manualmente.
- Por último, exploraríamos la posibilidad de internacionalizar la plataforma desde el inicio para permitir su escalado a otras ciudades o países con una mínima reestructuración del código.