

Diseño de un interfaz de monitorización y gestión para sistemas distribuidos



Grado en Ingeniería Multimedia

Trabajo Fin de Grado

Autor:

Martín Fierro de Pedro

Tutor/es:

José Vicente Berná Martínez

Lucía Arnau Muñoz



Universitat d'Alacant
Universidad de Alicante

Resumen

El proyecto se enfoca en el diseño y desarrollo de una interfaz para un sistema distribuido en concreto. Este sistema consta de tres tipos de nodos: balanceadores, controladores y procesadores. Como en todo sistema distribuido los nodos desarrollan una serie de funciones según su tipo, ya sea de control, gestión o procesamiento, e intercambian información entre ellos con el fin de que el sistema funcione correctamente.

Partiendo de esta base, el objetivo era crear una interfaz visual intuitiva y fácil de comprender que reflejase lo que sucede en el sistema en tiempo real. Al comienzo del proyecto se plantearon varias ideas de interfaces y debatiendo pros y contras se decidió elaborar la interfaz en forma de grafo ya que se ajustaba mejor a los requerimientos del sistema. Una vez comenzó el desarrollo se planteó que no solo fuera una interfaz de monitorización de información, sino que también permitiera gestionar los elementos del sistema. Además, con la idea de que fuera usable en cualquier situación se creó un apartado de configuración para adaptar el sistema a cualquier necesidad de representación.

El desarrollo del proyecto se llevó a cabo entre septiembre y diciembre de 2024. En este desarrollo se usó la metodología Scrum dividida en iteraciones de entre una y dos semanas. En estas iteraciones se planteaba lo que se iba a desarrollar, se desarrollaba y se revisaba, realizando así las tareas de diseño, implementación y pruebas del servicio.

El resultado es un servicio personalizable capaz de gestionar y monitorizar un sistema distribuido a través de una interfaz atractiva, intuitiva y fácil de usar.

Resum

El projecte s'enfoca en el disseny i desenrotllament d'una interfície per a un sistema distribuït en concret. Este sistema consta de tres tipus de nodes: balanceadors, controladors i processadors. Com en tot sistema distribuït els nodes desenrotllen una sèrie de funcions segons el seu tipus, ja siga de control, gestió o processament, i intercanvien informació entre ells amb la finalitat que el sistema funcione correctament.

Partint d'esta base, l'objectiu era crear una interfície visual intuïtiva i fàcil de comprendre que reflectira el que succeix en el sistema en temps real. Al començament del projecte es van plantejar diverses idees d'interfícies i debatent pros i contres es va decidir elaborar la interfície en forma de graf ja que s'ajustava millor als requeriments del sistema. Una vegada va començar el desenrotllament es va plantejar que no sols fora una interfície de monitoratge d'informació, sinó que també permetera gestionar els elements del sistema. A més, amb la idea que fora usable en qualsevol situació es va crear un apartat de configuració per a adaptar el sistema a qualsevol necessitat de representació.

El desenrotllament del projecte es va dur a terme entre setembre i desembre de 2024. En este desenrotllament es va usar la metodologia *Scrum dividida en iteracions d'entre una i dos setmanes. En estes iteracions es plantejava el que s'anava a desenrotllar, es desenrotllava i es revisava, realitzant així les tasques de disseny, implementació i proves del servei.

El resultat és un servei personalitzable capaç de gestionar i monitorar un sistema distribuït a través d'una interfície atractiva, intuïtiva i fàcil d'usar.

Summary

The project focuses on the design and development of an interface for a specific distributed system. This system consists of three types of nodes: balancers, controllers, and processors. As in any distributed system, the nodes perform a series of functions depending on their type, whether related to control, management, or processing, and exchange information among themselves to ensure the system operates correctly.

Based on this foundation, the goal was to create an intuitive and easy-to-understand visual interface that reflects what happens in the system in real time. At the beginning of the project, several interface ideas were proposed, and after discussing their pros and cons, it was decided to design the interface as a graph, as this approach better suited the system's requirements. Once development began, it was decided that the interface would not only monitor information but also allow the management of the system's elements. Additionally, with adaptability in mind, a configuration section was created to tailor the system to any representation needs.

The project development took place between September and December 2024. During this process, the Scrum methodology was used, divided into iterations lasting between one and two weeks. In each iteration, the tasks to be developed were planned, implemented, and reviewed, carrying out the design, implementation, and testing phases of the service.

The result is a customizable service capable of managing and monitoring a distributed system through an attractive, intuitive, and easy-to-use interface.

Motivación, justificación y objetivo general

Este Trabajo de Fin de Grado trata del desarrollo de una interfaz web para entornos distribuidos que facilite la interpretación de eventos y del cúmulo de características del sistema. De modo que sea un sistema que permita una interacción lo más limpia y rápida de las consultas del usuario que lo maneja.

Se buscaba un proyecto diferente, no una aplicación con una temática precisa, sino algo que resultará más motivador. En ese momento, se discutieron ideas con el tutor, quien propuso desarrollar la interfaz gráfica del sistema distribuido. La idea presentada resultó interesante, ya que reflejaba un tema de gran interés desde mitad de la carrera cuando empezamos a desarrollar interfaces más complejas, y esta es la sencillez, es decir, a veces lo más difícil para un ingeniero multimedia es crear una interfaz lo más sencilla de interpretar para el usuario, pero que a la vez le permita acceder a todas las características del sistema de forma rápida.

En ese caso en mi proyecto al tener que crear una interfaz de un sistema distribuido, teniendo en cuenta el abanico de datos que van a pasar por el sistema y buscar la representación más intuitiva, me ayudará a tener un mejor análisis de cómo dividir una interfaz compleja y un mejor criterio a la hora de repartir la información, potenciando así mi interpretación crítica para aportar mejores soluciones y más eficaces según los requerimientos del proyecto a la hora de diseñar interfaces.

Además, el hecho de trabajar con un sistema distribuido requiere que sea una interfaz dinámica que se refresque cada vez que el sistema realice una acción. Lo que me permitirá mejorar mi habilidad de gestión y diseño de interfaces para que cuando haya cambios en esta se representen de la manera más limpia sin dar lugar a interpretaciones erróneas por cambios mal realizados en la interfaz.

En resumen, este Trabajo de fin de Grado me ayudará a desarrollarme en el campo de interfaces de sistemas distribuidos permitiendo que mejore mi entendimiento general tanto de análisis como funcionamiento de los mismos.

Agradecimientos

Para empezar quiero agradecer a mis padres Josefina y Francisco Javier por apoyarme desde el minuto uno que empecé la carrera y por brindarme apoyo incondicional incluso cuando no entienden la mayoría de cosas que les explico.

Además, quisiera agradecer a mis amigos Andrés y Alejandro por haber sabido inspirarme y motivarme durante este último año y medio de carrera.

Por último, agradecer a mis tutores Jose Vicente Berna Martínez y Lucía Arnau Muñoz por ayudarme en todas las fases del proyecto y darme consejos para mejorar como ingeniero.

Citas

El talento es un precio que pagas por el esfuerzo.

Albert Einstein

Dedicatoria

A mis padres

Índice de contenidos

Resumen.....	1
Resum.....	2
Summary.....	3
Motivación, justificación y objetivo general.....	4
Agradecimientos.....	5
Citas.....	6
Dedicatoria.....	7
Índice de contenidos.....	8
Índice de figuras.....	12
1. Introducción.....	17
2. Planificación.....	19
3. Estado del arte.....	20
3.1 Llamada POST.....	20
3.1.1 Coordinador.....	20
3.1.2 Balanceador.....	21
3.1.3 Procesador.....	22
4. Objetivos.....	23
5. Metodología.....	24
6. Implementación.....	25
6.1 Definición del proyecto y mockups.....	25
6.1.1 Investigación y elección de interfaces de monitorización.....	25
6.1.2 Boceto inicial con interfaz timeline.....	26
6.1.3 Boceto inicial con interfaz grafo.....	26
6.1.4 Pantalla principal vista en forma de grafo.....	27
6.1.5 Pantallas de listas de nodos.....	29
6.1.6 Pantalla de información de los procesadores.....	31
6.1.7 Pantalla de errores de los nodos.....	32

6.2 Revisión y mejora de los mockups.....	33
6.2.1 Primera modificación de la pantalla principal.....	33
6.2.2 Primera modificación de la pantalla de lista de nodos.....	34
6.2.3 Primera modificación de la pantalla de información de los nodos.....	35
6.2.4 Revisión de los cambios.....	36
6.2.5 Segunda modificación de la pantalla principal.....	36
6.2.6 Segunda modificación de la pantalla de lista de nodos.....	38
6.2.7 Segunda modificación de la pantalla de información de los nodos.....	39
6.2.8 Última revisión de los mockups.....	40
6.3 Preparación del entorno de desarrollo.....	41
6.3.1 Preparación del repositorio y estructura del proyecto.....	41
6.3.2 Configuración general de la plantilla de FUSE para Angular.....	42
6.3.3 Creación de componentes de la página grafo.....	43
6.3.4 Creación de componentes de la página lista.....	46
6.3.5 Creación de componentes modales.....	47
6.3.6 Revisión de la estructura de la interfaz y cambios.....	49
6.4 Creación de la base de datos y desarrollo de la API.....	50
6.4.1 Creación de la base de datos.....	50
6.4.2 Preparación del entorno del backend.....	51
6.4.3 Configuración de la conexión con la base de datos.....	52
6.4.4 Desarrollo de la petición GET ALL.....	53
6.4.5 Desarrollo de la petición GET BY ID.....	55
6.4.6 Desarrollo de la petición POST.....	56
6.5 Creación y configuración del grafo.....	59
6.5.1 Creación del grafo con Echarts.....	59
6.5.2 Implementación de la llamada POST en el Grafo.....	60
6.5.3 Implementación de la actualización del grafo.....	61
6.5.4 Diseño del Grafo.....	62
6.5.5 Implementación de la información del nodo en el grafo.....	63

6.5.6 Incidencia con el grafo.....	64
6.5.7 Implementación de los Links en el Grafo.....	65
6.5.8 Deshabilitación de los links.....	66
6.6 Creación e implementación de la vista mapa.....	68
6.6.1 Cambios en el modo de almacenar la geolocalización.....	68
6.6.2 Implementación del mapa en ECharts.....	70
6.6.3 Incidencia con el mapa.....	71
6.6.4 Creación del mapa con Open Street Maps.....	72
6.6.5 Actualización del mapa e información de nodos.....	74
6.7 Mejoras I.....	75
6.7.1 Cambios de aspecto en la pantalla Lista.....	75
6.7.2 Implementación de la petición PUT.....	77
6.7.3 Implementación de la opción visualización.....	78
6.7.4 Implementación del formulario de actualizar nodo.....	80
6.7.5 Implementación de la función buscar.....	80
6.7.6 Implementación de la función descargar.....	83
6.8 Mejoras II.....	83
6.8.1 Implementación del estado interno.....	84
6.8.2 Desarrollo y gestión de la última consulta.....	85
6.8.3 Implementación del apartado de configuración.....	87
6.8.4 Implementación de las conexiones en el mapa.....	90
6.8.5 Solución al problema de recarga.....	91
6.9 Mejoras III.....	92
6.9.1 Mejoras en el modal de información.....	93
6.9.2 Visualización de errores en los formularios.....	96
6.9.3 Nuevos iconos.....	96
6.9.4 Implementación de distribución y ordenación de nodos.....	98
6.9.5 Configuración del mapa.....	102
6.9.6 Tratamiento de nombres.....	102

7. Pruebas y validación.....	104
7.1 Creación y activación de nodos.....	104
7.2 Desactivación de nodos principales.....	105
7.3 Funcionamiento del sistema para varios clústers.....	106
8 Resultados.....	108
8.1 Producto Final.....	108
8.2 Coste Temporal.....	110
9. Conclusiones y trabajo futuro.....	111
9.1 Conclusiones.....	111
9.2 Trabajo Futuro.....	111
Referencias.....	112
Anexo.....	113

Índice de figuras

Figura 1. Estímulo visual por cambio de color (Fuente Propia).....	18
Figura 2. Mockup Interfaz Principal Timeline (Fuente Propia).....	26
Figura 3. Mockup Interfaz Principal Grafo (Fuente Propia).....	27
Figura 4. Mockup Interfaz Principal Grafo 2 (Fuente Propia).....	28
Figura 5. Mockup Interfaz Principal Grafo 3(Fuente Propia).....	29
Figura 6. Mockup Interfaz Lista de Procesadores(Fuente Propia).....	30
Figura 7. Mockup Interfaz Lista de Balanceadores(Fuente Propia).....	30
Figura 8. Mockup Interfaz Lista de Controladores(Fuente Propia).....	31
Figura 9. Mockup Interfaz Información de Procesador(Fuente Propia).....	31
Figura 10. Mockup Interfaz Error de Procesador(Fuente Propia).....	32
Figura 11. Mockup Interfaz Error de Controlador(Fuente Propia).....	32
Figura 12. Mockup Interfaz Error de Balanceador(Fuente Propia).....	33
Figura 13. Mockup Interfaz Principal(Fuente Propia).....	34
Figura 14. Mockup Interfaz Lista de Nodos(Fuente Propia).....	34
Figura 15. Mockup Información de Procesadores(Fuente Propia).....	35
Figura 16. Mockup Información de Balanceadores(Fuente Propia).....	35
Figura 17. Mockup Información de Controladores(Fuente Propia).....	36
Figura 18. Mockup Pantalla Principal con Conexiones y Geolocalización(Fuente Propia).....	37
Figura 19. Mockup Pantalla Principal con Geolocalización Deshabilitada(Fuente Propia).....	37
Figura 20. Mockup Pantalla Principal con Conexiones Deshabilitadas(Fuente Propia).....	38
Figura 21. Mockup Pantalla de lista de Nodos(Fuente Propia).....	38
Figura 22. Mockup Información de Procesadores(Fuente Propia).....	39
Figura 23. Mockup Información de Balanceadores(Fuente Propia).....	39
Figura 24. Mockup Información de Controladores(Fuente Propia).....	40
Figura 25. Paleta de Colores para Eventos del Grafo(Fuente Propia).....	40

Figura 26. Organización del entorno en Visual Studio Code (Fuente Propia)....	41
Figura 27. Elección de la interfaz de la plantilla Fuse (Fuente: https://angular-material.fusetheme.com/).....	43
Figura 28. Creación de componentes Grafo y Lista en Visual Studio Code (Fuente Propia).....	44
Figura 29. Rutas de los componentes principales en la plantilla (Fuente Propia).. 44	
Figura 30. Componente “Button toggle” (Fuente Propia).....	45
Figura 31. Componente “Checkbox” (Fuente Propia).....	45
Figura 32. Cabecera de la Página Grafo (Fuente Propia).....	45
Figura 33. Componente de la Página Lista (Fuente Propia).....	47
Figura 34. Creación de la carpeta Dialogs en Visual Studio Code (Fuente Propia).....	47
Figura 35. Modal de la información de un nodo (Fuente Propia).....	48
Figura 36. Modal de Añadir Nodo (Fuente Propia).....	49
Figura 37. Rediseño de los botones de la lista (Fuente Propia).....	50
Figura 38. Estructura de la tabla “nodos” en PHPMyAdmin (Fuente Propia)...	50
Figura 39. Estructura de directorios del backend de la API en Visual Studio Code (Fuente Propia).....	51
Figura 40. Código de conexión con la base de datos usando Sequelize (Fuente Propia).....	52
Figura 41. Petición GET ALL correcta en Postman (Fuente Propia).....	53
Figura 42. Petición GET BY ID correcta en Postman (Fuente Propia).....	55
Figura 43. Correcto funcionamiento de los validadores de la llamada POST en Postman (Fuente Propia).....	57
Figura 44. Petición POST correcta en Postman (Fuente Propia).....	57
Figura 45. Representación de los nodos en Grafo (Fuente Propia).....	59
Figura 46. Función recursiva (Fuente Propia).....	61
Figura 47. Diseño de los nodos en Grafo (Fuente Propia).....	62
Figura 48. Configuración del diseño de los links en Grafo (Fuente Propia).....	64

Figura 49. Diseño de los links en Grafo (Fuente Propia).....	65
Figura 50. Conexiones activas (Fuente Propia).....	66
Figura 51. Conexiones desactivadas (Fuente Propia).....	66
Figura 52. Latitud y longitud en la Lista (Fuente Propia).....	67
Figura 53. Latitud y longitud en la Modal de información (Fuente Propia).....	68
Figura 54. Latitud y longitud en la Modal de añadir nodo (Fuente Propia).....	68
Figura 55. Geolocalización de nodo activa (Fuente Propia).....	70
Figura 56. Ejemplo de limitación del mapa Echarts (Fuente Propia).....	71
Figura 57. Nodos geolocalizados en mapa creado con OSM y Leaflet (Fuente Propia).....	73
Figura 58. Mapa con Nodos activos e inactivos (Fuente Propia).....	74
Figura 59. Cambios de diseño en la vista de Lista (Fuente Propia).....	75
Figura 60. Tooltip para la accesibilidad (Fuente Propia).....	75
Figura 61. Petición PUT correcta en Postman (Fuente Propia).....	77
Figura 62. Modificación del formulario con el campo visibilidad (Fuente Propia).....	77
Figura 63. Representación de visibilidad en la Lista (Fuente Propia).....	78
Figura 64. Formulario Actualizar Nodo (Fuente Propia).....	79
Figura 65. Petición SEARCH correcta en Postman (Fuente Propia).....	80
Figura 66. Funcionamiento del resultado de una búsqueda (Fuente Propia).....	81
Figura 67. Función que se encarga de descargar la lista de nodos (Fuente Propia).....	82
Figura 68. Modal de información con estado interno (Fuente Propia).....	83
Figura 69. Modal de información sin estado interno (Fuente Propia).....	83
Figura 70. Modal con la fecha de consulta (Fuente Propia).....	84
Figura 71. Primera consulta al modal (Fuente Propia).....	85
Figura 72. Archivo de configuración (Fuente Propia).....	86
Figura 73. Iconos de balanceadores (Fuente Propia).....	87
Figura 74. Iconos de controladores (Fuente Propia).....	87
Figura 75. Iconos de procesadores (Fuente Propia).....	87

Figura 76. Archivo de configuración modificado (Fuente Propia).....	88
Figura 77. Como crear una polyline (Fuente Propia).....	89
Figura 78. Vista mapa con conexiones (Fuente Propia).....	89
Figura 79. Función recursiva de comprobación almacenada en la instancia (Fuente Propia).....	91
Figura 80. Nuevo modal de información para balanceadores según estado (Fuente Propia).....	92
Figura 81. Nuevo modal de información para controladores según estado (Fuente Propia).....	93
Figura 82. Nuevo modal de información para procesadores según estado (Fuente Propia).....	93
Figura 83. Función que aplica el formato a la última consulta (Fuente Propia)	94
Figura 84. Representación de errores en los formularios (Fuente Propia).....	95
Figura 86. Iconos en las opciones de tipo nodo en los formularios (Fuente Propia).....	96
Figura 87. Cálculo de las zonas para los nodos (Fuente Propia).....	97
Figura 88. Distribución visual de nodos en el Grafo (Fuente Propia).....	97
Figura 89. Petición GET ALL BY ORDER correcta en Postman (Fuente Propia).....	98
Figura 90. Modal de información con el campo Orden (Fuente Propia).....	99
Figura 91. Formularios con campo Orden (Fuente Propia).....	99
Figura 92. Valores de configuración del mapa (Fuente Propia).....	100
Figura 93. Formateo de un nombre largo en la vista Grafo (Fuente Propia)...	101
Figura 94. Formateo de un nombre largo en la vista Mapa (Fuente Propia)....	101
Figura 95. Funcionamiento de conexiones del sistema en grafo (Fuente Propia). 102	
Figura 96. Funcionamiento de conexiones del sistema en mapa (Fuente Propia) 102	
Figura 97. Funcionamiento de nodos sustitutos del sistema en grafo (Fuente Propia).....	103

Figura 98. Funcionamiento de nodos sustitutos del sistema en mapa (Fuente Propia).....	103
Figura 99. Funcionamiento de varios cluster en la vista grafo (Fuente Propia)....	
104	
Figura 100. Funcionamiento de varios cluster en la vista mapa (Fuente Propia).	
104	
Figura 101. Interfaz anterior al desarrollo del proyecto (Fuente Propia).....	106
Figura 102. Interfaz final de la vista Nodo (Fuente Propia).....	107
Figura 103. Interfaz final de la vista Mapa (Fuente Propia).....	107
Figura 104. Interfaz final de la vista Lista (Fuente Propia).....	108
Figura 105. Tiempo dedicado por mes al producto (Fuente Propia).....	108

1. Introducción

Los sistemas distribuidos se crearon para tener redes de ordenadores que trabajen de manera conjunta para abordar el máximo número de tareas en el menor tiempo y de la manera más eficiente.

Hoy en día los sistemas distribuidos están presentes en casi todos lados aunque nosotros no los veamos, redes sociales como facebook, twitter, instagram..., plataformas de entretenimiento como netflix, hbo, spotify..., comercio electrónico como amazon, eBay..., usan este tipo de sistemas para gestionar sus redes y los volúmenes de datos gigantes que contienen. Dentro de un sistema tan grande lo que interesa es tener un sistema de monitorización eficiente y rápido que permita detectar los eventos del sistema.

El contexto en el que se ubica este proyecto es la realización de una interfaz gráfica para la monitorización de sistemas distribuidos.

Se investigaron las tecnologías utilizadas para representar estos sistemas, incluyendo tablas y listas, métricas gráficas, árboles y grafos. Tecnologías poco intuitivas y difícilmente interpretables a simple vista. Se identificaron las dimensiones que han llegado a alcanzar estos sistemas distribuidos ha llevado a que estos tres primeros sistemas de representación sufran problemas de monitorización.

Los problemas con los que se encuentran son: la alta cardinalidad de nodos, es decir, la gran cantidad de nodos implicados en la red, la necesidad de tener la seguridad de que estos nodos trabajan de manera correcta, la escalabilidad del sistema de representación y la dificultad de representar las relaciones complejas entre los nodos del sistema. Además del factor añadido que si el sistema de representación no es intuitivo y rápido para el encargado de revisar el mantenimiento del sistema, este debe realizar una tarea de investigación para dar con el problema antes de ponerse con su tratamiento.

El objetivo es eliminar los problemas de los anteriores sistemas de representación mediante una interfaz de monitorización qué permita una visualización centralizada y escalable del sistema distribuido, y ahí es donde entra el sistema de representación basado en grafos.

Esta arquitectura permite representar los nodos del sistema en una interfaz de manera que se puedan identificar de manera sencilla los diferentes componentes en una sola vista centralizada y las relaciones entre ellos. También permite añadir o eliminar tantos nodos como necesitemos representando las relaciones entre ellos independientemente del número de nodos que tenga el sistema.

Por otro lado, en sistemas distribuidos es esencial poder saber como se está comportando el sistema, se necesita poder identificar el correcto funcionamiento de cada elemento, es decir, nodos activos, suplentes y fallos.

Para esto se tomó como referencia el ABP, M4RKET, en el que se representa un supermercado 3D con los productos de una lista de la compra y cada vez que se interactuaba con la lista se añadían, eliminaban o marcaban como comprado los elementos de la interfaz. Gracias a ello surgió una idea, es cierto que un grafo no es una interfaz 3D, pero se puede usar una representación parecida para indicar los nodos activos y fallos convirtiendo el grafo en un elemento dinámico que cambie en base al funcionamiento del sistema distribuido.

Los cambios se modificarían en el grafo según los eventos que sufra el sistema y se reflejarían en la pantalla de manera rápida. Ya que estos estímulos visuales harán que se interpreten rápidamente y te fijes en ellos. Por ejemplo: si dentro de un grupo de objetos uno cambia de color lo detectamos visualmente rápido:

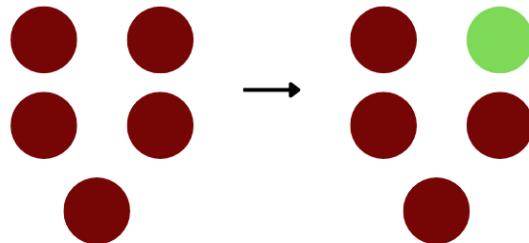


Figura 1. Estímulo visual por cambio de color (Fuente Propia)

Trabajar con este sistema de representación dinámica basada en grafo ayudará a mejorar y solucionar los problemas de escalabilidad, alta cardinalidad y visibilidad de los anteriores sistemas.

2. Planificación

Nada más comenzar el proyecto se planteó la idea de planificar el trabajo usando la metodología basada en iteraciones. Esta metodología encaja perfectamente con los requisitos del proyecto, ya que permite la división del proyecto en etapas previamente definidas.

En la reunión con el Product Owner se habló sobre los requisitos del proyecto. Se dejó claro que era un proyecto delicado ya que no se puede encargar un gran volumen de trabajo. Esto se debe a que cada aspecto cuenta y debe ser revisado para ver si encaja con los requisitos. Por tanto, mandar un gran volumen de trabajo implica que si un aspecto no encaja deba rehacerse gran parte de la iteración.

Tras conocer la complejidad del proyecto se decidió llevar a cabo la metodología planteada pero con iteraciones cortas. De esta manera se asegura que los fallos no tengan gran coste de tiempo y que las especificaciones del proyecto se adapten al desarrollo del mismo.

Con esta planificación se sigue el siguiente patrón de trabajo, primero se plantea la iteración, una vez acabada se realiza una reunión en la que se debate si esta iteración ha cumplido con los requisitos establecidos. De ser así se plantea la siguiente, de no serlo se revisa, se debaten los aspectos que no han cumplido con lo esperado y se incluyen las mejoras en la siguiente fase.

Una vez establecida la metodología de trabajo se procedió a establecer las tareas para la primera iteración.

3. Estado del arte.

Una vez aceptada la propuesta del proyecto se tuvo una reunión en la que se dejó claro que esta propuesta viene dada por la necesidad de una interfaz para el doctorado de mi tutora Lucía Arnau Muñoz. El doctorado trata de un sistema distribuido manejado por inteligencia artificial pero carente de una interfaz en condiciones que represente la información del sistema.

Una vez se introdujeron las necesidades de la interfaz mis tutores me proporcionaron, con el fin de llevar a cabo el desarrollo, la API de su sistema. Esta API es la que contiene la llamada que interactúa con los nodos, por tanto, es la que va a proporcionar los datos que serán representados por la interfaz a desarrollar. Por ello para crear la interfaz primero hay que entender los componentes de la API proporcionada.

3.1 Llamada POST

Para entender la estructura de la API hay que tener en cuenta que el sistema distribuido desarrollado consta de tres tipos de nodos: balanceadores, controladores y procesadores. De modo que la API proporcionada está implementada con el fin de recuperar los datos de estos nodos. Para ello se creó la llamada de tipo POST, esta recibe una URL por parámetro. Cuando se realiza esta llamada, dependiendo del tipo de nodo que sea, se realizan una serie de funciones que devuelven los datos de funcionamiento del sistema.

3.1.1 Coordinador

El coordinador es el componente encargado de gestionar los nodos del sistema distribuido y supervisar su estado. Cuando se realiza la llamada POST de un coordinador este coordinador realiza sus tareas internas, que son:

- Supervisión: Envía mensajes AYA para comprobar la actividad de los nodos.
- Gestión de Nodos: Registra o elimina los nodos del sistema según las respuestas de los mensajes de supervisión.
- Sincronización: Envía solicitudes a los balanceadores para actualizar información del sistema.

Una vez realiza sus tareas devuelve un su estado interno en formato SMFP que contiene la información del nodo estructurada de la siguiente forma:

- UID: Identificador único del nodo.
- Type: Tipo de nodo (procesador, balanceador).
- Operation: Operación realizada.
- Data: Información adicional estructurada que contiene el nodo:
 - Información adicional necesaria para la operación. Dentro de este apartado data viene la configuración interna del nodo con sus conexiones.
 - La lista actual de nodos gestionados, incluyendo su nombre, carga, capacidad y URL.

3.1.2 Balanceador

El balanceador es el componente encargado de gestionar la distribución de carga entre los nodos del sistema distribuido y mantener la sincronización del sistema. Cuando se procesa una solicitud POST dirigida al balanceador, este lleva a cabo las siguientes tareas internas:

- Supervisión: Envía mensajes AYA para verificar la actividad del coordinador correspondiente.
- Gestión de Carga:
 - Distribuye las solicitudes de procesamiento entre los nodos según su estado actual y carga.
 - Registra, actualiza o elimina nodos en la lista interna según los mensajes de configuración recibidos por el coordinador.
- Sincronización:
 - Envía mensajes al coordinador principal para actualizar el estado del balanceador y la lista de nodos gestionados.
 - Responde a solicitudes proporcionando información actualizada sobre el balanceador y los nodos supervisados.
- Configuración: Procesa mensajes para actualizar la lista de nodos gestionados o realizar ajustes en la configuración interna.

Al completar sus tareas, el balanceador devuelve un estado estructurado en formato SFMP que incluye:

- UID: Identificador único del mensaje o nodo.
- Type: Tipo de nodo, en este caso, "balanceador".

- Operation: Operación realizada
- Data: Información adicional estructurada que contiene:
 - La configuración interna del balanceador.
 - La lista actual de nodos gestionados, incluyendo su nombre, carga, y URL.

