UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Generación de mapas de calor mediante un sistema de geoposicionamiento

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: García Acidellas, Christian

Tutor/a: Silva Galiana, Josep Francesc

CURSO ACADÉMICO: 2021/2022

Resumen

Este proyecto construye un sistema de generación de mapas de calor para personas que transitan un centro comercial, o en general, en el interior de cualquier edificio.

El sistema se compone de dos partes. Por un lado, un módulo de geoposicionamiento basado en el sistema operativo Android que recoge y almacena la posición de un dispositivo móvil a lo largo del tiempo. Por otro lado, un módulo de generación de mapas de calor que se pueden visualizar en una aplicación web a partir de las posiciones registradas. Para ello, el sistema es capaz de manejar grandes cantidades de datos geo, y la información de cada edificio para poder ofrecer al usuario búsquedas según criterios de localización y rango temporal.

El presente documento detalla paso a paso cómo se ha realizado el desarrollo del sistema, pasando por los diferentes usos que tienen los mapas de calor y las tecnologías dirigidas a ese propósito. Seguidamente, se realiza un proceso completo de Ingeniería de Software, donde se empieza por la definición de requisitos y el análisis del sistema, seguido del diseño y la implementación, y se termina con las pruebas. Para finalizar, se realiza una conclusión con la evaluación final del sistema y valoración personal.

Palabras clave: Vue, Javascript, Node JS, API, PostgreSQL, programación, mapas de calor, HTML, Tailwind CSS

Abstract

This project builds a heat map generation system related to people that are transiting a shopping mall, or in general, inside any building.

The system is made up of two parts. On the one hand, a geopositioning module based on the Android operating system, that collects and stores the position of a mobile device over time. On the other hand, a heat map generation module than can be displayed in a web application based on the registered positions. To do this, the system is capable of handling large amounts of geo data, and the information of each building, to be able to offer the user searches according to location criteria and time range.

The following document details step by step how the development of the system has been carried out, going through the different users of heat maps and the technologies aimed for that purpose. Next, a complete software engineering process is carried out, starting with the definition of requirements and analysis of the problem, followed by design and implementation, and ending with testing. At the very end, a conclusion is made with the final evaluation of the system and some personal assessment.

Keywords: Vue, Javascript, NodeJS, API, PostgreSQL, programación, mapas de calor, HTML, Tailwind CSS

Índice de contenidos

Introducción **7**
Motivación 7
Objetivos 7
Impacto Esperado 7
Metodología 8
Estructura 10
Estado del Arte **12**
Análisis del Problema **14**
Especificación de Requisitos 14
Visión 14
Actores 14
Requisitos funcionales 14
Requisitos no funcionales 15
Modelo de dominio 15
Modelo de contexto 16
Casos de Uso 17
Gestión de Coordenadas (Servidor REST) 17
Gestión de Edificios (Servidor REST) **19**
Sistema de búsqueda (Aplicación web) 20
Gestión de Edificios (Aplicación web) 21
Ajustes del mapa (Aplicación web) 23
Análisis de Riesgos 25
Riesgos de aceptación 25
Riesgos de satisfacción 25
Riesgos tecnológicos y de integración 26
Identificación y análisis de soluciones posibles 26
Análisis de arquitectura 26
Análisis de tecnologías servidor 27

Frameworks de desarrollo 27
Base de datos 28
Entornos de desarrollo integrado 29
Análisis de tecnologías web 29
Frameworks de desarrollo 29
Frameworks CSS 30
Entornos de desarrollo integrado 31
Diseño de la Solución **32**
Arquitectura del Sistema 32
Diseño de Componentes 33
Servidor 33
Arquitectura REST API Server 33
Base de datos 34
Cliente web 34
Arquitectura Web 34
Wireframes de las interfaces 35
Desarrollo de la Solución **37**
Desarrollo del servidor 37
Rutas 40
Controladores 42
Repositorio y base de datos 43
Desarrollo de la aplicación web **45**
Componentes 45
Vistas 46
Router 47
Mapa Leaflet 47
Librerías Leaflet 48
Mapa de Calor (Leaflet.heat) 48
Distortable Image 49
Implantación **50**
Preparativos **50**

Instalación **50**

Ejecución **51**

Pruebas **52**

Pruebas unitarias 52

Pruebas de rendimiento 54

Pruebas de aceptación 57

Primera sesión **57**

Segunda sesión **57**

Tercera sesión **58**

Última sesión **59**

Conclusión **60**

Resultados del proyecto 60

Aprendizaje 60

Valoración personal **60**

Trabajos Futuros **61**

Índice de ilustraciones **62**

Índice de tablas **65**

Referencias **66**

1. Introducción

1.1. Motivación

Durante la realización de la rama de Ingeniería de Software he tenido la oportunidad de formar parte de dos aplicaciones para el sistema operativo Android[1], con temáticas relacionadas con la movilidad y el turismo. Por tanto, en esos sistemas la integración de mapas y GPS eran aspectos esenciales del desarrollo. Entre las distintas posibilidades que existían en la oferta pública de TFGs, me llamó la atención este, ya que tenía que ver con un tema ya trabajado anteriormente, lo cual me permitiría profundizar en ese área.

Otra gran motivación es que se trata de un entorno totalmente desconocido para mí, ya que se trata de desarrollo web, y lo tomo como una oportunidad para desafiarme a mí mismo y poder aprender más como alumno antes de finalizar los estudios.

1.2. Objetivos

Los objetivos principales de este TFG son, por un lado, el diseño y la implementación de una aplicación web y, por otro lado, un servidor *backend* que permita el almacenamiento de información y la comunicación entre los clientes web y móvil mediante llamadas API REST.

El cliente móvil será el encargado de enviar los datos geográficos al servidor y asegurarse de la veracidad de los mismos utilizando la información disponible sobre los edificios.

El cliente web será el encargado de recibir los datos geo y transformarlos en mapas de calor, y dispondrá también de un apartado donde se podrán consultar, crear, eliminar y editar (CRUD) los edificios en los que se implemente el sistema.

En el servidor se implementará una API para la recepción y el envío de información a ambos clientes.

1.3. Impacto Esperado

Al finalizar este proyecto se espera que este sistema pueda contribuir en gran medida al análisis de la movilidad y ocupación de un grupo de personas dentro de un edificio a lo largo del tiempo.

La implantación original se realizará en la Escuela Técnica Superior de Ingeniería Informática (ETSINF) de la Universitat Politècnica de València (UPV), donde se tiene previsto que ayudará a mejorar la ocupación de las aulas de forma más eficiente, y nos permitirá identificar las zonas con más afluencia para poder situar *stands* de empresas o publicidad dirigida al alumnado.

[1] Android for Developers: https://developer.android.com/

Además, se prevé utilizar este sistema dentro de otros edificios como centros comerciales o museos, donde los propietarios podrán identificar rápidamente las zonas importantes y realizar los cambios necesarios para mejorar sus servicios.

Todo esto deberá estar implementado en un entorno web, donde el usuario será capaz de acceder a la aplicación desde cualquier dispositivo que disponga, con total accesibilidad sin necesidad de instalarlo de forma local.

1.4. Metodología

El desarrollo de este proyecto ha constado de diferentes fases, y cada una de ellas se ha documentado en un tablero Trello[2] para poder monitorizar el progreso de forma organizada y visual.

La primera fase comenzó con una reunión inicial de planificación, donde se concretaron todos los detalles del TFG y las tareas iniciales a realizar durante las siguientes semanas, que constaban principalmente de diseñar el sistema e investigar las tecnologías a utilizar. Esta fase transcurrió desde principios de febrero hasta principios de marzo.

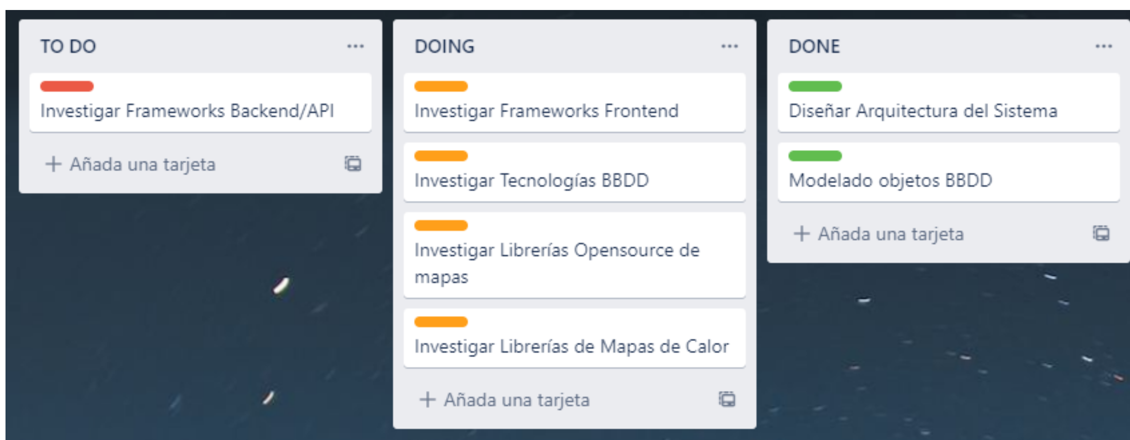

Ilustración 1. Tablero Trello de la primera fase

[2] Trello tour: https://trello.com/tour

La segunda fase comenzó con otra reunión donde se hizo un consenso de la arquitectura y las tecnologías a utilizar, y se establecieron los objetivos, donde se iba a llevar adelante el desarrollo del *backend* mientras se realizaba la formación en el *framework* utilizado para el *frontend*. Esta fase transcurrió desde principios de marzo hasta mediados de abril.

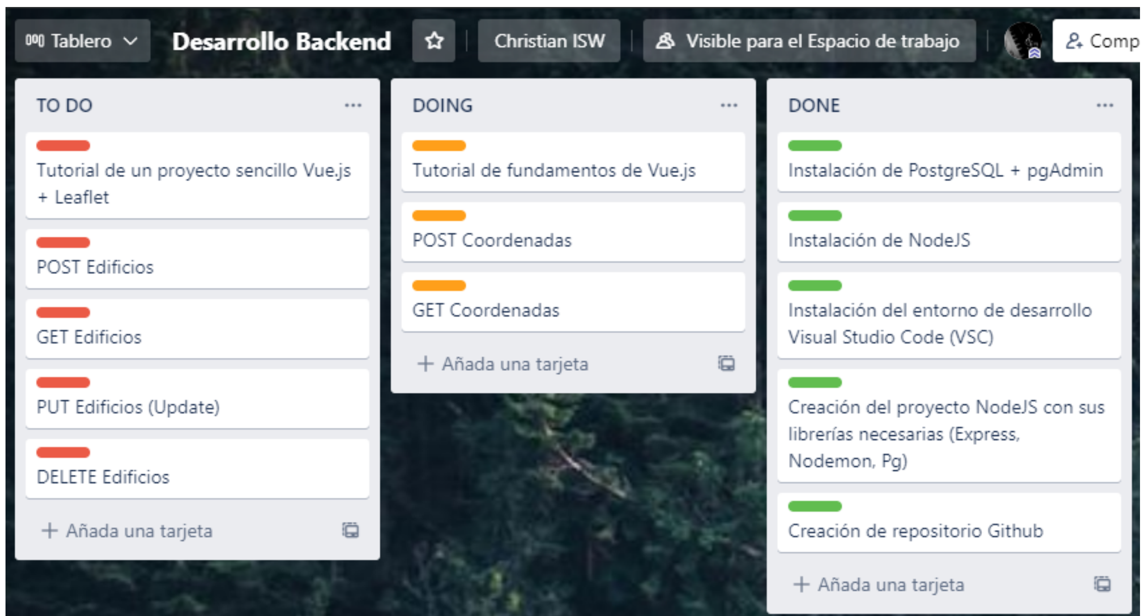

Ilustración 2. Tablero Trello de la segunda fase

En la tercera fase se realizó una revisión de todo lo desarrollado en la fase anterior, y se identificaron todas las correcciones a realizar. Además, se concretaron todos los objetivos a cumplir para la parte del *frontend.* Esta fue con diferencia la fase más larga, transcurriendo desde mediados de abril hasta finales de junio.

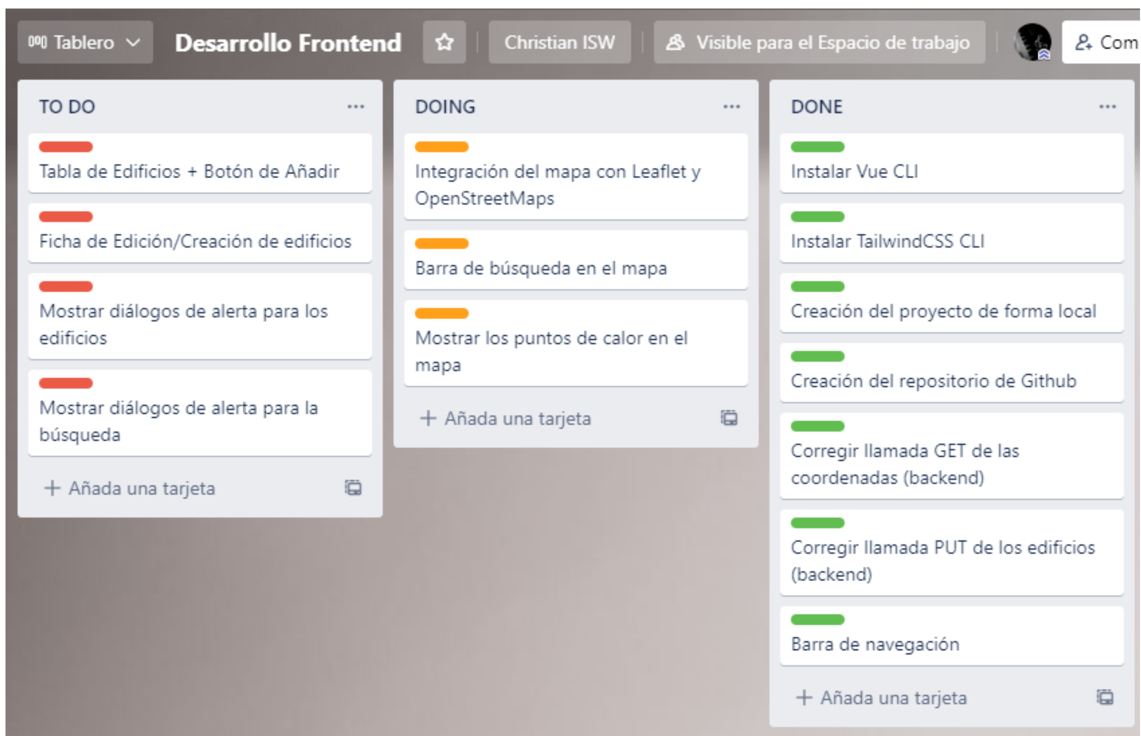

Ilustración 3. Tablero Trello de la tercera fase

Finalmente, en la última fase se realizó una validación del trabajo realizado, se identificaron todos los defectos del sistema y se organizaron en un último tablero para resolverlos y dar por terminado el proyecto.

Ilustración 4. Tablero Trello de la última fase

Las fases del proyecto se resumen en el diagrama de Gantt de la Figura 5.

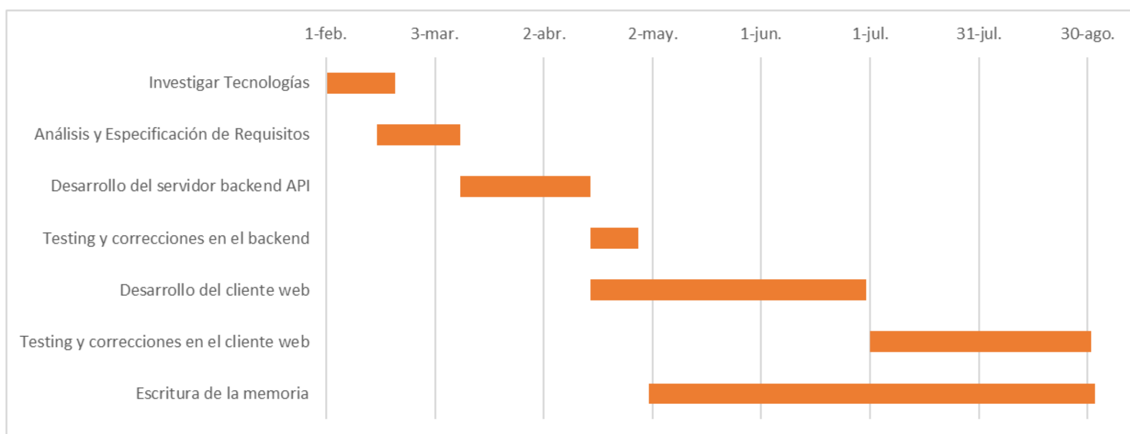

Ilustración 5. Diagrama de Gantt sobre el tiempo invertido en cada fase

1.5. Estructura

La memoria de este proyecto se divide en una serie de apartados que explicaremos a continuación:

En primer lugar, tras la introducción, presentaremos el estado del arte, donde veremos diversas aplicaciones actuales relacionadas con nuestro sistema, y detallaremos qué aspectos están relacionados con nuestra aplicación y en qué se diferencian. Además veremos las diferentes utilidades de los mapas de calor y los proveedores más importantes de mapas de código abierto junto a sus APIs.

En segundo lugar, pasaremos a la especificación de requisitos donde describiremos todas las características y funcionalidades que tiene que cumplir el sistema a desarrollar.

A continuación, realizaremos un análisis del problema para poder plantear una solución que se adapte a las necesidades descritas anteriormente.

En el siguiente punto pasaremos a la fase de diseño, donde especificaremos en detalle la arquitectura del sistema, el modelo de datos y todas las interfaces que componen la aplicación.

Tras la fase de diseño, explicaremos en detalle cómo se ha desarrollado la solución propuesta, haciendo énfasis en las partes más importantes y en las dificultades encontradas durante el desarrollo.

En penúltimo lugar, en la fase de implantación, explicaremos los pasos a seguir para que los usuarios puedan utilizar la aplicación.

Finalmente, pasaremos a detallar todas las pruebas llevadas a cabo para garantizar la calidad del sistema, el cumplimiento de los requisitos establecidos, y su rendimiento con cargas elevadas.

Para concluir este documento, resumimos brevemente todo el trabajo realizado y si cumple con todas las características propuestas en las primeras fases. Además, realizaremos una valoración personal tanto en lo personal como en lo académico, relacionando esto último con todo lo aprendido en la carrera.

2. Estado del Arte

En este punto vamos a comparar nuestra aplicación con otras que se utilizan en la actualidad.

