

PROYECTO

LUISTER

CICLO FORMATIVO DE GRADO SUPERIOR:
Desarrollo de Aplicaciones Web

AUTORES:

Sandra Fernández Sánchez

Fausto Obama Ngomo Afang

Henry David Orbe Cisneros

Licencia

Esta obra está bajo una licencia Reconocimiento-Compartir bajo la misma licencia 3.0 España de Creative Commons. Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-sa/3.0/es/> o envie una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

DEDICATORIAS/AGRADECIMIENTOS

RESUMEN

Luister se ha planteado como un proyecto de fin de ciclo de DAW. Con Luister se pretende integrar los conocimientos adquiridos durante la realización de este CFGS y aplicarlos en un caso práctico de desarrollo

Luister ha sido realizado por Sandra Fernández Sánchez, Fausto Obama Ngomo Afang y Henry David Orbe Cisneros.

Este proyecto consiste en la realización de una aplicación web dedicada a la música, en ella se podrán realizar diferentes actividades como creación de listas, consulta de novedades, detalles de artistas, nuevos lanzamientos, entre otros. Este proyecto ha constado de dos partes, diferenciadas en su ámbito pero que han compartido tiempo de desarrollo con el fin de garantizar una implementación correcta entre el entorno cliente y el entorno servidor.

La primera parte de este proyecto ha consistido en el desarrollo del entorno cliente realizado con Angular. Esta parte ha englobado el diseño de la interfaz de usuario en donde los usuarios podrán realizar las actividades anteriormente mencionadas.

La segunda parte ha comprendido el desarrollo del entorno servidor y de la base de datos usada por Luister usando MySQL, realizado usando Django Rest Framework.

Al finalizar estas partes se comenzó con el despliegue de la aplicación usando Docker, esta parte del proceso se ha aprovechado para la realización de pruebas y mejoras dentro de los entornos cliente y servidor.

Todo este trabajo se ha realizado con el fin de ofrecer a los amantes de la música un lugar en el que poder gestionar sus bibliotecas musicales sin depender de usar un solo servicio de música para lo que se ofrece la posibilidad de importar las listas en formatos usables en las plataformas de streaming. En un contexto en donde su uso es generalizado y donde la dependencia de estos servicios para construir una biblioteca musical es cada día mayor.

ABSTRACT

Luister has been proposed as a final project for the DAW cycle. The goal of Luister is to integrate the knowledge acquired during the completion of this cycle and apply it to a practical case of development.

Luister has been created by Sandra Fernández Sánchez, Fausto Obama Ngomo Afang, and Henry David Orbe Cisneros.

This project consists of developing a web application dedicated to music, where users can engage in various activities such as creating playlists, checking for new releases, accessing artist details, and more. The project is divided into two parts, each with its own focus, but they share development time to ensure proper implementation between the client and server environments.

The first part of the project involves developing the client-side environment using Angular. This includes designing the user interface where users can perform the aforementioned activities.

The second part involves developing the server-side environment and the database used by Luister, using MySQL and Django Rest Framework.

Once these parts were completed, the application has been deployed using Docker. This phase of the process was used for testing and making improvements within the client and server environments.

All this work has been done to provide music lovers with a place where they can manage their music libraries without relying on a single music service. Luister offers the ability to import playlists in formats compatible with streaming platforms. In a context where the use of these services is widespread and the dependence on them to build a music library is growing every day.

Contenido

1. INTRODUCCIÓN.....	14
2. NECESIDADES DEL SECTOR PRODUCTIVO	15
2.1 ANÁLISIS DE LA SITUACIÓN ACTUAL.....	15
2.2 NECESIDADES DEL CLIENTE Y OPORTUNIDAD DE NEGOCIO	16
2.3 EL NUEVO PROYECTO: LUISTER.....	18
3. DISEÑO DEL PROYECTO.....	20
3.1 FASES DEL PROYECTO.....	20
3.1.1 ANÁLISIS.....	20
3.1.2 DISEÑO	21
3.1.3 IMPLEMENTACIÓN	24
3.1.4 PRUEBAS	83
3.2 OBJETIVOS A CONSEGUIR	93
3.3 PREVISIÓN DE LOS RECURSOS MATERIALES Y HUMANOS NECESARIOS.....	94
3.4 PRESUPUESTO ECONÓMICO.	95
4. PLANIFICACIÓN DE LA EJECUCIÓN DEL PROYECTO.....	96
4.1 FASE DE ANÁLISIS.....	96
4.2 FASE DE DISEÑO.....	96
4.3 FASE DE IMPLEMENTACIÓN	97
4.4 FASE DE PRUEBAS	98
5. DEFINICIÓN DE PROCEDIMIENTOS DE CONTROL Y EVALUACIÓN.....	100
6. FUENTES	107
7. ANEXOS	109