3.1.3 Procesador

El procesador es el componente encargado de recibir, procesar y responder a mensajes relacionados con el sistema distribuido. Cuando se realiza la llamada POST a un procesador este realiza las siguientes tareas:

- Procesamiento de mensajes: El procesador recibe y procesa mensajes, realizando las acciones solicitadas.
- Respuestas de Estado: El procesador responde con un mensaje estructurado, indicando el resultado de las operaciones solicitadas.
- Gestión de Operaciones: El procesador maneja las operaciones que se le solicitan a través de diferentes tipos de mensajes, proporcionando la información o realizando la tarea correspondiente.

Al completar la operación solicitada, el procesador responde con un mensaje estructurado en formato SFMP que incluye:

- UID: Identificador único del mensaje o del nodo que realiza la operación.
- Type: Tipo de mensaje, en este caso, "procesador".
- Operation: Tipo de operación realizada.
- Data: Información adicional relacionada con la operación, que podría incluir:
 - La configuración interna del procesador (como los parámetros y estados actuales).
 - La confirmación de que se ha recibido un mensaje de tipo AYA.
 - El resultado de la operación solicitada (si se ha ejecutado correctamente o no).

4. Objetivos

Al principio del proyecto se estableció como objetivo principal el desarrollo de una interfaz de monitorización para sistemas distribuidos que fuera atractiva, intuitiva y fácil de manejar. Con el objetivo claro se comenzó a desarrollar y a lo largo del desarrollo se decidió que no solo que fuese una interfaz de monitorización sino también de gestión del sistema, dando así más utilidad a la herramienta permitiendo definir directamente los elementos que van a formar parte del sistema y su representación al instante.

Durante el avance de la implementación se definieron los siguientes subobjetivos con el fin de alcanzar el producto deseado:

- Diseño de la interfaz de monitorización
- Diseño de la interfaz de gestión
- Gestión de nodos
- Gestión de las conexiones entre nodos
- Representación de nodos en el grafo
- Representación de nodos en el mapa
- Consulta del estado interno de los nodos
- Apartado de configuración de la interfaz
- Actualización del sistema según su comportamiento

Estos subobjetivos son los que componen el objetivo principal, que es desarrollar la interfaz de monitorización que permita representar la información del sistema en tiempo real y la gestión del mismo.

5. Metodología

En cuanto a la metodología se tuvo en cuenta la planificación decidida anteriormente en la que se estableció la necesidad de revisiones periódicas semanales para que el avance del proyecto fuese de la mejor manera posible.

Teniendo en cuenta esto se llegó a la conclusión junto con mis tutores, José Vicente Berná y Lucía Arnau Muñoz, de llevar a cabo la metodología Scrum. Esto se debe a que esta se caracteriza por su enfoque ágil, permitiendo adaptarse rápidamente a cambios y nuevos requerimientos conforme el proyecto avanza. Se insiste mucho en revisiones constantes que permiten la mejora de un producto, priorizando la flexibilidad y la capacidad de reacción frente a imprevistos.

Para llevar a cabo esta metodología se decidió dividir el proyecto en iteraciones, es decir, fases de desarrollos controlados de una o dos semanas, con una planificación de tareas previa que se acaban revisando al final de la iteración para debatir si se han alcanzado los objetivos planteados. En estas reuniones, se introducen los avances, se verifica que funcionen de manera adecuada y que no afecten de manera negativa a otras partes del desarrollo y si corresponde se plantean mejoras o se aclaran las dudas sobre los apartados mal implementados y el porqué de su mala implementación. Todo esto con el objetivo de pulir todos los aspectos al máximo y con la finalidad de que se comprendan todos los requisitos.

Con esta metodología nos aseguramos de que el proyecto siga una evolución constante y evitamos que los errores de entendimiento de los objetivos o comunicación de ambas partes no tengan un impacto significativamente negativo en el desarrollo del producto.

6. Implementación

En esta parte se explicará el proceso de desarrollo del proyecto mediante la metodología ágil comentada anteriormente. Ya que ésta permitía adaptarse de la mejor manera a la implementación del proyecto.

6.1 Definición del proyecto y mockups

En la primera reunión con mis tutores en la que se presentó la idea del proyecto, me dijeron los requisitos que buscaban y me estuvieron explicando algunas ideas que tenían para el desarrollo de la interfaz. Una vez sentadas las bases estuvimos debatiendo y descartando algunas. Una vez terminado de explicar los aspectos claves y principales se decidió que en esta primera iteración la idea es buscar ideas para la interfaz del sistema distribuido que cumpliera con las características de monitorización que se habían acordado.

6.1.1 Investigación y elección de interfaces de monitorización

Para conseguir una interfaz de monitorización que cumpliera los requisitos se investigaron las principales interfaces de monitorización de sistemas distribuidos para evaluar sus características. Se vieron arquitecturas de tipo: árbol, lista, grafos y timeline con métricas. Una vez hecho el proceso de investigación se procedió a una segunda revisión donde se plantearon las cuatro ideas. La interfaz de lista se descartó por el simple hecho de que se buscaba una interfaz fácil de interpretar y esta no cumplía los requisitos. Por otro lado, la de árbol te obliga a seguir un orden de representación con lo cual limitaba mucho la escalabilidad en la pantalla, así que también fue descartada. De esta manera nos quedamos con dos posibles ideas: el grafo y el timeline.

Como no quedaba muy claro cuál podía ser la mejor opción se decidió desarrollar un par de interfaces simples de la pantalla principal del sistema a ver qué representación se ajustaba mejor y qué aspecto visual ayudaba más a la hora de recibir la información.

6.1.2 Boceto inicial con interfaz timeline

Para esta idea se implementó una interfaz que dejaba registrada las fechas clave con mensajes descriptivos del problema para una fácil interpretación de los eventos del sistema con posibilidad de retroceder y ver el registro de errores en el tiempo (Figura 2).

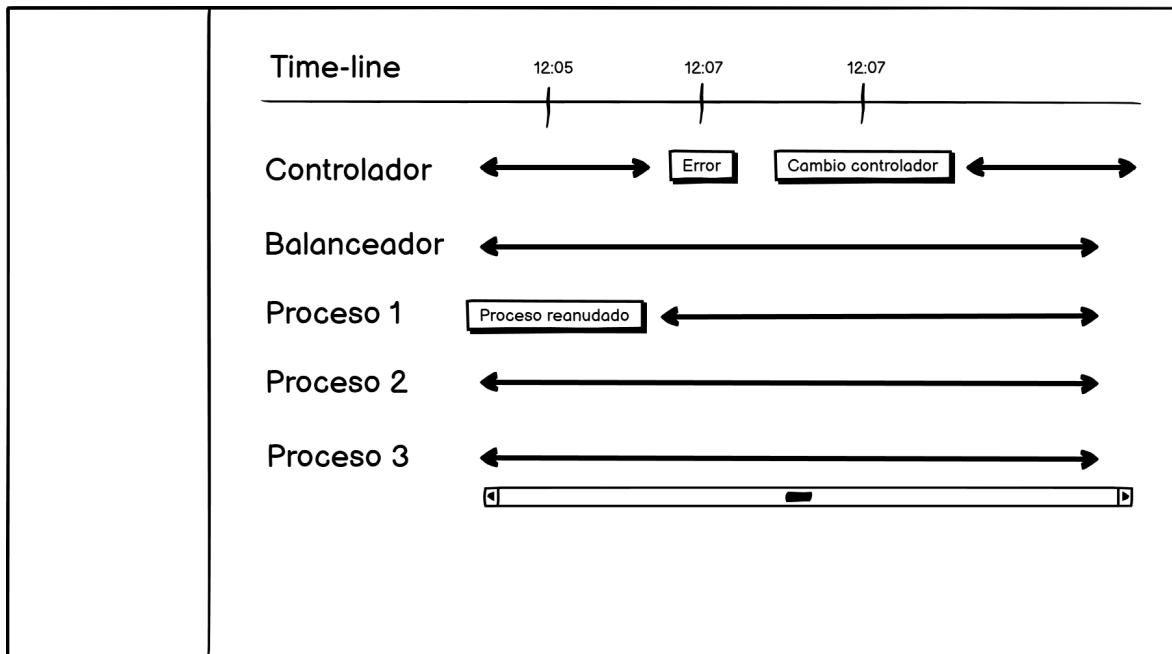


Figura 2. Mockup Interfaz Principal Timeline (Fuente Propia)

6.1.3 Boceto inicial con interfaz grafo

En cuanto a la idea del grafo la interfaz se encargaba de representar los elementos del sistema con sus letras para distinguir los nodos y los colores para identificar si estaba trabajando, apagado o había sufrido un fallo (Figura 3).

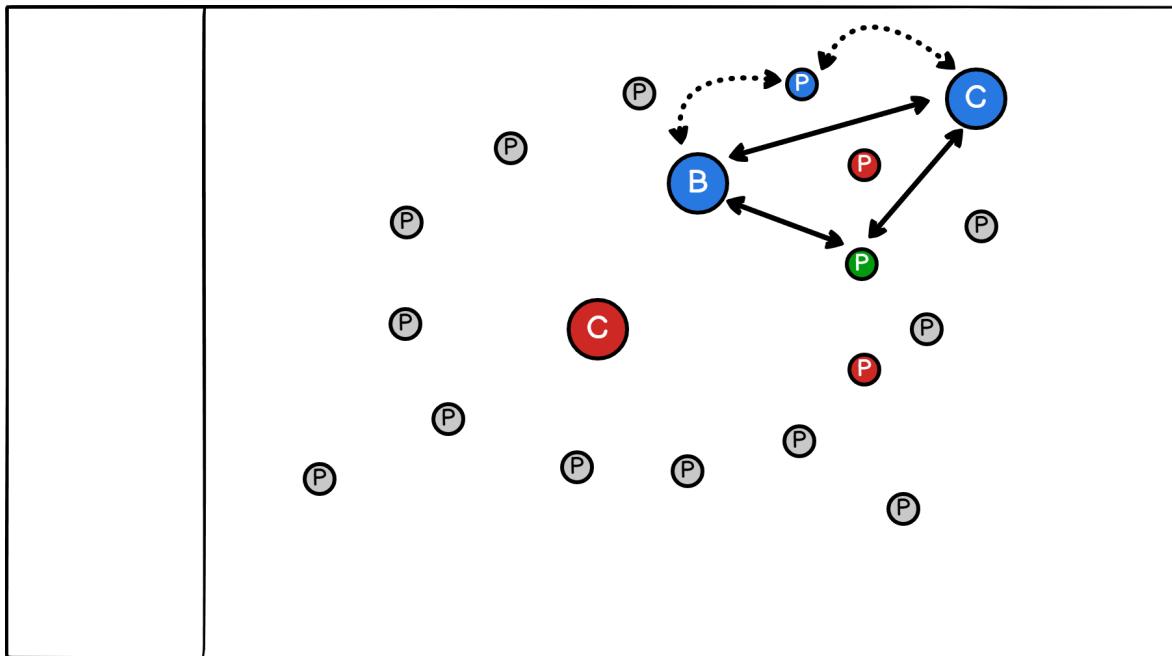


Figura 3. Mockup Interfaz Principal Grafo (Fuente Propia)

Tras el desarrollo de estas dos interfaces simples se procedió a una tercera reunión donde se presentaron las dos ideas y tras debatir sus pros y contras se optó por la idea de la pantalla en forma de grafo. Esto se debía a que en el timeline es difícil representar lo que va pasando todo el rato te quedaría una lista interminable de nodos que no caben en la pantalla todos a la vez y para monitorizar a todos se tardaría mucho, la única forma era representar sólo los nodos con mensajes relevantes como errores. Pero, por otro lado, la visión del grafo permitía una visión general y escalable que almacenaba todos los nodos en un pantalla y mediante eventos visuales se podía identificar el comportamiento de cada nodo del sistema.

De este modo se decidió terminar de diseñar los mockups restantes alrededor de esta idea de interfaz principal en forma de grafo.

6.1.4 Pantalla principal vista en forma de grafo

Teniendo de referencia la vista de grafo anterior se añadió un panel lateral con el que navegar entre la pantalla principal y pantallas de lista de nodos de tipo: controlador, balanceador y procesador. Además, en el caso de que sucediera un error en uno de estos tipos en esta barra lateral tendría un icono de error a la derecha del nombre.

Dentro de la pantalla en forma de grafo los balanceadores y procesadores serían más grandes de tamaño que los procesadores ya que hay menos. Su funcionamiento se distinguiría por colores: azul

significa que están funcionando, verde que están disponibles a la espera de recibir tareas y rojo que han tenido un fallo (Figura 4).

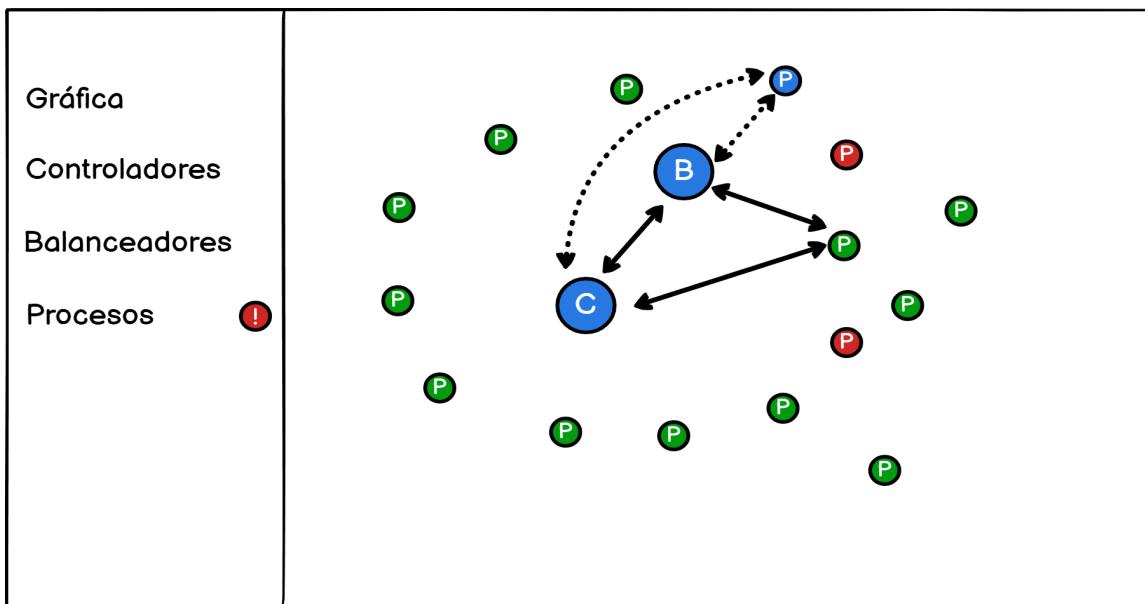


Figura 4. Mockup Interfaz Principal Grafo 2 (Fuente Propia)

En el caso de que un balanceador o controlador sufriera un fallo sus conexiones las tomaría el suplente asignado que se representaría en pantalla y este pasaría a color rojo, además, de que en la lista de su tipo aparecerá el ícono de error (Figura 5).

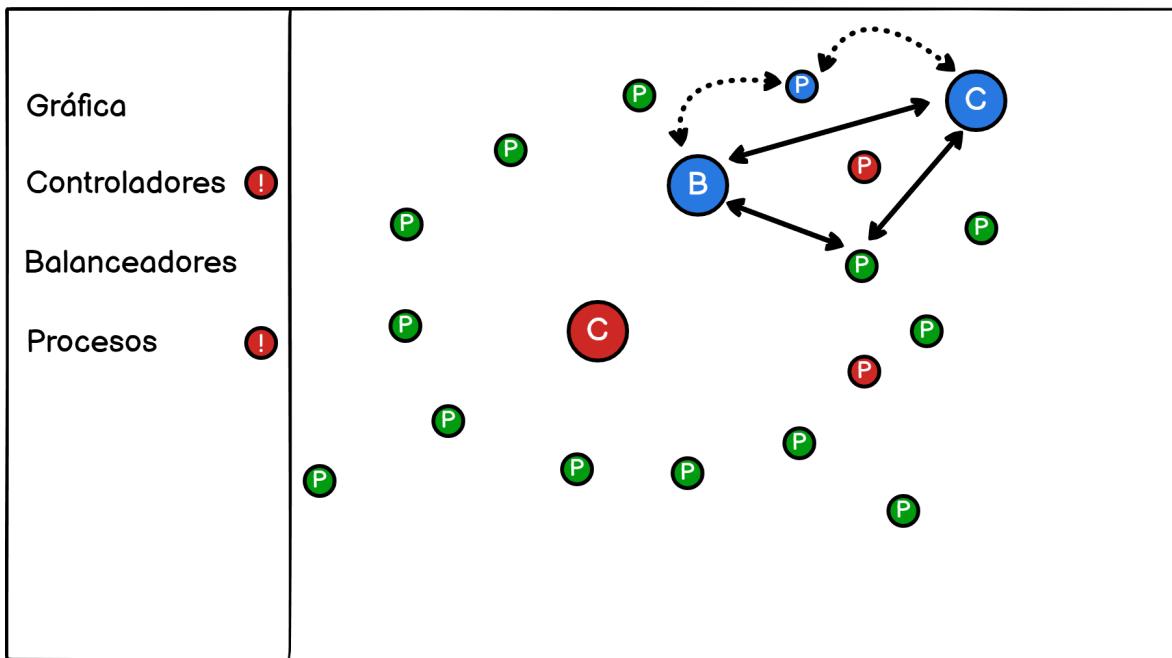


Figura 5. Mockup Interfaz Principal Grafo 3(Fuente Propia)

6.1.5 Pantallas de listas de nodos

En los apartados de listas de la barra lateral se almacenan las listas de nodos según su tipo. La información de cada uno es: su identificador, su estado y su registro de fallos. Para su gestión en las tres pantallas se añade un botón para añadir el tipo de nodo y una barra de búsqueda, además que para cada nodo se tendrá un botón de eliminar, uno de editar, en el caso de los procesadores uno de visualizar el estado interno de lo que procesan y en el caso de que el nodo haya sufrido un fallo que aún no se ha revisado aparecerá un ícono de alarma para consultar la información del error (Figura 6) (Figura 7) (Figura 8).

Gráfica	Lista de Procesos		
	ID Proceso	Estado	Nº Fallos
Controladores	1	Disponible	1
Balanceadores	2	Funcionamiento	0
Procesos	3	Fuera servicio	3

< [1] 2 3 >

Figura 6. Mockup Interfaz Lista de Procesadores(Fuente Propia)

Gráfica	Lista de Balanceadores		
	ID Balanceador	Estado	Nº Fallos
Controladores	1	Reserva	1
Balanceadores	2	Funcionamiento	0
Procesos	3	Fuera servicio	3

< [1] 2 3 >

Figura 7. Mockup Interfaz Lista de Balanceadores(Fuente Propia)

Gráfica	Lista de Controladores			Añadir Controlador	<input type="text"/> search
	Controladores	ID Controlador	Estado	Nº Fallos	
Balanceadores	1		Reserva	1	
Procesos	2		Funcionamiento	0	
	3		Fuera servicio	3	
			< 1 2 3 >		

Figura 8. Mockup Interfaz Lista de Controladores(Fuente Propia)

6.1.6 Pantalla de información de los procesadores

A la hora de visualizar la información de un procesador se podrá acceder clicando en los iconos del grafo de la pantalla principal (Figura 4)(Figura 5) o, en el caso de los procesadores desde el botón visualizar de la lista (Figura 6).

Se abre un modal donde se verá la información del procesador: la fecha de creación, el tiempo de trabajo que lleva, las tareas que ha realizado, la media de tiempo que tarda por tarea y la tarea que está realizando actualmente (Figura 9).

Proceso X
Fecha de creación: 22/10/2023
X

Tiempo de trabajo 300 seg	Tareas realizadas 24	Tarea Actual <small>Pantalla de datos de la tarea</small>
Tiempo/Tarea 12,5 seg		

Figura 9. Mockup Interfaz Información de Procesador(Fuente Propia)

6.1.7 Pantalla de errores de los nodos

Al clicar en el icono de exclamación que puede aparecer en la línea de las listas de nodos (Figura 6) (Figura 7) (Figura 8), se abre un modal que muestra la información del error: el tipo de nodo, la fecha y hora del error y el error (Figura 10) (Figura 11) (Figura 12).

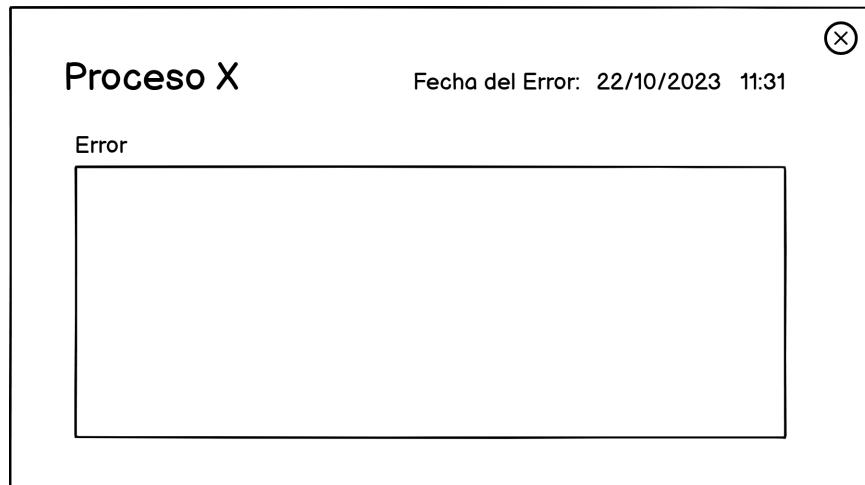


Figura 10. Mockup Interfaz Error de Procesador(Fuente Propia)

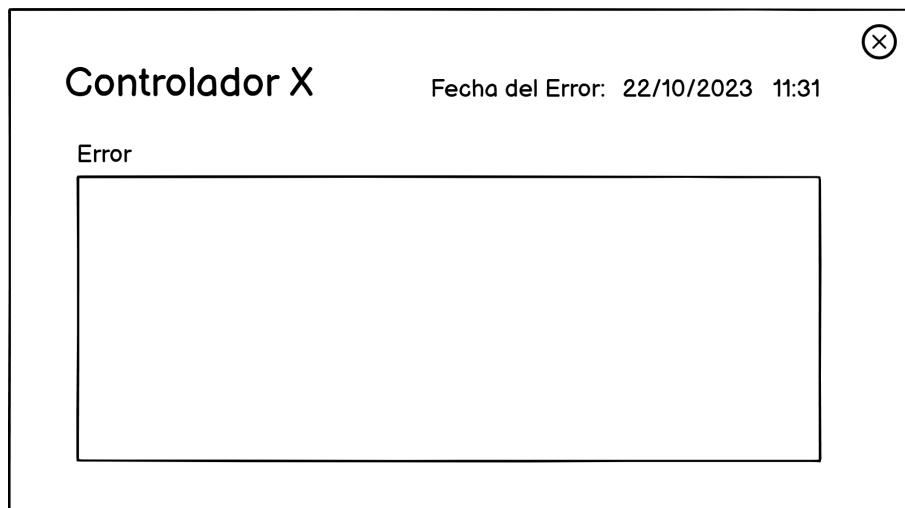


Figura 11. Mockup Interfaz Error de Controlador(Fuente Propria)

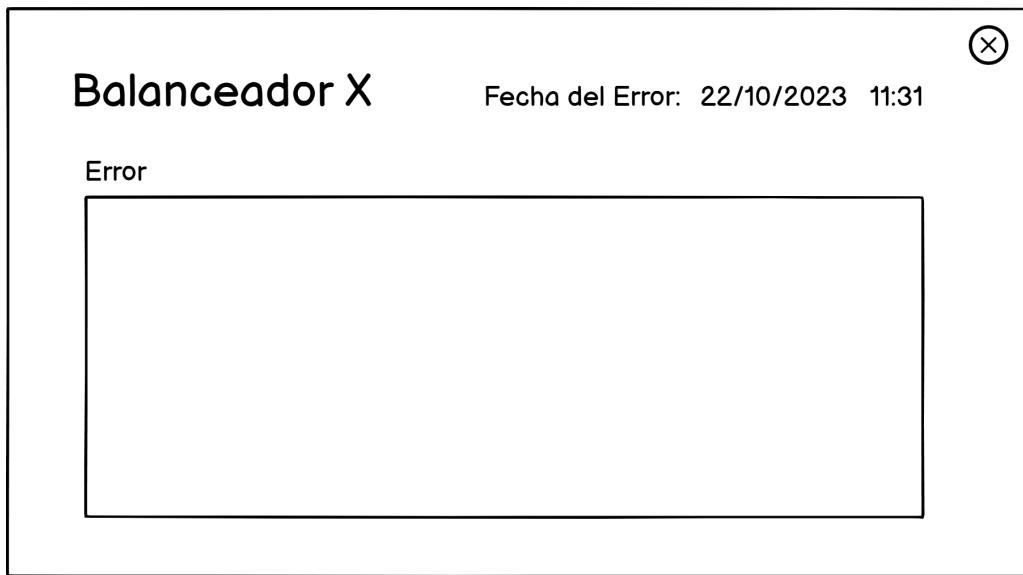


Figura 12. Mockup Interfaz Error de Balanceador(Fuente Propia)

6.2 Revisión y mejora de los mockups

El 2 de septiembre se llevó a cabo una reunión en la que evaluamos las interfaces realizadas en el apartado anterior. Después de evaluar junto con mis tutores los mockups se plantearon varias modificaciones. Ver si la interfaz principal se podía simplificar, juntar las listas de nodos, se indicaron los valores que se deberían de ver en cada nodo de manera definitiva, se propuso que es interesante ver la información de todos los nodos no de solo los procesadores y de juntar las pantallas de error y de información en una, ya que, están relacionadas.

6.2.1 Primera modificación de la pantalla principal

Como se quería simplificar la pantalla principal se eliminó la barra lateral con las opciones y se sustituyó con una vista simple con dos botones estáticos que cambian entre la pantalla del grafo y la de listas de nodos. También decidimos añadir los nodos que estuvieran dados de alta en el sistema pero pudieran estar apagados o ser suplentes en un tono apagado y con sus conexiones para reflejar que están operativos. Y por último, como hay grupos de nodos que pueden ayudarse se reflejó de esta manera para que no hubiera tantas conexiones en la pantalla (Figura 13).

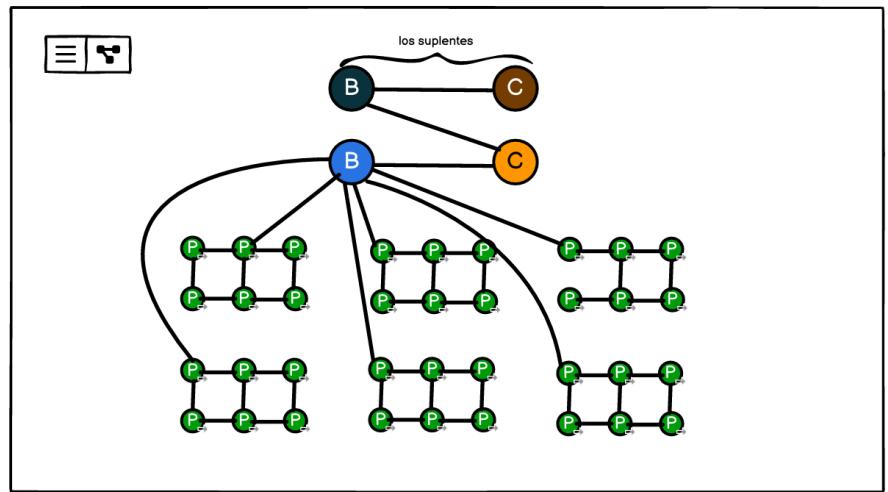


Figura 13. Mockup Interfaz Principal(Fuente Propria)

6.2.2 Primera modificación de la pantalla de lista de nodos

Para esta vista se decidieron agrupar todas las listas anteriores y con los valores: tipo de nodo que es, URL del nodo, puerto al que escuchan y nombre del nodo. Se siguen pudiendo apagar o editar nodos y el botón de “ver información” se ha implementado para todos los tipos de nodo. También quisimos implementar un botón “descargar” por si interesa importar la lista de nodos para otro sistema (Figura 14).

Tipo Nodo	URL	Puerto	Nombre			
Balanceador	https://www.example.com	3000	Nombre_nodo1			
Controlador	https://www.example.com	3001	Nombre_nodo1			
Procesador	https://www.example.com	3002	Nombre_nodo1			

< 1 2 3 >

Figura 14. Mockup Interfaz Lista de Nodos(Fuente Propria)

6.2.3 Primera modificación de la pantalla de información de los nodos

A la hora de representar la información se accede bien clicando en un nodo de la vista de grafo (Figura 13), o bien mediante el botón de “ver información” de la vista de la lista de nodos (Figura 14). De esta manera se abre un modal donde se verán los datos de los nodos.

Los nodos tienen la información de la lista no nodos, un apartado que indica la última consulta a ese nodo y un apartado de estado interno que contiene las tareas y fallos que se han dado desde la última consulta (Figura 15) (Figura 16), además en la vista del controlador también aparecerán el número de procesos activos en ese momento en el sistema (Figura 17).