En cuanto a la aplicación web, el framework front-end de Vue se puede ver en páginas bastante famosas como 9gag[3], Nintendo[4] y Gitlab[5], entre otras. Todas estas páginas web ofrecen una experiencia cómoda y rápida al usuario.

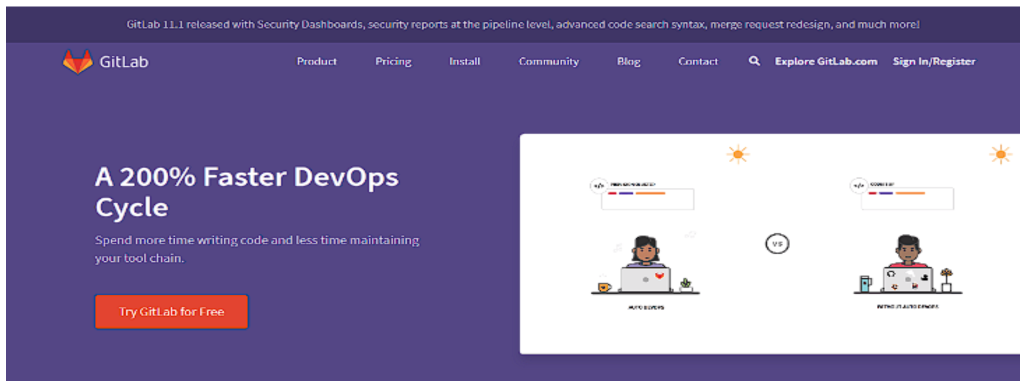

Ilustración 6. Página principal de GitLab

Otro de los aspectos más importantes de este sistema es la utilización de OpenStreetMap[6] (OSM), que se trata de un visualizador de mapas de uso libre bajo licencia abierta. Se puede encontrar en numerosas aplicaciones para diversas plataformas y finalidades totalmente distintas, como el transporte, la arqueología, y la cartografía marítima (véase la wiki de OSM: https://wiki.openstreetmap.org/wiki/List_of_OSM-based_services#Routing para encontrar una lista exhaustiva).

Los mapas de calor se han utilizado en diversos ámbitos y aplicaciones a lo largo de los años, desde representar la actividad de los usuarios en tu repositorio de Github[7] (véase como ejemplo la Figura 7), hasta representar el movimiento de miles de usuarios en el mapa de una ciudad o en una página web (véase como ejemplo las Figuras 8 y 9).

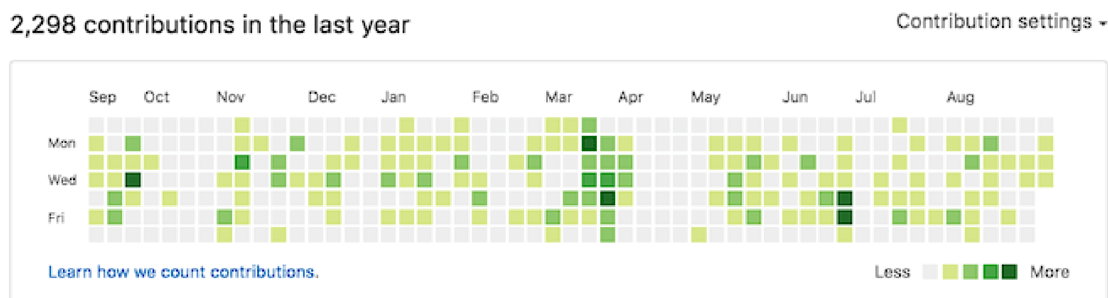

Ilustración 7. Calendario de contribuciones en un repositorio de Github a lo largo del año

[3] About 9GAG: https://about.9gag.com/app
[4] About us Nintendo: https://www.nintendo.com/about/
[5] About Gitlab: https://about.gitlab.com/company/
[6] About OpenStreetMap: https://www.openstreetmap.org/about
[7] Github Docs: https://docs.github.com/en

Ilustración 8. Mapa de calor sobre la ciudad de San Francisco

Ilustración 9. Mapa de calor sobre la página web de una tienda online

Estos ejemplos proponen soluciones bastante eficientes, pero para cumplir con necesidades muy específicas. Con este proyecto se pretende ofrecer una buena visualización e interacción de los mapas de calor, y un *dashboard* donde podamos gestionar los edificios que componen el sistema de manera sencilla y eficiente.

En un futuro se podrían implementar más funcionalidades dependiendo de las necesidades del cliente y el usuario. Podría resultar interesante ofrecer una visualización en tiempo real de los usuarios, ofrecer visualización de interiores en los edificios más importantes, o incluso ofrecer filtros más extensos relacionados con la edad, el género, patrones en los recorridos, etc.

3. Análisis del Problema

3.1. Especificación de Requisitos

3.1.1. Visión

En este apartado se proporciona una visión general de todo lo relacionado con este proyecto, y a partir de ella definimos el alcance del proyecto. Así bien, mostraremos los actores que interactuarán con el sistema, los requisitos funcionales y no funcionales, y los modelos de dominio y contexto.

3.1.1.1. Actores

Los actores en este sistema no se pueden diferenciar de momento, al no disponer de una mecanismo de autentificación ni usuarios en la base de datos. En todo caso, podríamos diferenciar entre los siguientes actores:

- Usuario. Persona que hace uso del sistema mediante su interfaz gráfica y envía datos al servidor.
- Administrador. Persona que hace uso del sistema y su finalidad es hacer cualquier modificación necesaria sobre la información contenida en el sistema.

3.1.1.2. Requisitos funcionales

Los requisitos funcionales se refieren a las diferentes funciones que tiene que ofrecer el sistema, que luego se tienen que descomponer en casos de uso. Los principales requisitos que podemos identificar son los siguientes:

- Servidor REST
    - Gestión de coordenadas: El sistema debe ofrecer un API para poder crear y obtener las coordenadas guardadas en el servidor.
    - Gestión de edificios: El sistema debe ofrecer un API para poder crear, modificar, obtener y eliminar los edificios guardados en el servidor. Un edificio es una localización del mundo real, y se distingue por su nombre, su número de plantas, y sus coordenadas.
- Aplicación web
    - Gestión de edificios: El sistema permitirá al usuario la consulta, modificación, creación y eliminación de edificios.
    - Sistema de búsqueda: El sistema permitirá al usuario hacer consultas al servidor enviando el identificador del edificio seleccionado y un rango de fechas, para luego recibir una respuesta en forma de mapa de calor.
    - Ajustes del mapa: El sistema permitirá al usuario personalizar el mapa a su gusto, ya sea ajustando el zoom, seleccionado la vista de mapa o satélite, o incluso insertar varias imágenes sobre el mapa..

3.1.1.3. Requisitos no funcionales

Los requisitos no funcionales se suelen denominar atributos de calidad de un sistema, ya que se refiere a las restricciones y condiciones que debe cumplir el sistema en sí, no a su comportamiento. Podemos identificar los siguientes:

- Tiempos de respuesta bajos. La aplicación debe ser capaz de realizar las operaciones y búsquedas de forma que el usuario no tenga que esperar más de 10 segundos, y el tiempo medio de respuesta sea inferior a un segundo.
- Usabilidad. El sistema debe ser sencillo de aprender y proporcionar al usuario mensajes de error que sean claros e informativos.
- Portabilidad. La aplicación web debe tener la capacidad de ser usada en cualquier dispositivo móvil Android con versión 4.4 o superior.
- Escalabilidad. El sistema debe ser capaz de manejar grandes cantidades de datos, y ampliarse de forma sencilla para poder soportar una carga mayor.

3.1.1.4. Modelo de dominio

El modelo de dominio nos permite identificar y nombrar los conceptos más importantes del sistema, establecer un lenguaje común entre el desarrollador y el cliente, e identificar y nombrar relaciones entre estos conceptos.

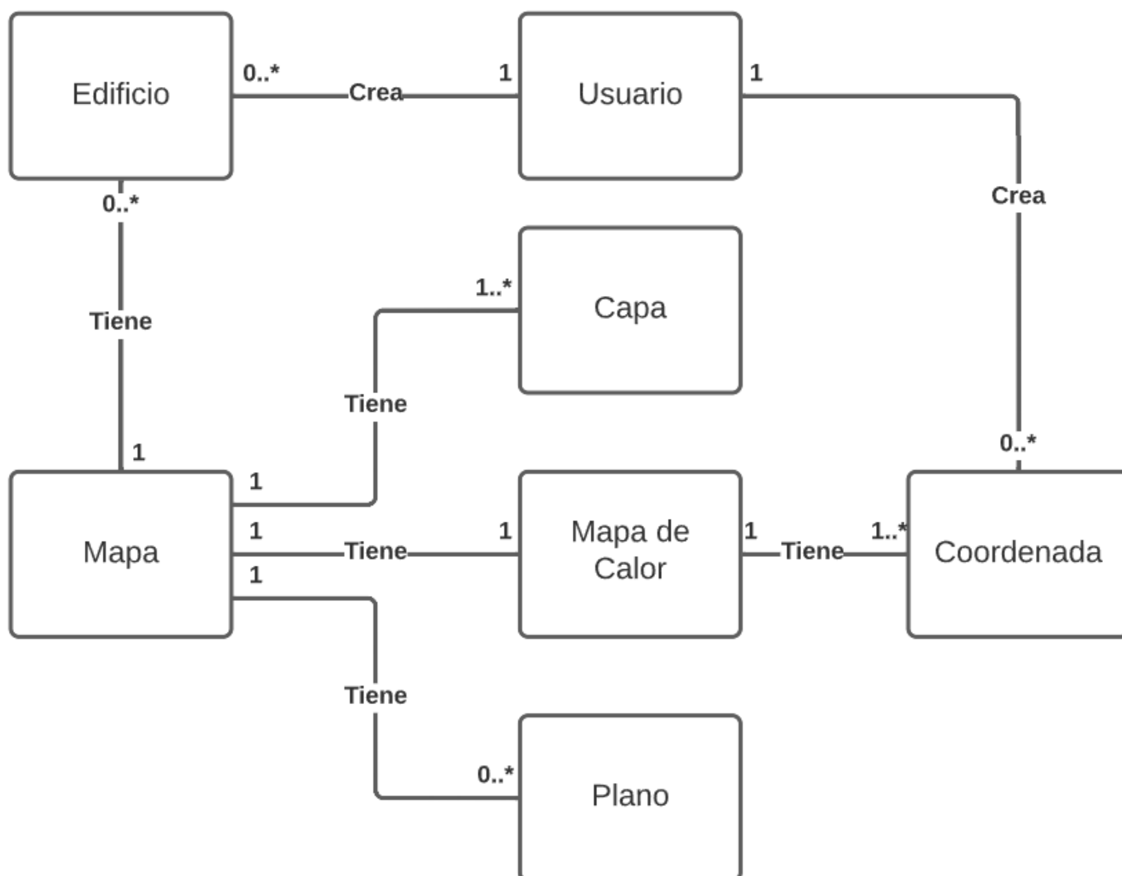

Ilustración 10. Modelo de dominio

Para evitar confusiones, a continuación se dará una breve explicación de cada concepto ubicado en cada caja del modelo de dominio:

- Edificio. Cualquier estructura del mundo real construida con materiales resistentes y con el propósito de ser habitada u otros usos. Que en nuestra aplicación es creada por un usuario y se puede ubicar en un mapa.
- Mapa. Un elemento visual con la representación geográfica de la Tierra. Que en nuestra aplicación es utilizada para ubicar los edificios y poner elementos por encima para mostrarnos información.
- Capa. Es la forma en la que elegimos visualizar el mapa, hay de distintos tipos y diversas fuentes, las más conocidas son la vista satélite y de carreteras.
- Mapa de Calor. Es un mapa con un gradiente de colores para diferenciar entre las distintas agrupaciones de puntos en el mapa.
- Plano. Es un plano de un edificio que podemos importar desde nuestro dispositivo y visualizar sobre el mapa.
- Coordenada. Es una representación de la posición de un usuario en un tiempo y lugar determinado.
- Usuario. Es una persona física que interactúa con el sistema.

3.1.1.5. Modelo de contexto

El modelo de contexto se utiliza para definir los límites entre los subsistemas o partes del sistema, y mostrando las entidades que interactúan con el mismo.

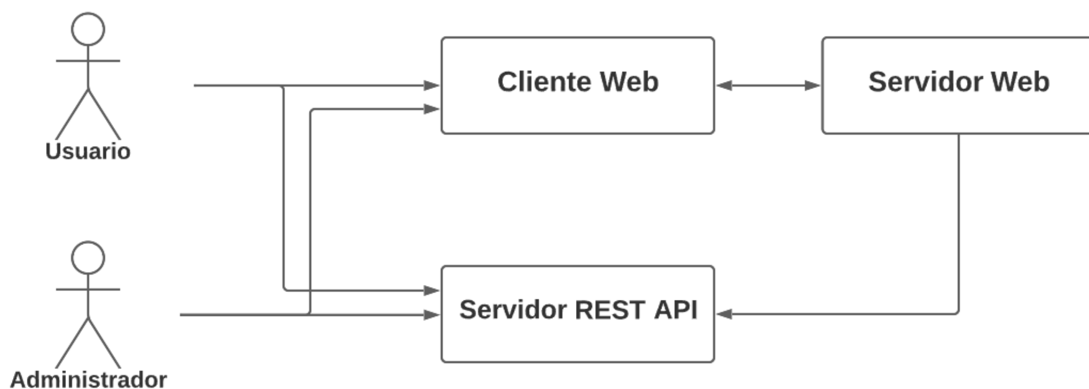

Ilustración 11. Modelo de contexto

Cada una de las partes del sistema se explican brevemente a continuación:

- Cliente Web. Es la parte del sistema orientada a mostrar una interfaz gráfica al usuario con la cual puede interactuar y mandar solicitudes al servidor web.
- Servidor Web. Es la parte del sistema que actúa como intermediaria entre el cliente web y el servidor REST, encargada de enviar y recibir solicitudes para posteriormente procesarlas.

- Servidor REST API. Es la parte del sistema que se encarga de recibir las solicitudes del servidor web, procesarlas y enviar una respuesta. Ya que en este sistema se encuentra contenida la base de datos, donde se almacena y modifica toda la información. También dispone de un endpoint donde cualquier usuario puede mandar coordenadas para que se almacenen.

3.1.2. Casos de Uso

En esta sección se definen los distintos casos de uso de acuerdo a las características del sistema, mostrando un diagrama para cada uno de ellos. Además, se proporciona una tabla con información sobre cada caso de uso (nombre, descripción, actores, precondición).

3.1.2.1. Gestión de Coordenadas (Servidor REST)

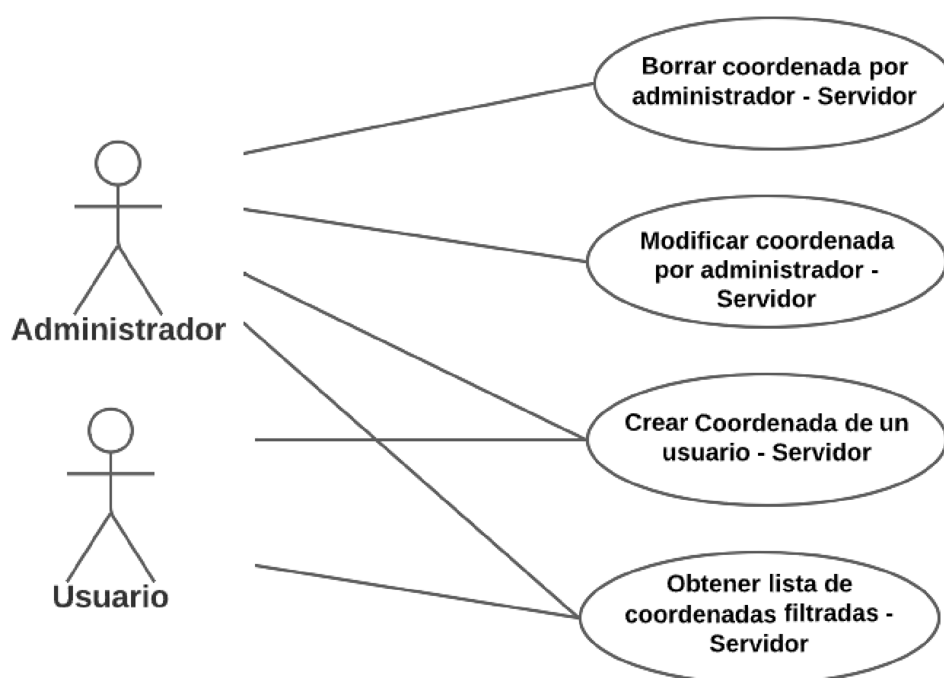

Ilustración 12. Diagrama de casos de uso gestión de coordenadas en el servidor

Borrar coordenada por administrador - Servidor

Descripción	El usuario deberá ser capaz de eliminar coordenadas proporcionando el identificador del mismo, o un grupo extenso proporcionando el identificador del edificio al que están vinculados.
Actor	Administrador
Precondición	Ninguna

Modificar coordenada por administrador - Servidor

Descripción	El usuario deberá ser capaz de modificar los atributos de una coordenada proporcionando su identificador y el resto de atributos que quiera modificar.
Actor	Administrador
Precondición	Ninguna

Crear coordenada de un usuario - Servidor

Descripción	El usuario deberá ser capaz de crear una coordenada proporcionando los atributos correspondientes con el formato correcto.
Actor	Administrador y usuario
Precondición	El usuario debe estar en posesión del listado de edificios para poder insertar el identificador correcto de cada edificio, y la planta donde se sitúa.

Obtener lista de coordenadas filtradas - Servidor

Descripción	El usuario deberá ser capaz de obtener una lista de coordenadas proporcionando el identificador del edificio y un rango de fechas.
Actor	Administrador y usuario
Precondición	El usuario debe estar en posesión del listado de edificios y mandar las fechas sin el *offset* horario del país donde se encuentre.