INDICE DE FIGURAS

Figura 1. Evolución del mercado musical.	15
Figura 2. Cuota de mercado los servicios de streaming de música.	16
Figura 3. Estructura base de las secciones.	21
Figura 4. Barra de navegación.	21
Figura 5. Estructura de "Inicio"	22
Figura 6. Estructura de "Novedades"	22
Figura 7. Estructura de "Biblioteca"	23
Figura 8. Estructura de "Descubrir"	23
Figura 9. Instalación de Node	24
Figura 10. Instalación de Angular	25
Figura 11. Versiones usadas del entorno cliente	25
Figura 12. Creación del proyecto en Angular	25
Figura 13. Estructura de ficheros inicial	26
Figura 14. Creación de los componentes principales	27
Figura 15. Fichero de rutas	27
Figura 16. Ejemplo de creación de un componente	27
Figura 17. Estructura inicial del fichero ts de un componente	28
Figura 18. Ejemplo de uso de router-outlet	28
Figura 19. Contenido del módulo components	29
Figura 20. Ejemplo de rutas básicas	30
Figura 21. Uso de routerLink	31
Figura 22. Comprobación del funcionamiento de las rutas	31
Figura 23. Creación de una aplicación en Spotify	32
Figura 24. Generación del token de autenticación de Spotify	33
Figura 25. Validación del token de autenticación	33
Figura 26. Ejemplo de funciones del servicio de Spotify	34
Figura 27. Funcionamiento del componente "Discover"	34
Figura 28. Menú contextual	35
Figura 29. Estructura de ficheros y fichero de rutas final	36
Figura 30. Modelo E/R de Luister	38
Figura 31. Configuración de base de datos en Django	39

Figura 32. Modelo de Usuario	40
Figura 33. Modelo de Admin de usuarios.	40
Figura 34. Tabla de usuarios en el panel de administración.	41
Figura 35. Detalles de usuario.....	41
Figura 36. Historial de usuario.....	41
Figura 37. Fichero permissions de usuario.....	42
Figura 38. Ejemplo importaciones de serializer de usuario.....	42
Figura 39. Modelo serializer de usuario.....	43
Figura 40. Campo Username, modelo de usuario	43
Figura 41. Serializador de login	44
Figura 42. Serializador de registro.....	44
Figura 43. Métodos de registro.....	45
Figura 44. Vistas de usuario, importaciones	45
Figura 45. Vistas de usuario	46
Figura 46. Decoradores de usuario	46
Figura 47. Tabla tokens en admin	47
Figura 48. Tabla usuarios en admin	47
Figura 49. Registro reset contraseña.....	47
Figura 50. Tabla reset contraseña en admin	47
Figura 51. FicheroUrls	48
Figura 52. Modelo de customList.....	48
Figura 53. Admin de customList	49
Figura 54. Modelo serializador de customList.....	49
Figura 55. Serializador de customList.....	50
Figura 56. Importaciones de vistas de customList	50
Figura 57. Vistas de customList.....	51
Figura 58. Respuesta de creación de customList	51
Figura 59. Tabla customList en admin	51
Figura 60. Fichero urls customList.....	52
Figura 61. Modelo customListTracks	52
Figura 62. Admin customListTrack	53
Figura 63. Modelo serializador customListTrack.....	53

Figura 64. Serializador customListTrack.....	54
Figura 65. Vistas customListTrack.....	54
Figura 66. Creación customListTrack	54
Figura 67.Urls customListTrack	55
Figura 68. Modelo favoriteTracks	55
Figura 69. Admin favoriteTracks.....	56
Figura 70. Modelo serializador favoriteTracks	56
Figura 71. Serializador favoriteTracks	57
Figura 72.Urls favoriteTracks.....	57
Figura 73. Modelo followedArtists.....	58
Figura 74. Modelo serializador followedArtists.....	58
Figura 75. Serializador followedArtists.....	59
Figura 76. Vistas followedArtists.....	59
Figura 77. Prueba registro de usuarios.....	60
Figura 78. Resultado prueba de registro	60
Figura 79. Prueba login	61
Figura 80. Resultado login.....	61
Figura 81. Token login.....	61
Figura 82. Logout usuario.....	62
Figura 83. Resultado logout	62
Figura 84. Password reset.....	62
Figura 85. Resultado password reset	63
Figura 86. Mensaje consola password reset.....	63
Figura 87. Confirmación cambio contraseña.....	63
Figura 88. Resultado cambio contraseña.....	63
Figura 89. Comprobación cambio contraseña	64
Figura 90. Creación customList	64
Figura 91. Cuerpo creación customList	65
Figura 92. Resultado creación customList.....	65
Figura 93. Cabecera consulta customList.....	65
Figura 94. Resultado consulta customList	66
Figura 95. Comprobación registro customList	66