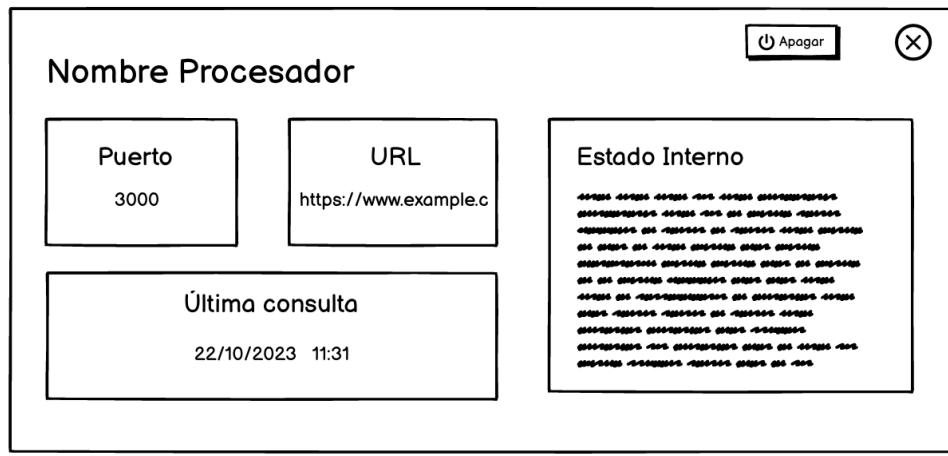


Figura 15. Mockup Información de Procesadores(Fuente Propia)

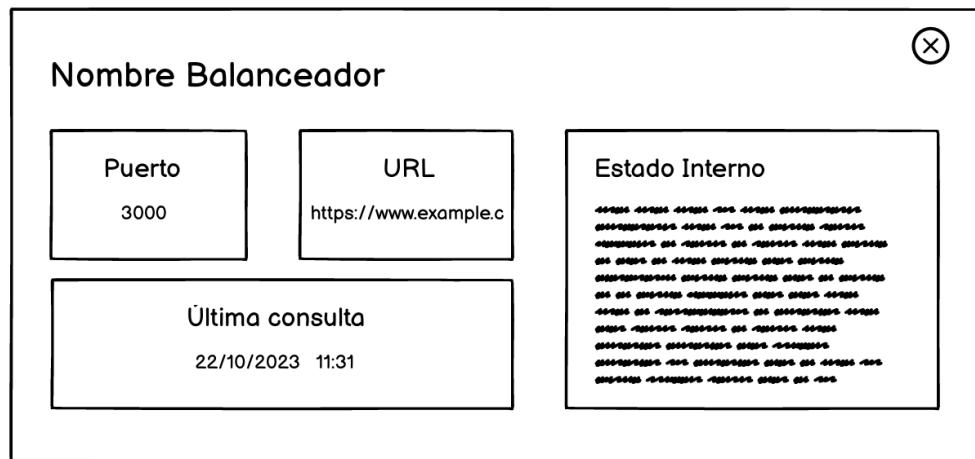


Figura 16. Mockup Información de Balanceadores(Fuente Propia)

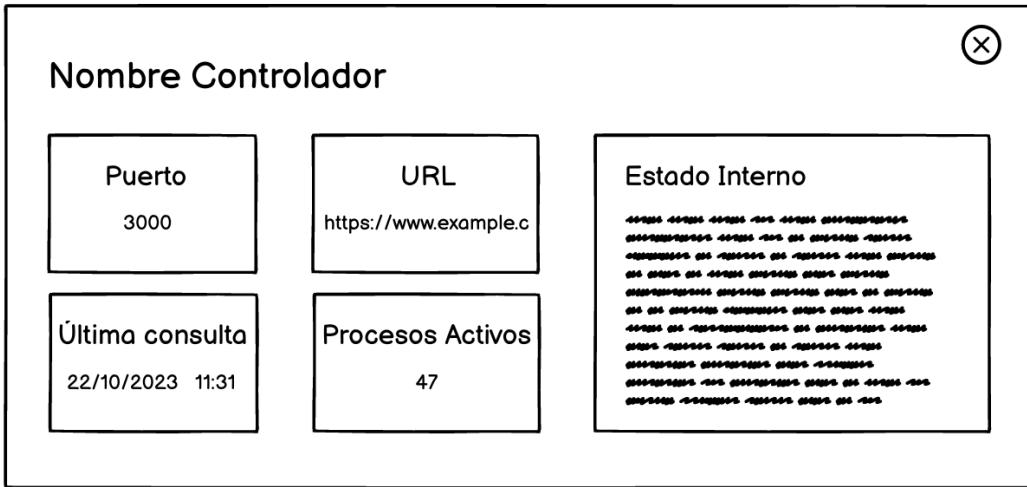


Figura 17. Mockup Información de Controladores(Fuente Propia)

6.2.4 Revisión de los cambios

En la reunión del 13 de septiembre junto con mis tutores volvimos a revisar los mockups. Entendí mal la idea de que los nodos trabajan juntos porque no suponía que no había que reflejar sus conexiones. Por otro lado, se planteó que sería interesante poder ver el sistema con los nodos geolocalizados. Estas revisiones llevaron a las últimas modificaciones de los mockups.

6.2.5 Segunda modificación de la pantalla principal

Con el fin de incluir la opción de ver la geolocalización de los nodos en el grafo y aclarada la idea de que todos los nodos deben tener sus conexiones representadas se añadió el mapa de fondo con los nodos y su geolocalización (Figura 18).

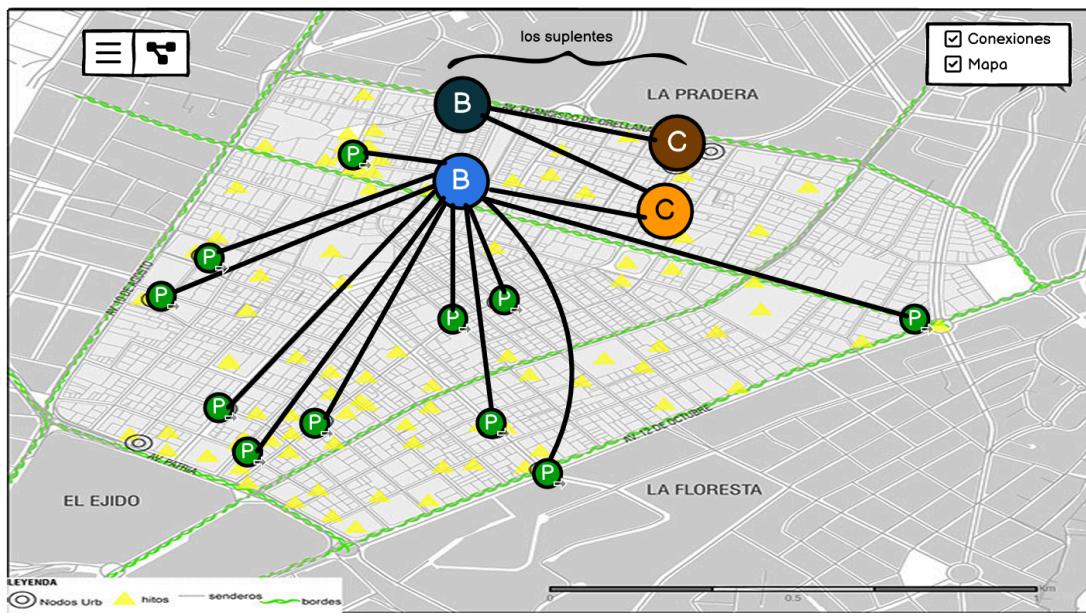


Figura 18. Mockup Pantalla Principal con Conexiones y Geolocalización(Fuente Propia)

Una vez hechas estas modificaciones se pensó que no siempre puede ser interesante ver el mapa y, por otro lado, que tener una visión sin las conexiones de los nodos podría dar una vista más limpia. Para ello, incluimos un “checkbox” con dos opciones que permiten “habilitar” o “deshabilitar”: uno, la vista del mapa con las geolocalizaciones(Figura 19) y dos las conexiones de relación de los nodos (Figura 20).

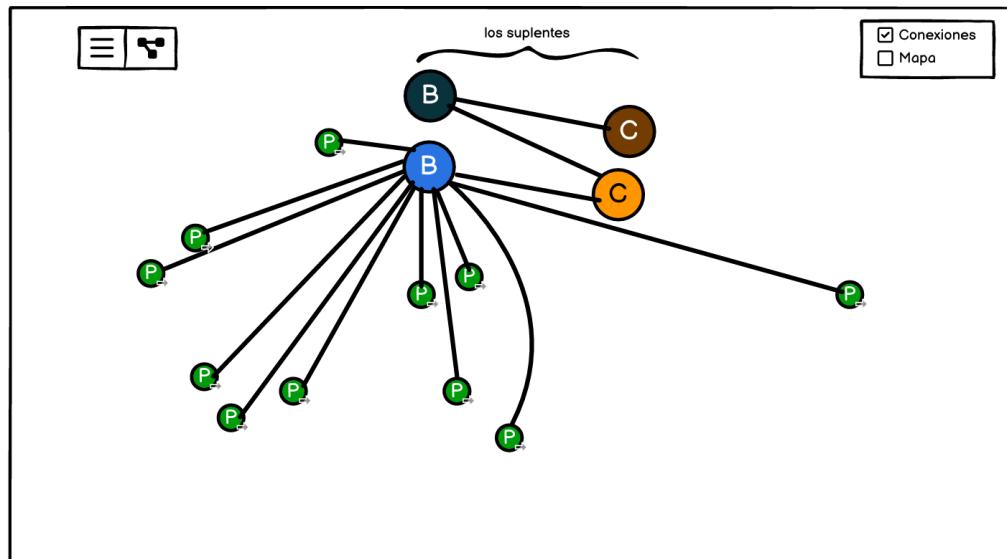


Figura 19. Mockup Pantalla Principal con Geolocalización Deshabilitada(Fuente Propia)

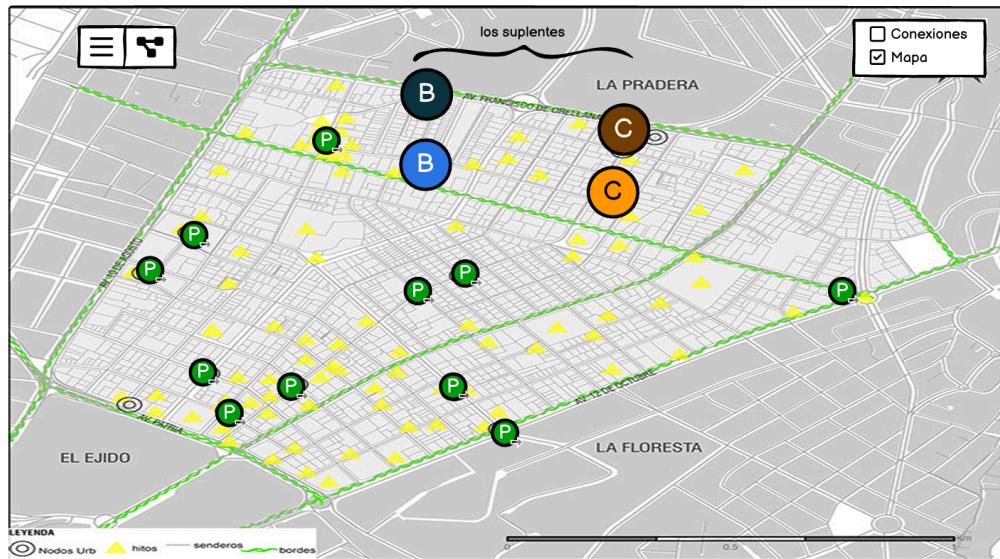


Figura 20. Mockup Pantalla Principal con Conexiones Deshabilitadas(Fuente Propia)

6.2.6 Segunda modificación de la pantalla de lista de nodos

Para esta interfaz se añadió el botón de “añadir nodo” junto al de descarga, ya que ese se olvidó en las anteriores revisiones, y la geolocalización como información del nodo (Figura 21).

		Añadir nodo	Descargar	search
Tipo Nodo	URL	Puerto	Nombre	Geolocalización
Balanceador	https://www.example.com	3000	Nombre_nodo1	Ubicación1
Controlador	https://www.example.com	3001	Nombre_nodo1	Ubicación1
Procesador	https://www.example.com	3002	Nombre_nodo1	Ubicación1

< [1] 2 3 >

Figura 21. Mockup Pantalla de lista de Nodos(Fuente Propia)

6.2.7 Segunda modificación de la pantalla de información de los nodos

Por último, para las pantallas modales de información de los nodos se añade el campo de la geolocalización a los campos de: url, puerto, nombre, última consulta, estado interno (Figura 22) (Figura 23) y en el caso del controlador los procesos activos del sistema (Figura 24).

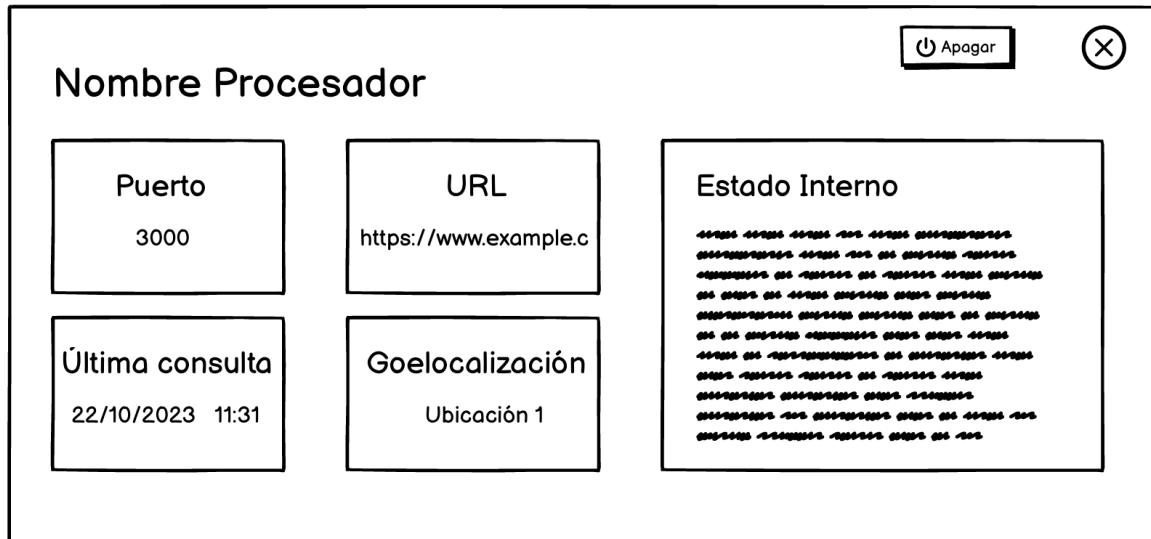


Figura 22. Mockup Información de Procesadores(Fuente Propia)

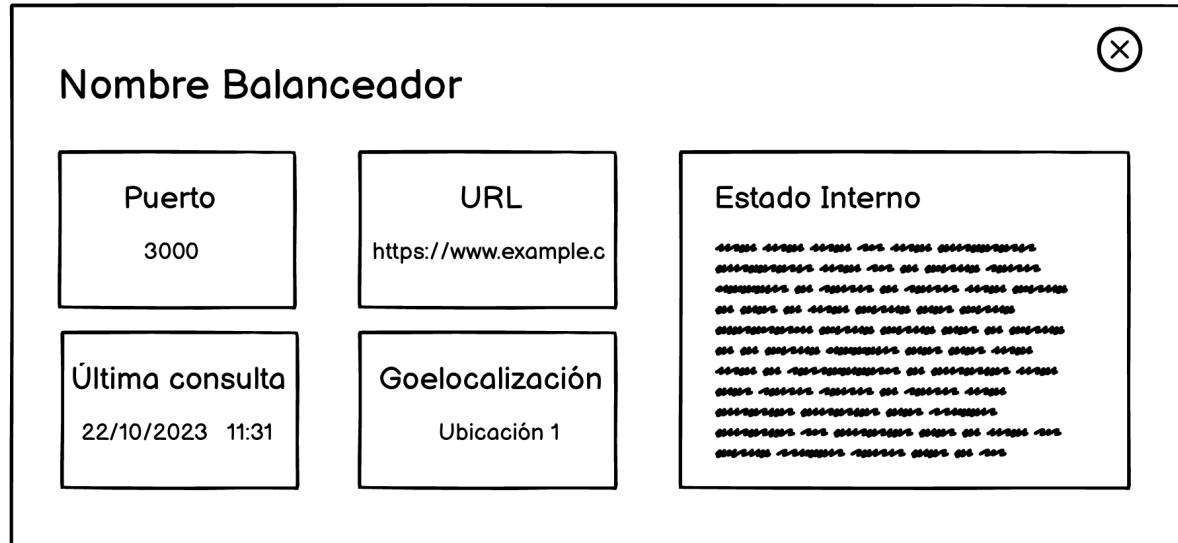


Figura 23. Mockup Información de Balanceadores(Fuente Propia)

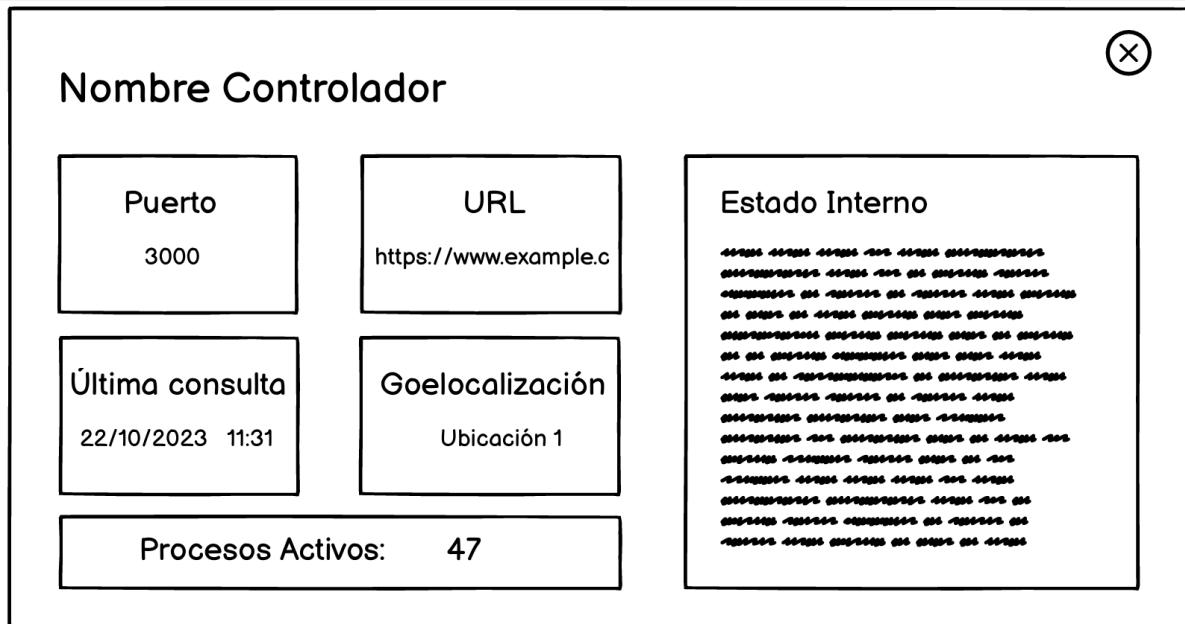


Figura 24. Mockup Información de Controladores(Fuente Propia)

6.2.8 Última revisión de los mockups

Al terminar los mockups fueron presentados al Product Owner en una reunión el 24 de septiembre. El único apunte que hicimos fue que para distinguir mejor los nodos, a parte de distinguir que tengan colores diferentes cuando están de suplentes o apagados, en este caso sean sus colores mezclados con gris, y cuando de fallos, rojos para balanceadores y controladores, y para los procesadores como si estuvieran apagados (Figura 25).

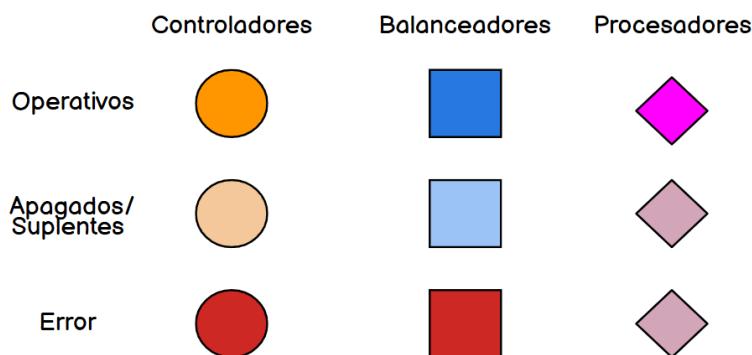


Figura 25. Paleta de Colores para Eventos del Grafo(Fuente Propia)

6.3 Preparación del entorno de desarrollo

Una vez terminada la fase de diseño de mockups inicia la preparación del entorno donde se va a desarrollar el proyecto. En esta parte se tiene como objetivo configurar todos los aspectos necesarios del entorno de trabajo.

6.3.1 Preparación del repositorio y estructura del proyecto

Para preparar el entorno de desarrollo lo primero fue la creación de un repositorio en GitHub[1], llamado TFG_MFP, para una mejor gestión de versiones del código. La estructura que se utilizó para el repositorio fue de tres tipos de ramas: “main”, “develop” y “feature”. En la rama “main” solamente se aloja funcional y revisado, en “develop” es donde se sube el código desarrollado que funciona pero no está revisado y en las ramas “feature/tarea_que_se_realiza” se implementan las nuevas funcionalidades que se quieren añadir al proyecto.

En cuanto a la estructura de directorios del proyecto se dividió en dos carpetas “frontend” y “backend”. La carpeta de “frontend” contiene la lógica de la plantilla Fuse[2] de Angular donde se va a la interfaz. La idea de usar una plantilla para el desarrollo fue en consenso con mis tutores con el objetivo de ahorrar tiempo de desarrollo en el CSS. Por otro lado, la carpeta “backend” contiene la API encargada de detectar la inicialización de nodos proporcionada por mis tutores en la carpeta “v2” y la API de gestión de nodos contenida en la carpeta “miback”.

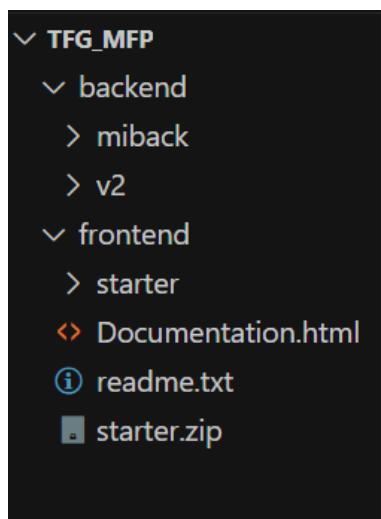


Figura 26.Organización del entorno en Visual Studio Code (Fuente Propia)

Para el desarrollo del proyecto se optó por el editor de Visual Studio Code [3] gracias a que proporciona una conexión directa con el repositorio desde su interfaz y gestiona los cambios desde su terminal. Además, cuenta con extensiones muy útiles como GitGraph[4] que te permite tener un muy buen control de las ramas.

Una vez se decidió usar Visual Studio Code procedimos a la instalación de todas las herramientas que vamos a utilizar, principalmente, en el entorno. En cuanto al frontend vamos a necesitar Aguilar CLI[5], y que la plantilla que vamos a usar trabaja con Angular, en su versión 17.0.1 y Typescript, que es el lenguaje que maneja Angular, en su versión 5.4.2. Para el backend, vamos a necesitar Node.js[6], donde se va a desarrollar la API, versión 20.17.0, el paquete de instalación npm en su versión 10.5.0 y la herramienta de nodemon, la cual revisa los cambios en los archivos y reinicia el programa sin necesidad de guardar los cambios, en su versión 3.1.7.

6.3.2 Configuración general de la plantilla de FUSE para Angular

Después de tener ya el entorno de desarrollo se comenzó la configuración de la plantilla de Fuse proporcionada por mis tutores.

La plantilla Fuse es una plantilla de interfaz de usuario (UI) basada en Angular hecha para ahorrar tiempo en la configuración y diseño de la interfaz. Ofrece una estructura flexible, componentes preconstruidos, además de, librerías como Angular Material Components[7] con sus respectivos componentes y TailwindCSS[8] para el CSS.

Una vez estudiada la documentación de la plantilla y entendido el funcionamiento de esta se procedió a su configuración de manera que se asemejara a la interfaz diseñada en los bocetos. La plantilla proporciona una gran variedad de interfaces sobre las que poder trabajar para el desarrollo de tu proyecto según tus necesidades. Como la interfaz de nuestros bocetos se divide en dos pantallas y ninguna tiene ni cabecera ni barra lateral se decidió elegir el layout “Empty” ya que el resto de plantillas están más enfocadas a la administración y sus elementos no los necesitamos en nuestra interfaz.

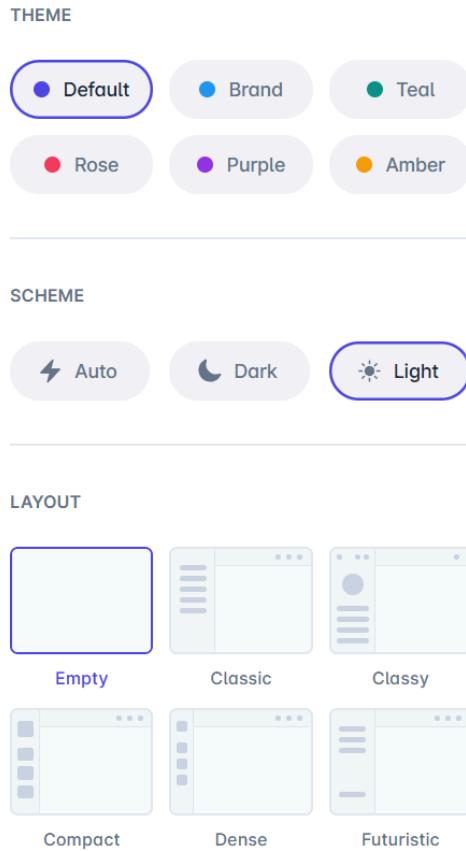


Figura 27. Elección de la interfaz de la plantilla Fuse (Fuente:
<https://angular-material.fusetheme.com/>)

6.3.3 Creación de componentes de la página grafo

Una vez terminada la configuración general de la plantilla se puso en marcha el desarrollo de todos los componentes del proyecto. Se comenzó por la pantalla principal, es decir, la que va a contener el grafo de monitorización. Como el grafo es una implementación futura, se procedió a implementar el resto de componentes de la pantalla.

Primeramente, se desarrolló el componente común en las dos pantallas que es el “toggle button” que permite cambiar entre las dos pantallas de la interfaz. Para ello, en la carpeta “frontend/starter/src/app/modules/admin” se crearon los componentes “lista” y “grafo”, con los comandos “ng g c lista –skip-tests” y “ng g c grafo –skip-tests” que se van usar para hacer referencia a las dos vistas de la interfaz.

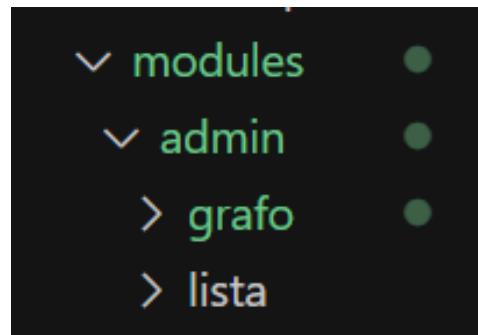


Figura 28. Creación de componentes Grafo y Lista en Visual Studio Code (Fuente Propia)

Una vez creados los componentes, haciendo use de la librería de Angular Materials, creamos un componente “mat-toggle-button-grupo” con dentro dos componentes de tipo “mat-toggle-button” en sus respectivos “.component.html”:

Una vez creados los componentes se desarrollaron las funciones “irALista()” e “irAGrafo()” las cuales redirigen a las rutas del navegador de la plantilla Fuse. Por último, estas rutas se deben añadir en el archivo: “frontend/starter/src/app/mock-api/navigation/data.ts” para que la plantilla las redirija automáticamente.

```
1  export const horizontalNavigation: FuseNavigationItem[] = [
2      {
3          id   : 'grafo',
4          title: 'Grafo',
5          type : 'basic',
6          icon  : 'heroicons_outline:chart-pie',
7          link  : '/grafo'
8      },
9      {
10         id   : 'lista',
11         title: 'Lista',
12         type : 'basic',
13         icon  : 'heroicons_outline:list-bullet',
14         link  : '/lista'
15     }
16 ];
17
```

Figura 29. Rutas de los componentes principales en la plantilla (Fuente Propia)

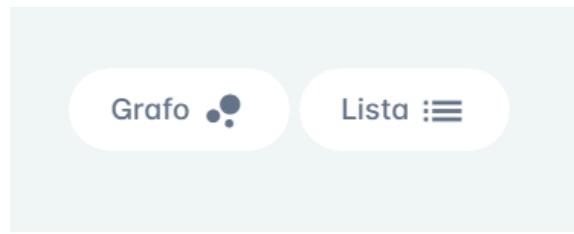


Figura 30. Componente “Button toggle” (Fuente Propria)

Por otro lado, para el checkbox que nos permite activar o desactivar las funciones de geolocalización de los nodos y las conexiones entre los nodos. Para ello usamos dos componentes de tipo “mat-checkbox”.