3.1.2.2. Gestión de Edificios (Servidor REST)

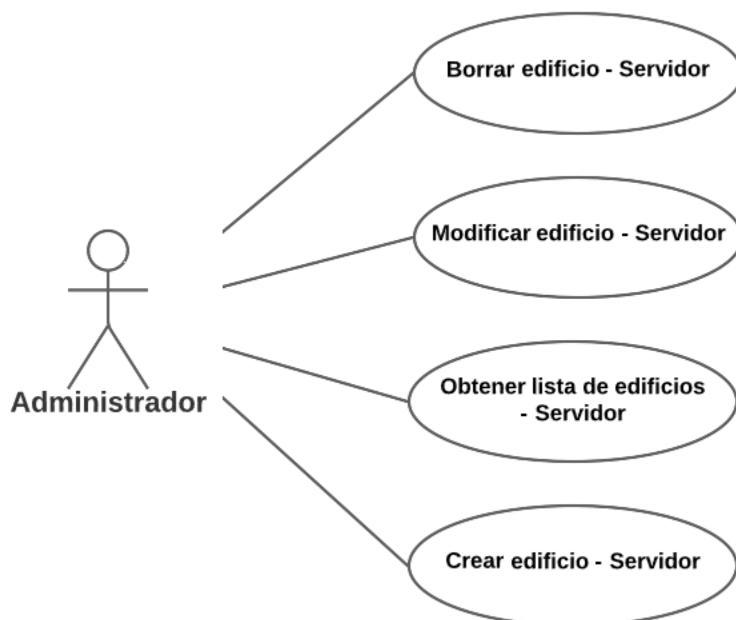

Ilustración 13. Diagrama de casos de uso gestión de edificios en el servidor

Borrar edificio - Servidor

Descripción	El administrador deberá ser capaz de eliminar un edificio proporcionando su identificador. Esta acción conlleva la eliminación de todas las coordenadas relacionadas con ese edificio.
Actor	Administrador
Precondición	Debe existir el edificio indicado en la base de datos

Modificar edificio - Servidor

Descripción	El administrador deberá ser capaz de modificar los atributos de un edificio proporcionando su identificador y el resto de atributos que quiera modificar.
Actor	Administrador
Precondición	Debe existir el edificio indicado en la base de datos

Crear edificio - Servidor

Descripción	El administrador deberá ser capaz de crear un edificio proporcionando los atributos correspondientes con el formato correcto.
Actor	Administrador

Precondición	Ninguna

Obtener lista de edificios - Servidor

Descripción	El administrador deberá ser capaz de obtener un listado completo de todos los edificios que se encuentran en la base de datos.
Actor	Administrador
Precondición	Debe existir como mínimo un edificio en la base de datos

3.1.2.3. Sistema de búsqueda (Aplicación web)

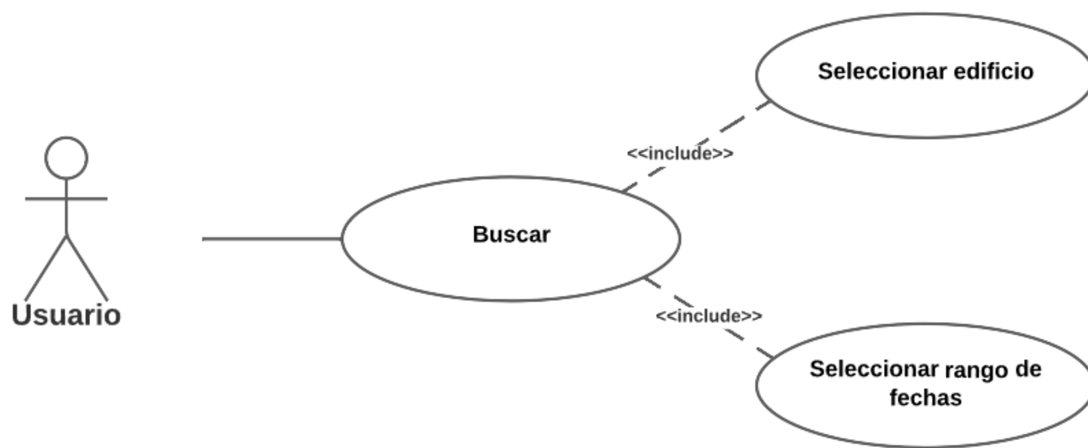

Ilustración 14. Diagrama de casos de uso sistema de búsqueda en el cliente web

Buscar - Cliente web

Descripción	El usuario deberá ser capaz de buscar todas las coordenadas que cumplan los parámetros establecidos en la búsqueda. En este caso sería el identificador del edificio y el rango de fechas.
Actor	Usuario
Precondición	Tener un edificio seleccionado y un rango de fechas válido

Seleccionar edificio - Cliente web

Descripción	El usuario deberá ser capaz de poder seleccionar el edificio que le interese incluir en la búsqueda. Cada vez que se seleccione un edificio, la aplicación centrará el mapa automáticamente en la posición del edificio.
Actor	Usuario

Precondición	Deben existir edificios en la base de datos

Seleccionar rango de fechas - Cliente web

Descripción	El usuario deberá ser capaz de poder seleccionar un rango de fechas que le interese incluir en la búsqueda. El calendario ofrecerá una variedad de rangos preseleccionados (hoy, ayer, última semana, último mes…) para poder agilizar el proceso de búsqueda.
Actor	Usuario
Precondición	Ninguna

3.1.2.4. Gestión de Edificios (Aplicación web)

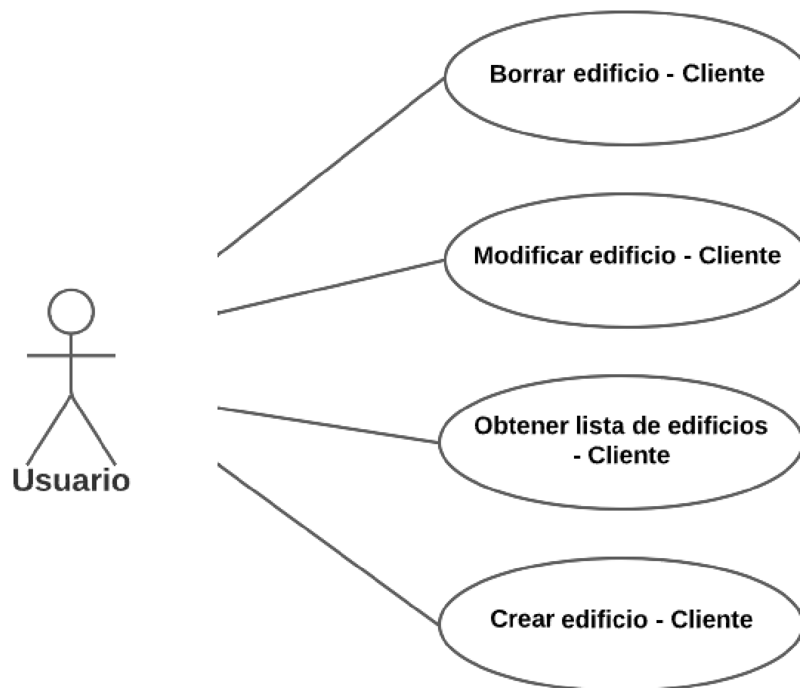

Ilustración 15. Diagrama de casos de uso gestión de edificios en el cliente web

Borrar edificio - Cliente web

Descripción	El usuario deberá ser capaz de poder eliminar un edificio, pero antes se le mostrará un mensaje de alerta donde tendrá que confirmar su acción.
Actor	Usuario
Precondición	Ninguna

Modificar edificio - Cliente web

Descripción	El usuario deberá ser capaz de modificar los datos de un edificio mediante un formulario, donde se mostrará el nombre, el número de plantas y un mapa donde se muestra la localización del edificio.
Actor	Usuario
Precondición	Debe existir un edificio en la base de datos, y el nombre no puede estar vacío.

Crear edificio - Cliente web

Descripción	El usuario deberá ser capaz de crear un nuevo edificio mediante un formulario donde podrá introducir el nombre, el número de plantas y la localización del edificio.
Actor	Usuario
Precondición	Ninguna

Obtener lista de edificios - Cliente web

Descripción	El usuario deberá ser capaz de obtener una lista actualizada de los edificios que se encuentran en el servidor, y mostrarla en un desplegable o una tabla.
Actor	Usuario
Precondición	Deben existir edificios en la base de datos

3.1.2.5. Ajustes del mapa (Aplicación web)

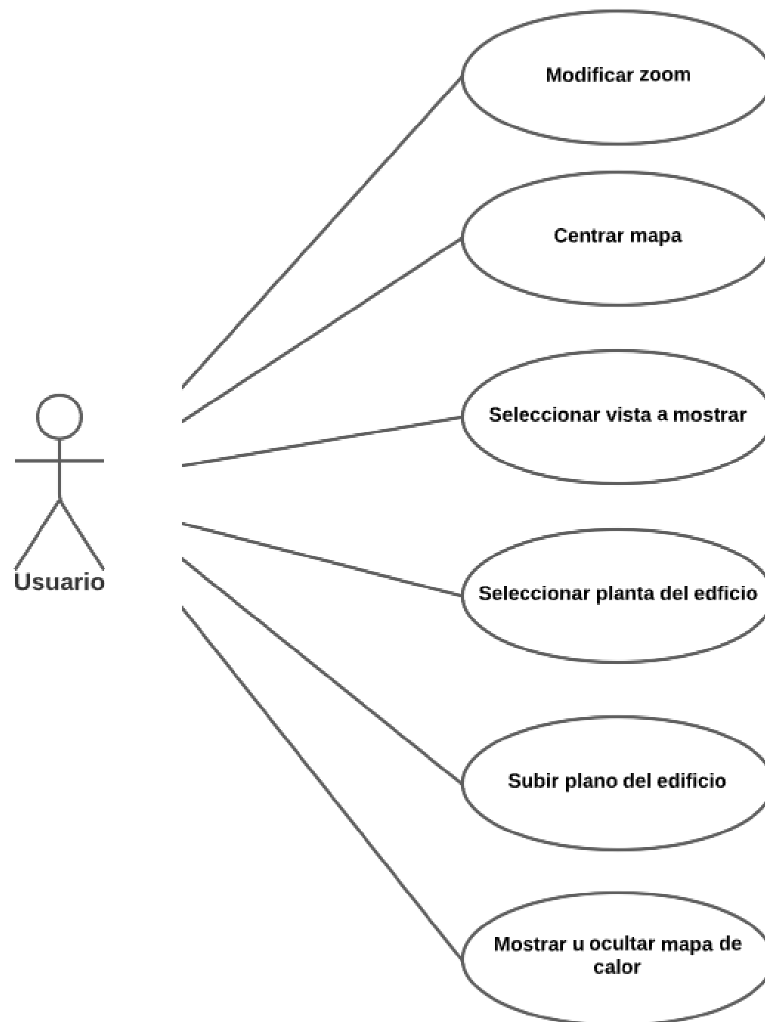

Ilustración 16. Diagrama de casos de uso ajustes del mapa en el cliente web

Modificar zoom - Cliente web

Descripción	El usuario deberá ser capaz de reducir o incrementar el zoom del mapa.
Actor	Usuario
Precondición	Ninguna

Centrar mapa - Cliente web

Descripción	El usuario deberá ser capaz de centrar el mapa en el edificio en el que está realizando la consulta.
Actor	Usuario

Precondición	Haber seleccionado un edificio

Seleccionar vista a mostrar - Cliente web

Descripción	El usuario deberá ser capaz de seleccionar el tipo de vista que quiere visualizar sobre el mapa.
Actor	Usuario
Precondición	Ninguna

Seleccionar planta del edificio - Cliente web

Descripción	El usuario deberá ser capaz de seleccionar la planta del edificio en la cual quiere ver el mapa de calor, y cambiar de forma rápida y cómoda entre ellas.
Actor	Usuario
Precondición	Haber seleccionado un edificio

Subir plano del edificio - Cliente web

Descripción	El usuario deberá ser capaz de subir una imagen con el plano de un edificio desde su dispositivo, y poder moverla, rotarla, ajustar su transparencia…
Actor	Usuario
Precondición	Ninguna

Mostrar u ocultar mapa de calor - Cliente web

Descripción	El usuario deberá ser capaz de activar y desactivar el mapa de calor, sin perder los resultados de su búsqueda.
Actor	Usuario
Precondición	Haber realizado una búsqueda

3.2. Análisis de Riesgos

En este apartado analizaremos las posibles amenazas y repercusiones que se puedan originar en este proyecto. Para ello, se han seguido los siguientes pasos:

- Identificar los posibles riesgos.
- Clasificarlos en: de aceptación, de satisfacción, y tecnológicos o de integración.
- Describir el impacto de cada uno.
- Describir las posibles soluciones para reducir el impacto.

3.2.1. Riesgos de aceptación

Título	El cliente no está satisfecho con el diseño de la interfaz gráfica.
Impacto	Retrabajo que implicaría un aumento en el tiempo de desarrollo.
Soluciones	Reuniones periódicas con el cliente antes y después del desarrollo para obtener su validación.

Título	El cliente no está satisfecho con el rendimiento del sistema.
Impacto	Aumento de gastos al tener que invertir en una mejor infraestructura. Aumento del tiempo de desarrollo y retrabajo. Estudiar y contemplar nuevas arquitecturas a las que migrar.
Soluciones	Identificar los límites del sistema en la fase de pruebas y realizar pruebas periódicas mientras se trabaja en tareas de optimización.

3.2.2. Riesgos de satisfacción

Título	El sistema es complicado de utilizar para los usuarios.
Impacto	Mayor tiempo de aprendizaje del usuario que puede llevar a la frustración y al abandono del sistema.
Soluciones	Realizar un manual de usuario. Realizar pruebas periódicas con usuarios en la fase de pruebas para obtener su opinión e identificar sus dificultades, para luego realizar los cambios necesarios.

Título	El sistema desarrollado resulta demasiado complejo a nivel de arquitectura y código.
Impacto	Mayor dificultad en la integración de nuevos programadores en el equipo. Descontento de los mismos desarrolladores, que puede llevar a un rendimiento

	más bajo, e incluso puede surgir la necesidad de desarrollar un sistema nuevo desde cero. Incremento del tiempo para añadir nuevas funcionalidades o corregir errores.
Soluciones	Crear una arquitectura simple y eficiente, realizando refactorizaciones periódicas en cada iteración a ser posible.

3.2.3. Riesgos tecnológicos y de integración

Título	Las tecnologías utilizadas dejan de recibir soporte.
Impacto	Se dejan de recibir actualizaciones de las librerías utilizadas. Alto riesgo a crear vulnerabilidades.
Soluciones	Elegir herramientas que tengan LTS (*Long Term Support*) ya que son estables y reciben actualizaciones durante varios años.

Título	El sistema desarrollado no es escalable.
Impacto	Se dejan de recibir actualizaciones de las librerías utilizadas. Alto riesgo a crear vulnerabilidades.
Soluciones	Elegir herramientas que tengan LTS (*Long Term Support*) ya que son estables y reciben actualizaciones durante varios años.

3.3. Identificación y análisis de soluciones posibles

En este apartado explicaremos la arquitectura, los lenguajes de programación y los IDE empleados para llevar a cabo el desarrollo de este sistema. Además, se analizarán las diferentes alternativas en la actualidad y por qué se decidió elegir una tecnología en concreto.

3.3.1. Análisis de arquitectura

Desde un primer momento se pretendía desarrollar el sistema con una arquitectura de cliente - servidor donde cada uno es completamente independiente del otro. Por tanto, cualquier cambio realizado en una parte del proyecto no se propagaría a la otra parte.

En esta arquitectura las tareas se reparten entre los servidores y los clientes, donde los servidores se encargan de recibir, procesar y enviar una respuesta, mientras que los clientes se encargan de enviar las solicitudes, recibir las respuestas y mostrarlas al usuario en una interfaz gráfica.

La comunicación entre estas partes del sistema se basa en REST (*representational state transfer*), un estilo de arquitectura *software* basado en la utilización del protocolo HTTP[8] para obtener o indicar la ejecución de operaciones sobre datos. Y destaca por lo siguiente:

- Un protocolo sin estado. Cada mensaje HTTP lleva toda la información necesaria para realizar la petición. Por tanto, ni el servidor ni el cliente necesitan recordar el historial de comunicaciones entre ambas.
- Operaciones bien definidas. A todos los recursos de información referenciados se les aplican operaciones bien definidas, las más importantes son POST, GET, PUT y DELETE.
- Sintaxis universal. Cada recurso es único a través de su URI definida.

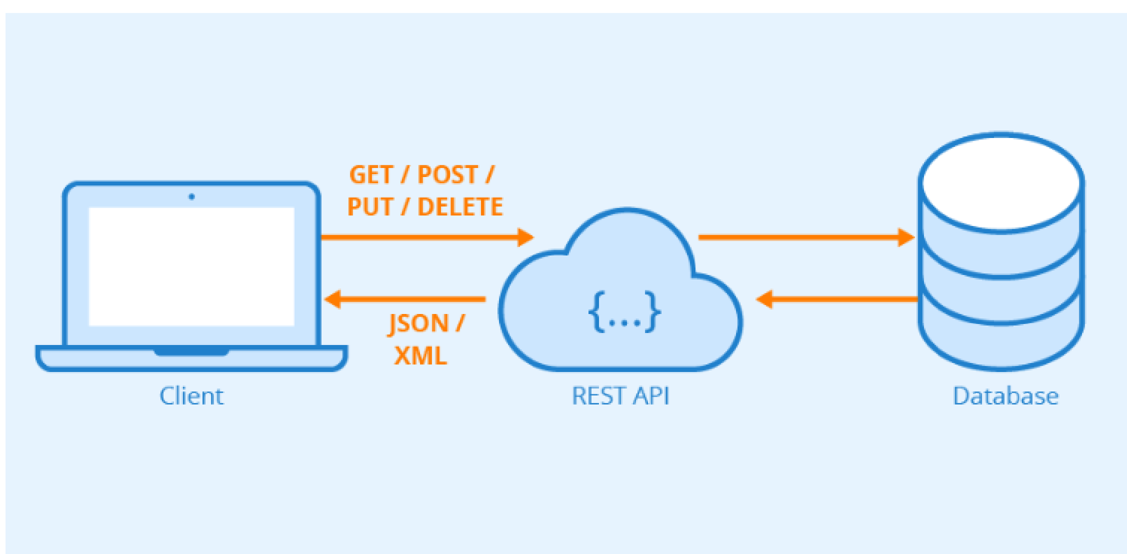

Ilustración 17. Arquitectura cliente - servidor con REST API

3.3.2. Análisis de tecnologías servidor

3.3.2.1. Frameworks de desarrollo

Hoy en día las tecnologías más relevantes para desarrollar un *backend* son los *frameworks* como Spring[9] (basado en Java) y Django[10] (basado en Python), y las librerías que nos ofrece Node.js[11] cómo Express. Cada una tienen sus ventajas y desventajas, como podemos ver en la siguiente tabla:

	Ventajas	Desventajas
Django	• Buena documentación. • Alta seguridad.	• Conocimiento previo en Python. • No utilizado anteriormente. • Es monolítico.
Spring	• Bastante completo por su gran cantidad de librerías. • Previo conocimiento en Java.	• Es complejo. • No utilizado anteriormente. • Requiere mucho XML.