Figura 96. Borrado customList.....	66
Figura 97. Resultado borrado customList	67
Figura 98. Comprobación borrado customList	67
Figura 99. Cabecera creación followedArtists.....	67
Figura 100. Cuerpo creación followedArtists	68
Figura 101. Consulta followedArtists	68
Figura 102. Resultado consulta followedArtists	68
Figura 103. Registro middleware CORS.....	69
Figura 104. Whitelist CORS	69
Figura 105. Post permitidos CORS.....	69
Figura 106. Base de datos settings mySQL.....	70
Figura 107. Credenciales de sesión CORS	70
Figura 108. Clase Meta Modelos.....	70
Figura 109. dbcolumn customlistID	70
Figura 110. FollowedArtists	71
Figura 111. CMD DockerFile	71
Figura 112. Error instalación Django	71
Figura 113. Instalación mySQL en requirements.txt	71
Figura 114. Actualización librería	72
Figura 115. Instalación certificados	72
Figura 116. Instalación repositorio.....	73
Figura 117. Ajuste repositorio.....	73
Figura 118. Instalación Docker	73
Figura 119. Test Docker	73
Figura 120. Ficheros aplicación.....	74
Figura 121. Ficheros Docker	74
Figura 122. Fichero DockerFile	74
Figura 123. Fichero NginX.....	76
Figura 124. Fichero de configuración de proxy	77
Figura 125. Certificados	78
Figura 126. DockerIgnore	78
Figura 127. Construcción de imagen de sitio web	78

Figura 128. Listado imagen	78
Figura 129. Imagen contenedor.....	79
Figura 130. Ejecución contenedor	79
Figura 131. Fichero DockerFile BBDD.....	80
Figura 132. Ficheros BBDD.....	80
Figura 133. Construcción de imagen	80
Figura 134. Listado imagen	80
Figura 135. Ejecución contenedor	80
Figura 136. Vista contenedor.....	80
Figura 137. BBDD	81
Figura 138. DockerFile	82
Figura 139. Requirements.txt	82
Figura 140. Construcción imagen API	82
Figura 141. Vista contenedor.....	83
Figura 142. Ejecución contenedor	83
Figura 143. Logs contenedor.....	83
Figura 144. Prueba API	84
Figura 145. Prueba login	84
Figura 146. Resultado login.....	84
Figura 147. Prueba Registro	85
Figura 148. Enlace registro	85
Figura 149. Registro correcto	85
Figura 150. Prueba cierre sesión.....	86
Figura 151. Prueba cookies.....	86
Figura 152. Creación lista.....	86
Figura 153. Guardado lista	87
Figura 154. Visualización lista	87
Figura 155. Prueba menú contextual.....	87
Figura 156. Visualización Pista añadida	88
Figura 157. Visualización detalle lista.....	88
Figura 158. Prueba seguir artista	89
Figura 159. Vista BBDD artista.....	89

Figura 160. Dejar se seguir artista.....	90
Figura 161. Impresión datos API	90
Figura 162. Menú contextual custom.....	90
Figura 163. Menú contextual custom 2.....	91
Figura 164. Menú contextual 3	91
Figura 165. Prueba registro.....	91
Figura 166. Restricción acceso	92
Figura 167. Restricción en menú contextual.....	93
Figura 168. Prueba de previsualización.....	93
Figura 169. Todo Tree.....	94
Figura 170. Planificación de tareas	96
Figura 171. Fase de análisis.....	96
Figura 172. Fase de diseño.....	97
Figura 173. Fase de implementacion s1	97
Figura 174. Fase de implementación s2	97
Figura 175. Fase de pruebas.....	99
Figura 176. Scripts 1	100
Figura 177. Scripts 2	101
Figura 178. Crontab	101
Figura 179. Script renovación certs	102
Figura 180. Configuración Crontab.....	103
Figura 181. Actualización dependencias	103
Figura 182. Prueba de funcionamiento.....	103
Figura 183. Configuración Crontab 2.....	104
Figura 184. Script respaldo	104
Figura 185. Prueba respaldo	104
Figura 186. Confirmación respaldo.....	105
Figura 187. Servidor de respaldo	105
Figura 188. Configuración Crontab respaldo	105
Figura 189. Visualización Logs.....	106
Figura 190. Diagrama Casos de uso	109
Figura 191. Sketching Inicio	110