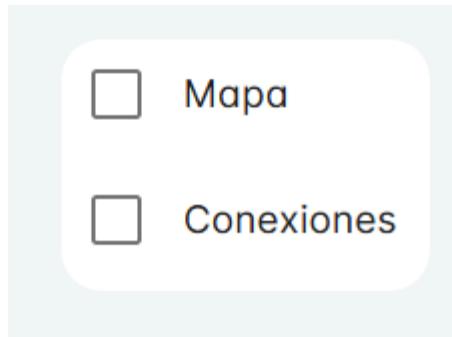


Figura 31. Componente “Checkbox” (Fuente Propria)

Finalmente, para dejar los dos componentes a la misma altura a modo de cabecera se metieron dentro de un elemento div de html cada uno y estos dos elementos en un div con clase cabecera y mediante CSS en el archivo “styles.scss” de la plantilla se obtuvo el siguiente resultado:

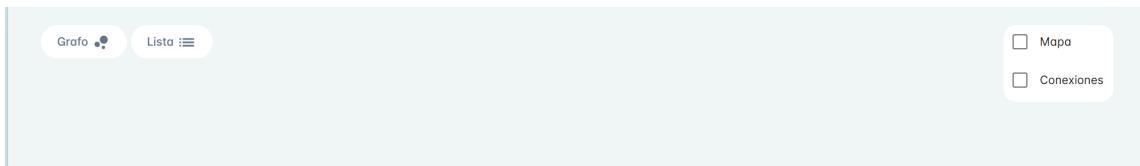


Figura 32. Cabecera de la Página Grafo (Fuente Propria)

6.3.4 Creación de componentes de la página lista

Una vez finalizados los componentes de la página de grafo se procedió a desarrollar los de la página lista. El acceso a la vista “lista” y su componente de redirección, el “mat-toggle-button”, se crearon en el apartado anterior así que solo quedaban por implementar los componentes internos de la página. Estos componentes son: el buscador, el botón de descargar, el botón de añadir nodo y la lista de nodos.

En el caso del buscador, el botón de descargar y el botón de añadir nodo se implementó un elemento div para colocarlos en línea en la parte de arriba de la “lista” con CSS. Dentro de este elemento de HTML5 se implementaron los componentes mediante HTML5 y la librería de angular Materials. Para el nombre de la lista se usó un elemento span, para los botones de añadir nodo y descargar se implementó un botón al que se le aplicó la característica de “mat-fab extended”, la cual nos permite que al agregar un “mat-icon” dentro del elemento se vean tanto el texto como el icono agregado. Finalmente, para el buscador se usó un input de tipo search.

Por otro lado, para el componente “lista” se creó una tabla con la característica “mat-table” cosa que nos permite gestionar los datos de manera fácil y agregarle elementos como “paginación” y “ordenación” de manera sencilla. La “lista” contará con las siguientes columnas: “tipo_nodo”, “url”, “puerto”, “nombre”, “geo (geolocalización)” y ”actions”. Para ellos se implementó un “ng-container” donde dentro se crearon un th para el título de la columna y un “td” para los valores. En el caso de los cinco primeros se implementó un “DataSource” con datos de prueba con la misma estructura que seguirá la base de datos de la API y se le pasó como fuente de datos a la tabla. Por otro lado, la columna “actions” contiene las acciones que se podrán hacer sobre cada fila de la tabla. Dentro del elemento td de esta columna se crearon tres botones con un elemento “mat-icon” dentro y la característica “mat-mini-fab”, la cual solo deja que se muestre el icono que se ha elegido. Finalmente, se desarrolló una paginación para gestionar todos los nodos que pueda haber en la base de datos en un futuro. Para ello se creó un elemento de tipo “mat-paginator” al cual se le pasó como parámetro “length” la longitud del “DataSource” que a futuro se conectará con los datos de la base de datos y no con los de prueba.

Tipo Nodo	URL	Puerto	Nombre	Geolocalización	Actions
controlador	localhost:3000	3000	Controlador 1	X	
procesador	localhost:3001	3001	Procesador 1	Y	
balanceador	localhost:3002	3002	Balanceador 1	Z	

Items per page: 5 1 - 5 of 100 < >

Figura 33. Componente de la Página Lista (Fuente Propia)

6.3.5 Creación de componentes modales

Al ya tener las dos páginas principales del sistema quedaban por realizar los componentes modales de la interfaz. Por un lado, está el que muestra información del nodo seleccionado y, por el otro, está el modal del formulario del botón de añadir nodo.

Para la creación de estos componentes se creó una carpeta nueva llamada “dialogs” en la ruta “frontend/starter/src/app/modules/admin” junto a los componentes “lista” y “grafo”. En esta carpeta almaceno únicamente los archivos HTML de los modales. Gracias a que la plantilla de Fuse utiliza “StandAlone Components”, puedo desarrollar componentes independientes sin la necesidad de declararlos en un módulo. Esto me permite implementar la funcionalidad directamente en los componentes “grafo” y “lista”, y vincularlos a los HTML de la carpeta “dialogs”, evitando la creación de componentes adicionales y simplificando el desarrollo.

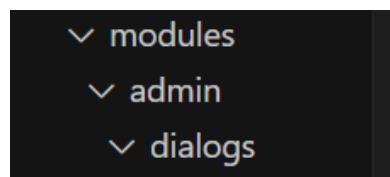


Figura 34. Creación de la carpeta Dialogs en Visual Studio Code (Fuente Propia)

Una vez creada la carpeta procedí a la implementación del modal que muestra la información del nodo. Como la única diferencia de información entre los diferentes tipos de nodo es que el

controlador tiene un campo más que es el de “Procesos Activos” simplemente desarrollé un HTML, ya que, cuando haga la API podré gestionar la aparición de ese campo con un “*ngIf” del elemento que le pase a la función para recuperar los datos. Una vez se tuvo esto en cuenta se desarrolló el código del modal compuesto por: un elemento div que contiene un h1 con el nombre del nodo y un botón para cerrar a modo de cabecera del modal, y un componentes “mat-dialog-content” dividido en dos partes. La primera parte es un grid para los valores del nodo, que son: “puerto”, “url”, “última consulta”, “geolocalización” y “procesos activos”, en caso de ser un controlador, y un “ng-container” para lo que sería el “Estado Interno” del nodo.

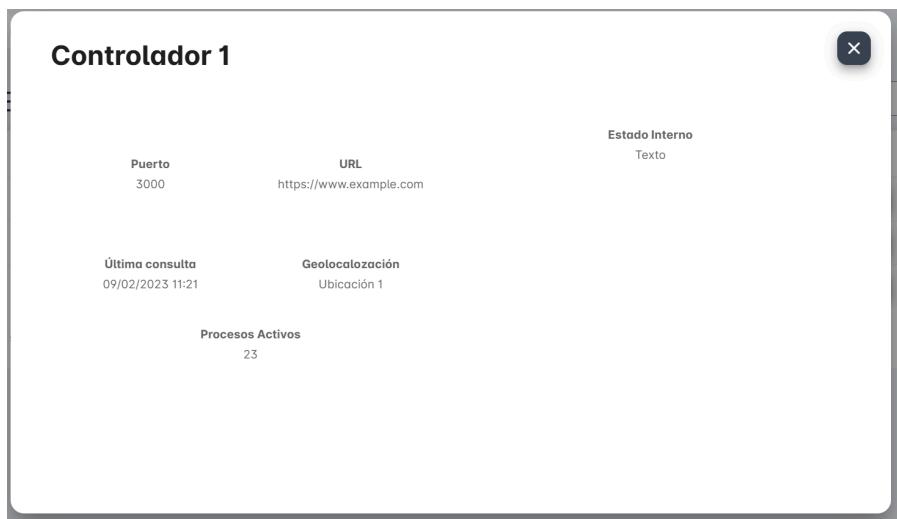


Figura 35. Modal de la información de un nodo (Fuente Propia)

Para el modal de añadir nodo el código se compone de un elemento div que contiene un h1 que indica la función de la pantalla Añadir Nodo y un botón para cerrar a modo de cabecera del modal. Debajo de la cabecera se creó un “mat-dialog-content” que se dividió en un componente de tipo “mat-grid-list” de dos columnas donde dentro de cada espacio del grid se implementó un “ng-container” con un componente “mat-form-field” para cada campo del formulario. Por último, se añadió un botón que se encargará de realizar la petición.

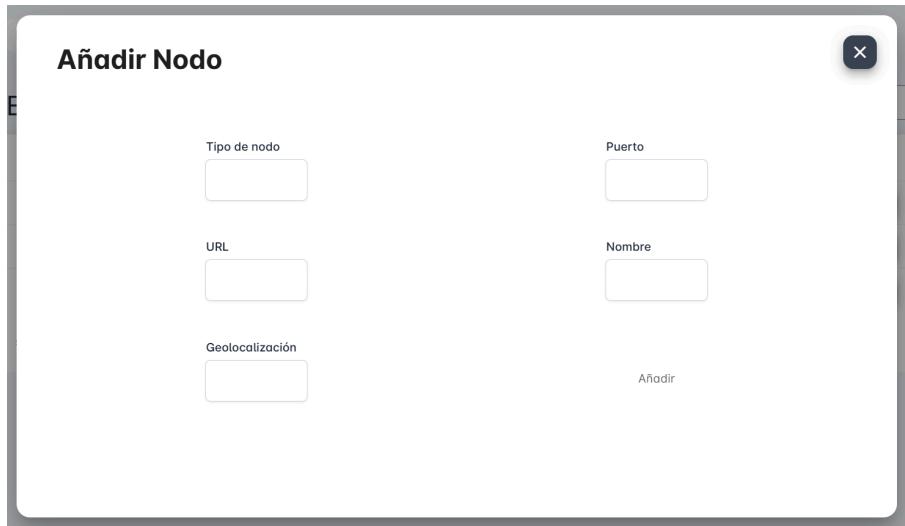


Figura 36. Modal de Añadir Nodo (Fuente Propia)

Después de tener los dos tipos de modales creados se desarrolló la lógica para acceder a ellos mediante Angular Standalone Components. Gracias a esta funcionalidad creamos dos componentes nuevos en el archivo “lista.component.ts” e importamos tanto las librerías de Angular Materials utilizadas como la librería de CommonModule para las directivas estructurales que usaremos como “ngIf”, “ngFor”, etc. Una vez importadas las librería creamos las funciones “dialogo()” y “anadirNodo()” que se encargan de abrir respectivamente el componente de su modal.

6.3.6 Revisión de la estructura de la interfaz y cambios

Al terminar de desarrollar la base del frontend se realizó una revisión con mis tutores el 10 de octubre. En esta reunión se revisó el frontend y una vez se le dió el visto bueno se organizó el siguiente desarrollo. No obstante, hablando sobre la implementación de las funcionalidades se hizo una pequeña corrección. Cuando se diseñaron los mock-ups de los bocetos hubo un malentendido y lo que eran los botones de acción de cada fila de la lista debían ser para: mostrar su información, dejar el nodo visible en el grafo y editar el nodo. Pero por la confusión se implementaron los diseños para: ver la información, apagar el nodo, es decir, quitarlo del sistema, y editar el nodo. Como vemos en el segundo botón no coincidían las ideas, por tanto, una vez aclarada su funcionalidad se planteó un cambio en los iconos de los botones para representar mejor su funcionalidad. Este cambio consistía en cuando el nodo estuviera visible tuviera el símbolo de un ojo en el segundo botón y cuando no fuera visible un ícono de un ojo tachado. Por otro lado, al usar

el símbolo del ojo en este segundo botón se acordó para el primero, que es el de representación de la información, usar el icono de abrir pestaña.

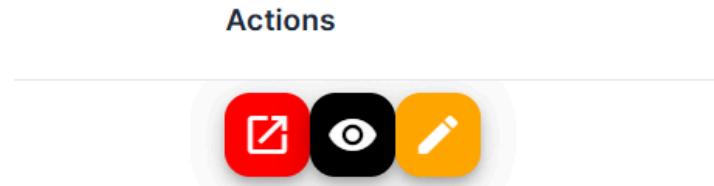


Figura 37. Rediseño de los botones de la lista (Fuente Propia)

6.4 Creación de la base de datos y desarrollo de la API

6.4.1 Creación de la base de datos

Una vez terminados los elementos del frontend y revisados con datos estáticos se comenzó el desarrollo de la API para agregar datos dinámicos al frontend. Para ello lo primero fue crear la base de datos que almacena los datos de los nodos que con lo que vamos a trabajar en la API.

En cuanto a la base de datos se eligió usar una base de datos relacional, ya que en principio, solamente consta de una tabla “nodos”. Para la gestión de esta se optó por PHPMyAdmin[9] en local junto al software de XAMPP [10] para conectar la aplicación con la API. Después se creó la tabla “nodos”, usando la interfaz de PHPMyAdmin, con las siguientes columnas: “id” de tipo integer que incrementa solo, “tipo_nodo” de tipo varchar(100), “nombre” de tipo varchar(100), “url” de tipo varchar(100), “puerto” de tipo varchar(50) y “geolocalizacion” de tipo varchar(100). Este último campo está de manera provisional hasta que se decida cómo se obtendrá la latitud y longitud.

Figura 38 muestra la estructura de la tabla “nodos” en PHPMyAdmin. La tabla tiene seis columnas: #, Nombre, Tipo, Cotejamiento, Atributos, Nulo, Predeterminado, Comentarios, Extra y Acción. Los datos son:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int(11)			No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	tipo_nodo	varchar(100)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
3	nombre	varchar(100)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
4	url	varchar(100)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
5	puerto	varchar(50)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
6	geolocalizacion	varchar(100)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más

Al pie de la tabla hay botones para: Seleccionar todo, Examinar, Cambiar, Eliminar, Primaria, Único, Índice, Espacial, Texto complejo, Agregar a columnas centrales y Eliminar de las columnas centrales.

Figura 38. Estructura de la tabla “nodos” en PHPMyAdmin (Fuente Propia)

6.4.2 Preparación del entorno del backend

Tras crear la base de datos que vamos a conectar a las llamadas de la API se siguió con la preparación del entorno del backend. Para empezar en la carpeta “miback” dentro de la carpeta “backend” del entorno se ejecutó la orden “npm init” para crear un proyecto de Node.JS. Después se instaló el framework de Express.JS [11] en la versión 2.21.0 para facilitar el desarrollo de la API. Después se creó el archivo de “index.js” para configurar y añadir las rutas principales y se instaló la librería “dotenv” [12], para guardar las variables de entorno en su archivo “.env”. Una vez creados estos archivos se configuró la variable de entorno “PORT” a 8080 que la aplicación escuche en ese puerto.

Posteriormente, se creó la estructura de directorios para la API la cual consta de: “controllers”, “database”, “middleware”, “models” y “routes”. En la carpeta “controllers” se encuentran las funciones CRUD (GET, POST, PUT Y DELETE), en “database” el archivo de conexión con la base de datos, en “middleware” se encuentra la lógica de validación de las peticiones, respecto a la carpeta “models” se encuentra la estructura de los datos de las tablas, para ello se instaló Sequelize [13] en su versión 6.37.4, un ORM (Object-Relational Mapping) de Node.js que nos permite interactuar con bases de datos relacionales sin tener que hacer sentencias completas SQL directamente y, por último, la carpeta “routes” donde están las declaraciones de las rutas de cada tipo de petición.

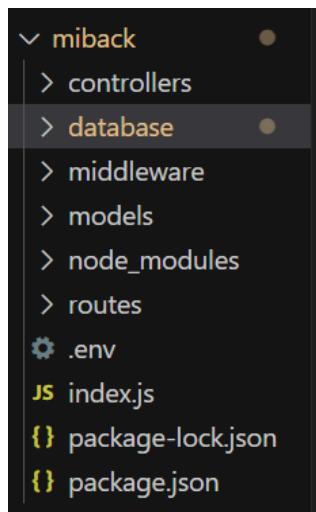


Figura 39. Estructura de directorios del backend de la API en Visual Studio Code (Fuente Propia)

6.4.3 Configuración de la conexión con la base de datos

Con la estructura de carpetas de la API creada y el proyecto de Node.js escuchando configurado para el puerto 8080, se comenzó el desarrollo para la petición GET ALL, que recupera todos los nodos y sus datos de la base de datos.

Lo primero fue crear y configurar el archivo de conexión de la base de datos de la carpeta "database". Lo primero fue crear y configurar el archivo de conexión a la base de datos en la carpeta "database". Para ello, se definieron las siguientes variables de entorno:

- **PORTBD**: almacena el puerto en el que XAMPP escucha para las conexiones de base de datos locales, que por defecto es el puerto 3306.
- **BD**: contiene el nombre de la base de datos.
- **PWDDBD**: almacena la contraseña de acceso a la base de datos.

Después teniendo en cuenta que usamos los modelos de Sequelize y los datos van a pasar por su estructura hay que hacer la conexión creando una conexión con esta lógica.



```
1 const { Sequelize } = require('sequelize');
2
3 const sequelize = new Sequelize(process.env.BD, 'root', process.env.PWDDBD, {
4   host: 'localhost',
5   dialect: 'mysql',
6   port: process.env.PORTBD,
7 });
8
9 module.exports = sequelize;
```

Figura 40. Código de conexión con la base de datos usando Sequelize (Fuente Propia)

6.4.4 Desarrollo de la petición GET ALL

Una vez establecida la conexión, se creó el archivo "nodo.js" en la carpeta "models", el cual define la estructura de la tabla "nodos" utilizando Sequelize. Este archivo servirá como referencia en las peticiones, permitiendo interactuar con los datos de la tabla de manera más sencilla. Después se definió la ruta principal de las peticiones en el "index.js", que es "http://localhost:8080/api" y se creó el archivo "nodo.js" en la carpeta "routes" donde se definió la ruta de la llamada:

- **Ruta GET ALL**: "http://localhost:8080/api/nodos".

Una vez creada la ruta se desarrolló la lógica de la función que trataría esta petición en el archivo “nodo.js” creado en la carpeta “controllers”. Lo primero fue importar la clase Nodo definida en el archivo “nodo.js” de la carpeta “models” para tratar con los datos de la tabla, una vez importada la función realizaba la petición Sequelize “Nodo.findAll()”. En caso de que la petición sea correcta devuelve todos los nodos de la base de datos, sino se controla el error con un try catch y devuelve el mensaje de error:

- **Status:** 500.
- **Mensaje de Error:** “Error en get all nodos”.

Tras el desarrollo de la lógica de la petición GET ALL se inició el proyecto mediante el comando “npm start” configurado en el archivo “package.json” del proyecto con la función “node index.js” y usando la herramienta Postman [14] se comprobó el funcionamiento de la petición antes de incorporarse al servicio del frontend.

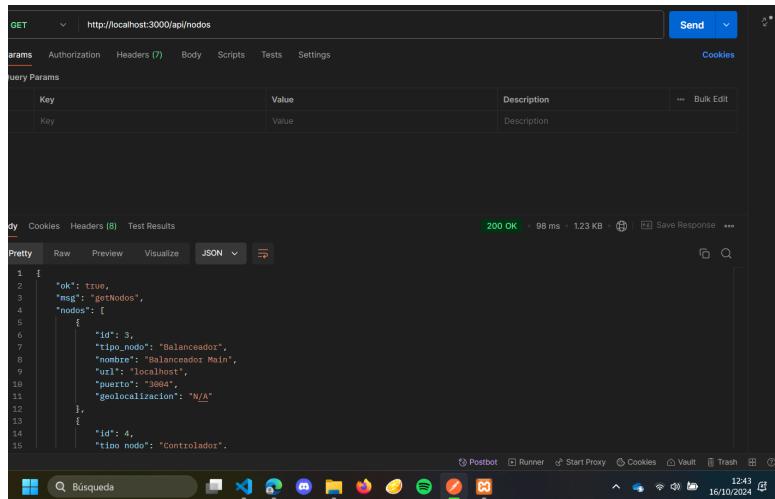


Figura 41. Petición GET ALL correcta en Postman (Fuente Propia)

Comprobado que la lógica de la petición era correcta se procedió a implementarla en el frontend de la plantilla Fuse. Para ello se usó la pantalla de lista, ya que la pantalla grafo aún no está acabada.

Primeramente, se creó un servicio de Angular en la carpeta “frontend/starter/src/app/services” con el comando “ng g s api”. Una vez creado el servicio se instaló la librería RxJS [15], esta facilita el manejo de la asincronía y ayuda a recuperar los datos de manera eficiente en observables. Después se creó la variable “urlAPI”, que contiene la url con el puerto que escucha la aplicación junto a la variable “headers” de tipo HttpHeaders, para definir las cabeceras que se van a enviar junto a las

peticiones. Con ya las variables declaradas se creó la función “getNodos()” que realiza la petición a la url definida anteriormente añadiendo “/nodos”, para que coincida con la ruta definida en la carpeta “routes” para la petición GET ALL, junto con las cabeceras definidas.

Posteriormente, se importó y se declaró en el constructor de la clase Lista del archivo “lista.component.ts” el servicio de la API. Tras importar y declarar el servicio se creó la función “getNodos()” que se encarga de llamar a la función “getNodos()” del servicio para realizar la petición y una vez recogidos los datos llama a la función “actualizarDataSource()” que se encarga de guardar los datos en el DataSource creado anteriormente con datos estáticos para las pruebas pero esta vez con los datos que devuelve la API. De esta manera los datos que se muestren en pantalla ya serán dinámicos.

Por último, como la función debe ser llamada al cargar la página para que se muestre con los datos cuando cargue, la función “getNodos()” del archivo “lista.component.ts” se llama en la función “ngOnInit()” para que recupere los datos antes de cargar la página.

6.4.5 Desarrollo de la petición GET BY ID

La petición GET BY ID se desarrolló para mostrar los datos de un nodo en su pantalla de información. Para crear esta petición se definió, en el archivo “nodo.js” de la carpeta “routes”, la ruta que iba a seguir:

- **Ruta GET BY ID:** “<http://localhost:8080/api/nodos/:id>”. Donde el “id” del nodo se pasa como parámetro en la url.

Tras definir la ruta de la petición se creó la función “obtenerNodoID()” en el archivo “nodo.js” de la carpeta “controllers” que se encarga de la lógica de la petición. Esta recoge el “id” pasado por parámetro y realiza la petición de Sequelize “Nodo.findOne(id)”, después comprueba si se ha encontrado el nodo y en caso de que la petición sea correcta pero no existe un nodo con ese “id” devuelve:

- **Status:** 404.
- **Mensaje de Error:** “Nodo con id ‘\${id}’ no existe”.

Por otro lado, si el nodo existe devuelve:

- **Status:** 200.
- **Nodo:** El nodo recuperado.

Y, por último, si hay algún error en la petición devuelve:

- **Status:** 500.
- **Mensaje de Error:** “Error en get nodo by ID”.

Tras crear la lógica de la petición en el backend se comprobó el correcto funcionamiento de la petición en Postman iniciando el proyecto con “npm start”:

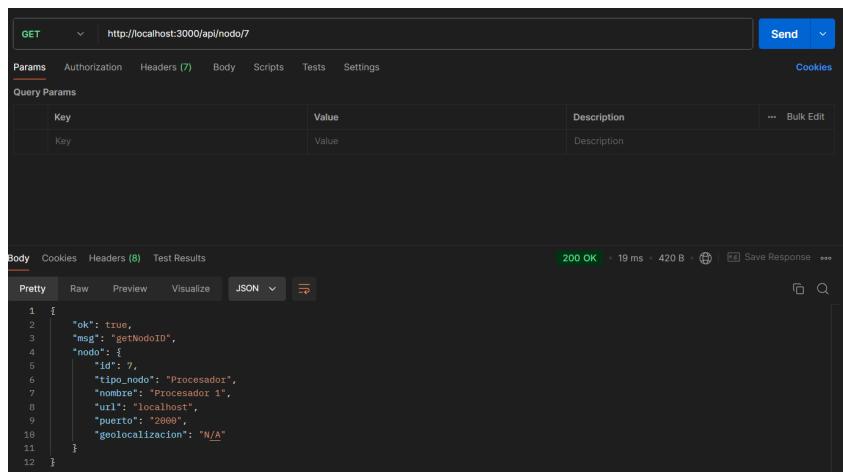


Figura 42. Petición GET BY ID correcta en Postman (Fuente Propia)

Finalmente, después de verificar el buen funcionamiento de la petición se comenzó su incorporación al frontend. Se creó la función “getNodo(id)” en el servicio de la API la cual llamaba a la petición de la url que escucha la aplicación añadiendo la ruta definida en el “routes” y el id pasado por parámetro: “`http://localhost:8080/api/nodo/id`”. Tras esto, como la información del nodo se va a visualizar en el modal que abre el botón de visualizar información del nodo de cada, se desarrolló la función “dialogo(tipo_nodo, id)” que se encarga de abrir el diálogo al hacer click en el botón y le pasa por parámetro al componente el “id” y el “tipo_nodo” del nodo seleccionado.

Una vez en el componente diálogo se declaran en el constructor las variables pasadas por parámetro para posteriormente, en la función “ngOnInit()”, se llame a la función “getNodo(id)” del servicio para recuperar los datos del Nodo y asignar estos datos y el “tipo_nodo” a variables del componente. Una vez asignadas se importa la librería CommonModule para poder usar estos datos en el archivo “dialog.html” y así representar los valores en pantalla, en el caso de los datos del nodo, y en caso del “tipo_nodo” ayudar a detectar cuando es un nodo de tipo controlador, mediante “*ngIf” y mostrar el dato de procesos activos que es único de estos nodos.

6.4.6 Desarrollo de la petición POST

La petición POST tiene como finalidad añadir nodos en el sistema y se usa en el formulario modal del botón “añadir nodo” de la página “lista”. Siguiendo los pasos de las peticiones GET realizadas anteriormente, se definió en el archivo “nodo.js” de la carpeta “routes”, la ruta que iba a seguir esta petición:

- **Ruta POST:** “`http://localhost:8080/api/nodo`”.

Con la ruta ya creada se desarrolló la función “`crearNodo()`” en el archivo “nodo.js” en la carpeta “controllers”. Esta función recibe por un request body con la información del nodo que se quiere crear. Seguidamente, comprueba que en la base de datos no existe un nodo que tenga el tipo de nodo y nombre a la vez, si es así devuelve:

- **Status:** 409.
- **Mensaje de Error:** “Ya existe un nodo con ese tipo de nodo y ese nombre”.

En caso de no haya un nodo así en el sistema hace una petición de tipo “`Nodo.create(req.body)`” donde crea el nodo en la base de datos y devuelve:

- **Status:** 200.
- **Nodo:** El nodo creado.

Por último, en caso de que hay un problema en la petición que no esté involucrada directamente con los valores de tipo de nodo y nombre devolverá:

- **Status:** 500.
- **Mensaje de Error:** “Error creando el nodo”.

Después de crear la ruta y la función que se encargan de la petición se pensó implementar validadores para los campos de las peticiones POST. Para ello se instaló Express-Validator [16] una librería de Express.js que nos ayudará a validar los datos. Una vez instalada se importó la función “`check`” de la librería en el archivo “nodo.js” de la carpeta “routes” y se usó para cada campo de la petición POST seguido de “`.not().isEmpty()`” para asegurarnos de que no se pasen campos vacíos a la petición.

Tras definir las validaciones que queríamos sobre los campos de la petición se creó el archivo “validar-campos.js” en la carpeta “middleware”. En este archivo se creó la función “validarCampos()”, la cual recoge los resultados de las validaciones que se hayan definido para los campos de la petición. Guarda los errores de validación, si los hay, que en caso de haber devolverá:

- **Status:** 400.
- **Errores:** Los errores encontrados en la petición.

Posteriormente, se empleó la herramienta Postman para verificar el correcto funcionamiento de la petición y sus validadores.

The screenshot shows a POST request to `http://localhost:3000/api/nodo`. The request body is a JSON object with missing fields:

```

1 {
2   "tipo_nodo": "Procesador",
3   "nombre": "",
4   "url": "localhost",
5   "puerto": "",
6   "geolocalizacion": "N/A"
7 }

```

The response is a 400 Bad Request with the following JSON error message:

```

2   "ok": false,
3   "errores": [
4     {
5       "name": "field",
6       "type": "field",
7       "value": "",
8       "msg": "El nombre está vacío",
9       "path": "nombre",
10      "location": "body"
11    },
12    {
13      "name": "field",
14      "type": "field",
15      "value": "",
16      "msg": "El puerto está vacío",
17      "path": "puerto",
18      "location": "body"
19    }
20 ]

```

Figura 43. Correcto funcionamiento de los validadores de la llamada POST en Postman (Fuente Propia)

The screenshot shows a POST request to `http://localhost:3000/api/nodo`. The request body is a valid JSON object:

```

1 {
2   "tipo_nodo": "Procesador",
3   "nombre": "Procesador 6",
4   "url": "localhost",
5   "puerto": "2008",
6   "geolocalizacion": "N/A"
7 }

```

The response is a 200 OK with the following JSON success message:

```

1   {
2     "ok": true,
3     "msg": "Nodo creado correctamente",
4     "nodo": [
5       {
6         "id": 15,
7         "tipo_nodo": "Procesador",
8         "nombre": "Procesador 6",
9         "url": "localhost",
10        "puerto": "2008",
11        "geolocalizacion": "N/A"
12      }
13   ]

```

Figura 44. Petición POST correcta en Postman (Fuente Propia)

Por último, tras comprobar el buen funcionamiento de la petición se comenzó su incorporación al frontend. Se creó la función asíncrona “postNodo(datosNodo)” en el servicio de la API. Esta función realiza la petición POST de la ruta asignada anteriormente pasando la variable “datosNodo” como cuerpo de la sentencia.