[8] HTTP Methods: https://developer.mozilla.org/es/docs/Web/HTTP/Methods
[9] Spring Framework: https://spring.io/projects/spring-framework
[10] Django Framework overview: https://www.djangoproject.com/start/overview/
[11] About Node.js: https://nodejs.org/en/about/

	• Gran reutilización de código.	• Ofrece tantas posibilidades a los desarrolladores que puede resultar. confuso.
Express (Node.js)	• Previo conocimiento en Javascript y Node.js. • Utilizado anteriormente. • Alta velocidad de desarrollo. • Un único hilo de ejecución pero altamente escalable. • Sin *buffering* y muy rápido.	• No es adecuado para aplicaciones con cargas elevadas. • Menos seguro. • La programación asíncrona puede ser compleja.

Tabla 1. Comparativa de los diferentes frameworks para el servidor

Para este proyecto se descartó utilizar en primer lugar Django, ya que no se adaptaba a nuestras necesidades, y el desconocimiento del lenguaje Python habría incrementado el tiempo de desarrollo. Esta decisión nos dejaba con el framework Spring y el entorno de ejecución NodeJS.

Finalmente decidí utilizar NodeJS ya que lo había utilizado anteriormente en la carrera y conocía de primera mano las capacidades de dicho entorno, y el proceso de desarrollo sería muchísimo más rápido en comparación con aprender Spring desde cero.

3.3.2.2. Base de datos

En primer lugar habría que diferenciar entre los dos tipos de bases de datos que existen en la actualidad: las relacionales y las no relacionales. En la siguiente tabla podemos observar las ventajas y desventajas de cada una:

	Ventajas	Desventajas
Relacional	• Perfecta para datos estructurados. • Utiliza el lenguaje SQL. • Perfecta para *queries* complejas. • Alto nivel de integración, debido a sus relaciones y restricciones entre tablas. • Transacciones con alta seguridad. • Gran fiabilidad.	• El esquema debe estar bien definido previamente. • Es complicado hacer cambios al modelo de datos. • El procesamiento de datos puede ser lento.
No Relacional	• Modelo de datos flexible. • Los cambios de un objeto no afectan al resto. • Alta capacidad de almacenamiento de objetos con una estructura sencilla. • Alto rendimiento.	• Poca fiabilidad • Las *queries* son manuales. • Poca integridad y consistencia de los datos.

Tabla 2. Ventajas y desventajas de las bases de datos relacionales y no relacionales

En la siguiente tabla podemos ver de forma resumida las características no funcionales de cada uno:

Característica	Relacionales	No Relacionales
Disponibilidad	Buena	Buena
Consistencia	Buena	Mala
Almacenamiento	Buena	Buena
Rendimiento	Mala	Buena
Fiabilidad	Buena	Mala
Escalabilidad	Buena	Buena

Tabla 3. Comparativa según sus características de las bases de datos relacionales y no relacionales

Tras realizar este análisis se tomó la decisión de utilizar una base de datos relacional, ya que nos ofrece una mayor fiabilidad y consistencia para el modelo de datos, y nos abre las puertas a poder escalar el sistema en un futuro si fuera necesario.

Dentro de las bases de datos relacionales podemos encontrar las más importantes como PostgreSQL[12] y MySQL[13]. Y se tomó la decisión de utilizar PostgreSQL ya que nos ofrece una mayor integridad y fiabilidad de datos, es decir, podemos estar seguros de almacenar datos sin perderlos.

3.3.2.3. Entornos de desarrollo integrado

Para el desarrollo del *backend* hemos utilizado Visual Studio Code[14] como entorno de desarrollo, debido a que es muy ligero y dispone de una gran cantidad de extensiones que podemos instalar para agilizar y facilitar el desarrollo.

A la hora de administrar la base de datos hemos utilizado pgAdmin4[15], una herramienta que viene justo a PostgreSQL, la cual nos ofrece una interfaz gráfica donde realizar todo tipo de operaciones y búsquedas.

3.3.3. Análisis de tecnologías web

3.3.3.1. Frameworks de desarrollo

Hoy en día existen una gran cantidad de librerías y *frameworks* de desarrollo web, entre los más importantes podemos encontrar Angular[16], React[17] y Vue.js[18]. Hemos recopilado y reflejado las ventajas y desventajas de cada una en la siguiente tabla:

---

[12] About PostgreSQL: https://www.postgresql.org/about/
[13] Why MySQL?: https://www.mysql.com/why-mysql/
[14] About Visual Studio Code: https://code.visualstudio.com/docs/editor/whyvscode
[15] pgAdmin Introduction: https://www.pgadmin.org/
[16] Introduction to Angular concepts: https://angular.io/guide/architecture
[17] Getting started with React: https://reactjs.org/docs/getting-started.html
[18] Vue.js introduction: https://vuejs.org/guide/introduction.html

	Ventajas	Desventajas
Angular	● Gran cantidad de librerías	● Pesado ● Mayor curva de aprendizaje ● Conocimientos de Typescript ● No es muy flexible
React	● Bastante ligero ● Es muy flexible	● Utiliza la licencia de Facebook
Vue.js	● Bastante ligero ● Fácil de aprender y utilizar ● Es algo flexible ● Es de código libre ● Extensa documentación	● Escasez de foros y páginas donde se resuelvan dudas importantes.

Tabla 4. Comparativa de los frameworks de desarrollo para el cliente web

Para este proyecto se decidió finalmente utilizar Vue.js ya que nos ofrece todas las herramientas adecuadas para desarrollar una aplicación ligera, y de una forma sencilla y rápida.

3.3.3.2. Frameworks CSS

Para manejar el CSS de la aplicación web se tomó la decisión de utilizar un *framework* para agilizar y facilitar el desarrollo, y para ello realizamos una comparación entre los más famosos del mercado ahora mismo, que son Bootstrap[19] y Tailwind CSS[20]:

	Ventajas	Desventajas
Bootstrap	● Viene con plantillas y componentes incorporados. ● Páginas web totalmente responsive y con un diseño impecable.	● No es muy flexible. ● Es más pesado. ● Baja personalización. Todas las páginas web son similares entre ellas.
Tailwind CSS	● Es bastante flexible. ● Es más ligero. ● Muy personalizable. Gran número de clases de utilidad.	● No tiene plantillas o componentes incorporados. ● En proyectos grandes puede provocar que tu código sea difícil de leer.

Tabla 5. Comparativa de los frameworks CSS para el cliente web

[19] Getting started with Bootstrap: https://getbootstrap.com/docs/4.0/getting-started/introduction/
[20] Get started with Tailwind CSS: https://tailwindcss.com/docs/installation

Teniendo en consideración todos estos aspectos se tomó la decisión de utilizar Tailwind CSS, ya que al tratarse de una aplicación de pequeña envergadura y ligera, este *framework* satisface con creces nuestras necesidades.

3.3.3.3. Entornos de desarrollo integrado

Para el desarrollo del *frontend* hemos utilizado Visual Studio Code como entorno de desarrollo, debido a que es muy ligero y dispone de una gran cantidad de extensiones que podemos instalar para agilizar y facilitar el desarrollo.

A la hora de testear la aplicación hemos utilizado los navegadores web Google Chrome[21], y Mozilla Firefox[22], ya que ambos disponen de herramientas para desarrolladores como el inspector de elementos, la consola, y el análisis de tráfico en la red. Y también se han realizado pruebas periódicas en otros navegadores como Microsoft Edge[23], Opera[24] y Safari[25].

[21] Google Chrome introduction: https://www.google.com/intl/en/chrome/browser-features/#
[22] Mozilla Firefox features: https://www.mozilla.org/es-ES/firefox/features/
[23] Microsoft Edge features: https://www.microsoft.com/es-es/edge/features
[24] About Opera: https://www.opera.com/es/about
[25] Página oficial Safari: https://www.apple.com/es/safari/

4. Diseño de la Solución

En este apartado se va a realizar un diseño de alto nivel, es decir, vamos a definir la arquitectura general del sistema en base a los requisitos y el análisis detallados anteriormente.

4.1. Arquitectura del Sistema

Tal y como se ha descrito en el apartado anterior, la arquitectura global del sistema se basa en una de tipo cliente - servidor, donde en la parte del cliente encontramos dos tipos totalmente distintos:

- Clientes Web. Son los clientes que utilizan la aplicación web para visualizar los mapas de calor y utilizar el gestor de edificios.
- Clientes Android. Son los clientes que envían datos geográficos al servidor de forma periódica. Estos no forman parte del sistema a desarrollar.

Para ello el servidor *backend* se sitúa en medio de ellos, actuando como un servidor totalmente independiente y comunicándose con ambas partes mediante protocolo HTTP y enviando las respuestas en formato JSON[26].

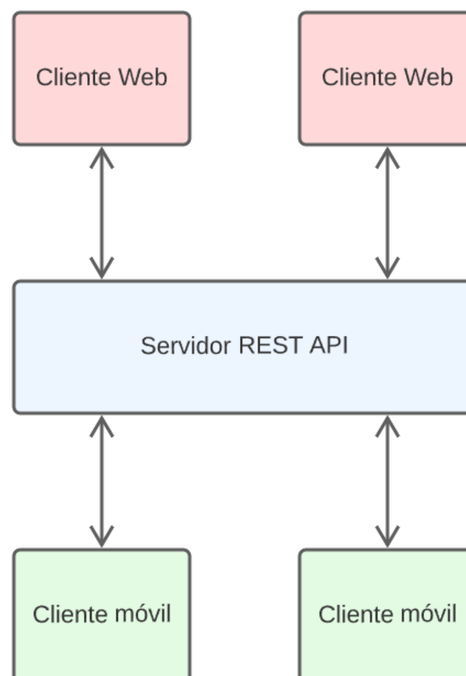

Ilustración 18. Arquitectura del sistema a desarrollar

[26] Introducing JSON: https://www.json.org/json-en.html

4.2. Diseño de Componentes

En este apartado explicaremos en detalle las características de cada componente que forma parte del sistema en este proyecto. Cada uno de estos es totalmente independiente del otro, y pueden existir múltiples instancias de cada uno, brindando la oportunidad de escalar el sistema.

4.2.1. Servidor

4.2.1.1. Arquitectura REST API Server

Para este sistema se ha optado por utilizar una arquitectura vertical basada en capas, donde cada una de ellas cumple una función y la comunicación se realiza de forma directa con la capa superior o inferior.

Esta estructura vertical está divida en Rutas, donde se definen los distintos *endpoints* del servidor, en el Controlador, donde se ejecutan las operaciones a realizar, y en la capa de Repositorio, donde se realiza toda la lógica de acceso a datos.

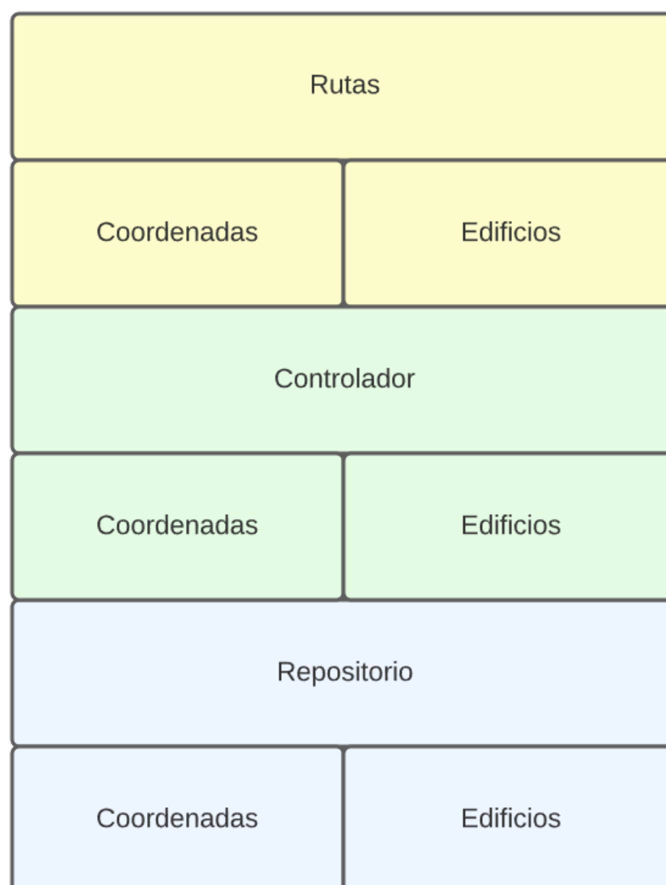

Ilustración 19. Arquitectura del servidor API REST

4.2.1.2. *Base de datos*

A continuación se muestra el modelo de datos ubicado en la base de datos.

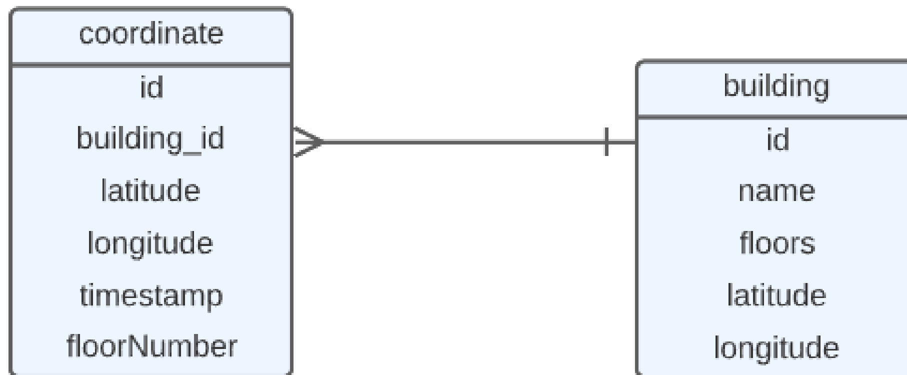

Ilustración 20 Modelo de datos del sistema

Describiremos brevemente cada una de las tablas y las relaciones que figuran en el anterior diagrama.

- coordinate: Esta tabla se utiliza para almacenar puntos geográficos en un edificio, planta, día y hora determinado. El campo id es utilizado como clave primaria, y el campo building_id actúa como una clave foránea referenciando a la tabla building. El resto de campos tienen la restricción de *NOT NULL* para evitar tener objetos con atributos vacíos.

- building: Esta tabla se utiliza para almacenar los datos de todos los edificios que se van a introducir en el sistema. El campo id es utilizado como clave primaria y se referencia con la tabla coordinate. El resto de campos tienen la restricción de *NOT NULL* para evitar tener objetos con atributos vacíos.

4.2.2. Cliente web

4.2.2.1. *Arquitectura Web*

Al utilizar el *framework* de desarrollo web Vue.JS tenemos que adaptarnos a la arquitectura que nos ofrece. Para usuarios que no han utilizado este tipo de tecnologías puede parecer un poco confuso, pero en este apartado intentaremos detallar brevemente las características más importantes del mismo.

La idea principal de esta arquitectura es utilizar vistas que se componen de componentes, estas vistas están vinculadas por un Router que se encarga de redireccionar al usuario a la URL correspondiente. En otras palabras, cada vista es una interfaz completa, y se construye con componentes que cumplen unas funciones determinadas y se comunican entre ellos.

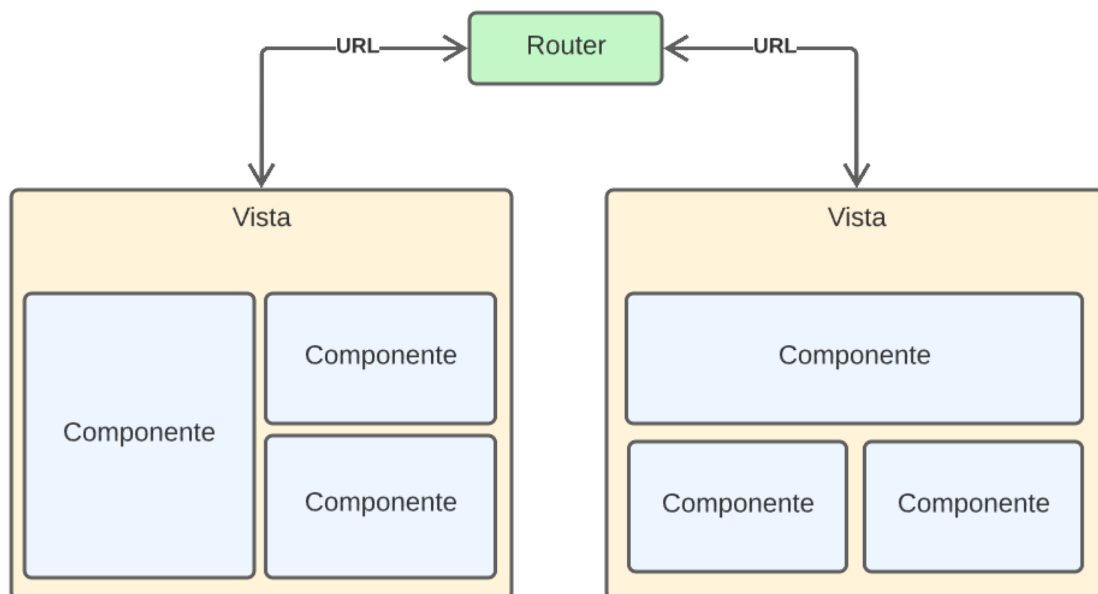

Ilustración 21. Arquitectura del cliente web

4.2.2.2. *Wireframes de las interfaces*

A continuación mostraremos los wireframes diseñados para cada una de las interfaces que componen la aplicación.