Una vez creada la función que se encarga de hacer la petición se desarrolló la lógica para asignar los datos del formulario del modal “Añadir Nodo” con esta. Para ello, se crearon en el archivo “lista.component.ts”, dentro del componente del diálogo “Añadir Nodo”, las propiedades: “tipoNodo”, donde se incluyeron las opciones “Controlador”, “Balanceador” y “Procesador”, “nombre”, “url”, “puerto” y “geolocalizacion”. Estas propiedades se asignaron a los inputs del formulario del archivo “anadirNodo.html” mediante el atributo “ngModel”, de este modo los valores introducidos en los campos del formulario se almacenan en las propiedades del componente. Además, se emplea el atributo “ngIf” para comprobar que selección se ha en el tipo de nodo para modificar los posibles valores del nombre en caso de ser de tipo “Controlador” o “Balanceador” convirtiendo el campo nombre del formulario en un select con dos opciones: “Main” y “Subs”, que hacen referencia al principal y sustituto, ya que, de estos nodos sólo pueden haber de estos tipos y solo uno de cada, pero esta segunda característica la cubren las comprobaciones de la petición.

Finalmente, tras asignar los valores del formulario a las propiedades del componente se programó la función “anadirNodo()” y se asignó a la acción click sobre el botón “añadir” del formulario. Esta función primero crea una variable llamada “nodoData” donde recoge los campos de un nodo y les asigna los valores recogidos en el formulario, luego llama a la función asíncrona “postNodo(nodeData)” del servicio al que le pasa la variable con los datos para ejecutar la sentencia y, por último, una vez se trata la petición cierra el modal con la función “cerrarDialog()” de la librería MatDialogClose.

6.5 Creación y configuración del grafo

Una vez terminadas las llamadas GET ALL y POST de la API se inició con la configuración del grafo que va a mostrar la información del sistema distribuido. Para ello se decidió utilizar la herramienta de Echarts[17], gracias a la gran variedad de opciones que da y la personalizable que es.

6.5.1 Creación del grafo con Echarts

Para comenzar instalamos Echarts en su versión 5.5.1. Una vez tuvimos la herramienta instalada y configurada procedimos a la creación de un grafo básico que primeramente mostrase los datos de la base de datos por pantalla.

Para ello se creó el elemento HTML que iba a contener el grafo. Una vez creado, se pasó al desarrollo de la función “initChart(data)” que se encarga de establecer las características del grafo y vincularlo con el elemento HTML. Esta función es declarada nada más iniciarse la aplicación. Lo primero que lleva a cabo es la vinculación del elemento HTML, si existe, con la función de Echarts “init()”. Despues configura las opciones que se pasan al grafo en la variable “option” siguiendo la estructura de ECharts. En Echarts el atributo “series” es el responsable de definir cómo se van a mostrar los datos, en nuestro caso los nodos del grafo. De esta manera configuramos este atributo para que el tipo de representación fuera un grafo, activamos la propiedad “roam” que permite hacer zoom y moverte libremente por el grafo, se añadió una etiqueta con el nombre del nodo de manera que se pudieran diferenciar y por último al apartado “data” se la añadieron los nodos pasados por parámetro recuperados por la función encargada de realizar la petición GET ALL.

Por último, para aplicar las características del grafo se usa la función de ECharts “setOption(option)” a la que se le pasa por parámetro la variable que contiene las características que hemos configurado. De esta manera tenemos un grafo que muestra los nodos en pantalla pero sin estar interconectados.

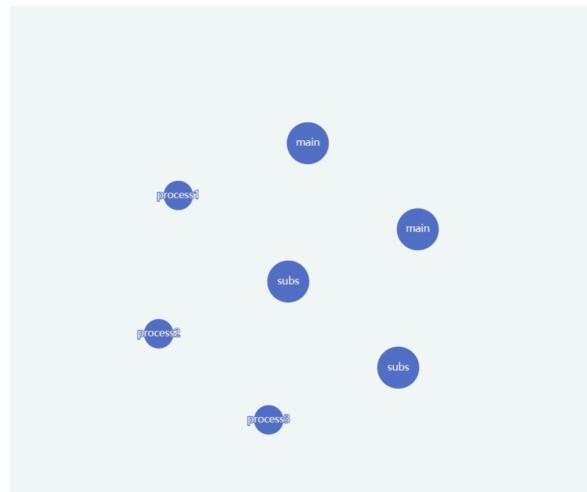


Figura 45. Representación de los nodos en Grafo (Fuente Propia)

6.5.2 Implementación de la llamada POST en el Grafo

Después de tener los nodos representados en el mapa se pasó a crear la función que añade los nuevos nodos al grafo. Para entender cómo se hizo esta tarea previamente se tienen que comentar varias cosas. Al igual que se proporcionó una API para las peticiones e información de los nodos del proyecto, la cual recupera información de los nodos y sus tipos, también se proporcionaron una serie de funciones recursivas y una estructura que contiene los datos de los nodos disponibles del sistema en un componente de Angular para usar en la parte de frontend.

El funcionamiento de este código proporcionado es el siguiente: por un lado la estructura proporcionada contiene los datos de configuración del puerto y URL de los nodos del sistema y los variables para almacenar los datos del nodo, por otro lado, las funciones recursivas, que son: “updateProcessors()”, “updateMainBalancerState()”, “updateSubsBalancerState()”, “updateMainCoordinatorState” y “updateSubsCoordinatorState()”, se encargan de hacer la llamada “debug” a las URL’s dadas por la estructura para cada nodo cada cierto tiempo para comprobar continuamente si hay cambios en la activación de los nodos. Esta llamada “debug” comprueba para la URL pasada como parámetro si esta URL está recibiendo información y lo devuelve a la función. En caso de estar recibiendo información, para ese nodo, se almacenará en la estructura la información devuelta y se pondrá la variable “status” de la estructura a “true”. En caso contrario, no se almacena información porque la llamada no devuelve nada y la variable “status” se establece a “false”. De esta manera conseguimos la configuración de los nodos en la estructura.

Una vez entendido el funcionamiento se pasó al desarrollo de la lógica de cómo implementar la función POST al grafo empleando la estructura. Para ello se ha hecho uso de las funciones mencionadas anteriormente. Dentro de la función, en caso de estar funcional si es la primera vez que se lanza esa URL, ya que cada nodo tiene una URL diferente, se hará una llamada POST de los datos del nodo que se guardará en la estructura. De esta manera cada vez que se lance un nuevo nodo se añadirá a la base de datos y, por tanto, al grafo.

6.5.3 Implementación de la actualización del grafo

Tras haber hecho la implementación de las funciones POST y GET ALL en el grafo se pasó a desarrollar una función que vaya actualizando los datos del grafo sin tener que recargar la página manualmente. Los datos que se tienen que reflejar en el grafo son: el estado del nodo, que refleja si está o no activo, y los datos del nodo.

Para ello se ha desarrollado la función “updateGrafo()”, una función recursiva que se encarga de recoger y aplicar los cambios que se produzcan en el grafo. Esta función se inicia tras la primera

configuración del grafo y su traza de funcionamiento es la siguiente: primero hace una llamada GET ALL por si se ha realizado añadido algún nodo a la base de datos y hay que añadirlo al grafo. Una vez devuelve la respuesta de la llamada GET ALL mediante un “switch” se comprueba qué nodo es para asignarle de la estructura proporcionada su estado y su configuración interna. Una vez se han asignado estas variables nuevas al los datos del nodo se guardan en un array. Este array se pasa como parámetro “data” en el atributo “series” del grafo y haciendo uso de la función “setOption()” se realizan los cambios en el grafo guardados en la variable “data”.

Por último, para que esta función se recursiva se ha usado la librería “timer” de rxjs, que es la misma que se usó en el código proporcionado para sus funciones recursivas. De esta manera se creó instanciallamada “readTime” con valor de 3 segundos y al final de la función se incluyó la función “timer(this.readTime)” para que cuando acabe la función se vuelva a llamar a la misma.



```
1 timer(this.readTime).subscribe(() => this.updateGrafo());
```

Figura 46. Función recursiva (Fuente Propia)

6.5.4 Diseño del Grafo

El grafo ya se actualiza de manera recursiva pero de momento no está implementado el diseño que haga visible los cambios de estas actualizaciones. Ni tampoco el diseño que permita diferenciar que tipo de nodo es cada uno.

El diseño que se iba a usar ya se especificó en el apartado de los mock-ups, por tanto, solo queda dejar reflejado ese diseño en código. Lo primero que se realizó del diseño fue la diferenciación entre los tipos de nodo. Para ello había que cambiar la forma y el color según su tipo. Como ya hemos mencionado anteriormente, al apartado “data” del grafo se le pasan los datos recuperados de la base de datos donde se incluye el tipo de nodo. Por tanto, dentro de este apartado se harán las modificaciones. “Data” es un array de nodos así que se le aplicó la función “map()” para recuperar todos los datos de cada nodo. Una vez recuperados se realizaron dos ternarias para el asignar variables al color y a la forma dependiendo del tipo de nodo. Si era Balanceador el color es azul y la forma cuadrada, si es Coordinador color naranja y forma circular y si es Procesador color lila y forma de rombo. Una vez hecha la ternaria se devuelven estas variables junto con el nombre para la etiqueta ya configurada. De esta manera se aplicarán las modificaciones a cada nodo.

Por último, para reflejar en el grafo si está activo o inactivo se recupera el valor pasado al nodo de la estructura de configuración que ha sido asignado por la llamada “debug”. En el caso de ser “true” está activo por tanto el color no cambia, pero si es “false” se llama a la función “grey(color)” que coge el color y lo pasa a una escala de gris volviendo este color más oscuro.

Así conseguimos un diseño que distinga los tipos de nodo por color y forma, a la vez que nos deja ver que nodos están funcionando y cuales no por el tono de color del nodo.

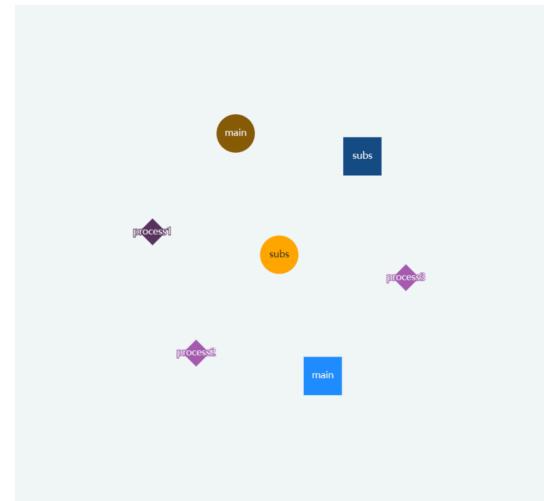


Figura 47. Diseño de los nodos en Grafo (Fuente Propia)

6.5.5 Implementación de la información del nodo en el grafo

Tras la implementación del diseño del grafo se comenzó con el desarrollo de abrir el diálogo de información del nodo seleccionado en el grafo.

Como el diálogo encargado de mostrar la información del nodo está declarado en un componente simplemente hay que llamar a su función de abrir diálogo pasándole por parámetro las características que requiere, en este caso, el id del nodo y su tipo de nodo. Para pasarle estos dos datos el nodo creado en el apartado “data” debe devolver el id y el tipo de nodo al igual que devuelve el color y la forma. Estos datos se sacan de la llamada GET ALL que posteriormente se analiza uno a uno con la función “map()” como se comentó en el apartado anterior. Una vez recuperados estos datos se añaden al return y ya estarían almacenados en el nodo.

Por último, se desarrolló la función que se encarga que al hacer click en un nodo se abra el diálogo correspondiente. Lo primero fue importar el componente del diálogo para poder acceder a sus funciones. Una vez importado mediante la función de Echarts “on(click)” se creó un evento que al hacer click recuperaba el elemento seleccionado y sus variables. Después buscaba las variables “id” y “tipo_nodo” y las pasaba por parámetro a la función “this.dialog(id, tipo_nodo)” que se encarga de abrir el diálogo y hacer el GET BY ID del nodo que se pasa y proporcionar su información.

6.5.6 Incidencia con el grafo

El siguiente paso era crear los enlaces del grafo, y se planteó la siguiente manera para hacerlo. Como la llamada “debug” devolvió un estado interno con las URL’s que podía escuchar cada nodo se intentó coger estos enlaces de la configuración interna de la estructura proporcionada para realizar los links. Pero una vez me puse a desarrollarlo vi incongruencias: como que link debe escoger cual no debe escoger, ya que enlazaba con varios, dependiendo de los nodos activos.

Después de no encontrar soluciones se solicitó una tutoría el día 24 de octubre. En esta reunión se habló sobre el planteamiento y se llegó al punto de que había habido un problema de entendimiento y comunicación a la hora de explicar cómo iba a funcionar la aplicación. Las funciones y la estructura que se habían proporcionado no era para uso explícito, era un ejemplo de datos limitados para que yo me hiciera una idea general, pero lo que se buscaba era no una estructura fija y limitada de nodos sino que esta estructura fuera la base de datos con todos sus nodos.

De modo que solo hubiera una función “debug” que se aplicará a todas las URL’s de la base de datos y en caso de funcionar se recuperarán los datos internos y el estado del nodo que está devolvía. Y que la manera de añadir nodos fuera la llamada POST de la vista “Lista”.

Una vez aclarado este malentendido se realizaron los cambios correspondientes al componente y su flujo, pero la fase de diseño quedó igual.

Después de realizar los cambios el flujo quedó de la siguiente manera: primeramente, al iniciar la aplicación se llama a la función “initChart()” que realiza la llamada GET ALL de la API, una vez recuperados los nodos, para cada nodo se realiza la llamada “debug” en base a la URL recuperada. En caso de estar activo se le añaden los valores devueltos por la llamada que son: el estado igual a “true” y el estado interno del nodo, por otro lado, si está desactivado se devuelve: estado igual a “false” y el estado interno vacío. Tras hacerlo para todos los nodos se inicializa el grafo de la misma manera que antes de la incidencia. Una vez inicializado el grafo lo último que hace la función “initChart()” es llamar a la función “updateGrafo()”.

Por último, las modificaciones aplicadas a la función recursiva “updateGrafo()” son las mismas que las aplicadas a la función “initChart()” se hace el GET ALL y posteriormente para cada nodo, ya estuviera añadido o sea nuevo, se realizarla petición “debug” para recuperar sus datos y representarlos en el grafo.

6.5.7 Implementación de los Links en el Grafo

Ahora, fase tras realizar los cambios en el grafo, si que se comenzó el desarrollo de los links en el grafo. Para definir los links en un grafo de ECharts debe asignarse al grafo en el apartado “links” un array con todas los enlaces entre nodos que contenga: “source” que es el nodo que apunta y “target” que es el nodo apuntado.

Lo primero tener claro cuando un nodo apuntaba a otro, es decir, que condiciones debe cumplir para así poder aplicarlas a la hora de completar el array de “links”. Las condiciones son las siguientes:

- **Subs→Main:** Los balanceadores del mismo tipo que sean sustitutos y estén activos apuntan a los de su tipo que sean main que estén activos. Esto se debe a que esperan respuestas de él para que en el caso de fallo poder tomar su rol.
- **Balanceadores→Procesadores:** El balanceador activo con rol de “main” apunta a los procesadores activos. Ya que es el encargado de enviar tareas a los procesadores.
- **Balanceadores→Controlador:** El balanceador activo con rol de “main” apunta al controlador activo activos con rol de “main”. Esto se debe a que el balanceador comunica al procesador los cambios en los procesadores.

Después de tener en cuenta las condiciones se realizaron tanto en la función “initChart()” como en “updateGrafo()” estas comprobaciones para crear el array de “links” que se pasarán al grafo. Estas comprobaciones se aplicarán a los datos recuperados de la petición GET ALL y se devolverá el array con los links.

Finalmente, para darle aspecto a los links en el grafo en el apartado “series” se declararon las siguientes características:



```
1  lineStyle: {
2      color: '#aaa',
3      width: 2,
4      opacity: 0.9,
5      curveness: 0.1
6  },
```

Figura 48. Configuración del diseño de los links en Grafo (Fuente Propia)

Estás directrices establecen el color, el ancho, la curvatura de los links, además de especificar el que el nodo que apunta su lado del link no tiene símbolo mientras el apuntado está señalado con una flecha dejando claro el flujo de datos. Como resultado obtenemos un grafo conectado según el funcionamiento requerido del sistema.

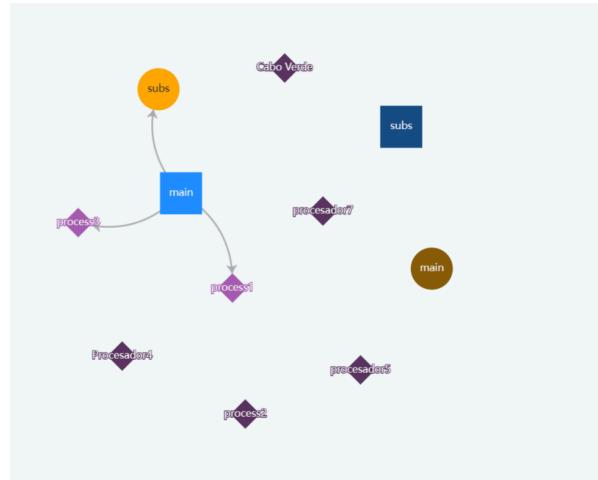


Figura 49. Diseño de los links en Grafo (Fuente Propia)

6.5.8 Deshabilitación de los links

Una vez desarrollados los links se pasó a completar el apartado correspondiente permitiendo la desactivación de los links en la pantalla y su reactivación.

Para ello se creó la función “conexiones(event)” que se vinculó al evento change del checkbox “Desactivar conexiones”. Esta función recibe el evento del checkbox y dependiendo si está seleccionado le pasa al grafo la función “setOption()” con el array del apartado “links” vacío y guarda el estado del checkbox en la instancia “activarConexiones”.

También para asegurarnos que al cambiar de página o al ejecutarse la función recursiva que actualiza el grafo no se vuelvan a cargar los links si no se ha solicitado se ha puesto una condición en las funciones “updateGrafo()” para que a la hora de calcular los links solo entre sí la instancia “activarConexiones” se cumple. De esta manera controlamos el correcto funcionamiento del checkbox.

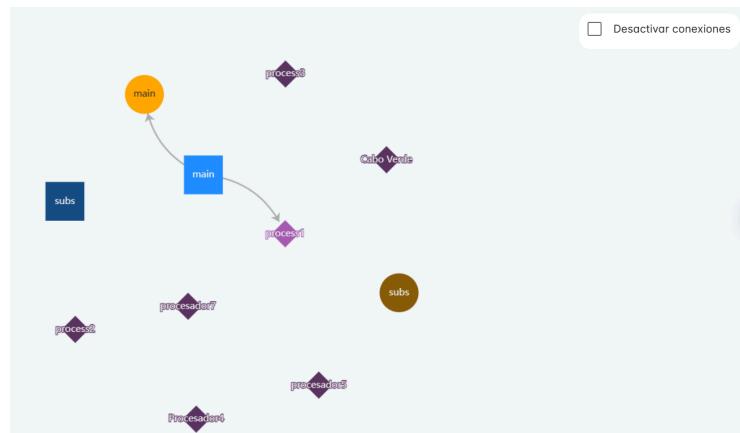


Figura 50. Conexiones activas (Fuente Propia)

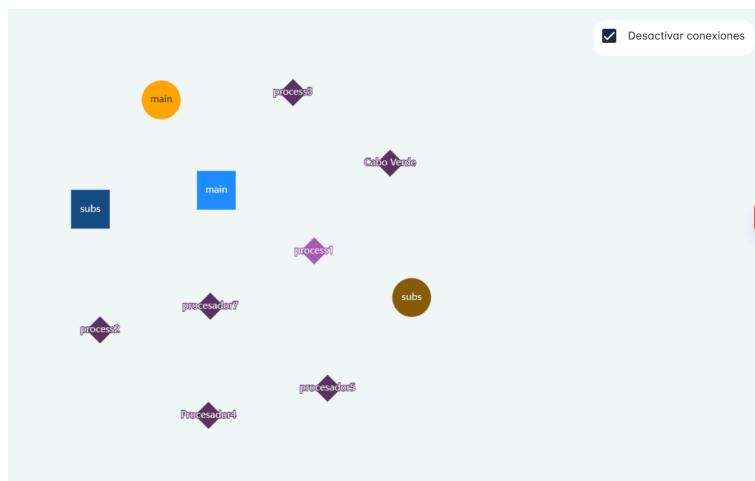


Figura 51. Conexiones desactivadas (Fuente Propia)

6.6 Creación e implementación de la vista mapa

Una vez completado el grafo con sus nodo y sus links actualizando correctamente se planteó para esta iteración crear la vista de mapa que se habilita con el checkbox y que muestra los nodos con su geolocalización global en un mapamundi.

6.6.1 Cambios en el modo de almacenar la geolocalización

Antes de ir con la implementación del mapa primero tenemos que almacenar bien la latitud y longitud de los nodos. Hasta ahora hemos tenido el campo geolocalización en la base de datos pero era simplemente indicativo.

Para empezar se eliminó el campo de la base de datos y se crearon dos campos de tipo “varchar(100)” llamados latitud y longitud. Seguidamente se hicieron los cambios correspondientes en la API, en la carpeta “models” dentro del archivo “nodo.js” se añadieron los campos y se eliminó el campo geolocalización y las validaciones de la carpeta ”routes” del archivo “nodo.js” se añadió un check para cada nuevo campo y el campo de geolocalización se volvió a eliminar.

Después de realizar los cambios en el backend se procedió a hacer lo propio en el frontend. Para la vista “Lista” se añadieron los campos en la interfaz “NodeData”, que es donde organizamos los datos de la lista, para que a la hora de recuperar los datos no haya problemas de asignación. Tras hacer esto en el HTML se asignó en el apartado de geolocalización las variables de latitud y longitud para que se visualizarán en la columna de geolocalización.

Geolocalización
Latitud: 19.4326
Longitud: -99.1332
Latitud: 40.7128
Longitud: -74.0060
Latitud: 51.5074
Longitud: -0.1278
Latitud: 35.6762
Longitud: 139.6503
Latitud: 52.5200
Longitud: 13.4050

Figura 52. Latitud y longitud en la Lista (Fuente Propia)

Del mismo modo se modificó para el modal de información de cada nodo dejando ver en el apartado de geolocalización tanto longitud como latitud.

main

Puerto	URL
3100	http://localhost:3100
Última consulta	Geolocalización
09/02/2023 11:21	Latitud: 19.4326 Longitud: -99.1332

Figura 53. Latitud y longitud en la Modal de información (Fuente Propia)

Por último, se modificó el formulario creando las instancias latitud y longitud en el componente del modal “añadirNodo” y creando los campos conectados a la instancias con “NgModel”. Con la intención de no cambiar mucho el diseño original y hacer otra columna se dejó el apartado como geolocalización formado por dos campos con el atributo “placeholder” para cada uno con tal de indicar a cual pertenece el valor de latitud y cual de longitud.

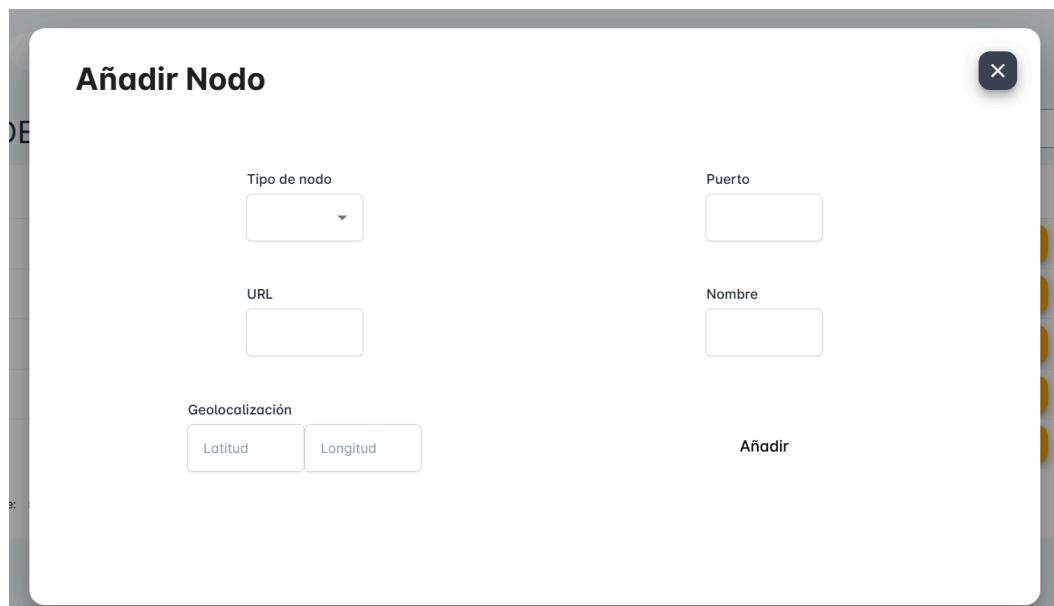


Figura 54. Latitud y longitud en la Modal de añadir nodo (Fuente Propia)

6.6.2 Implementación del mapa en ECharts

El objetivo de esta siguiente parte de la iteración es crear el mapa en Echarts para ello se ha estado investigando y vi que hay una característica en Echarts que te permite establecer un mapa para representar y también ajustar la escala del mapa a la geolocalización de los nodos. Este atributo es “geo”, para usar este atributo debes tener un mapa de tipo “json” o “geojson” que Echarts pueda leer y asignarlo a la variable “map” dentro del atributo.

Después de leer vi que Echarts tenía un mapa por defecto que se aplicaba si a la variable “map” la igualas a “world” pero no dibujaba nada e investigando un poco más descubrí que Echarts ya no daba soporte a esta metodología. De esta manera para cargar el mapa teníamos que conseguir un archivo “json” o “geojson” que tuviera las coordenadas de todos los países. Llegados a este punto, crear el archivo era inviable por tanto me puse a buscar donde podía encontrar alguno.

Tras buscar di con un repositorio que tenía un archivo “geojson” del mapamundi, este es el repositorio:https://github.com/datasets/geo-boundaries-world-110m/blob/main/countries.geojson?short_path=2fcecb. Una vez descargado el archivo se guardó en la carpeta “assets/data” con nombre “map.json”.

Al tener el archivo del mapa solo quedaba asignarlo en el Chart para que se dibujara. Para asignarlo a la variable “map” del atributo “geo”, primeramente, había que convertir el mapa en un objeto de javascript, de modo que se importó y al iniciar la vista de “Grafo” en la función “ngOnInit()” se creó la instancia “jsonMap” a la que se le pasó como valor el mapa convertido a objeto javascript con la función “JSON.parse(map)”. Seguidamente, había que registrar el mapa en Echarts con la función “registerMap(‘nombre’, this.jsonMap)”, la cual asignaba el mapa con el nombre que tu le pusieras, en nuestro caso fue “mimap”. Después, se asignó a la variable “map” del atributo “geo” el mapa con nombre “mimap” cargado anteriormente.

Tras haber hecho la carga del mapa, queda pintar los nodos en su geolocalización, para ello el atributo “data” debía devolver junto a sus variables una variable más llamada “value” compuesta por la latitud y la longitud del nodo de la siguiente manera: “value=[longitud, latitud]”. De esta manera se asignaría esa posición exacta en el grafo. Pero para que las coordenadas del nodo y las del mapa encajen hay que añadirle al grafo el atributo “coordinateSystem: ‘geo’” dentro de “series”, que su función coger el mapa cargado en “geo” como referencia a la hora de pintar las coordenadas de los nodos.

Finalmente, queda solo que el mapa se active cuando el checkbox del “Mapa” esté seleccionado. De modo que se creó la función vinculada al elemento HTML del checkbox mediante un evento “change” que cada vez que cambiaba le pasaba al grafo mediante la función de Echarts

“setOption()” la variable “show” del atributo “geo” a “true” o “false” correspondientemente. De este modo si estaba seleccionado pintaba el mapa y si no no lo hacía.



Figura 55. Geolocalización de nodo activa (Fuente Propia)

6.6.3 Incidencia con el mapa

Finalizado el mapa se organizó una reunión el 4 de noviembre para evaluar el producto desarrollado hasta la fecha.

En esta reunión se comprobó por encima el correcto funcionamiento de los cambios de la anterior incidencia y se comprobó que todo quedó claro. Después se pasó a evaluar el mapa y nos encontramos con el siguiente problema de comunicación. Yo entendí que lo que se requería era geolocalizar el grafo por eso empleé la misma herramienta de ECharts para hacerlo, pero el Product Owner tenía otro enfoque. Quería emplear herramientas distintas para la vista de mapa y de grafo, ya que al ampliar mucho el grafo sólo tiene las líneas de los países y si tengo muchos nodo en un país y amplío para verlo no se ve exactamente en qué zona está. Por ejemplo, en la siguiente imagen puedo ver que el procesador está por la zona de Madrid pero no puedo ver el barrio exacto ni la comunidad autónoma con claridad:

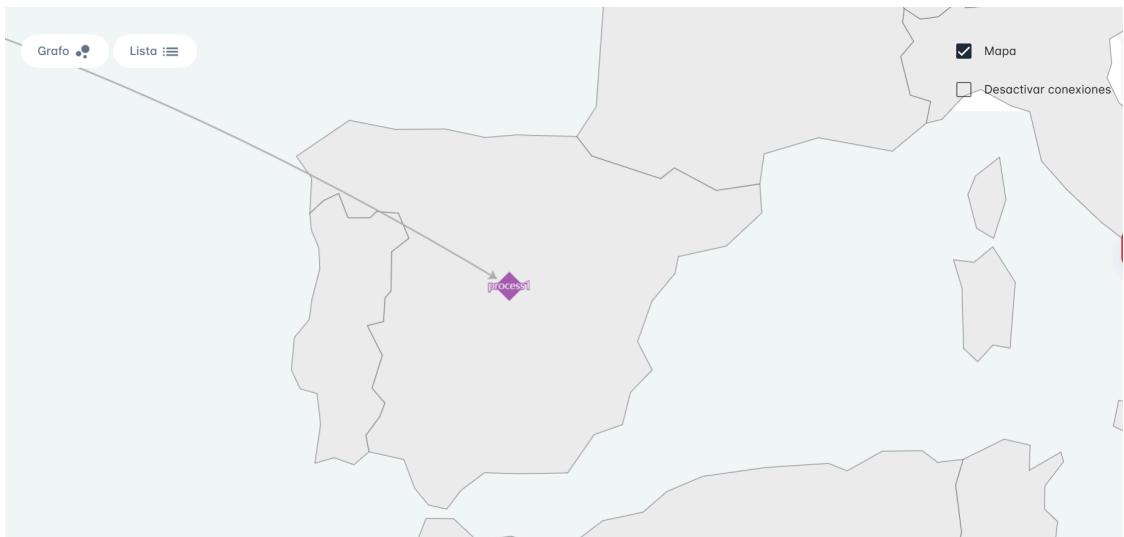


Figura 56. Ejemplo de limitación del mapa Echarts (Fuente Propia)

De modo que me sugirieron que en vez de hacer el checkbox para activar o desactivar el mapa hiciera otra vista más a parte y dejase el grafo hecho con Echarts por un lado, y por otro, que usase otra herramienta para el mapa como puede ser OpenStreetMaps.

Una vez aclarado este punto se eliminaron del componente grafo las funciones implementadas en el apartado “9.6.2 Implementación del mapa en ECharts”.

6.6.4 Creación del mapa con Open Street Maps

Definido ya tanto el nivel de detalle como las características necesarias para el mapa que se usará para la geolocalización de los nodos se investigó la herramienta de Open Street Maps[18]. Una vez analizada gustó gracias a la facilidad que ofrece para crear el mapa, editarlo a su gusto y añadir o representar elementos externos, en nuestro caso los nodos.

Lo primero fue proceder con la instalación de Open Street Maps, para ello se instaló la librería de Leaflet[19] en su versión 1.9.4. Esta librería permite crear mapas interactivos y es compatible con Open Street Maps.

Después se creó la nueva vista donde se representará el mapa. Dentro de la carpeta “src/app/modules/admin” se ejecutó el comando “ng g c mapa” para crear un componente para el mapa. Una vez creado el componente para el mapa se añadió la nueva ruta en el archivo

“app.routes.ts” y se añadió la ruta en la carpeta “src/app/mock-api/common/navigation” dentro del archivo “data.ts” que es el encargado de redirigir las rutas de la plantilla.

Una vez configurada la ruta se añadió a la cabecera creando un nuevo “toggle button” junto a los otros con las redirecciones a su vista y en los componentes “Lista” y “Grafo” se creó la función “irAMapa()” que redirige a la ruta de la nueva vista “Mapa”. Por último, se incluyeron en el componente “mapa”: el componente “Dialog” y la función encargada de abrir el modal de información de cada nodo junto a las funciones “irALista()” e “irAGrafo()”, para completar la navegación de la vista.

Añadida ya la vista al frontend del servicio se pasó al desarrollo del mapa con OSM(Open Street Maps) y Leaflet. Primeramente, se importó la librería Leaflet al componente para poder trabajar con ella junto con el servicio que recupera los datos de la API para recuperar los nodos que se tienen que representar en el mapa. Tras estas importaciones se creó la función “initMap()”, dentro de “ngOnInit()”, que crea un mapa asignándolo a la variable “this.map” y lo centra en las coordenadas [0,0] (ubicación del ecuador y meridiano de Greenwich) con una vista global. A continuación, añade una capa de dibujo mediante “titleLayer()” que hace que cargue de sus servidores los mapas mediante tu amplias o te desplazas por el mapa.

Seguidamente, al tener ya cargado el mapa creamos una función llamada “cargarNodos()” que como su nombre indica carga los nodos. Primero recupera los nodos de la base de datos mediante la llamada GET ALL del servicio de la API. Después para cada uno crea una variable de tipo “icon” con la librería Leaflet al que le pasamos una imagen de un ordenador como icono, debido a que de momento no tenemos los png de las figuras del grafo y OSM solo acepta png no figuras, y adaptamos el tamaño al deseado, de esta manera conseguimos diseñar el icono que queremos que se muestre por pantalla. Una vez creado el icono, con la misma librería, la función crea una variable de tipo “marker” a la que se le asigna la latitud y longitud del nodo, junto al icono creado y lo añade al mapa con la función “addto(this.map)” de la librería. Por último, al marcador le añadimos una etiqueta con su nombre con la función “bindToolTip()” q la que le pasamos el nombre y como queremos que se posiciones respecto al icono.



Figura 57. Nodos geolocalizados en mapa creado con OSM y Leaflet (Fuente Propia)

6.6.5 Actualización del mapa e información de nodos

Creado ya el mapa se procedió a implementar dos funcionalidades que también están presentes en el grafo. Estas son: la funcionalidad de mostrar qué nodos están activos y cuáles no y la funcionalidad de consultar la información del nodo clicando en su ícono del mapa.

Para ello vamos a tomar como referencia las funciones de la vista “Grafo” ya que el flujo de actuación es el mismo. Estas dos funcionalidades se implementaron modificando la función “cargarNodos()” del apartado anterior. En cuanto a la funcionalidad de abrir el modal de información, como ya se importó el componente junto a su función de abrir el modal, se asignó a cada nodo recuperado un evento “click” que ejecute la función “dialogo()”, la cual abre el modal. Por otro lado, para controlar el funcionamiento de los nodos se desarrolló un bucle que realiza la petición “debug” y guarda los valores devueltos más el estado junto con los datos recuperados en la API, al igual que se hace en el componente “Grafo”. Una vez guardados los valores devueltos por la petición recorre los nuevos valores y en caso de que el estado sea “true” al ícono le asigna un archivo PNG a color y si el valor es “false” el mismo PNG pero en escala de grises. Finalmente, llama a la función “timer()” de la biblioteca “RxJS” a la que le pasa un margen de 3 segundos para que cada ese tiempo vuelva a llamar a la función y haga las correspondientes revisiones y actualizaciones.

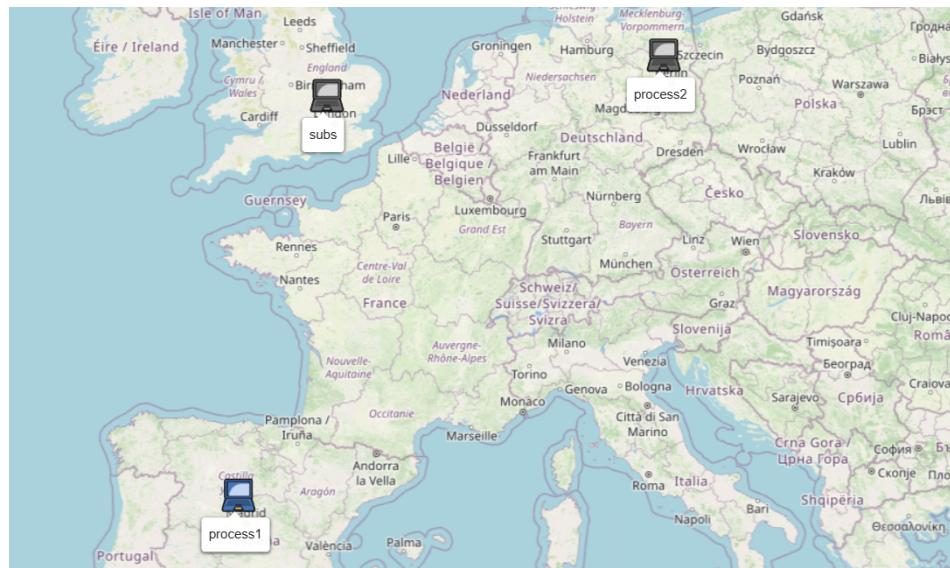


Figura 58. Mapa con Nodos activos e inactivos (Fuente Propia)

6.7 Mejoras I

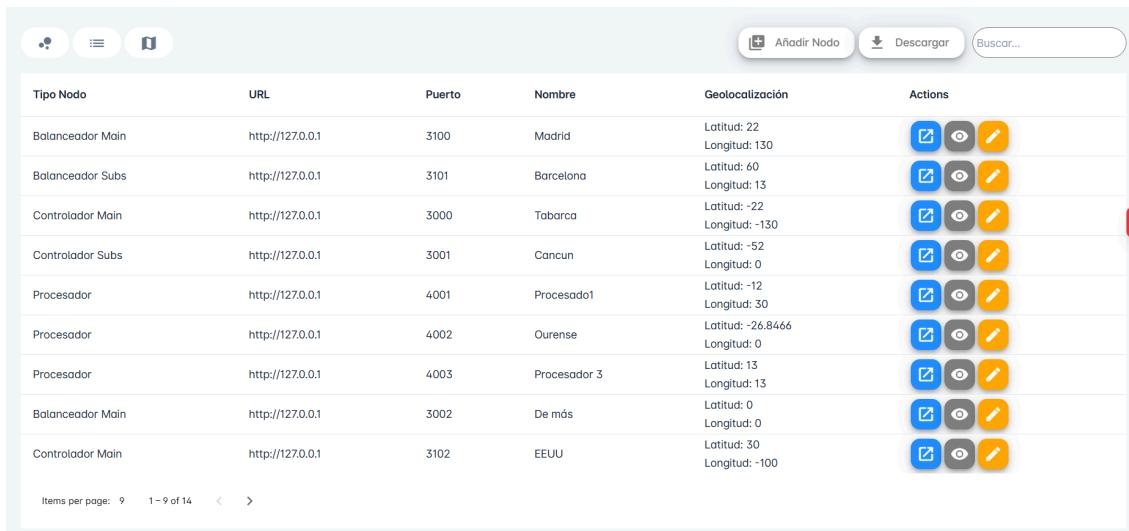
Tras llegar a una aproximación funcional del producto con sus partes fundamentales desarrolladas se decidió que la siguiente iteración se dedicaría a acabar las funcionalidades complementarias de la vista lista, con sus peticiones de actualizar y buscar, la función de descargar y pulir ciertos aspectos de diseño.

6.7.1 Cambios de aspecto en la pantalla Lista

Después de repasar con mis tutores el producto que hay hasta la fecha se decidieron varios cambios de aspecto en la pantalla de Lista para dar una mayor accesibilidad y diseño a cualquier persona que use el servicio.

La finalidad de la pantalla Lista es ver de forma ordenada los datos de la pantalla pero dado al espacio que ocupa la cabecera de la lista, formada por el encabezado de esta misma, los botones de “añadir nodo” y “descargar” y el campo de búsqueda, se decidió que ese espacio se podía aprovechar para mostrar más líneas de datos de la tabla. Como solución se eliminó el encabezado de la tabla, ya que era “Lista de Nodos”, y consideramos que la persona que use la aplicación sabrá del uso de todas las pantallas, por tanto, es información redundante. Además los botones que también formaban parte de la cabecera de la lista se pusieron a la altura de los botones de cambio de pantalla a la derecha formando una cabecera conjunta que permite dedicar espacio a la tabla.

Por otro lado, se decidió que los botones tenían colores que no representaban bien las funcionalidades que desempeñan y se decidió cambiar estos colores por unos más acordes: para el botón de mostrar información al ser una función de consulta frecuente se eligió el color azul claro que va acorde con acciones usuales o primarias, para el botón de visualizar se cambió a un gris, ya que como representa una opción habilidades o deshabilitada un color neutro acompaña bien y por último el de editar cambió a ser de color amarillo ya que llama facilmente la atención.



Tipo Nodo	URL	Puerto	Nombre	Geolocalización	Actions
Balanceador Main	http://127.0.0.1	3100	Madrid	Latitud: 22 Longitud: 130	  
Balanceador Subs	http://127.0.0.1	3101	Barcelona	Latitud: 40 Longitud: 13	  
Controlador Main	http://127.0.0.1	3000	Tabarca	Latitud: -22 Longitud: -130	  
Controlador Subs	http://127.0.0.1	3001	Cancun	Latitud: -52 Longitud: 0	  
Procesador	http://127.0.0.1	4001	Procesador1	Latitud: -12 Longitud: 30	  
Procesador	http://127.0.0.1	4002	Ourense	Latitud: -26.8466 Longitud: 0	  
Procesador	http://127.0.0.1	4003	Procesador 3	Latitud: 13 Longitud: 13	  
Balanceador Main	http://127.0.0.1	3002	De más	Latitud: 0 Longitud: 0	  
Controlador Main	http://127.0.0.1	3102	EEUU	Latitud: 30 Longitud: -100	  

Figura 59. Cambios de diseño en la vista de Lista (Fuente Propia)

Además, para facilitar la accesibilidad del usuario, los iconos de información y visualización pueden ser diferentes según la aplicación. Se nos ocurrió poner un texto con la funcionalidad del botón cuando se ponga el ratón encima con un elemento tooltip de la librería Angular Materials.

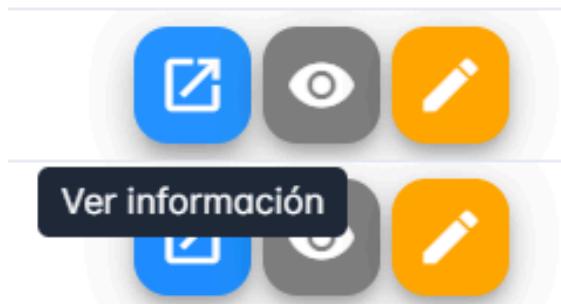


Figura 60. Tooltip para la accesibilidad (Fuente Propia)

6.7.2 Implementación de la petición PUT

Una vez realizados los cambios de diseño se decidió acabar las funcionalidades complementarias de la vista Lista. Para ello quedan las funcionalidades de los botones de visualizar y actualizar. Al ser ambas dependientes de la petición de PUT se empezó con el desarrollo de la petición en la API.

Para ello en el archivo “nodo.js” de la carpeta “backend/miback/routes” implementó la ruta a la llamada PUT:

- **Ruta PUT:** “`http://localhost:8080/api/nodo/:id`”.

Tras crear la ruta se asignaron los validadores: “`validarCampos`” desarrollado previamente y la función “`check`” de Express-Validator, usados en la llamada POST para asegurar que los campos no estén vacíos.

Con la ruta ya creada se desarrolló la función “`actualizarNodo()`” en el archivo “nodo.js” en la carpeta “controllers”. Esta función recibe un body con los nuevos datos del nodo y el id del nodo por URL. Seguidamente, comprueba que en la base de datos no existe un nodo con su mismo nombre y tipo de nodo. En caso de si existir:

- **Status:** 409.
- **Mensaje de Error:** “Ya existe un nodo con ese tipo de nodo y ese nombre”.

En caso de no haya un nodo así en el sistema hace una petición de tipo “`Nodo.create(req.body)`” donde crea el nodo en la base de datos y devuelve:

- **Status:** 200.
- **Nodo:** Nodo actualizado correctamente.

Por último, en caso de que hay un problema en la petición que no esté involucrada directamente con los valores de tipo de nodo y nombre devolverá:

- **Status:** 500.
- **Mensaje de Error:** “Error actualizando el nodo”.

Para comprobar el correcto funcionamiento de la petición se usó Postman:

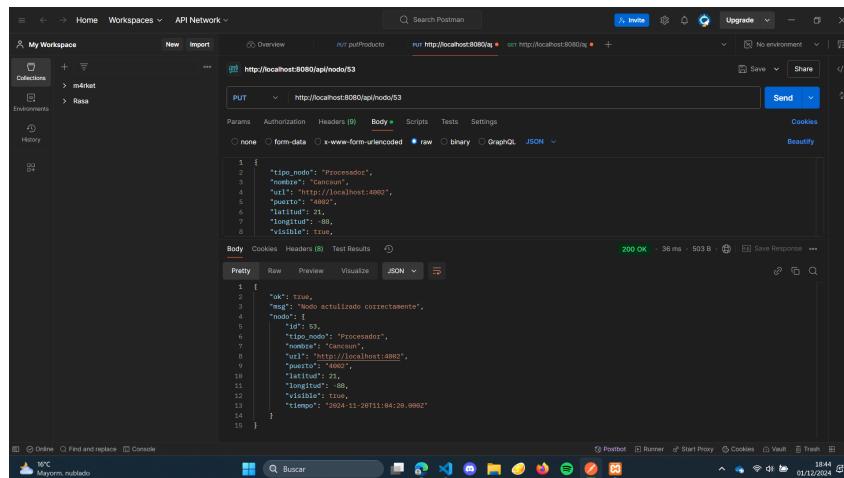


Figura 61. Petición PUT correcta en Postman (Fuente Propia)

6.7.3 Implementación de la opción visualización

Con la lógica de la llamada PUT terminada se pasó a la implementación de la opción visualización que permite, al desactivarla, que los nodos dejen de ser visibles tanto en el mapa como en el grafo. Para este desarrollo se creó en la base de datos, en la tabla “nodos”, el campo “visible” de tipo booleano y se implementó en las demás peticiones de la aplicación. Junto con este cambió se tuvo que modificar el formulario de “Añadir nodo” para que incluyera el campo con un select con dos opciones que trae por defecto la opción activada.

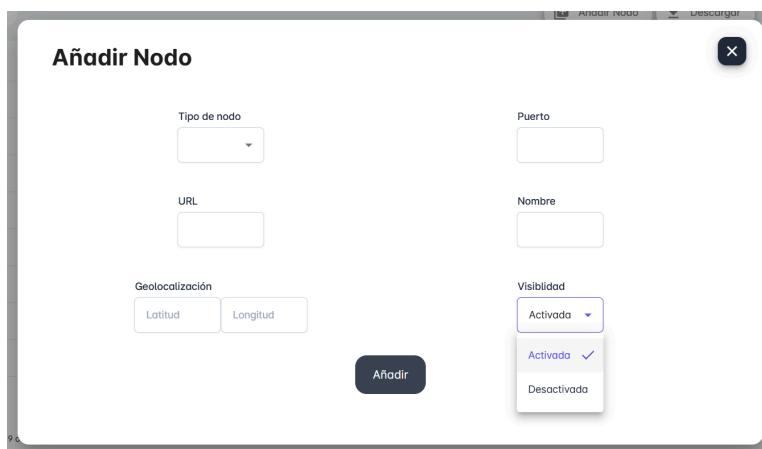


Figura 62. Modificación del formulario con el campo visibilidad (Fuente Propia)

El último cambio para incluir el campo de visualización en el sistema fue representarlo en la lista. Cómo está función va enlazada con el botón visualizar de la lista se decidió representarlo con el

ícono del ojo normal cuando esté visible y con el ojo tachado cuando esté invisible, mediante un “ngIf” que controla el valor del campo “visible” de la fila .

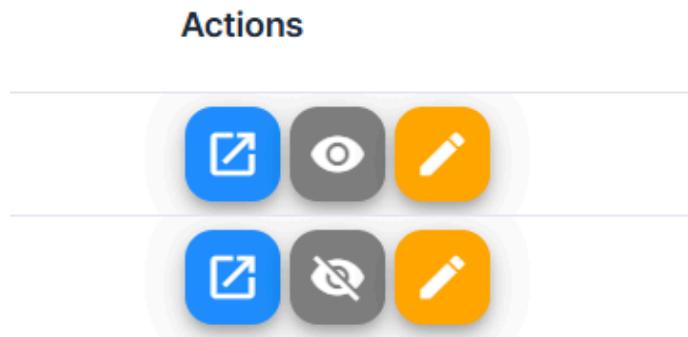


Figura 63. Representación de visibilidad en la Lista (Fuente Propia)

Una vez hechas las modificaciones de aspecto se pasó a la implementación de la función. Al clicar sobre el botón se lanza la función “visible(element)” o “noVisible(element)”, dependiendo de qué valor tenga el campo “visible”, que recibe todos los datos del elemento de la fila y realiza una función PUT a la que le pasa los mismos valores que ya tiene el nodo menos el campo “visible” que si está a verdadero pasará a falso y viceversa actualizando de esta manera el nodo.

Por último, falta que la visualización encaje correctamente con el mapa y el grafo cuando cambia su valor de visibilidad y se representen correctamente en ellos. Para el grafo sólo se desarrolló un bucle que a la hora de pasar los datos al este recorra los nodos y compruebe si el campo visible es true, de ser así se añade, si no no. De esta manera no aparece en el grafo y los links tampoco se representan porque Echarts lo tiene así programado. Pero para el mapa, a parte de antes de dibujar el ícono que representa el nodo comprobar que el campo visible sea true, se tuvieron que modificar todas las comprobaciones de los links asegurándonos de que tanto el nodo que apunta como el apuntado fueran visibles. De esta manera se representan correctamente si son o no visibles.

6.7.4 Implementación del formulario de actualizar nodo

Prosiguiendo con la petición actualizar lo último que quedaba por implementar de esta funcionalidad de esta petición es el modal que permita cambiar los datos del nodo no sólo su visualización.

Con este fin se usó el formulario “añadir nodo” y se duplicó cambiando algunos aspectos para crear el formulario ”actualizar nodo”. El primer cambio fue el encabezado del modal, se modificó por el tipo del nodo acompañado de su nombre. Y el segundo cambio fue que nada más abrir el modal se llama a la función “getNodo(id)” para recuperar los datos del nodo y se guardan en instancias de ese componente para que cuando se muestre el formulario los campos estén llenos con los datos almacenados del nodo.

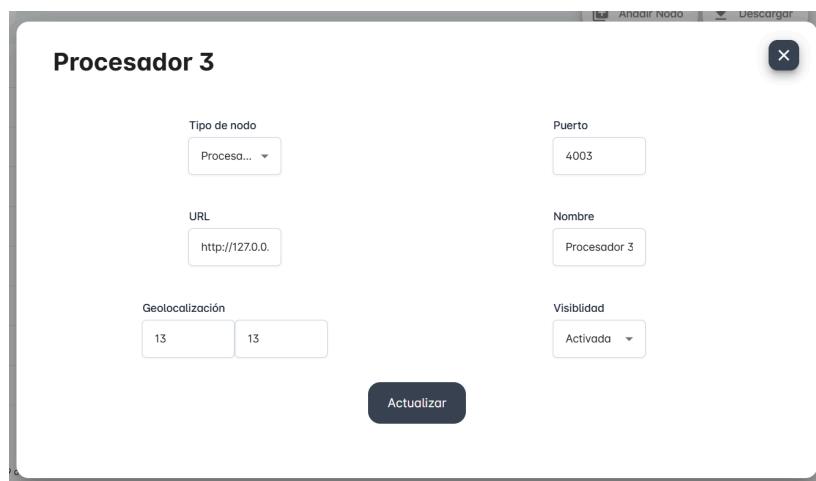


Figura 64. Formulario Actualizar Nodo (Fuente Propia)

Finalmente, al botón de actualizar se creó la función “actualizarNodo()” que recoge los datos de los campos y llama a la función PUT de la API para actualizar el nodo.

6.7.5 Implementación de la función buscar

Tras el desarrollo de la petición PUT y las funciones encargadas de actualizar los nodos y de la visualización de estos se procedió al desarrollo de la función buscar que permita encontrar los nodos en la lista según su tipo, su nombre y su puerto.

Lo primero fue crear una petición nueva de tipo GET. Para ello en el archivo “nodo.js” de la carpeta “backend/miback/routes” implementó la ruta a la llamada GET:

- **Ruta GET SEARCH:** “<http://localhost:8080/api/nodos/:datos>”.

Con la ruta ya creada se desarrolló la función “buscarNodos()” en el archivo “nodo.js” en la carpeta “controllers”. Esta función recibe el parámetro por URL con el valor a buscar y los compara con los campos: puerto, tipo y nombre de todos los nodos. Comprueba que nodos coinciden y los devuelve si es que existe alguno que coincida:

- **Status:** 200.
- **Nodo:** buscarNodos.

En caso de no haya un nodo con ese valor en los campos:

- **Status:** 404.
- **Nodo:** No se encontraron nodos con esos parámetros de búsqueda.

Por último, en caso de que hay un problema en la petición que no esté involucrada directamente con los valores de tipo de nodo y nombre devolverá:

- **Status:** 500.
- **Mensaje de Error:** “Error actualizando el nodo”.

Una vez programada la lógica se comprobó el correcto funcionamiento de la petición mediante Postman:

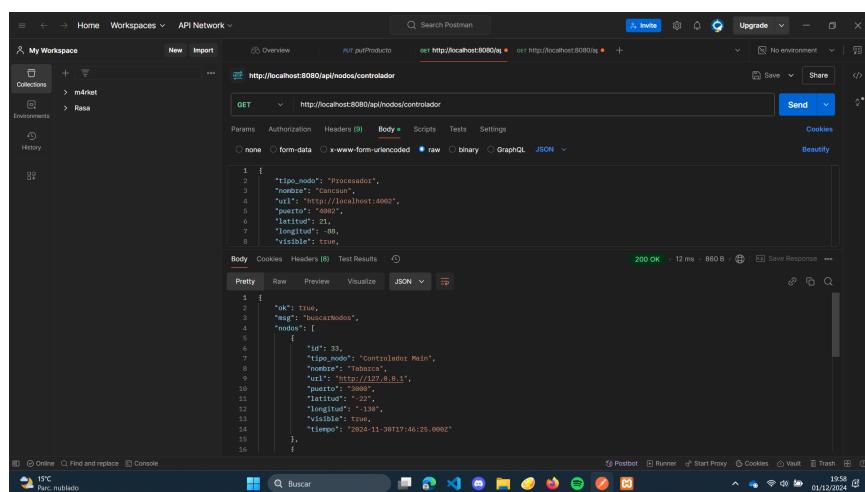


Figura 65. Petición SEARCH correcta en Postman (Fuente Propia)

Una vez se tuvo la lógica funcional se pasó a enlazarla con el frontend. Primero se creó en el servicio de la API la función “buscarNodos(datos)” que le pasa a la URL definida en el routes la el dato a buscar. Seguidamente, se creó la función “buscarNodos(\$event.target.value)” en el componente lista y se enlazó con el input dedicado al buscador. Esta función recoge el valor introducido en el campo y llama la función del servicio que se encarga de la petición buscar pasándole el valor a encontrar. Se realiza la petición y se llama a la función “actualizarDataSource(datos)” a la que se le pasan los datos recuperados por la petición y actualiza la lista con lo encontrado.

Tipo Nodo	URL	Puerto	Nombre	Geolocalización	Actions
Balanceador Main	http://127.0.0.1	3100	Madrid	Latitud: 22 Longitud: 130	
Balanceador Subs	http://127.0.0.1	3101	Barcelona	Latitud: 40 Longitud: 13	
Balanceador Main	http://127.0.0.1	3002	De más	Latitud: 0 Longitud: 0	

Figura 66. Funcionamiento del resultado de una búsqueda (Fuente Propia)

6.7.6 Implementación de la función descargar

Una vez implementadas las funciones de actualizar, activar o desactivar visibilidad y buscar para acabar con las características de la vista de Lista solo falta la función de descargar.

Esta función permite descargar un JSON con los nodos del sistema y sus valores con el fin de poder transportarlo de un sistema a otro. Para su desarrollo se le asignó al botón “Descargar” de la interfaz una función que realiza la petición GET ALL y recupera los datos de todos los nodos. Después con la función JSON.stringify(data, null, 2) los convierte en un formato legible y apto para un archivo de tipo .json y crea un Blob donde adjunta los datos bien identados por la anterior función. Crea la url del Blob y el elemento a descargar y los vincula y, por último, lo descarga asignando el nombre “listaNodos.json” al archivo.



```

1  descargar(): void {
2      this.http.getNodos().subscribe(
3          (data) => {
4              const jsonData = JSON.stringify(data, null, 2);
5              const blob = new Blob([jsonData], { type: 'application/json' });
6              const url = window.URL.createObjectURL(blob);
7
8              const a = document.createElement('a');
9              a.href = url;
10             a.download = 'listaNodos.json';
11             a.click();
12
13             window.URL.revokeObjectURL(url);
14         },
15         (error) => {
16             console.error('Error al descargar el JSON:', error);
17         }
18     );
19 }

```

Figura 67. Función que se encarga de descargar la lista de nodos (Fuente Propia)

6.8 Mejoras II

Una vez acabadas las mejoras de la vista Lista se plantearon los siguientes avances del proyecto. Estos van a ser: desarrollar el estado interno junto a la última consulta de la información del nodo, que se dibujen las conexiones de los nodos en el mapa, implementar el apartado de configuración del mapa y solucionar un problema con la recarga de la información.

6.8.1 Implementación del estado interno

La razón por la cual se pensó en hacer un modal con la información del nodo era ver los datos estáticos de este, que son los almacenados en la base de datos, y los dinámicos que son los que en ese momento está manejando el nodo en ese momento. Estos segundos son los que llamamos estado interno y en esta parte de la iteración vamos a recogerlos y mostrarlos en pantalla.

La información que necesitamos del estado interno viene dada por la petición “DEBUG” que coge la URL de los nodos y con ella realiza la consulta obteniendo los datos del estado interno. De momento al modal de información solo le pasábamos el “id” del nodo seleccionado para obtener su información de la base de datos, pero ahora en las tres pantallas: grafo, lista y mapa, se han modificado para también la URL del nodo cuando se abre el modal. Se decidió realizarlo así debido a que es una información que ya tenemos en las vistas y pasar solo el “id” y realizar otra vez la petición costaba más tiempo de carga. Con la URL ya pasada por parámetro al modal se creó la

instancia “estado_interno” y la función “getEstadoInterno()”. Esta última se declara nada más se abre la página y hace la petición debug con la que obtiene los datos y los almacena en la instancia.