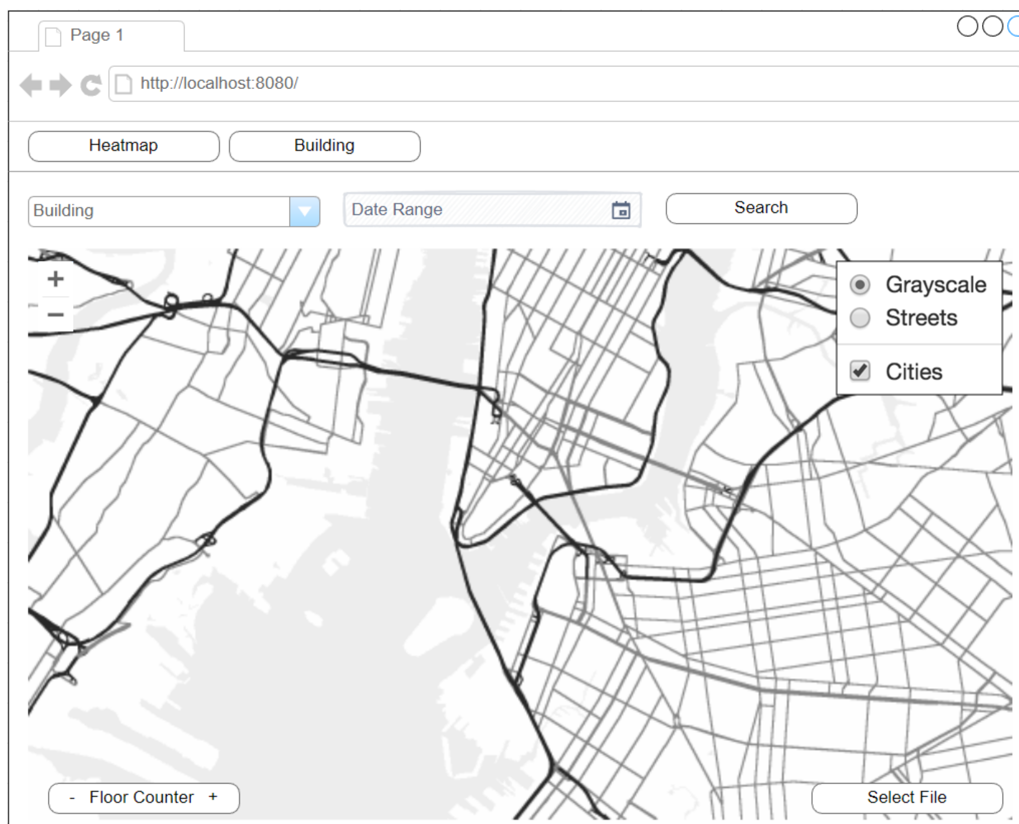

Ilustración 22. Wireframe de la interfaz de heatmaps

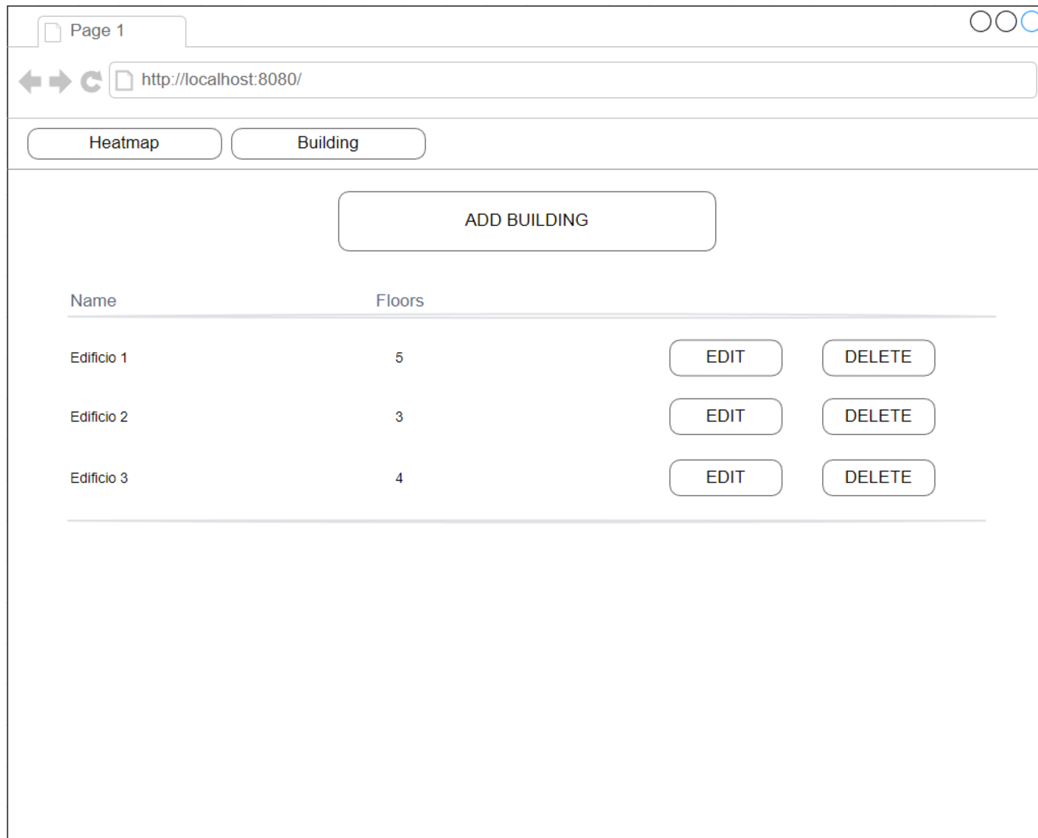

Ilustración 23. Wireframe de la interfaz de los edificios

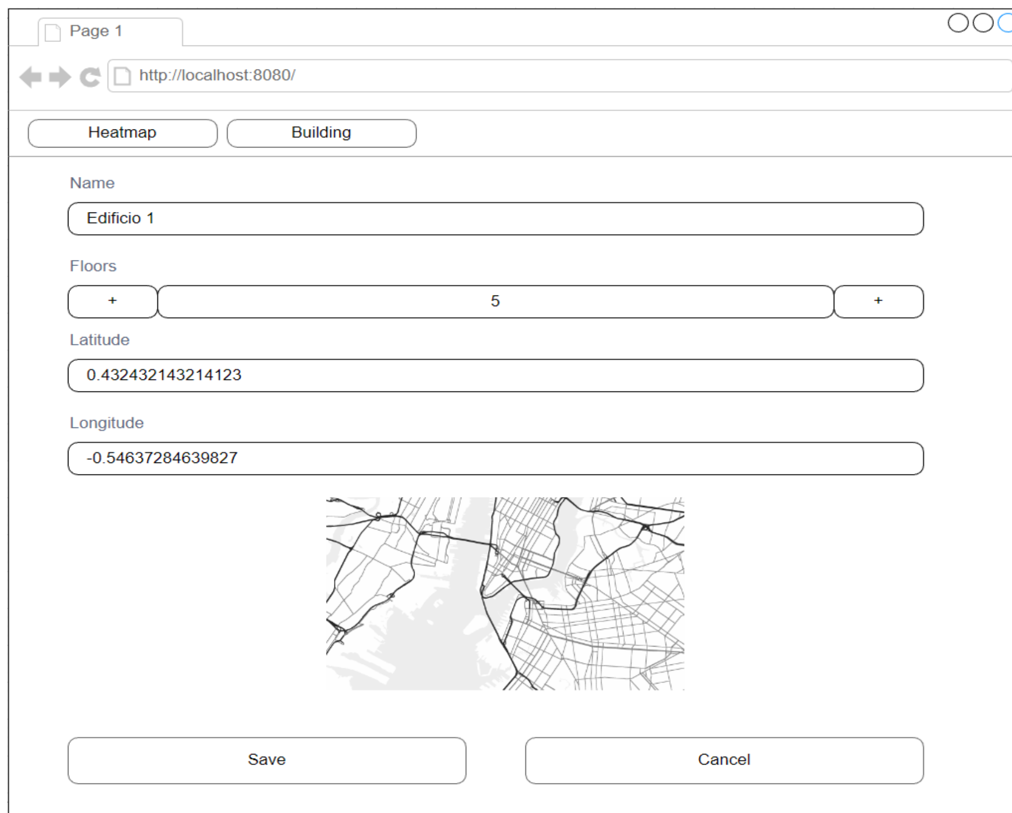

Ilustración 24. Wireframe del editor de edificios

5. Desarrollo de la Solución

En este apartado explicaremos cómo se ha realizado el desarrollo y la implementación de la solución propuesta para este sistema. Para ello se presentará cada parte del proyecto de forma separada, y se hará una exploración de los ficheros que los componen, dando énfasis en las partes más importantes.

Para finalizar veremos las dificultades encontradas durante el proceso de desarrollo, y las soluciones implementadas para conseguir un resultado satisfactorio.

5.1. Desarrollo del servidor

Para empezar hablaremos sobre el *backend* que se encuentra en el servidor. Como hemos podido ver anteriormente en la fase de diseño, este está dividido en tres capas, pero antes analizaremos la estructura general de los ficheros.

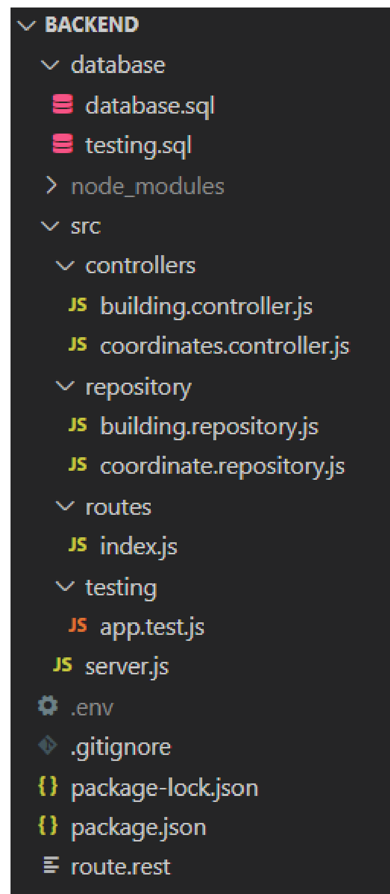

Ilustración 25. Estructura general de los ficheros en el backend

Empezando desde arriba del todo, dentro de la carpeta *database* encontraremos las sentencias SQL para poder crear la base de datos con sus respectivas tablas.

Seguidamente encontraremos la carpeta *node_modules*, la cual podemos ignorar, ya que contiene las librerías que hemos importado utilizando Node.js y no se deben modificar.

A continuación, tenemos la carpeta más importante del proyecto, la cual hemos llamado *src* ya que va a contener los ficheros con todo el código fuente de la aplicación. Cabe destacar el archivo *server.js, el cual se* ejecutará al iniciar la aplicación y establecerá la conexión entre el puerto y las rutas utilizadas para las llamadas a la API.

```
server.js
src > server.js > ...
1  const express = require('express');
2  const cors = require('cors');
3  const app = express().use(cors());
4
5  // middlewares
6  app.use(express.json());
7  app.use(express.urlencoded({ extended: false }));
8
9  // routes
10 app.use(require('./routes/index'));
11
12 app.listen(3000);
13 console.log('Server on port 3000');
```

Ilustración 26. Contenidos del fichero server.js

Además, hemos creado una carpeta *testing* donde poner todos los ficheros relacionados con el testeo de la aplicación, pero entraremos más en detalle en el apartado siguiente de pruebas.

Finalmente, podemos encontrar unos ficheros sueltos que son esenciales para el funcionamiento de la aplicación, que explicaremos a continuación:

- .env: Este fichero contiene las variables del entorno que vamos a utilizar para hacer la conexión a la base de datos. Su color grisáceo indica que no forma parte de *gitignore* por tanto no se subirá al repositorio de Github, esto nos proporciona seguridad, ya que no queremos dar nuestras credenciales a otro usuario que pueda descargar nuestro código.

```
.env
.env
1 BD_HOST=localhost
2 BD_USER=postgres
3 BD_PASSWORD=admin
4 BD_DATABASE=tfg
5 BD_PORT=5432
```

Ilustración 27. Contenidos del fichero .env

- .gitignore: Este fichero contiene el nombre de los archivos que no queremos subir al repositorio de Github, ya sea por seguridad o por evitar subir archivos innecesarios que el usuario podría descargarse de forma local al clonar el repositorio. Este último caso se podría aplicar a la carpeta *node_modules* que contiene miles de archivos, que podrían ralentizar el proceso de sincronización con el repositorio.

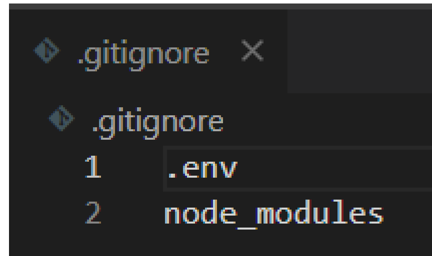

Ilustración 28. Contenidos del fichero .gitignore

- package-lock.json: Es un fichero que almacena el árbol de dependencias de los paquetes instalados en el proyecto. En este proyecto no ofrece ninguna ventaja, pero en un entorno colaborativo permitiría a otros usuarios descargarse las mismas versiones de las librerías.

- package.json: Es un fichero que contiene información acerca del proyecto y los paquetes que lo componen.

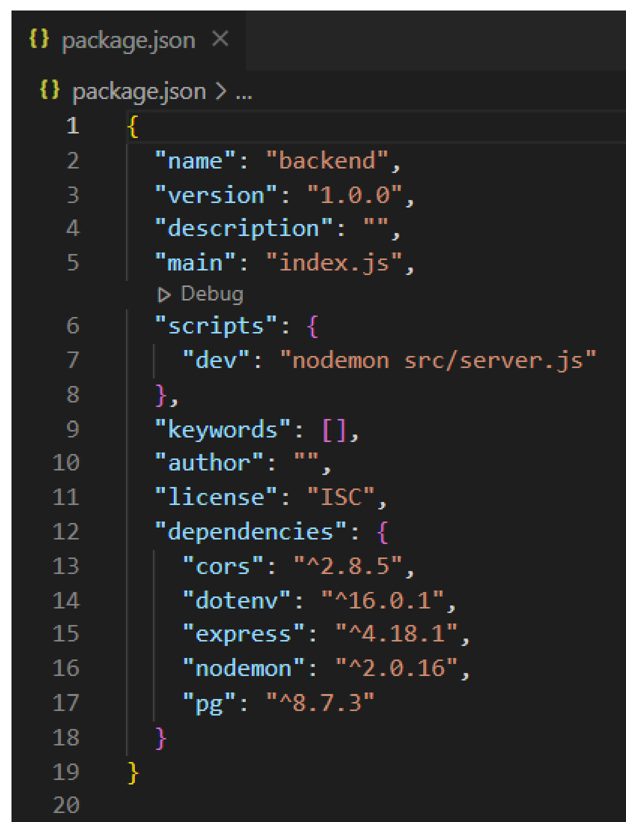

Ilustración 29. Contenidos del fichero package-json

- routes.rest: Este fichero contiene peticiones HTTP para poder testear las llamadas a la API durante el proceso de desarrollo, sin tener que diseñar pruebas unitarias para los mismos. Esto es gracias a una extensión de Visual Studio Code llamada REST Client.

```
route.rest
route.rest > GET /coordinates/?building=1&startDate=2022-09-11%2019:
15
16  ###
17
    Send Request
18  GET http://localhost:3000/buildings
19
20  ###
    Send Request
21  POST http://localhost:3000/buildings
22  Content-Type: application/json
23
24  {
25      "name": "Test Building",
26      "floors": 1,
27      "latitude": 39.48067057885303,
28      "longitude": -0.3398773008264785
29  }
30
31  ###
32
    Send Request
33  PUT http://localhost:3000/buildings/2
34  Content-Type: application/json
35
36  {
37      "name": "ETSINF - EDIFICIO 1E",
38      "latitude": 39.482716712422764,
39      "longitude": -0.346685476430633
40  }
41
42  ###
43
    Send Request
44  DELETE http://localhost:3000/buildings/1
```

```
Response(20ms)
1   HTTP/1.1 200 OK
2   X-Powered-By: Express
3   Access-Control-Allow-Origin: *
4   Content-Type: application/json; charset=utf-8
5   Content-Length: 421
6   ETag: W/"1a5-dW3iZilNbDcue/Nff058aS3Ib6w"
7   Date: Tue, 23 Aug 2022 19:13:33 GMT
8   Connection: close
9
10  [
11    {
12      "id": 1,
13      "name": "ETSINF - Edificio 1G",
14      "floors": 20,
15      "latitude": 39.4827129135827,
16      "longitude": -0.346745252609253
17    },
18    {
19      "id": 2,
20      "name": "ETSINF - Edificio 1E",
21      "floors": 4,
22      "latitude": 39.4827087727874,
23      "longitude": -0.34829556738385
24    },
25    {
26      "id": 3,
27      "name": "ETSINF - Edificio 1H",
28      "floors": 5,
29      "latitude": 39.4829240709159,
30      "longitude": -0.347595512866974
31    },
```

Ilustración 30. Ejemplo de una llamada GET en el fichero route.rest

5.1.1. Rutas

El fichero donde podemos encontrar las diferentes rutas con las que podemos acceder a la API se encuentra en el directorio '*src/router/index.js*'.

```
index.js
src > routes > index.js > ...
1   const { Router } = require('express');
2   const router = Router();
3
4   // Import coordinates controller
5   const { getCoordinates, createCoordinate } = require('../controllers/coordinates.controller');
6   // Import buildings controller
7   const { getBuildings, updateBuilding, createBuilding, deleteBuilding } = require('../controllers/building.controller');
8
9   // Coordinates
10  router.get('/coordinates', getCoordinates);
11  router.post('/coordinates', createCoordinate);
12
13  // Buildings
14  router.get('/buildings', getBuildings);
15  router.post('/buildings', createBuilding);
16  router.put('/buildings/:id', updateBuilding);
17  router.delete('/buildings/:id', deleteBuilding);
18
19  module.exports = router;
```

Ilustración 31. Contenidos del fichero src/router/index.js

Hay que tener en cuenta que todas las llamadas se componen de un método HTTP, acompañadas de una URL base http://localhost:3000/, una URL con la ruta correspondiente al recurso que queremos acceder, y una serie de atributos en la cabecera o en el cuerpo de la petición.

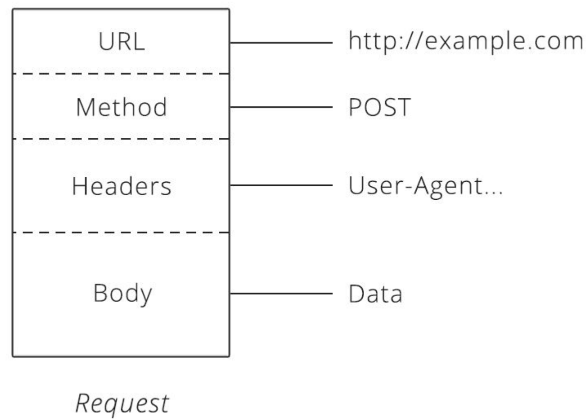

Ilustración 32. Estructura de una petición HTTP a una API

En la siguiente tabla podemos ver las diferentes llamadas que podemos realizar a nuestro servidor:

Método HTTP	URL	Atributos en la cabecera	Atributos en el cuerpo	Resultado
GET	/coordinates	building startDate endDate		Lista con coordenadas
POST	/coordinates		building latitude longitude timestamp floorNumber	Crear una nueva coordenada
GET	/buildings			Obtener la lista de edificios
POST	/buildings		name floors latitude longitude	Crear un nuevo edificio
PUT	/buildings	id	name floors latitude longitude	Actualizar un edificio
DELETE	/buildings	id		Borrar un edificio

Tabla 6. Todas las llamada API del servidor

5.1.2. Controladores

Para cada objeto que hemos modelado en la base de datos se ha creado un controlador, que tendrá la función de realizar diferentes operaciones sobre los mismos.