Una vez almacenados los datos se recuperan en el html del componente usando la instancia. Por otro lado, el nodo puede estar no funcional lo que hace que no se devuelva a un estado interno. Para diferenciar esta condición se usa “ngIf” que determina si el nodo está funcionando. En caso de que lo esté muestra es estado interno si no lo está mostrará el texto: “Nodo fuera de funcionamiento”.

Por último, para formatear el estado interno y hacerlo más legible antes de almacenar los datos devueltos por la petición en la instancia se convirtieron con la función JSON.stringify(res, null, 2) en una cadena escrita como un archivo JSON para una mejor lectura de la información y se decoró con CSS haciendo un recuadro negro con color de letras blancas dando la sensación de código. Además, se le dió el atributo “overflow-y: auto” para que en el caso de ser un texto muy largo el modal no perdiera la forma y tuviera un scroll dentro del cuadrado de información.



Figura 68. Modal de información con estado interno (Fuente Propia)



Figura 69. Modal de información sin estado interno (Fuente Propia)

6.8.2 Desarrollo y gestión de la última consulta

Una vez implementado el estado interno lo siguiente fue realizar la gestión de la última consulta. Este campo de información ayuda al usuario a saber cuándo fue la última vez que se consultó el modal de información del nodo.

Para este desarrollo se añadió en la base de datos la variable “tiempo” de tipo DATETIME que permite guardar los datos de la última consulta. Después se hicieron las modificaciones correspondientes a las peticiones ya creadas. En el caso de las peticiones GET solo se incluyó el campo para recuperarlo, por parte de la petición POST se decidió que como se creaba el nodo y no había consulta el campo se pasaría vacío y para la petición PUT se decidió que sólo se actualizará el tiempo cuando se cierra el modal del nodo consultado. Para esto último, se creó la función “actualizarTiempo()” la cual llama a la petición PUT del servicio de la aplicación y le pasa los mismos datos que tiene salvo el tiempo que le pasa la fecha actual con la función “new Date()”.

Finalmente, como hemos dicho antes se deja la fecha vacía cuando se crea un nodo, así que decidimos usar otro “ngIf” que compruebe el valor del tiempo en la base de datos y si no es vacío pon la fecha que almacena, en caso de ser vacío devuelve el mensaje: “Primera vez que se consulta”.

Procesador Procesador4

Puerto	URL
4004	http://127.0.0.1
Última consulta	Geolocalización
2024-11-30T11:45:34.000Z	Latitud: 50 Longitud: 80

Figura 70. Modal con la fecha de consulta (Fuente Propia)

Procesador España

Puerto	URL
4008	http://127.0.0.1
Última consulta	Geolocalización
Primera vez que se consulta	Latitud: 12 Longitud: 30

Figura 71. Primera consulta al modal (Fuente Propia)

6.8.3 Implementación del apartado de configuración

En la última reunión que se realizó se acordó establecer un archivo JSON que contuviera los valores estéticos de los iconos del grafo y del tiempo de recarga de la función que se encarga de comprobar si hay actualizaciones en el sistema.

Los campos estéticos de los nodos son: tamaño, forma, color asignado a los tipos según si está activo o no, un valor para un PNG como ícono de cada tipo de nodo y nombre que van a tomar los平衡adores y controladores según su tipo. Los PNG's de momento estarán vacíos ya que se debe consultar con los tutores que imagen buscan. Una vez se tuvieron claros los valores se creó el archivo "config.json" dentro de la carpeta "assets/theme" del proyecto.

```

1  {
2      "controller": {
3          "colorA": "#FFA500",
4          "colorD": "#CCC0AC",
5          "nameMain": "C main",
6          "nameSubs": "C subs",
7          "shape": "circle",
8          "png": "",
9          "size": 50
10     },
11     "balancer": {
12         "colorA": "#1E90FF",
13         "colorD": "#97B7D7",
14         "nameMain": "B main",
15         "nameSubs": "B subs",
16         "shape": "square",
17         "png": "",
18         "size": 50
19     },
20     "processor": {
21         "colorA": "#A75DB3",
22         "colorD": "#A281A8",
23         "shape": "diamond",
24         "png": "",
25         "size": 35
26     },
27     "timer": {
28         "valor": 3000
29     }
30 }

```

Figura 72. Archivo de configuración (Fuente Propia)

Creado el archivo de configuración se desarrolló un servicio llamado “config.service.ts” alojado en la carpeta “src/app/services” con la función “getConfig()” la cual realiza una llamada http de tipo GET al archivo de configuración y devuelve su contenido. Una vez realizado el servicio este se importó a los componentes de la aplicación y se declaró en el constructor junto a la instancia “configuración” y la función “initConfig()”. Esta función usa la llamada GET del servicio y los valores que obtiene los almacena en la instancia creada. De esta manera tenemos los datos en la instancia y solo queda declarar las comprobaciones tanto en grafo como en mapa para cambiar el aspecto de los iconos.

No obstante, al recordar que OSM no tiene la capacidad de ECharts de definir figuras, y por eso hemos estado usando el icono del portátil en el mapa, los valores de forma del apartado de configuración no se podían implementar en el mapa. Tras esta incompatibilidad se hizo una reunión el día 14 de noviembre y se tomó la decisión de que como Echarts si permite meter iconos cambiar las figuras por PNG’s que se pudieran usar en ambas vistas. Con este cambio el archivo de configuración cambiaba ya que no se iban a definir ni forma ni color y se reemplazaron por los

PNG's de nodo activado y desactivado para cada tipo. Para probar el apartado de configuración se descargaron los PNG's de la página de Flaticon y se almacenaron en la carpeta "assets/map".



Figura 73. Iconos de balanceadores (Fuente Propria)

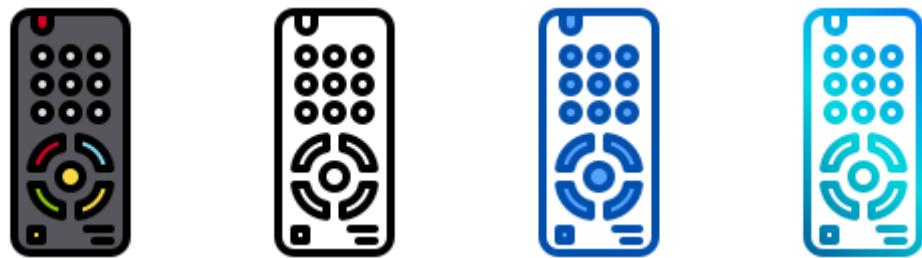


Figura 74. Iconos de controladores (Fuente Propria)

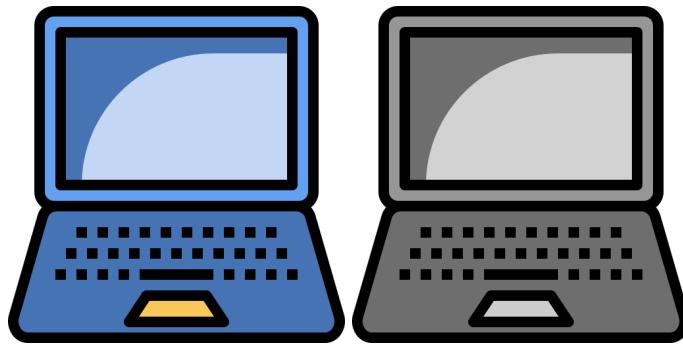


Figura 75. Iconos de procesadores (Fuente Propria)

```

1  "controller": {
2      "nameMain": "C main",
3      "nameSubs": "C subs",
4      "pngMain": "cMainAct.png",
5      "pngSubs": "cSubsAct.png",
6      "pngMainDes": "cMainDes.png",
7      "pngSubsDes": "cSubsDes.png",
8      "size": 60
9  },
10  "balancer": {
11      "nameMain": "B main",
12      "nameSubs": "B subs",
13      "pngMain": "bMainAct.png",
14      "pngSubs": "bSubsAct.png",
15      "pngMainDes": "bMainDes.png",
16      "pngSubsDes": "bSubsDes.png",
17      "size": 60
18  },
19  "processor": {
20      "png": "procesadorAct.png",
21      "pngDes": "procesadorDes.png",
22      "size": 60
23  },
24  "espera": {
25      "valor": 5000
26  }
27 }
```

Figura 76. Archivo de configuración modificado (Fuente Propia)

Finalmente, una vez se tuvo el archivo de configuración cambiado y los PNG's descargados se implementaron en los componentes grafo y mapa cogiendo sus valores de las instancias creadas anteriormente que almacenan los datos de configuración. Tanto para el mapa como el grafo se han usado ternarias que comprueban que tipo de nodo es, si es sustituto o principal y si está activo o no para asignarle el icono de configuración pertinente. Y en caso del tiempo de comprobación de los cambios se pagina a la función “timer()” de RxJS el valor de la instancia “configuracion” definida.

6.8.4 Implementación de las conexiones en el mapa

Lo primero fue informarse sobre cómo representar líneas en OSM (Open Street Maps) para dibujarlas en nuestro mapa. Tras investigar la manera de dibujar líneas vimos que igual que puedes agregar iconos puedes agregar líneas. Para hacerlo se usa el elemento “polyline” de OSM al que se le debe pasar dos puntos que contengan la latitud y la longitud del mapa. Además, con atributos como color, grosor, opacidad... .

```

1 L.polyline([pointA, pointB], {
2     color: 'grey',
3     weight: 3,
4     opacity: 0.7,
5 })

```

Figura 77. Como crear una polyline (Fuente Propria)

Teniendo en cuenta que necesitamos saber la posición con latitud y longitud en el mapa, no como en el grafo que enlazaba los nodos por el “id”, se cogieron las comprobaciones hechas para el grafo se pasaron al componente mapa dentro de la función “updateMapa()”, donde también se obtienen los nodos de la base de datos, y se adaptaron para que recogiera la latitud y longitud del nodo que apunta y el nodo objetivo en un array. Una vez recogidas las conexiones se llama a la nueva función “updateLineas(conexiones)” que recibe el array de conexiones y crea para cada valor del array un “polyline” con las posiciones y lo añadir al mapa con “addTo(this.map)”.

Después de esto se vió que dibujaba bien las líneas pero no es como ECharts que cuando actualiza borra los que ya están y pinta los nuevos, en OSM funciona como un dibujo así que hay que borrar la conexión si ya no existe. Para ello se creó la instancia “links” donde se guardan las líneas dibujadas en el mapa y se modificó la función “updateLineas(conexiones)” para que compare con un bucle los dos arrays cada vez que haya una actualización. De este modo si la conexión se ha cortado porque uno de los elementos se ha apagado se borra del mapa con la función “removeLayer(línea)”.



Figura 78. Vista mapa con conexiones (Fuente Propria)

Una vez se tuvieron las conexiones y su actualización según los cambios del sistema se pasó a la última tarea de conexiones en el mapa, desactivar las conexiones. Para ello se trajo el elemento checkout de la vista del grafo a la del mapa. Después se creó la instancia “activarConexiones” de tipo booleana que almacena si las conexiones deben estar activas o no y se declara al principio siempre a true. Por otra parte, a la acción de marcar el checkbox se le vinculó la función “conexiones(event)” que recoge el evento y cambia de valor la instancia booleana. Una vez cambia el valor, si pasa a ser false la función recorre con un bucle for la instancia “lineas” donde se guardan las líneas que se están pintadas en ese momento y las borra con la función de Leaflet “removeLayer(línea)”, en cambio, si pasa a ser true llama a la función hace la petición Debug para que obtenga los valores del sistema y se los pasa a la función “updateMapa(data)” que es la que se encarga de actualizar los datos del mapa y pintar los nuevos datos.

6.8.5 Solución al problema de recarga

Una vez acabamos las conexiones se abordó un fallo que surgió en la reunión del 14 de noviembre con el fin de corregirlo. El fallo consiste en que si se cambia de pantallas durante un tiempo considerado se van activando las funciones de comprobar que lanzan la petición Debug para obtener si hay actualizaciones en el sistema y como no están programadas para parar cuando cambias de pantalla se van acumulando y llega el momento que son tantas que no carga ninguna, se genera una sobrecarga de peticiones que no deja funcionar el servicio. De este fallo nos dimos cuenta cuando llevábamos tiempo probando el servicio ya que de normal cuando se desarrolla se va recargando continuamente los nuevos avances y debe pasar un tiempo para que se acumulen.

Con el fin de arreglar esta posible sobrecarga de peticiones se pensó que cada vez que cambie de pantalla destruir la petición de comprobación así al cambiar de pantalla no se acumularía y surgiría el problema. De modo que en los componentes grafo y mapa que son donde se realizan las comprobaciones se declaró la instancia “timerSubscription” que guarda la función “timer()” de RxJS y se usa “ngOnDestroy()” de angular que se lanza cada vez que se sale de la pantalla vinculada al componente. Dentro de esta función se comprueba si la instancia existe y en caso de ser así, cómo es una petición se usa la función “unsubscribe()” para detenerla.

No obstante, aunque ya no se produce esta sobrecarga se implementó la instancia “datosDelSistema” y la función “comprobar()” con el fin de que solo se actualice cuando haya cambios en el sistema. Ahora el ciclo de la aplicación sería el siguiente tanto en el grafo como en el mapa: se inicia el componente y se representan los datos que se guardan en la variable instancia “datosDelSistema”, seguidamente se llama a la nueva función “comprobar()”. Esta función realiza la petición Debug que devuelve los datos del sistema en tiempo real, después los compara con los

almacenados en la instancia, en caso de haber cambio llama a las respectivas funciones de cada componente que se encargan de realizar los cambios, por el lado del grafo: “updateGrafo(data)”, y por el lado del mapa “updateMapa(data)”, y en caso contrario si no hay cambios no los realiza. Por último al final de la función se llama de nuevo a sí misma con la función “timer()” y el tiempo de espera definido en la configuración previa.

```
1 this.timerSubscription = timer(this.configuracion.espera.valor).subscribe(() => this.comprobar());
```

Figura 79. Función recursiva de comprobación almacenada en la instancia (Fuente Propia)

6.9 Mejoras III

Tras la anterior iteración se tuvo una reunión el 25 de noviembre en la que se evaluó el proyecto hasta la fecha. En esta reunión se puso en práctica el funcionamiento del sistema y se sacaron los últimos aspectos a mejorar. Por tanto planeó esta iteración para abordar estos últimos retoques.

6.9.1 Mejoras en el modal de información

Las mejoras acordadas se dividen en mejoras de diseño y lógica de la aplicación. Se propuso empezar por los de diseño ya que las mejoras de lógica eran con el fin de mejorar la aplicación.

La primera mejora es el rediseño del modal de información del nodo. En este modal anteriormente se mostraban los datos del nodo de la base de datos por un lado y por otro lado, mediante una representación en forma de código, los datos del estado interno que tiene en ese momento. No obstante, en esta disposición se gastaba mucho espacio para los datos del nodo que se podrían resumir más y la información del estado interno no era fácil de interpretar visualmente a primera vista.

Para mejorar este aspecto se decidió distribuir la información de otra manera en la pantalla. Por un lado, a la izquierda del modal en forma de columna se representan los datos del nodo de la base de datos y los atributos del estado interno menos la lista de balanceadores. La lista de balanceadores es un atributo que tienen los balanceadores y controladores activos y almacena los procesadores que están recibiendo carga de trabajo en el sistema, su capacidad y su porcentaje de carga. Como la información que contiene es la más relevante se recoge y se muestra en la derecha del modal en forma de tabla para una mejor accesibilidad. Para ello se crearon las interfaces de los tres tipos de nodo, ya que cada nodo tiene diferente información. Después se creó la función

“getEstadoInterno()” que con el nodo que se le pasa al modal hace la llamada Debug para obtener el estado interno y lo almacena en una instancia del tipo de la interfaz correspondiente. Luego en el “html” del componente se crean los campos de información con varios “ngIf” que comprueban que tipo de nodo es y si está activo o no para mostrar la información del estado interno tanto la tabla que contiene la lista de nodos como los campos que dependen este mismo.

Resumiendo, todos los procesadores tendrán en la columna de la izquierda sus datos, si están activos se le añadirá a esa columna los datos del estado interno y en caso de ser balanceadores o controladores también se creará la tabla con la lista de procesadores.

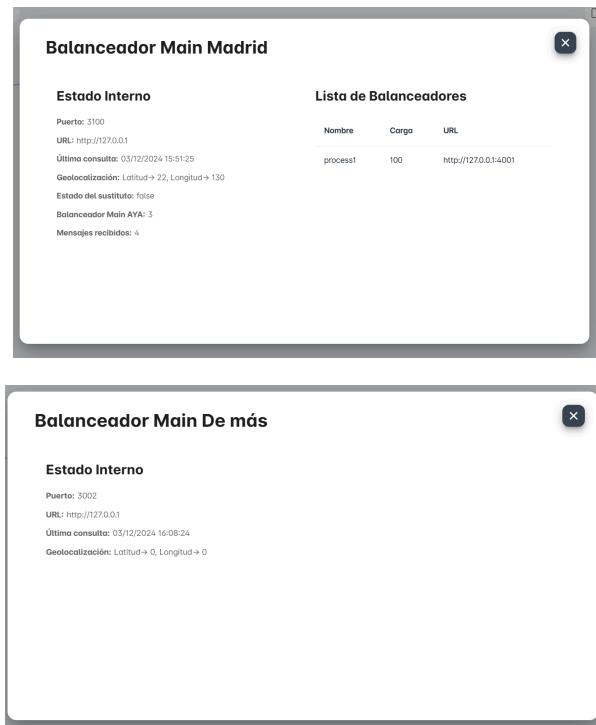


Figura 80. Nuevo modal de información para平衡adores según estado (Fuente Propia)

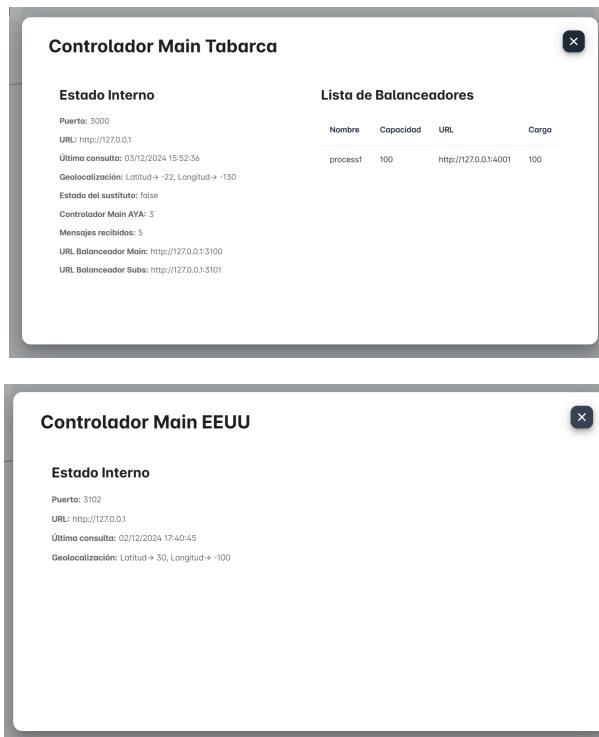


Figura 81. Nuevo modal de información para controladores según estado (Fuente Propia)

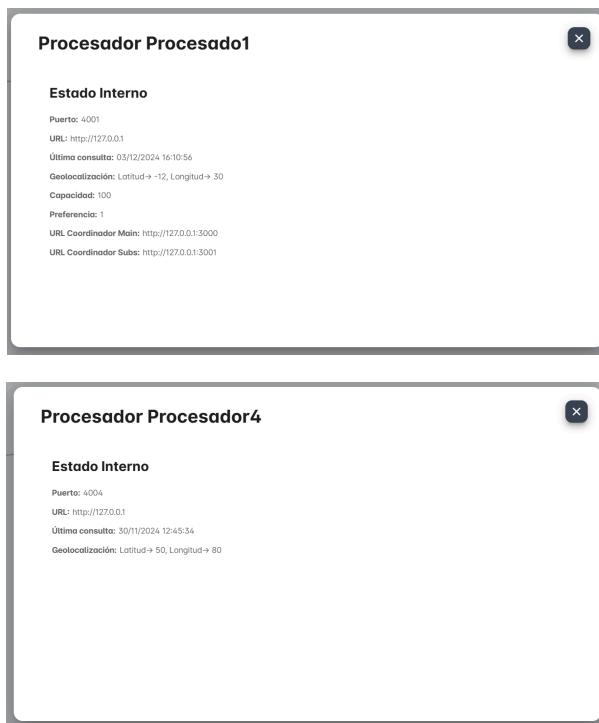


Figura 82. Nuevo modal de información para procesadores según estado (Fuente Propia)

Por último, en la fecha de la última consulta se modificó para que mostrara la fecha con la hora minutos y segundos para que fuera más exacta, ya que si por ejemplo se había consultado ese mismo día solo iba a salir la fecha pero no la hora y eso podía generar confusión. Se importó la librería DatePipe de Angular y dentro de “ngOnInit()” del modal donde se cargan los datos del nodo con la llamada GET BY ID se transforma el formato de la fecha de la última consulta con la función “transform()” de la librería importada que recibe el parámetro a formatear y el formato que se le quiere dar. Posteriormente, se guarda en la instancia “nodo” que se llama desde el “html” para cargar los valores del nodo en los campos. De esta manera ya tenemos la fecha con horas, minutos y segundos.



```
1 this.nodo.tiempo = this.datePipe.transform(this.nodo.tiempo, 'dd/MM/yyyy HH:mm:ss') || '';
```

Figura 83. Función que aplica el formato a la última consulta (Fuente Propia)

6.9.2 Visualización de errores en los formularios

Hasta la fecha las peticiones controlan las condiciones definidas. Estas condiciones son que la URL de dos nodos del mismo tipo no sean iguales y que no haya campos vacíos. Pero el usuario a la hora de usar la aplicación no hay nada en la interfaz de los formularios que le indique porque no se realizan los cambios.

Con el fin de representar los cambios tanto en el modal de actualizar como en el de agregar se modificaron las funciones que se encargan de llamar a las peticiones POST y PUT. Se añadió la instancia “peticionError” en los componentes y al ejecutarse la función se implementó un try catch que en caso de haber un error en la petición iguala la instancia a los errores que han sucedido. Para manejar estos errores, al ser dos tipos los de campo vacío y el de URL incorrecta, el segundo error solo se muestra cuando todos los campos están completos.

Estos errores se reflejan en el formulario en color rojo debajo de los campos correspondientes con el fin de que sea visible y así el usuario pueda corregir los errores.

Figura 84. Representación de errores en los formularios (Fuente Propia)

6.9.3 Nuevos iconos

En la revisión en la reunión se repasaron los iconos para los nodos según sus tipos y estado. Se llegó a la conclusión de que era mejor usar iconos más simples con un solo color.

Decidida esta idea se buscaron nuevos iconos más simples en la página de Flaticon. Una vez se encontraron los iconos que encajaban con los que se buscaba se descargaron en diferentes colores para indicar si era de un tipo u otro, para saber si era principal o sustituto y para saber si estaba activo a indicativo. Con tal de representar si era principal o sustituto de su tipo se eligieron iconos llenos para los principales y otros con solo el borde para los sustitutos. Para la actividad e inactividad se decidió el color negro cuando estuvieran inactivos y para cuando estuvieran activos se decidieron usar los colores de la paleta de figuras de los bocetos, es decir, lila para los procesadores, naranja para balanceadores y azul para controladores. De todos modos estos valores se asignan a la configuración con el fin de que se elijan los que se quieran.

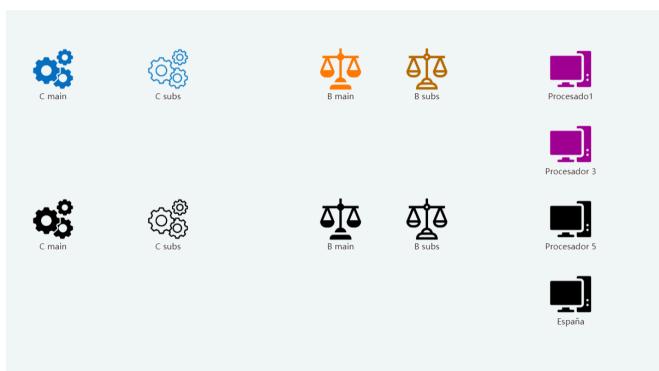


Figura 85. Nuevos iconos para los nodos (Fuente Propia)

A parte, se decidió incluir estos iconos en las opciones de tipo nodo como mejora de aspecto. Para esto se incluyó un elemento imagen dentro de cada opción con el tamaño correspondiente.

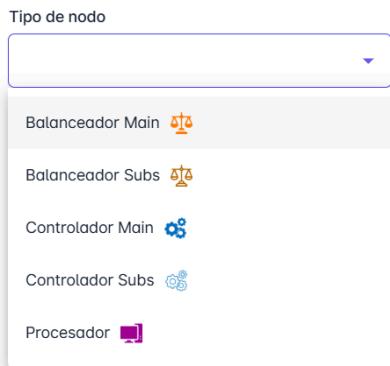


Figura 86. Iconos en las opciones de tipo nodo en los formularios (Fuente Propia)

6.9.4 Implementación de distribución y ordenación de nodos

Con la idea de que la vista del grafo no fuese muy difícil de interpretar a simple vista se propuso hacer que dependiendo el tipo de nodo se representará en una parte de la pantalla. Para esta distribución se tuvieron en cuenta las conexiones. Como los balanceadores son los nodos que conectan tanto con controladores como procesadores se decidió que estos irían en medio de la pantalla y a los lados los otros tipos.

Para hacerlo se creó la instancia “zonas” que contiene los valores de ancho y alto que delimita la zona de representación para cada tipo de nodos en el grafo. Al suponer que los procesadores van a ser más numerosos en el sistema se les dió una zona más grande para su representación. Una vez definidas las zonas, cuando se establecen los parámetros del grafo, se calculan las variables “x” e “y” que establecen las variables del nodo. Para ello se compara el tipo de nodo y se selecciona la zona que le corresponde y una vez se sabe la zona en la que se va a representar se calculan sus coordenadas en el grafo, las cuales posteriormente se asignan a los valores de nodo para que ECharts los ubique.

```

1 // Determinar la zona correspondiente al tipo de nodo
2     const zona = this.zonas[tipo_nodo];
3     const anchoZona = zona.xFin - zona.xInicio;
4     const altoZona = this.alto - 2 * this.margenZona;
5
6     // Índice local dentro de la zona
7     const nodosZona = res.filter((n: any) => n.nodo.tipo_nodo === tipo_nodo);
8     const indexZona = nodosZona.findIndex((n: any) => n.nodo.id === nodo.id);
9
10    // Calcular la cantidad dinámica de columnas
11    const nodosPorColumna = Math.floor(Math.sqrt(nodosZona.length)); // Basado en un layout cuadrado
12    const espacioX = anchoZona / nodosPorColumna; // Espacio horizontal entre columnas
13    const espacioY = altoZona / Math.ceil(nodosZona.length / nodosPorColumna); // Espacio vertical entre filas
14
15    // Calcular posición del nodo
16    const columna = indexZona % nodosPorColumna; // Columna actual
17    const fila = Math.floor(indexZona / nodosPorColumna); // Fila actual
18
19    const x = zona.xInicio + columna * espacioX + this.margenZona;
20    const y = this.margenZona + fila * espacioY;

```

Figura 87. Cálculo de las zonas para los nodos (Fuente Propia)

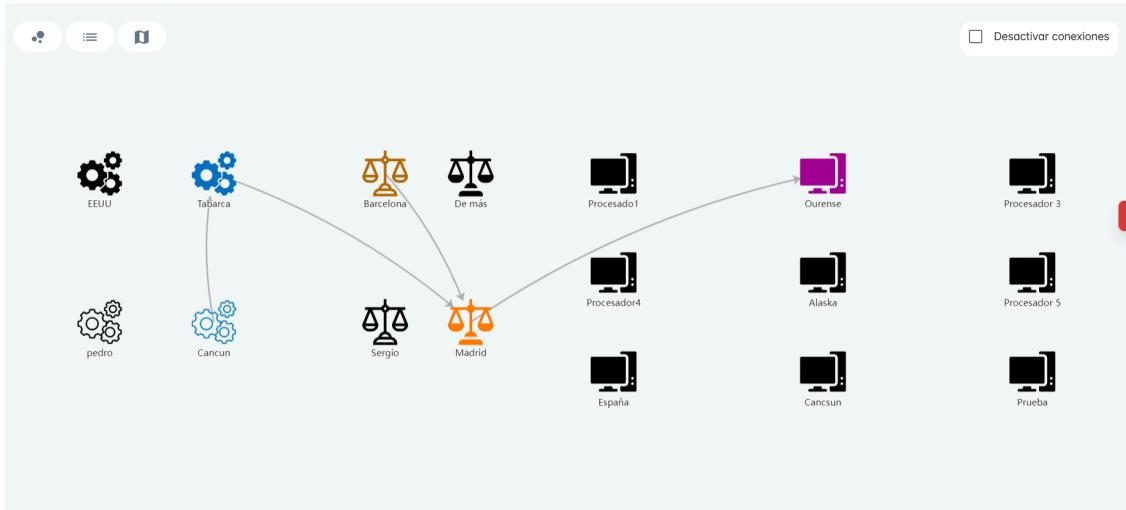


Figura 88. Distribución visual de nodos en el Grafo (Fuente Propia)

Una vez se terminó la distribución se planteó la cuestión de que sería interesante poder establecer el orden de los nodos ya que los recupera en el orden de la base de datos. Para abordar esta solución se creó un campo nuevo de tipo “number” en la base de datos llamado “orden” con el fin de que se pueda dar un orden específico a los nodos en el grafo.