Estos ficheros los podemos encontrar en *'src/controllers/building.controller.js'* y *'src/controllers/coordinates.controller.js'*.

En la parte superior importamos las librerías necesarias y realizamos la conexión a la base de datos, proporcionando los credenciales con las variables del entorno que hemos definido previamente en el archivo *.env*

```
const { Pool } = require('pg');
require('dotenv').config();

const pool = new Pool({
    host: process.env.BD_HOST,
    user: process.env.BD_USER,
    password: process.env.BD_PASSWORD,
    database: process.env.BD_DATABASE,
    port: process.env.BD_PORT
});
```

Ilustración 33. Conexión a la base de datos en el controlador

En la parte de en medio tendremos los métodos que se encargaran de procesar las solicitudes de forma asíncrona, y devolviendo una respuesta tras ejecutarse.

```
const getBuildings = async (req, res) => {
    const response = await pool.query(
        'SELECT * FROM building ORDER BY id ASC');
    res.status(200).json(response.rows);
}

const createBuilding = async (req, res) => {…
}

const updateBuilding = async(req, res) => {…
}

const deleteBuilding = async (req, res) => {…
}
```

Ilustración 34. Métodos asíncronos en el controlador

Finalmente, en la parte inferior, exportamos estos métodos para que puedan ser utilizados por la capa de Rutas, donde cada URL será redireccionada a un método en concreto del controlador correspondiente.

5.1.3. Repositorio y base de datos

Al inicio del proceso de desarrollo era necesario crear y poblar la base de datos de forma manual, para ello se han creado unos ficheros en *'src/database'* para ese propósito.

```
CREATE DATABASE geodatabase;

CREATE TABLE coordinates(
    id SERIAL PRIMARY KEY,
    building integer,
    floornumber integer,
    latitude float,
    longitude float,
    timedate TIMESTAMP
);

CREATE TABLE building(
    id SERIAL PRIMARY KEY,
    name TEXT,
    floors integer,
    latitude float,
    longitude float
);
```

Ilustración 35. Contenidos del fichero src/database/database.sql

Más adelante, para realizar cambios sobre la base de datos y realizar consultas se utilizó pgAdmin4, una aplicación que venía junto a la instalación de PostgreSQL, que nos ofrece una interfaz gráfica con una gran cantidad de información y operaciones que podemos realizar.

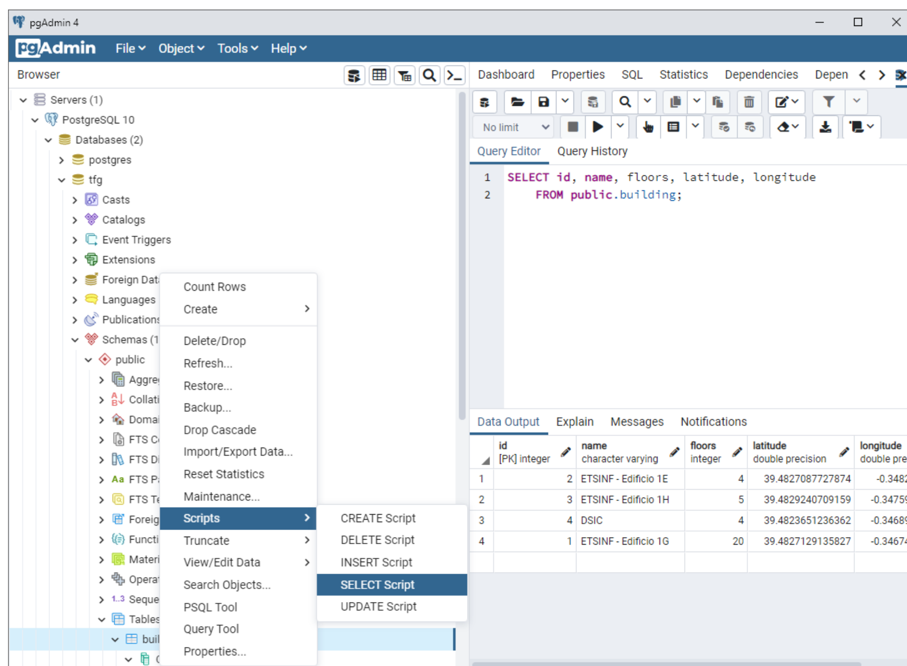

Ilustración 36. Ejemplo de cómo realizar una query SELECT sobre una tabla en pgAdmin4

Dentro del servidor, todo el código que actúa de intermediario entre la base de datos y el controlador se encuentra en la carpeta "*src/repository*", donde cada tipo de dato tiene su fichero especializado y contiene las queries necesarias para el funcionamiento de la aplicación.

```
const getAllBuildingsDB = async () => {
    try {
        const response = await pool.query(
            'SELECT * FROM building ORDER BY id ASC');
        return response.rows;
    } catch(err) {
        throw err;
    }

}
```

Ilustración 37. Ejemplo de un método en la capa Repositorio para obtener la lista de edificios

5.2. Desarrollo de la aplicación web

La aplicación web se ha desarrollado con el *framework* de Vue.js, por tanto nos hemos tenido que adaptar a su arquitectura basada en la construcción de interfaces de usuario mediante vistas compuestas por componentes, las cuales explicaremos en los siguientes apartados.

La estructura del proyecto es la siguiente:

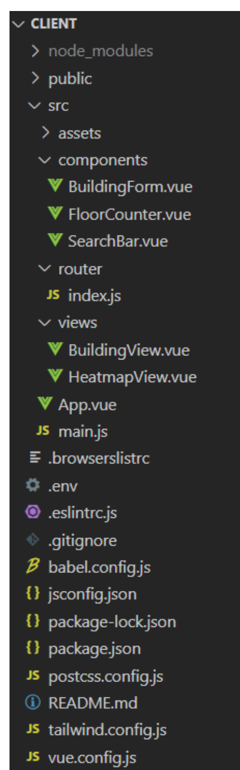

Ilustración 38. Estructura de los ficheros en el proyecto del cliente web

5.2.1. Componentes

Los componentes nos permiten dividir la interfaz en piezas totalmente independientes y reutilizables. Esto nos permite dividir la funcionalidad de la aplicación entre fragmentos muy pequeños, y esto nos proporciona muchas ventajas en el proceso de desarrollo y mantenimiento de software.

Una de las características más importantes que nos ofrece es la posibilidad de reutilizar componentes. Esto lo hemos implementado en distintos lugares de la aplicación, y un claro ejemplo lo podemos ver en el selector de pisos de los edificios.

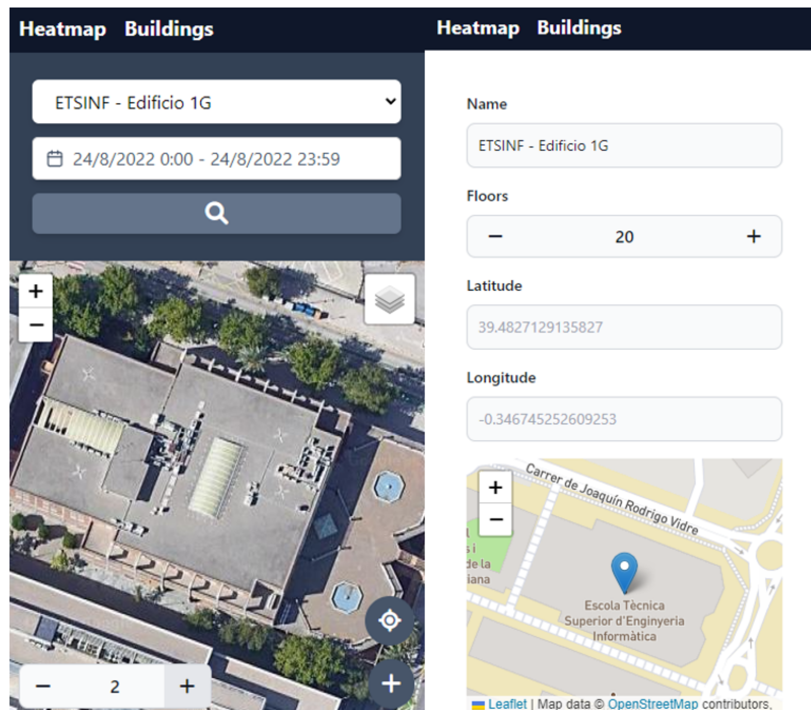

Ilustración 39. Selector de pisos siendo reutilizado sobre el mapa y en la ficha de los edificios

```
<!-- Floor Counter -->
<div class="fixed left-3 bottom-10 sm:b
  <FloorCounter v-if="currentBuilding"
  :floors="currentBuilding.floors"
  :fromBuildingTable="false"
  :key="currentBuilding"
  @onFloorUpdated="updateFloor" />
</div>
```

```
<!-- Floors Field -->
<div>
  <label for="floors" class="block mb-2
  <FloorCounter
    :floors="building.floors"
    :fromBuildingTable="true"
    @onFloorUpdated="updateFloor" />
</div>
```

Ilustración 40. Declaración del selector de pisos en el mapa y en la ficha de los edificios

Para que la reutilización de componentes cobre sentido, es necesario que exista una comunicación entre los componentes padre e hijos. Por tanto, Vue.js nos da la posibilidad de enviar parámetros desde el fragmento padre mediante nombres precedidos del símbolo : y recibir información con nombres precedidos del símbolo @, tal y como podemos ver en las imágenes anteriores.

5.2.2. Vistas

Las vistas en Vue.js son un tipo de componente que representan una página entera, y se utilizan como plantilla para distribuir los componentes y gestionar el intercambio de información entre ellos.

En nuestra aplicación solo nos ha hecho falta implementar dos vistas en toda la aplicación, una para visualizar el mapa y otra para gestionar los edificios.

5.2.3. Router

Cuando hacemos *click* en un enlace en una aplicación web tradicional, el navegador recibe una respuesta HTML desde el servidor web y refresca la página entera con el nuevo HTML.

En una aplicación de una sola página (SPA Single-Page Application) el propio cliente Javascript puede interceptar la navegación, cargar la información de forma dinámica y actualizar la página sin tener que hacer un refresco completo. Esto resulta en una experiencia mucho más fluida para el usuario y da la sensación de que se está utilizando una aplicación de verdad.

Es por ello que Vue.js nos ofrece una librería llamada Router para poder manejar esta navegación, en nuestra aplicación la hemos utilizado de la siguiente manera:

```
JS index.js ×
src > router > JS index.js > ...
1  import { createRouter, createWebHistory } from 'vue-router'
2  import HeatmapView from '../views/HeatmapView.vue'
3
4  const routes = [
5    {
6      path: '/',
7      name: 'heatmap',
8      component: HeatmapView
9    },
10   {
11     path: '/building',
12     name: 'building',
13     component: () => import('../views/BuildingView.vue')
14   }
15 ]
16
17 const router = createRouter({
18   history: createWebHistory(process.env.BASE_URL),
19   routes
20 })
21
22 export default router
```

Ilustración 41. Construcción del Router en nuestra aplicación web

5.2.4. Mapa Leaflet

Leaflet[27] es una librería de código abierto para la construcción de mapas compatibles para cualquier plataforma de escritorio o móvil. Destaca por simplicidad, rendimiento y usabilidad.

Además, esta librería puede ser extendida con una gran cantidad de *plugins*, dispone de una documentación extensa y entendible, y un código fuente estructurado que permite una fácil colaboración de cientos de programadores de todo el mundo.

Cabe mencionar que esta librería solo nos proporciona los métodos y funcionalidades para hacer funcionar el mapa, ya que la parte visual tenemos que importarla desde otras fuentes.

---

[27] Leaflet overview: https://leafletjs.com/

En nuestra aplicación hemos utilizado los recursos visuales que nos ofrece OpenStreetMap, un mapa de uso libre bajo licencia abierta, y las imágenes de Google Maps (OJO: No es lo mismo utilizar el API de Google Maps que es de pago, que utilizar solo las imágenes de sus mapas).

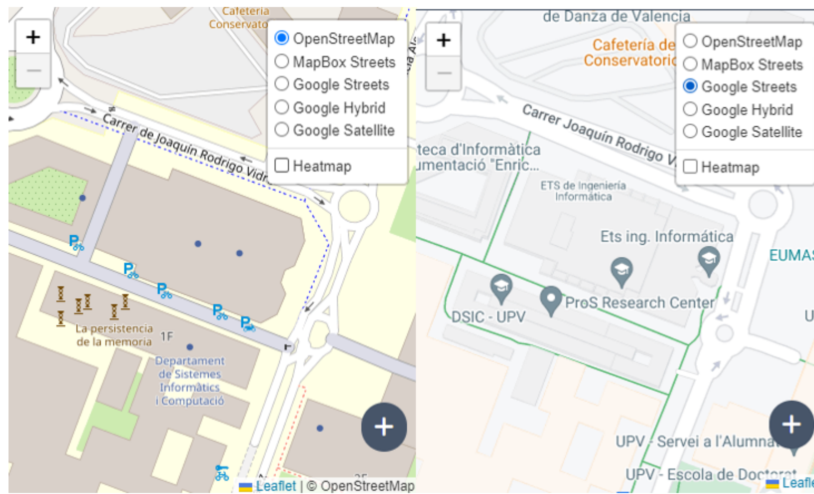

Ilustración 42. Vista de OpenStreetMap y Google Streets

5.2.5. Librerías Leaflet

Como hemos mencionado anteriormente, Leaflet nos ofrece la posibilidad de extender su funcionalidad con librerías y *plugins* externos. En los siguientes apartados detallaremos su funcionalidad y el motivo de su selección.

5.2.5.1. *Mapa de Calor (Leaflet.heat[28])*

Esta librería es la que nos ha permitido poder visualizar los puntos en el mapa en forma de un mapa de calor. Viene con una gran cantidad de atributos que podemos modificar para personalizar la visualización de los puntos, y unos métodos para actualizarlo en tiempo real.

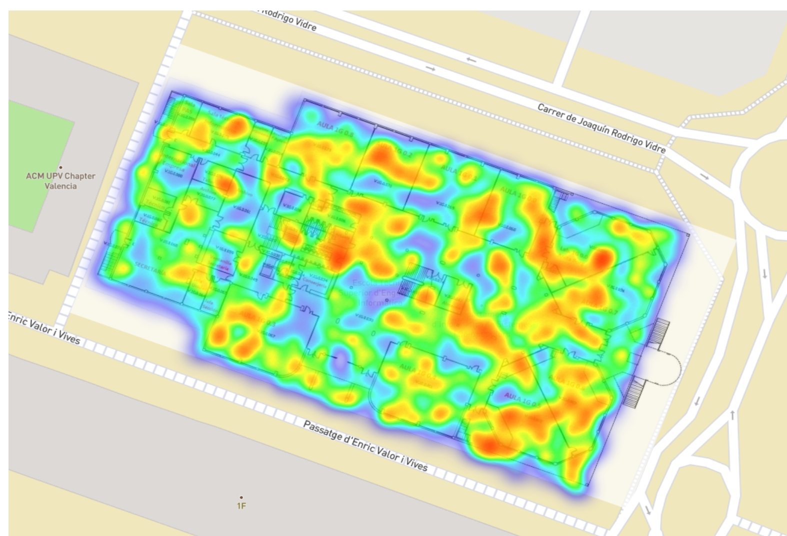

Ilustración 43. Visualización del mapa de calor

[28] Repositorio Github de Leaflet.heat: https://github.com/Leaflet/Leaflet.heat

5.2.5.2. *Distortable Image*[29]

Esta librería nos permite importar y visualizar imágenes sobre el mapa, además de ofrecernos una barra de herramientas donde podremos modificar la imagen como queramos, ya sea rotarla, ampliarla, reducir la opacidad, etc.

Ilustración 44. Imagen de la primera planta del edificio 1G de la ETSINF sobre el mapa

[29] Repositorio Github de DistortableImage: https://github.com/publiclab/Leaflet.DistortableImage

6. Implantación

En este apartado se explicará cómo realizar los preparativos necesarios, el proceso de instalación y los pasos necesarios para poder ejecutar el servidor junto a la aplicación web.

6.1. Preparativos

Para hacer que el servidor funcione es necesario instalar la base de datos PostgreSQL, para ello seguiremos los siguientes pasos:

1. Abriremos la página https://www.postgresql.org/download/ y descargaremos la versión correspondiente a nuestro sistema operativo.
2. Seguiremos el proceso de instalación y nos aseguraremos de instalar los componentes de pgAdmin4 y Stack Builder..
3. Lanzaremos Stack Builder y seleccionaremos la opción de *PostgreSQL XX (xXX) on port XXX.*
4. En la siguiente ventana desplegaremos la lista de *Database Drivers* y seleccionaremos los 2 últimos.
5. Al terminar el proceso de instalación, abriremos pgAdmin4 y nos logueamos con la contraseña que hemos establecido anteriormente.
6. Desplegaremos la lista y haremos *click* derecho en *Database,* donde seleccionaremos "*Create > Database*".
7. Una vez creada la base de datos podemos cerrar el programa y pasar la siguiente fase.

A continuación, necesitamos instalar Node.js para poder instalar todas las librerías tanto en el servidor como en la aplicación web. Para ello visitaremos la página https://nodejs.org/en/download/ y descargaremos la versión correspondiente con nuestro sistema operativo. Es posible que haya que reiniciar el sistema al finalizar la instalación.

6.2. Instalación

Antes de instalar los proyectos y sus librerías, es recomendable disponer de un IDE como Visual Studio Code, aunque esto es completamente opcional.

En la máquina donde se planea desplegar las aplicaciones simplemente tendremos que descargar y descomprimir los proyectos en el directorio que deseemos. A continuación, abriremos una terminal en el directorio fuente de cada uno, y ejecutaremos el comando **npm install** para descargar todas las dependencias y librerías necesarias para su funcionamiento.

Para el servidor necesitaremos crear un fichero .env donde introducir todas las variables necesarias para conectarnos a la base de datos.

6.3. Ejecución

En primer lugar tendremos que ejecutar el servidor *backend* para que la aplicación web pueda utilizar sus APIs. Para ello, abriremos la terminal y ejecutaremos el comando **npm run dev** que se encargará de arrancarlo y mantenerlo en ejecución aunque realicemos cambios en el proyecto.

La URL base del servidor sería en este caso http://localhost:3000/ o http://192.168.18.118:3000 en caso de querer acceder desde otro dispositivo en la misma red.

A continuación, podremos ejecutar la aplicación web abriendo una terminal en el proyecto y ejecutando el comando **npm run serve** que se encargará de arrancarlo y también lo mantendrá en ejecución aunque se realicen cambios en el proyecto.

La URL base de la aplicación web sería en este caso http://localhost:8080/ o https://192.168.18.118:8080 en caso de querer acceder desde otro dispositivo en la misma red.

7. Pruebas

En este apartado explicaremos cómo se han realizado los diferentes tipos de pruebas para comprobar el correcto funcionamiento del sistema durante todas las etapas de su desarrollo.

En primer lugar, se detallarán las pruebas unitarias, las cuales tienen el propósito de validar el correcto funcionamiento de todos los componentes que forman las aplicaciones.

A continuación, se mostrarán las pruebas de integración, las cuales permiten probar el correcto funcionamiento de los componentes desarrollados una vez ya dentro del sistema.

Finalmente, se realizaron pruebas con el usuario final en cada entrega, para identificar los posibles defectos del mismo y que partes son aceptables para el producto final.

7.1. Pruebas unitarias

Estas pruebas nos permiten verificar que las distintas partes del programa cumplen su función. Esto se hace mediante la construcción de pruebas, donde cada una ejecuta unas instrucciones concretas y define unos resultados esperados, que fallarán en el caso de que sean incorrectas.

Para que estas pruebas sean de buena calidad deben cumplir unos criterios:

- Automatizables. Se deben ejecutar de forma automática, sin intervención manual.
- Completas. Deben abarcar la mayor cantidad de código.
- Rápidas. Deben poder ejecutarse en milésimas de segundo.
- Reutilizables: Se pueden ejecutar más de una vez.
- Independientes: La ejecución de una prueba no debe afectar a otra.

Para construir estas pruebas en el servidor hemos utilizado un *framework* de JavaScript llamado Jest[30].

Antes de ejecutar las pruebas hemos creado un edificio de prueba, y hemos llamado al método *beforeAll()* para subirlo en nuestra base de datos.

```
const newBuilding = {
    name: "Test Building",
    floors: 5,
    latitude: 39.48067057885303,
    longitude: -0.3398773008264785
}

beforeAll(async () => {
    // Populate database with one building
    await request(app).post("/buildings").send(newBuilding);
});
```

Ilustración 45. Instrucciones que se ejecutan al comienzo de las pruebas

[30] Getting started with Jest: https://jestjs.io/es-ES/docs/getting-started

A continuación, cada endpoint de la API se separa en un bloque, para así tener el código estructurado y poder identificar los errores con mayor rapidez. En cada uno de estos bloques también se puede ejecutar los métodos *beforeAll*, *beforeEach*, *afterAll* y *afterEach* por si queremos ejecutar unas instrucciones antes de cada test.

```
describe("GET /buildings", () => {

    // HTTP Request
    let response;

    beforeAll(async () => {
        response = await request(app).get("/buildings");
    });

    // Check correct functionality

    it("returns 200", async () => {
        expect(response.statusCode).toBe(200);
    });

    it("returns the array of buildings", async () => {
        expect(response.body.length >= 1).toBe(true);
    });

});
```

Ilustración 46. Bloque de casos de prueba destinos para el endpoint GET /buildings

Finalmente, al terminar de ejecutarse todos los tests unitarios se ejecuta una función llamada *afterAll()* que será la encargada de vaciar la base de datos de prueba, y así prepararla para la siguiente ejecución.

Al ejecutar los tests con el comando **npm run test** obtendremos en la terminal la lista de tests y si se han ejecutado correctamente o no.

```
  GET /coordinates
    √ returns 200 for existing building
    √ returns the array of coordinates for existing building
    √ returns 500 and error for empty query (2 ms)
    √ returns 500 and error for missing building attribute (3 ms)
    √ returns 500 and error for missing startDate attribute (2 ms)
    √ returns 500 and error for missing endDate attribute (2 ms)
  POST /coordinates
    √ returns 200
    √ returns JSON
    √ returns success message
    √ returns created coordinate (1 ms)
    √ returns 400 and error for empty coordinate (2 ms)
    √ returns 400 and error for latitude under -90 (2 ms)
    √ returns 400 and error for latitude over 90 (1 ms)
    √ returns 400 and error for longitude under -180 (2 ms)
    √ returns 400 and error for longitude over 180 (1 ms)
    √ returns 500 and error for wrong building id (25 ms)
    √ returns 500 and error when floornumber is a string (28 ms)
    √ returns 500 and error when latitude is a string (2 ms)
    √ returns 500 and error when longitude is a string (2 ms)

Test Suites: 1 passed, 1 total
Tests:       41 passed, 41 total
Snapshots:   0 total
Time:        1.402 s
Ran all test suites.
```

Ilustración 47. Resultado de ejecutar los tests en el servidor

Para la aplicación web se ha seguido un procedimiento similar, pero en este caso se ha utilizado un *framework* distinto llamado Vitest[31], ya que está enfocado a las aplicaciones desarrolladas con Vue.

Una de las características más importantes es la capacidad de poder testear cada componente de forma independiente, de una forma bastante parecida, donde lo único que tenemos que hacer es montar el componente, pasarle datos y comprobar que los elementos en la interfaz y los datos que devuelve son correctos.

```
test("not going below 0", async () => {
    expect(FloorCounter).toBeTruthy();

    const wrapper = mount(FloorCounter, {
        props: {
            floors: 1,
            fromBuildingTable: true
        }
    });

    expect(wrapper.text()).toBe('1');

    await wrapper.find('#decreaseButton').trigger("click");

    expect(wrapper.text()).toBe('1');
});
```

Ilustración 48. Test unitario para comprobar que el selector de pisos no baja de 0

7.2. Pruebas de rendimiento

Uno de los factores más críticos de este sistema es la capacidad de manejar una gran cantidad de datos, tanto en la parte del servidor como en la del cliente web. Este tipo de pruebas nos permitirá ver su evolución e identificar las limitaciones del sistema llegados a un número determinado de puntos.

Para ello, se ha realizado un script que se encargará de generar miles de puntos y llamar al *endpoint* de manera simultánea para crear puntos en la base de datos. Esto nos permitirá también analizar el número de peticiones que puede manejar nuestro servidor al mismo tiempo.

Tras realizar varias modificaciones al script y ejecutarlo varias veces, podemos determinar que el endpoint soporta entre 4000 y 5000 peticiones simultáneas.

A la hora de realizar la query en PostgreSQL hemos podido ver que el rendimiento se ve claramente afectado al acercarnos al millón de puntos, como podemos ver en la gráfica siguiente.

[31] Página oficial Vitest: https://vitest.dev/

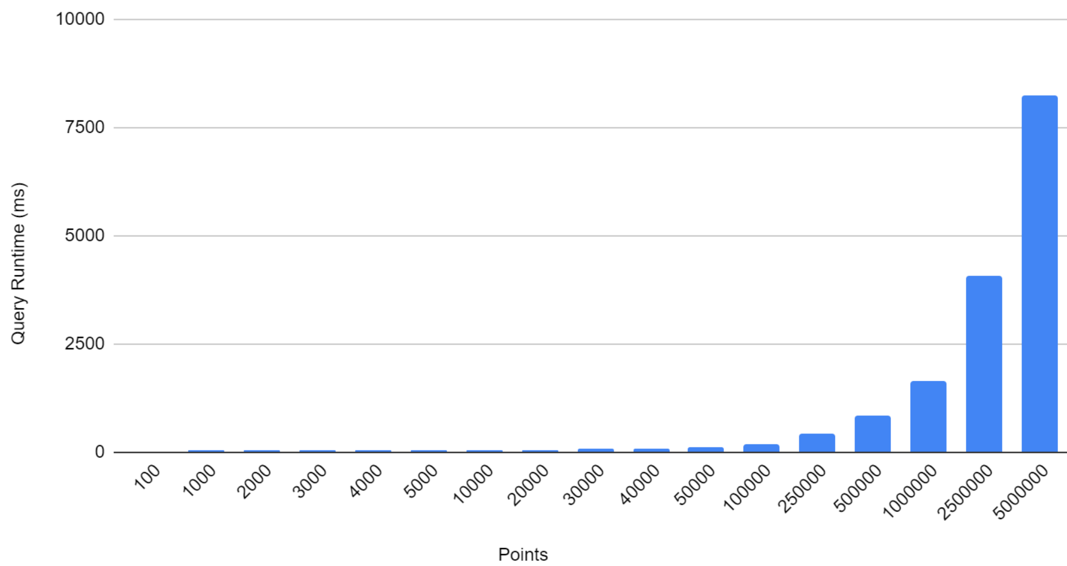

*Ilustración 49. Gráfico de barras sobre la latencia de una query SELECT * sobre la tabla de coordenadas*

Aunque si nos fijamos en el gráfico de líneas podremos ver que la tendencia es completamente lineal, dando la conclusión de que PostgreSQL es estable incluso manejando grandes cantidades de datos.

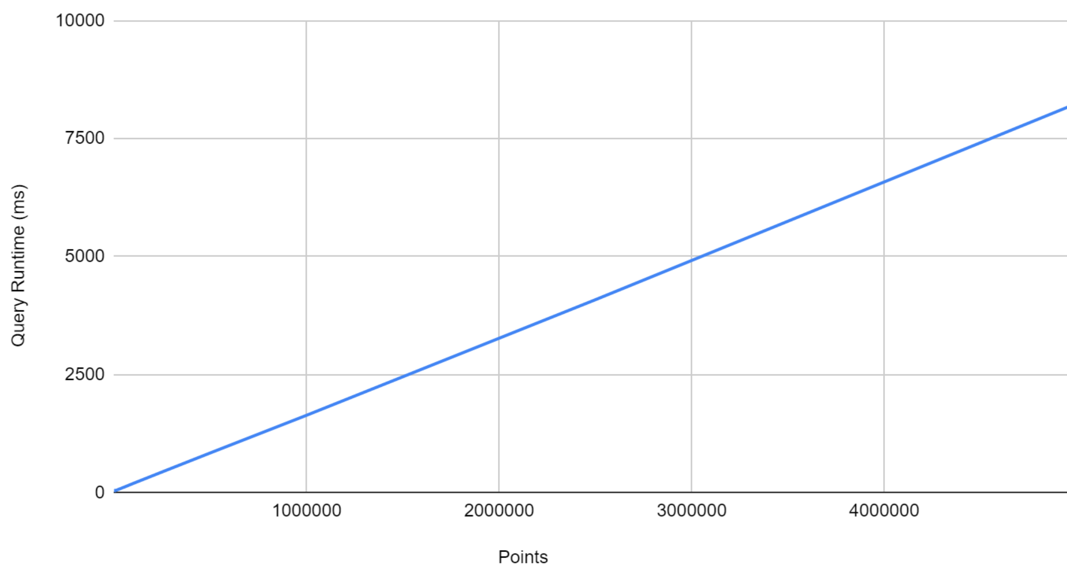

*Ilustración 50. Gráfico de líneas sobre la latencia de una query SELECT * sobre la tabla de coordenadas*

En la parte del cliente web hemos medido cuánto tarda la respuesta del servidor a una petición HTTP , y como podemos ver en la siguiente figura, una vez pasado el medio millón de puntos podemos esperar tiempos de respuesta superiores a un segundo.

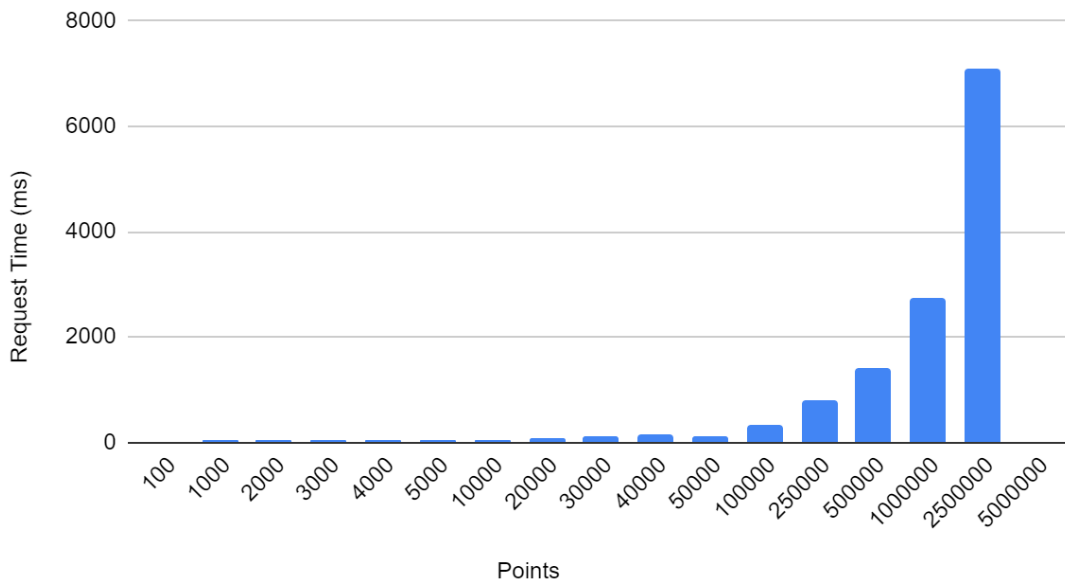

Ilustración 51. Tiempo de respuesta a una petición HTTP al servidor REST API

Cabe destacar que llegados a los cinco millones de puntos el servidor tarda tanto en responder que se produce un *timeout* y se devuelve un error al cliente. Por tanto, podemos decir que el servidor tiene una capacidad máxima de tres millones de puntos al mismo tiempo.

Otra comparativa que nos podría interesar contemplar es el tamaño del fichero JSON devuelto en la respuesta del servidor, ya que dependiendo de nuestra conexión a Internet es posible que los tiempos de respuesta sean más elevados, ya que tendremos que descargar dicho fichero antes de visualizar el mapa de calor. En la siguiente tabla se muestran los diferentes casos que se han probado.

Puntos	Tamaño
100	31 B
1000	68 B
10000	1,3 MB
100000	13,5 MB
1000000	136 MB

Tabla 7. Tamaño del fichero JSON adjunto a la respuesta del servidor respecto a los puntos que contiene

7.3. Pruebas de aceptación

En las pruebas de aceptación, cada entrega de la aplicación ha sido probada por el usuario final, que en este caso ha sido mi tutor Josep Silva. Estas pruebas sirven para medir el grado de satisfacción del cliente respecto a sus necesidades, detectar los fallos y sugerir las siguientes modificaciones que se pueden realizar.

Estas pruebas se han repartido en cuatro sesiones, al principio de cada una de ellas se ha realizado una pequeña demo explicando el trabajo realizado, luego los errores y sugerencias se han recopilado en un informe, y para finalizar se han establecido unos objetivos para la siguiente sesión.

7.3.1. Primera sesión

En la primera sesión se probaron las funcionalidades desarrolladas, como la vista del mapa y la comunicación con el servidor mediante peticiones HTTP.

Sin embargo, la aplicación carecía de características importantes como:

- El servidor no almacena información sobre los edificios, sólo sobre las coordenadas.
- El buscador solo servía para el identificador de una coordenada.
- El mapa no tenía el zoom suficiente.
- No había un selector de fechas para realizar las búsquedas.

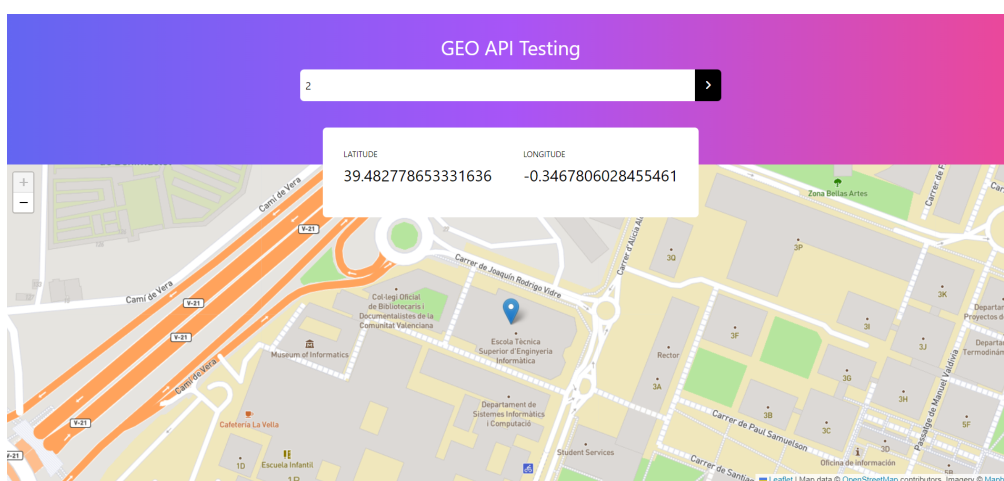

Ilustración 52. Captura de pantalla de la primera entrega de la aplicación

Para solucionar estos defectos se puso como objetivo trabajar en la barra superior donde se realizan las búsquedas.

7.3.2. Segunda sesión

En la segunda sesión se comprobó que todos los defectos de la entrega anterior se habían solucionado, y se identificaron los siguientes problemas:

- Al iniciar la aplicación, la fecha de inicio y de final es la misma, dando paso a una búsqueda predeterminada de 0 minutos.

- El sistema no muestra un mensaje de aviso cuando se le devuelve una búsqueda sin resultados.
- No existe una interfaz donde poder gestionar los edificios.
- Los puntos de calor en el mapa tienen demasiada intensidad.

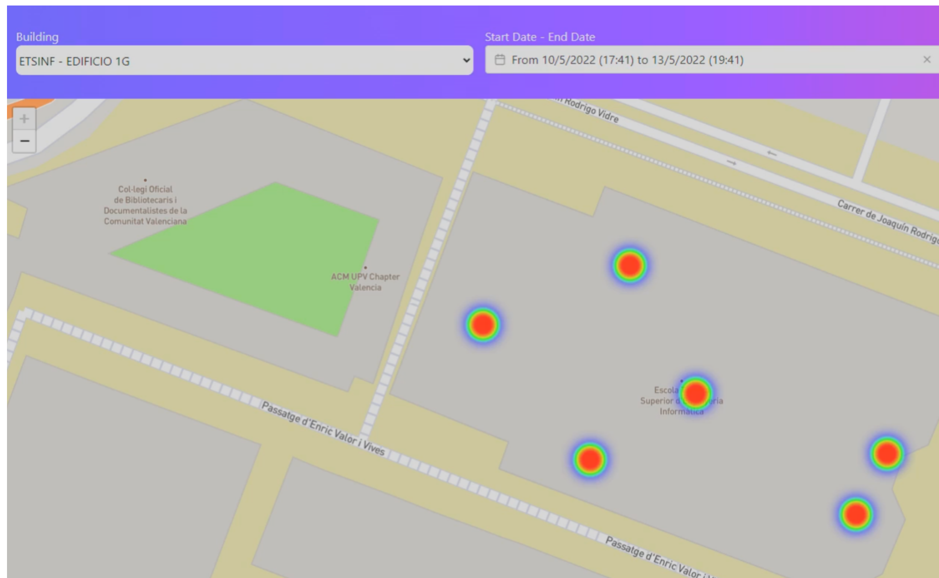

Ilustración 53. Captura de pantalla de la segunda entrega de la aplicación

Los objetivos marcados para la siguiente sesión fueron desarrollar un gestor de edificios y mejorar el calendario de la barra de búsquedas.