Una vez creado el campo lo primero fue incluir este campo en el modelo de Sequelize y en el resto de peticiones. Después se creó una petición nueva de tipo GET. Para ello en el archivo “nodo.js” de la carpeta “backend/miback/routes” implementó la ruta a la llamada GET:

- **Ruta GET ALL BY ORDER:** “<http://localhost:8080/api/nodosOrden>”.

Con la ruta ya creada se desarrolló la función “buscarNodosOrdenados()” en el archivo “nodo.js” en la carpeta “controllers”. La función realiza un “findAll()” al que se le pasa la condición de que estén ordenados por el campo “orden” de manera ascendente:

- **Status:** 200.
- **Nodo:** getNodosOrdenados.

En caso de que hay un problema en la petición devolverá:

- **Status:** 500.
- **Mensaje de Error:** “Error al recuperar los nodos por orden”.

Una vez programada la lógica se comprobó el correcto funcionamiento de la petición mediante Postman:

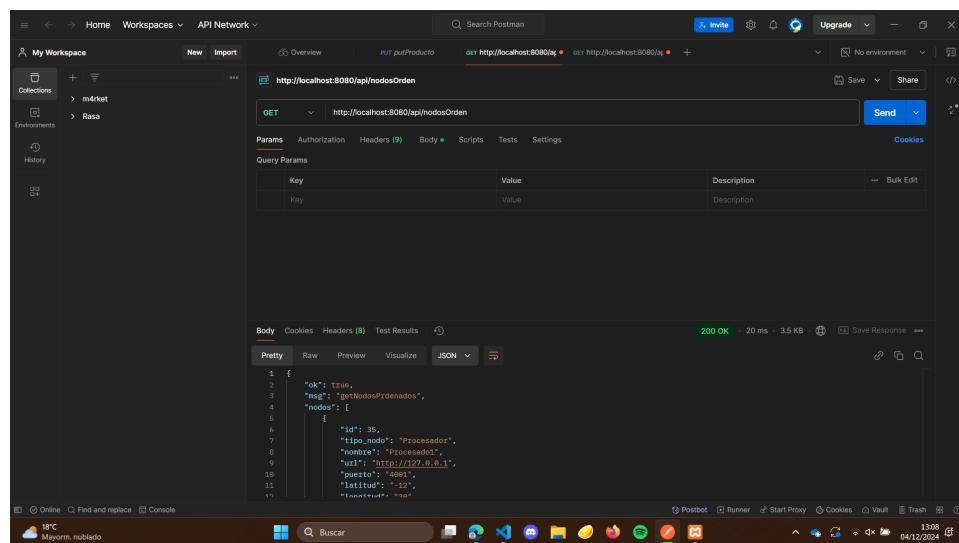


Figura 89. Petición GET ALL BY ORDER correcta en Postman (Fuente Propia)

Al tener la distribución en zonas y la ordenación en la llamada de la API no hay que preocuparse por comprobar de qué tipo son ya que de eso se encarga la distribución en zonas, es decir, si un balanceador y un controlador tienen el mismo valor en el orden dará igual porque lo pintará en su orden correspondiente su zona asignada en el nodo. Por otro lado, se creó la llamada a la petición GET ALL BY ORDER en el servicio y se cambiaron todas las llamadas GET ALL del componente Grafo.

Finalmente, lo último que falta es incluir este campo en los formularios, tanto en añadir nodo como en actualizar nodo, y en la pantalla de información. De modo que se añadió una fila más en la parte de columna de estado interno del modal de información con el dato orden del nodo y se implementó el nuevo campo en los formularios.

Estado Interno

Puerto: 3101
URL: http://127.0.0.1
Última consulta: 04/12/2024 12:19:05
Geolocalización: Latitud→ 60, Longitud→ 13
Orden: 2
Estado del sustituto: false
Balanceador Main AYA: 3
Mensajes recibidos: -1
URL Main: http://127.0.0.1:3100

Figura 90. Modal de información con el campo Orden (Fuente Propia)

Añadir Nodo

Tipo de nodo	Nombre	
<input type="text"/>	<input type="text"/>	
URL	Puerto	Orden
<input type="text"/>	<input type="text"/>	<input type="text"/>
Geolocalización	Visibilidad	
<input type="text"/> Latitud	<input type="text"/> Longitud	<input type="text"/> Activada
Añadir		

Balanceador Main: Madrid

Tipo de nodo	Nombre	
<input type="text"/> Balanceador Main	<input type="text"/> Madrid	
URL	Puerto	Orden
<input type="text"/> http://127.0.0.1	<input type="text"/> 3100	<input type="text"/> 123
Geolocalización	Visibilidad	
<input type="text"/> 22	<input type="text"/> 130	<input type="text"/> Activada
Actualizar		

Figura 91. Formularios con campo Orden (Fuente Propia)

6.9.5 Configuración del mapa

Después de esta última serie de implementaciones se tuvo una reunión para repasar el servicio. En esta reunión se propuso que como sus nodos están contenidos en España hacer un apartado configurable para la vista inicial del mapa de esta manera el mapa carga como se establece.

Los datos a guardar son la latitud y longitud del centro del mapa y el zoom de este. Se crearon los datos en el archivo “config.json” en la carpeta “assets/theme” debajo del campo que establece el periodo de recarga de cambios.



```
1
2 "espera": {
3     "valor": 3500
4 },
5 "map": {
6     "latitud": 0,
7     "longitud": 0,
8     "zoom": 2
9 }
```

Figura 92. Valores de configuración del mapa (Fuente Propia)

Por último, en el componente Mapa, en el que ya estaban importados los datos por el servicio de configuración, se cambió la función “setView()”, que se encarga de descargar las dimensiones del mapa, y se le pasaron los parámetros de configuración. De este modo se puede cambiar el enfoque del mapa a voluntad.

6.9.6 Tratamiento de nombres

Además en esa reunión se probó a poner un nombre muy largo en un nodo, ya que el Product Owner sugirió que iban a tener nombres largos. Esto supuso que en el grafo y en el mapa salieran las etiquetas con los nombre muy alargadas y quedará visiblemente poco atractivo y ocupaba espacio innecesario.

Con el fin de evitar este error se propuso la idea de cortar el nombre y poner puntos suspensivos en las etiquetas si pasaban un máximo. Para el grafo se aplicó al elemento “label”, que contiene el nombre, el atributo “formatter” que aplica el formato que se le va a dar a la etiqueta. Este atributo

se vinculó a una función que coge el parámetro nombre del nodo y si supera la longitud máxima establecida en 10 caracteres se corta lo que venga detrás de este último y se ponen los puntos suspensivos.



Figura 93. Formateo de un nombre largo en la vista Grafo (Fuente Propia)

Sin embargo, para Open Street Maps se usó CSS. Esto se debe a que los elementos de OSM se pueden manejar en CSS mediante la librería Leaflet. Por tanto, a los elementos de clase “.leaflet-tooltip” se le asignaron un ancho máximo y el atributo “text-overflow: ellipsis” que hace que cuando el texto supere el ancho establecido lo representa con puntos suspensivos.



Figura 94. Formateo de un nombre largo en la vista Mapa (Fuente Propia)

7. Pruebas y validación

Una vez se ha terminado con la implementación del sistema y los Product Owners dieron el visto bueno se pasó a comprobar el correcto funcionamiento del mismo.

7.1 Creación y activación de nodos

Para comenzar las pruebas del sistema se fue aumentando la dificultad de esta para ver como respondía el sistema. Primeramente, se añadieron los nodos del sistema en un orden y una geolocalización concretos. Una vez se crearon los nodos se procedió a encender unos nodos específicos del sistema para ver si activaba que se habían configurado correspondientemente y por ende sus conexiones pertinentes.

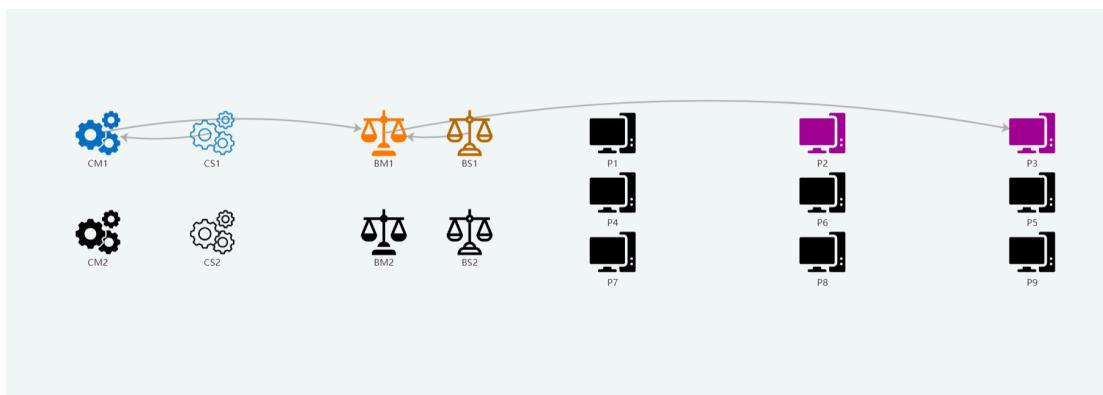


Figura 95. Funcionamiento de conexiones del sistema en grafo (Fuente Propia)

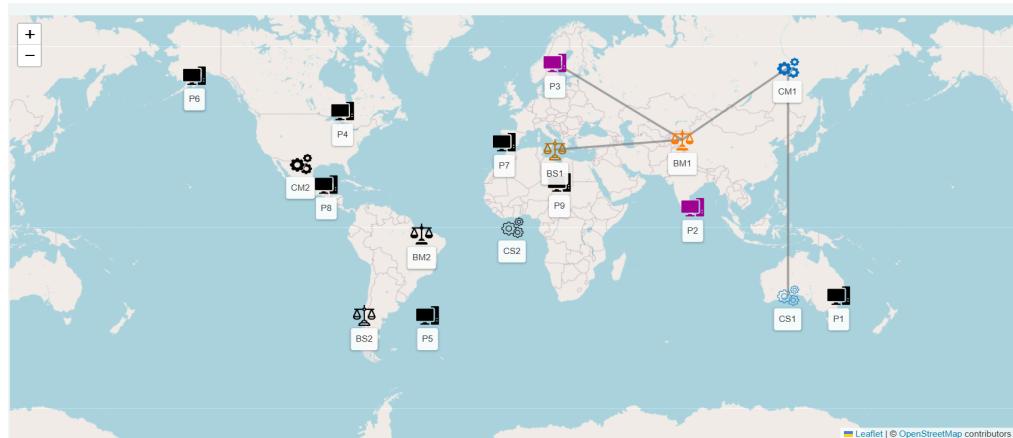


Figura 96. Funcionamiento de conexiones del sistema en mapa (Fuente Propia)

Resultados: La prueba se realizó de manera correcta. Los nodos se crearon y se almacenaron correctamente. La representación corresponde a los valores indicados tanto en el mapa como en el grafo. Por último, el sistema detectaba bien el arranque de los nodos correctos y así lo reflejaba cambiando los iconos correspondientes y generando las conexiones entre ellos.

7.2 Desactivación de nodos principales

Una de las necesidades más importantes es que los nodos sustitutos funcionen en caso de necesidad. Por eso el siguiente paso fue poner a prueba el cambio de rol cuando se apaga un principal. Para ello, se apagaron tanto balanceador como controlador principales.

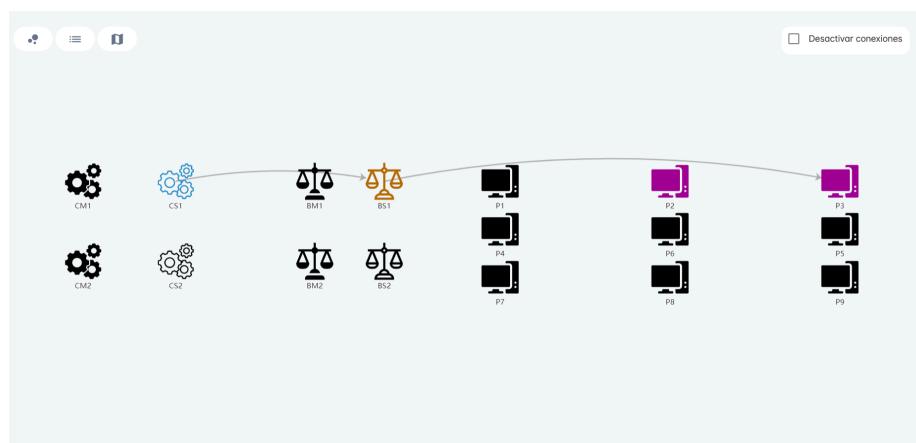


Figura 97. Funcionamiento de nodos sustitutos del sistema en grafo (Fuente Propia)

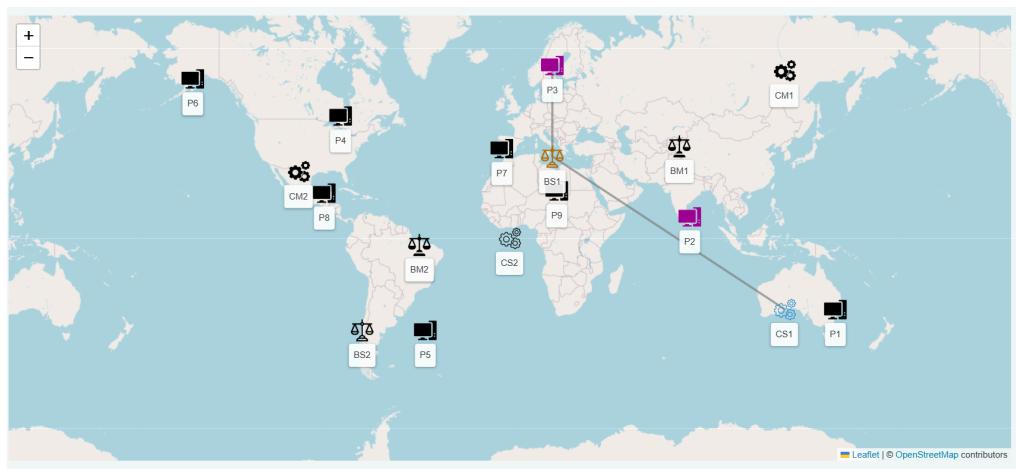


Figura 98. Funcionamiento de nodos sustitutos del sistema en mapa (Fuente Propia)

Resultados: se comprobó que si el principal fallaba el sustituto cogía su rol tanto en balanceadores como en controladores y se representaba correctamente en ambos sistemas de representación del servicio.

7.3 Funcionamiento del sistema para varios clústers

Después de comprobar las funcionalidades del sistema para un clúster los Product Owners proporcionaron otro, ya que uno de sus requerimientos era que el sistema funcionara para varios. Se crearon los nodos de la configuración del nuevo cluster y se arrancaron los dos para probar su representación. Una vez establecida su representación se desactivaron nodos y volvieron a activar para probar que no confundiese en algún momento las conexiones.

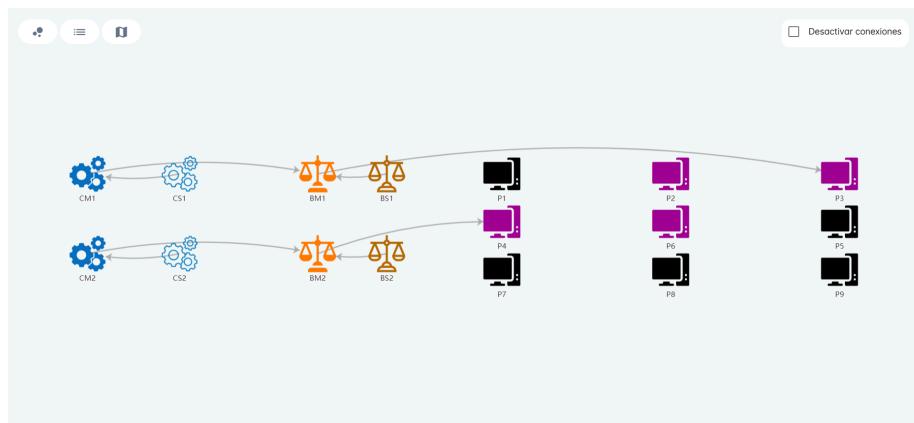


Figura 99. Funcionamiento de varios cluster en la vista grafo (Fuente Propia)



Figura 100. Funcionamiento de varios cluster en la vista mapa (Fuente Propia)

Resultados: se comprobó que el sistema sigue las conexiones establecidas en las configuraciones sin cometer errores de interpretación dejando una representación limpia que no da pie a equivocaciones.

Conclusiones: tras realizarse todas las pruebas exigidas por los Product Owners para poner a prueba la fiabilidad de la representación del servicio se vió que el sistema superó todas las pruebas. Esto quiere decir que se ha logrado alcanzar los requisitos establecidos al principio del trabajo quedando una interfaz limpia, atractiva, intuitiva, fácil de manejar y robusta.

8. Resultados

8.1 Producto Final

La propuesta inicial hecha por parte de José Vicente Berna y Lucía Arnaud era mejorar la interfaz de monitorización que existía para representar sistemas distribuidos. Esta interfaz era pobre y poco atractiva, ya que solamente mostraba la información del sistema de manera textual en párrafos.

Balancers	Subs Balancer		
Main Balancer	Subs Balancer		
Type:main - Port:3100 urlCoordinatorMain: balancerSubsActive:false - receivedMessages:4 - balancerMainMaxAYA:3 balancerList:[{"name":"process1","load":100,"url":"http://127.0.0.1:4001"}]	Type:subs - Port:3101 urlMain:http://127.0.0.1:3100 balancerSubsActive:false - receivedMessages:-1 - balancerMainMaxAYA:3 balancerList:[{"name":"process1","load":100,"url":"http://127.0.0.1:4001"}]		
Coordinators	Subs Coordinator		
Main Coordinator	Subs Coordinator		
Type:main - Port:3000 urlCoordinatorMain: urlBalancerMain:http://127.0.0.1:3100 - urlBalancerSubs:http://127.0.0.1:3101 coordinatorSubsActive:false - coordinatorMainMaxAYA:3 - receivedMessages:3 balancerList:[{"name":"process1","capacity":100,"url":"http://127.0.0.1:4001","load":100}] processorsList: [{"name":"process1","capacity":100,"url":"http://127.0.0.1:4001","preference":1,"AYACOUNT":3,"AYAUID":""}, {"name":"process2","capacity":100,"url":"http://127.0.0.1:4002","preference":0.3,"AYACOUNT":3,"AYAUID":""}]	Type:subs - Port:3001 urlCoordinatorMain:http://127.0.0.1:3000 urlBalancerMain:http://127.0.0.1:3100 - urlBalancerSubs:http://127.0.0.1:3101 coordinatorSubsActive:false - coordinatorMainMaxAYA:3 - receivedMessages:3 balancerList:[{"name":"process1","capacity":100,"url":"http://127.0.0.1:4001","load":100}] processorsList: [{"name":"process1","capacity":100,"url":"http://127.0.0.1:4001","preference":1,"AYACOUNT":3,"AYAUID":""}, {"name":"process2","capacity":100,"url":"http://127.0.0.1:4002","preference":0.3,"AYACOUNT":3,"AYAUID":""}]		
Processors	Precessor element 1 of 3	Precessor element 2 of 3	Precessor element 3 of 3
	name:process1 - port:4001 capacity:100 - preference:1 urlCoordinatorMain:http://127.0.0.1:3000	name:process2 - port:4002 capacity:100 - preference:0.3 urlCoordinatorMain:http://127.0.0.1:3000	name: - port:0 capacity:0 - preference:0 urlCoordinatorMain:

Figura 101. Interfaz anterior al desarrollo del proyecto (Fuente Propia)

Finalmente, me complace decir que el proyecto ha cumplido el objetivo principal de manera satisfactoria. Se ha creado una interfaz de monitorización y gestión de sistemas distribuidos teniendo en cuenta los requisitos establecidos durante la elaboración del producto. Esta se divide en tres vistas que conjuntamente representan toda la información esperada de modo accesible, intuitivo, fácil de interpretar y de manera atractiva.

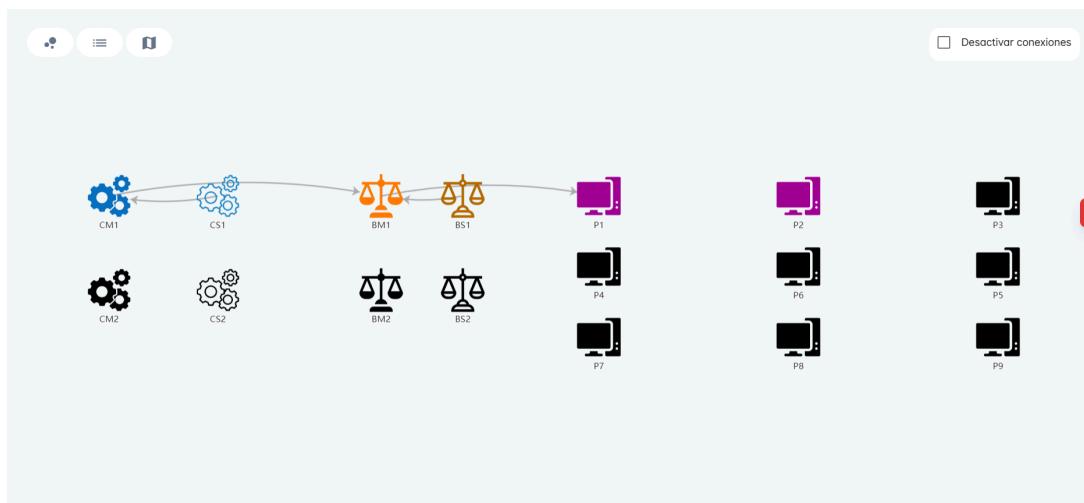


Figura 102. Interfaz final de la vista Nodo (Fuente Propia)

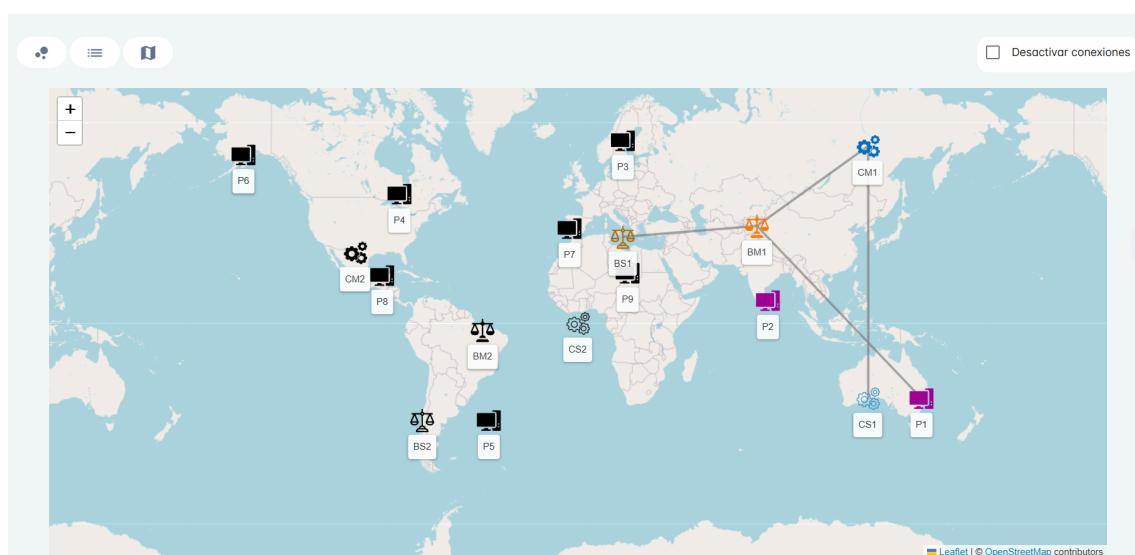


Figura 103. Interfaz final de la vista Mapa (Fuente Propia)

Tipo Nodo	Nombre	URL	Puerto	Geolocalización	Acciones
Balanceador Main	BM1	http://127.0.0.1	3100	Lat: 40 Lon: 80	
Balanceador Subs	BS1	http://127.0.0.1	3101	Lat: 36 Lon: 20	
Controlador Main	CM1	http://127.0.0.1	3000	Lat: 36 Lon: 130	
Controlador Subs	CS1	http://127.0.0.1	3001	Lat: -32 Lon: 130	
Procesador	P1	http://127.0.0.1	4001	Lat: -32 Lon: 154	
Procesador	P2	http://127.0.0.1	4002	Lat: 10 Lon: 85	
Procesador	P3	http://127.0.0.1	4003	Lat: 63 Lon: 20	
Balanceador Main	BM2	http://127.0.0.1	4100	Lat: -3 Lon: -43	
Controlador Main	CM2	http://127.0.0.1	5000	Lat: 30 Lon: -100	

Items per page: 1 – 10 of 24 < >

Figura 104. Interfaz final de la vista Lista (Fuente Propia)

8.2 Coste Temporal

En este punto se muestra el tiempo total del proyecto usando la herramienta Clockify. En la siguiente figura se ve una gráfica que refleja el tiempo dedicado al proyecto por mes.



Figura 105. Tiempo dedicado por mes al producto (Fuente Propia)

Como se aprecia el tiempo de elaboración del proyecto va desde Septiembre hasta Diciembre, ya que se decidió presentar el proyecto para su presentación en la convocatoria extraordinaria C1. El tiempo dedicado a la elaboración de este proyecto va dividido en meses. En el primer mes se investigó y se sentaron las bases, después en Octubre se desarrolló la estructura del proyecto, una vez se tuvo esta estructura revisada y aceptada se pasó a la implementación de funcionalidades en Noviembre, que fue el mes de más trabajo, y por último, en Diciembre se ultimaron los detalles del proyecto.

9. Conclusiones y trabajo futuro

Una vez finalizada la memoria se evalúa el proyecto en base a los logros alcanzados en relación con los objetivos, además de plantear cambios a futuro.

9.1 Conclusiones

El proyecto ha alcanzado los objetivos establecidos al comienzo de la memoria de manera gratificante al desarrollar una interfaz intuitiva y fácil de analizar para representación de sistemas distribuidos. Además, fuera de los objetivos previos se ha conseguido implementar una configuración que permite una personalización según necesidades individuales de cada sistema. Desde un punto de vista más personal, este proyecto ha supuesto un desafío diferente para mí, ya que al tratarse de un encargo en el que me dieron las especificaciones pero no una idea en concreto, sino que me dieron la posibilidad de abarcar el problema como creyese conveniente, ha hecho que desarrolle la capacidad de pensar en una interfaz completa y organizar las fases de desarrollo de la manera más práctica teniendo en cuenta las partes claves del proyecto. En este TFG he conseguido ampliar mis conocimientos de Angular y Node.js, además de aprender a usar las herramientas de representación de Echarts y Open Street Maps. Tengo la certeza de que estos conocimientos serán de gran ayuda para mí en el futuro.

9.2 Trabajo Futuro

Con respecto a futuras mejoras, se plantea desarrollar un apartado de errores de los nodos. Este apartado permitiría que en caso de fallo de un nodo, al consultar su estado interno apareciera el problema que ha sufrido con su información correspondiente. Además, para completar este sistema de fallos se implementaría un apartado complementario en la lista que almacene todos los errores que ha sufrido el nodo a lo largo de su funcionamiento. Esta mejora viene pensada ya que actualmente si un nodo falla se apaga al igual que si se apagara manualmente, ya que no llegan los errores que sufren directamente a través del estado interno. Con esta mejora se busca facilitar las labores de mantenimiento al proporcionar la causa que generó el error.

Referencias

1. Github. Disponible: <https://github.com/>
2. Plantilla FUSE de Angular. Disponible:
<https://angular-material.fusetheme.com/sign-in?redirectURL=%2Fdashboards%2Fproject>
3. Editor de código Visual Studio Code. Disponible: <https://code.visualstudio.com/>
4. Extensión GitGraph para Visual Studio Code. Disponible:
<https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph>
5. Angular CLI. Disponible: <https://v17.angular.io/cli>
6. Node.js. Disponible: <https://nodejs.org/en>
7. Angular Materials. Librería de componentes de angular. Disponible:
<https://v7.material.angular.io/>
8. TailwindCSS. Librería de aspectos usada en la plantilla FUSE. Disponible:
<https://tailwindcss.com/>
9. PHPmyAdmin. Disponible: <https://www.phpmyadmin.net/>
10. XAMPP. Disponible: <https://www.apachefriends.org/>
11. Express.js. Disponible: <https://expressjs.com/>
12. Dotenv. Disponible: <https://www.npmjs.com/package/dotenv>
13. Sequelize. Disponible: <https://sequelize.org/>
14. Postman. Disponible: <https://www.postman.com/>
15. RxJS. Disponible: <https://rxjs.dev/>
16. Express-validator. Disponible: <https://www.npmjs.com/package/express-validator>
17. Echarts. Herramienta de representación gráfica. Disponible:
<https://echarts.apache.org/en/index.html>
18. Open Street Maps. Herramienta de representación de mapas. Disponible:
<https://www.openstreetmap.org/#map=2/27.5/79.5>
19. Leaflet. Librería que maneja Open Street Maps en Angular. Disponible:
<https://leafletjs.com/>

Anexo

En este apartado se deja un enlace al repositorio donde se ha subido el código elaborado:

https://github.com/Martiinfierro/TFG_MFP