7.3.3. Tercera sesión

En la tercera sesión se revisaron las funcionalidades implementadas, que en este caso tenían bastante énfasis en el gestor de edificios, tal y como vemos en las figuras siguientes.

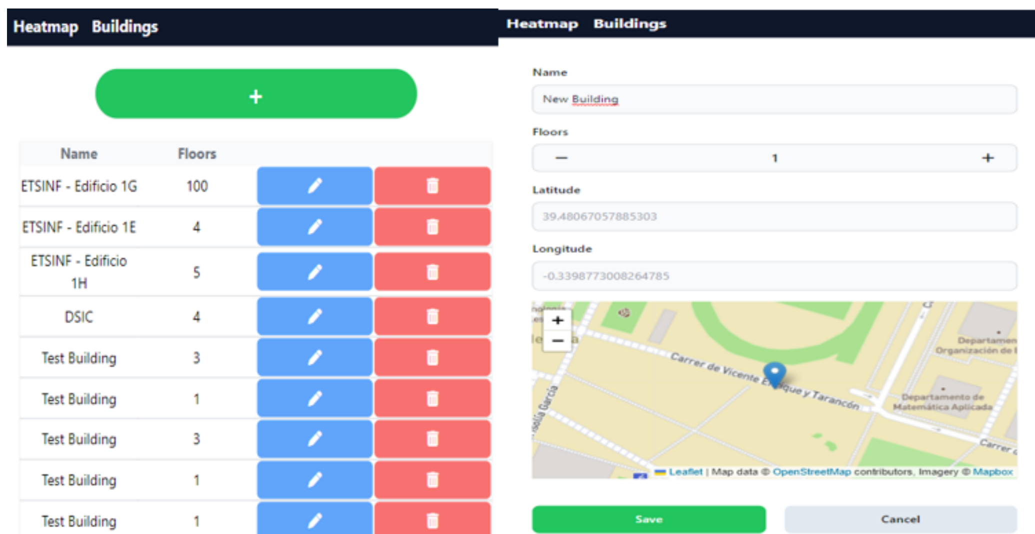

Ilustración 54. Gestor de edificios en la tercera sesión

Como en todas las sesiones realizadas hasta el momento, mi cliente (Josep Silva) realizó unas pruebas exhaustivas en la aplicación y pudo detectar los siguientes defectos:

- En el calendario las fechas deberían actualizarse automáticamente, por tanto, la barra inferior con los botones de cancelar y seleccionar deben ser eliminados.
- Al abrir la aplicación, la fecha final debería ser la actual y la fecha de inicio la misma pero 24 horas antes.
- Añadir rangos de fechas (hoy, ayer, última hora, última semana, este mes…) para poder realizar las búsquedas más rápido.
- Crear un selector de pisos para los edificios con los botones más y menos a los lados.
- Hacer las interfaces responsive para que se pueda ver en una tablet o móvil, en ambas orientaciones.

Con estos problemas ya identificados se procedió a realizar las correcciones necesarias para la siguiente sesión.

7.3.4. Última sesión

En la última sesión, se realizó una comprobación con el cliente sobre todas las funcionalidades implementadas y arregladas en la aplicación, y finalmente se mostró satisfecho para dar el punto bueno y presentar este proyecto.

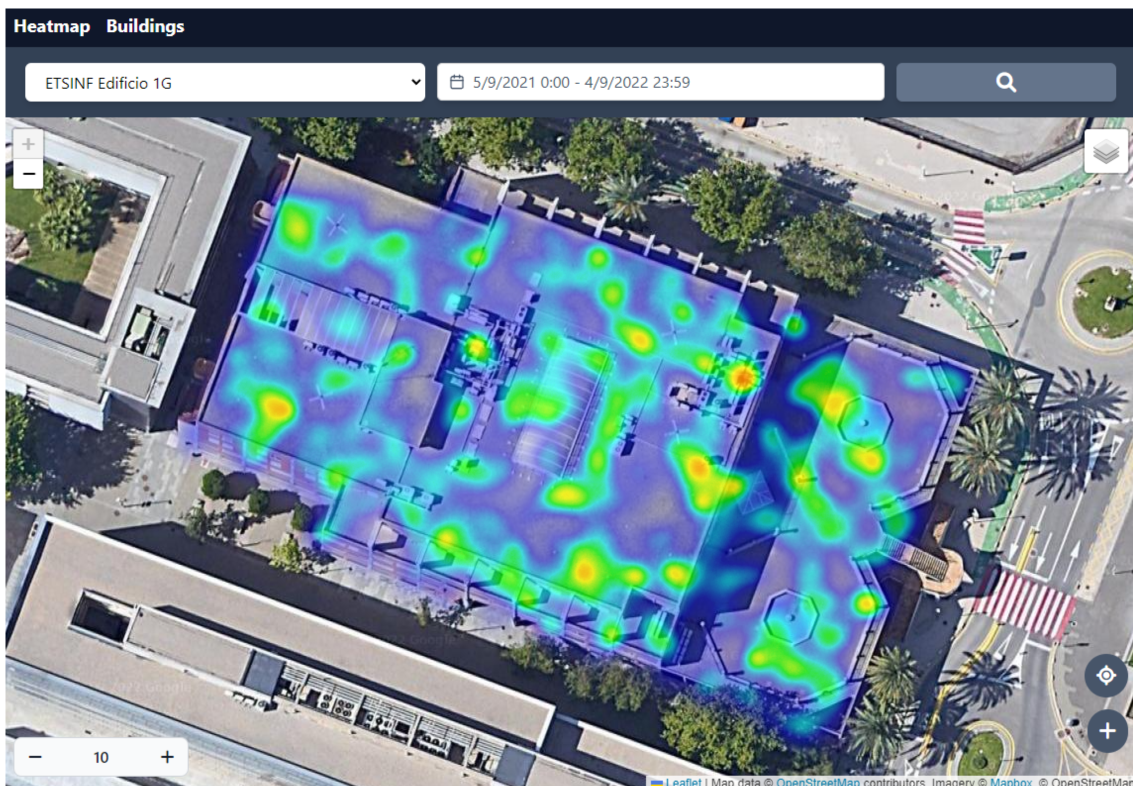

Ilustración 55. Captura de pantalla de la última entrega de la aplicación

8. Conclusión

En este apartado presentaremos los resultados del proyecto desarrollado, y la comparación con el sistema originalmente propuesto. Además, se realizará una retrospección para analizar todo lo que se ha aprendido durante el desarrollo del mismo, y finalmente una valoración personal

8.1. Resultados del proyecto

Originalmente se planteó un sistema muchísimo más ambicioso, donde se iban a implementar funcionalidades como la posibilidad de ver el mapa de calor en tiempo real, y poder mandar ofertas o anuncios a los transeúntes del edificio en ese mismo momento.

Debido al escaso tiempo disponible y la nula experiencia en desarrollo web, al final se optó por implementar las funcionalidades más básicas para que el usuario pueda ver los mapas de calor realizando búsquedas, y una interfaz sencilla donde gestionar los edificios.

8.2. Aprendizaje

Como hemos mencionado anteriormente, mi experiencia en desarrollo web era completamente inexistente al inicio de este proyecto. Pero por esta misma razón tomé este TFG como una oportunidad para formarme en un nuevo ámbito de la informática, y estar más preparado para el mundo laboral.

Por suerte, ya había desarrollado un servidor con API previamente en la universidad, aunque nunca con una conexión a una base de datos local ni los tests unitarios, esto último me ha ayudado a mejorar la calidad de mi código y una estructura más organizada.

En aspecto generales, este proyecto me ha ayudado a entender cómo funcionan y cómo se estructuran las aplicaciones web actuales, algo que se complementa a la perfección con mis conocimientos de desarrollo en Android adquiridos durante la carrera.

8.3. Valoración personal

Un proyecto de estas dimensiones ha resultado un gran desafío como estudiante en los últimos 6 meses, pero tengo que dar gracias a mi tutor Josep Silva en primer lugar por orientarme y ayudarme desde el inicio hasta el final.

La organización del tiempo y las tareas ha sido un factor clave para llevar a cabo todo lo planificado, y las reuniones periódicas me han ayudado a identificar los errores más importantes y marcar los objetivos para las siguientes semanas.

En mi opinión, en este proyecto se podrían haber implementado cosas más avanzadas si hubiera tenido conocimientos más avanzados y haberle dedicado más tiempo, pero me encuentro satisfecho con el trabajo realizado, y espero poder aprender más en un futuro cercano.

9. Trabajos Futuros

En un entorno comercial, este sistema se podría decir que se encuentra en una etapa bastante temprana, ya que todavía se podría implementar un listado bastante extenso de características.

En el caso que este proyecto se llevara adelante, se podrían integrar las siguientes funcionalidades:

- Un sistema de gestión de usuarios.
- La posibilidad de ver los mapas de calor en tiempo real.
- Guardar la localización de un edificio con una array de coordenadas que definen un polígono que corresponda con los límites del mismo.
- Añadir información de los usuarios en las coordenadas, para luego poder aplicar filtros en el mapa de calor dependiendo del género, el rango de edad, sus recorridos habituales…
- Añadir un reproductor o slider donde poder ver el avance del mapa de calor por minutos u horas.
- Poder disponer de vistas personalizadas donde ver el interior de un edificio sin tener que importar una imagen del plano.

10. Índice de ilustraciones

Ilustración 1. Tablero Trello de la primera fase 8

Ilustración 2. Tablero Trello de la segunda fase 9

Ilustración 3. Tablero Trello de la tercera fase 9

Ilustración 4. Tablero Trello de la última fase 10

Ilustración 5. Diagrama de Gantt sobre el tiempo invertido en cada fase 10

Ilustración 6. Página principal de GitLab 12

Ilustración 7. Calendario de contribuciones en un repositorio de Github a lo largo del año 12

Ilustración 8. Mapa de calor sobre la ciudad de San Francisco 13

Ilustración 9. Mapa de calor sobre la página web de una tienda online 13

Ilustración 10. Modelo de dominio 15

Ilustración 11. Modelo de contexto 16

Ilustración 12. Diagrama de casos de uso gestión de coordenadas en el servidor 17

Ilustración 13. Diagrama de casos de uso gestión de edificios en el servidor 19

Ilustración 14. Diagrama de casos de uso sistema de búsqueda en el cliente web 20

Ilustración 15. Diagrama de casos de uso gestión de edificios en el cliente web 21

Ilustración 16. Diagrama de casos de uso ajustes del mapa en el cliente web 23

Ilustración 17. Arquitectura cliente - servidor con REST API 27

Ilustración 18. Arquitectura del sistema a desarrollar 32

Ilustración 19. Arquitectura del servidor API REST 33

Ilustración 20 Modelo de datos del sistema 34

Ilustración 21. Arquitectura del cliente web 35

Ilustración 22. Wireframe de la interfaz de heatmaps 35

Ilustración 23. Wireframe de la interfaz de los edificios 36

Ilustración 24. Wireframe del editor de edificios 36

Ilustración 25. Estructura general de los ficheros en el backend 37

Ilustración 26. Contenidos del fichero server.js 38

Ilustración 27. Contenidos del fichero .env 38

Ilustración 28. Contenidos del fichero .gitignore 39

Ilustración 29. Contenidos del fichero package-json 39

Ilustración 30. Ejemplo de una llamada GET en el fichero route.rest 40

Ilustración 31. Contenidos del fichero src/router/index.js 40

Ilustración 32. Estructura de una petición HTTP a una API 41

Ilustración 33. Conexión a la base de datos en el controlador 42

Ilustración 34. Métodos asíncronos en el controlador 42

Ilustración 35. Contenidos del fichero src/database/database.sql 43

Ilustración 36. Ejemplo de cómo realizar una query SELECT sobre una tabla en pgAdmin4 43

Ilustración 37. Ejemplo de un método en la capa Repositorio para obtener la lista de edificios 44

Ilustración 38. Estructura de los ficheros en el proyecto del cliente web 44

Ilustración 39. Selector de pisos siendo reutilizado sobre el mapa y en la ficha de los edificios 45

Ilustración 40. Declaración del selector de pisos en el mapa y en la ficha de los edificios 45

Ilustración 41. Construcción del Router en nuestra aplicación web 46

Ilustración 42. Vista de OpenStreetMap y Google Streets 47

Ilustración 43. Visualización del mapa de calor 48

Ilustración 44. Imagen de la primera planta del edificio 1G de la ETSINF sobre el mapa 48

Ilustración 45. Instrucciones que se ejecutan al comienzo de las pruebas 51

Ilustración 46. Bloque de casos de prueba destinos para el endpoint GET /buildings 52

Ilustración 47. Resultado de ejecutar los tests en el servidor 52

Ilustración 48. Test unitario para comprobar que el selector de pisos no baja de 0 53

Ilustración 49. Gráfico de barras sobre la latencia de una query SELECT * sobre la tabla de coordenadas 54

Ilustración 50. Gráfico de líneas sobre la latencia de una query SELECT * sobre la tabla de coordenadas 54

Ilustración 51. Tiempo de respuesta a una petición HTTP al servidor REST API 55

Ilustración 52. Captura de pantalla de la primera entrega de la aplicación 56

Ilustración 53. Captura de pantalla de la segunda entrega de la aplicación 57

Ilustración 54. Gestor de edificios en la tercera sesión 57

Ilustración 55. Captura de pantalla de la última entrega de la aplicación 58

11. Índice de tablas

Tabla 1. Comparativa de los diferentes frameworks para el servidor 28

Tabla 2. Ventajas y desventajas de las bases de datos relacionales y no relacionales 28

Tabla 3. Comparativa según sus características de las bases de datos relacionales y no relacionales 29

Tabla 4. Comparativa de los frameworks de desarrollo para el cliente web 30

Tabla 5. Comparativa de los frameworks CSS para el cliente web 30

Tabla 6. Todas las llamada API del servidor 41

Tabla 7. Tamaño del fichero JSON adjunto a la respuesta del servidor respecto a los puntos que contiene 55

12. Referencias

[01] Android, “Android for Developers”,
https://developer.android.com/
(accessed Feb. 15, 2022)

[02] Trello, “Trello makes it easier for teams to manage projects and tasks”,
https://trello.com/tour
(accessed Feb. 21, 2022)

[03] 9GAG, “About 9GAG”,
https://about.9gag.com/app
(accessed May. 5, 2022)

[04] Nintendo, “About us”,
https://www.nintendo.com/about/
(accessed May. 5, 2022)

[05] GitLab, “About GitLab,
https://about.gitlab.com/company/
(accessed May. 5, 2022)

[06] OpenStreetMap, “About us”,
https://www.openstreetmap.org/about
(accessed May. 5, 2022)

[07] Github, “Viewing contributions on your profile”,
https://docs.github.com/en/account-and-profile/setting-up-and-managing-your-github-profile/managing-contribution-settings-on-your-profile/viewing-contributions-on-your-profile
(accessed May. 5, 2022)

[08] Mozilla, “HHTP Methods”,
https://developer.mozilla.org/es/docs/Web/HTTP/Methods
(accessed May, 18, 2022)

[09] Java, “Spring Framework overview”,
https://spring.io/projects/spring-framework
(accessed May. 25, 2022)

[10] Django, “Why Django? Framework overview”,
https://www.djangoproject.com/start/overview/
(accessed May, 25, 2022)

[11] Node.js, “About Node.js”,
https://nodejs.org/en/about/
(accessed May, 25, 2022)

[12] PostgreSQL, “What is PostgreSQL? Why use PostgreSQL?”,
https://www.postgresql.org/about/
(accessed June 1, 2022)

[13] MySQL, “Why MySQL?”,
https://www.mysql.com/why-mysql/
(accessed June 1, 2022)

[14] Visual Studio Code, “Why did we built visual studio code?”,
https://code.visualstudio.com/docs/editor/whyvscode
(accessed June 5, 2022)

[15] pgAdmin, “Introduction of pgAdmin4”,
https://www.pgadmin.org/
(accessed June 5, 2022)

[16] Angular, “Introduction to angular concepts”,
https://angular.io/guide/architecture
(accessed June 11, 2022)

[17] React, “Getting started with React”,
https://reactjs.org/docs/getting-started.html
(accessed June 11, 2022)

[18] Vue.js, “What is Vue? An introduction to Vue”,
https://vuejs.org/guide/introduction.html
(accessed June 11, 2022)

[19] Bootstrap, “Getting started with Bootstrap”,
https://getbootstrap.com/docs/4.0/getting-started/introduction/
(accessed June 15, 2022)

[20] Tailwind, “Get started with Tailwind CSS”,
https://tailwindcss.com/docs/installation
(accessed June 15, 2022)

[21] Google, “Meet the features that set Chrome apart”,
https://www.google.com/intl/en/chrome/browser-features/#
(accessed June 28, 2022)

[22] Mozilla, “Mozilla Firefox features”,
https://www.mozilla.org/es-ES/firefox/features/
(accessed June 28, 2022)

[23] Microsoft, “Microsoft Edge features”,
https://www.microsoft.com/es-es/edge/features
(accessed June 28, 2022)

[24] Opera, “Why Opera?”,
https://www.opera.com/es/about
(accessed June 28, 2022)

[25] Apple, “Safari browser”,
https://www.apple.com/es/safari/
(accessed June 28, 2022)

[26] JSON, “Introducing JSON”
https://www.json.org/json-en.html
(accessed July 14, 2022)

[27] Vladimir Agafonkin, “Leaflet overview”,
https://leafletjs.com/
(accessed Feb. 25, 2022)

[28] Repositorio Github de Leaflet.heat,
https://github.com/Leaflet/Leaflet.heat
(accessed Apr. 13, 2022)

[29] Repositorio Github de DistortableImage,
https://github.com/publiclab/Leaflet.DistortableImage
(accessed Apr. 15, 2022)

[30] Jest, “Getting started with Jest”,
https://jestjs.io/es-ES/docs/getting-started
(accessed Aug. 20, 2022)

[31] Vitest, “Getting started with Vitest”
https://vitest.dev/guide/
(accessed Aug. 20, 2022)