Figura 192. Sketching Popular	110
Figura 193. Sketching Novedades.....	111
Figura 194. Sketching Biblioteca	111
Figura 195. Sketching Detalles.....	112
Figura 196. Sketching Lista	113
Figura 197. Sketching Login.....	113

1. INTRODUCCIÓN

Este documento responde a la realización del módulo de Proyecto del CFGS en Desarrollo de Aplicaciones Web, cuyo objetivo es la integración de las diversas capacidades y conocimientos adquiridos en el resto de módulos del ciclo.

Nuestro proyecto consiste en la creación de una aplicación web de música, cuyo nombre es Luister. En esta aplicación se podrán elaborar listas, consultar novedades y explorar detalles de álbumes, canciones o artistas. Esto puede hacerse sin necesidad de tener cuenta en un servicio de streaming de música.

El objetivo final es ofrecer un sitio a los amantes de la música para elaborar listas de música y gestionar su biblioteca musical de forma independiente a los servicios de streaming. Consiguiendo así un incremento de la comodidad del usuario en el caso de que quiera cambiarse de servicio y facilitando la conservación de su biblioteca musical.

Luister se ha realizado usando **Angular**, framework realizado en **TypeScript**, en la parte de frontend. En la parte de servidor hemos utilizado **Python** mediante **Django API Rest** con una base de datos en **MySQL**, en donde almacenamos los datos de los usuarios registrados. La información usada en Luister se obtiene mediante peticiones a diferentes **APIs** de diferentes servicios de streaming de música, por lo que no almacenamos este tipo de información y se obtiene directamente de la fuente. En cuanto al despliegue de nuestro proyecto lo hemos realizado usando **Dockers**, herramienta de creación de contenedores generados a través de imágenes en donde se incluyen solamente los recursos necesarios para el servidor, hemos elegido esta alternativa debido a su portabilidad, escalabilidad, portabilidad y eficiencia tamaño respecto a otras alternativas.

Con Luister hemos unido los conocimientos adquiridos a lo largo del ciclo, hemos añadido otros adquiridos durante la realización de la FCT y otros de investigación y formación propia. En este proyecto aúna aspectos de programación tanto en entorno cliente como en entorno servidor, el uso de Frameworks, diseño de interfaces, la gestión y administración de bases de datos y el despliegue de aplicaciones web.

2. NECESIDADES DEL SECTOR PRODUCTIVO

A continuación, se identifican las necesidades detectadas en el sector productivo que originan la oportunidad de negocio que se detalla en los siguientes puntos.

2.1 Análisis de la situación actual

La industria musical ha sufrido un largo proceso de cambio desde principios de los años 80 en lo que se refiere a su difusión, con el CD-ROM como primer medio de reproducción digital de difusión masiva (Cascudo, 2014: 304).

A finales de 1990 nace internet y con ello la música online y la lucha de la industria musical en base a tres aspectos: los derechos de autor, el control de los medios de distribución y la fuerza de las técnicas de marketing y promoción musical (Abeillé, 2013: 121). El MP3 fue el primer formato musical estandarizado en internet, y obtuvo una gran popularidad (Cascudo, 2014: 307) que acrecentó con la llegada de Napster, un servicio lanzado en Estados Unidos a finales de 1999 que supuso el punto de partida para las plataformas de streaming musical. Su éxito consistía en un grupo de usuarios que compartían o descargaban música en formato mp3. (Crossan, M & cols, 2001: 18)

Estas plataformas nos permiten conocer su catálogo musical a partir de una suscripción que puede ser gratuita a cambio de publicidad o bien de pago.

Los consumidores acceden a la música a través de bibliotecas digitales formadas por bases de datos con música de todos los tiempos.

Según IFPI, la organización que representa a la industria de la música grabada en todo el mundo, el streaming ha ido cogiendo fuerza desde 2005, y actualmente es la principal fuente de ingresos musicales (IFPI, 2023), como se puede observar en la siguiente gráfica:

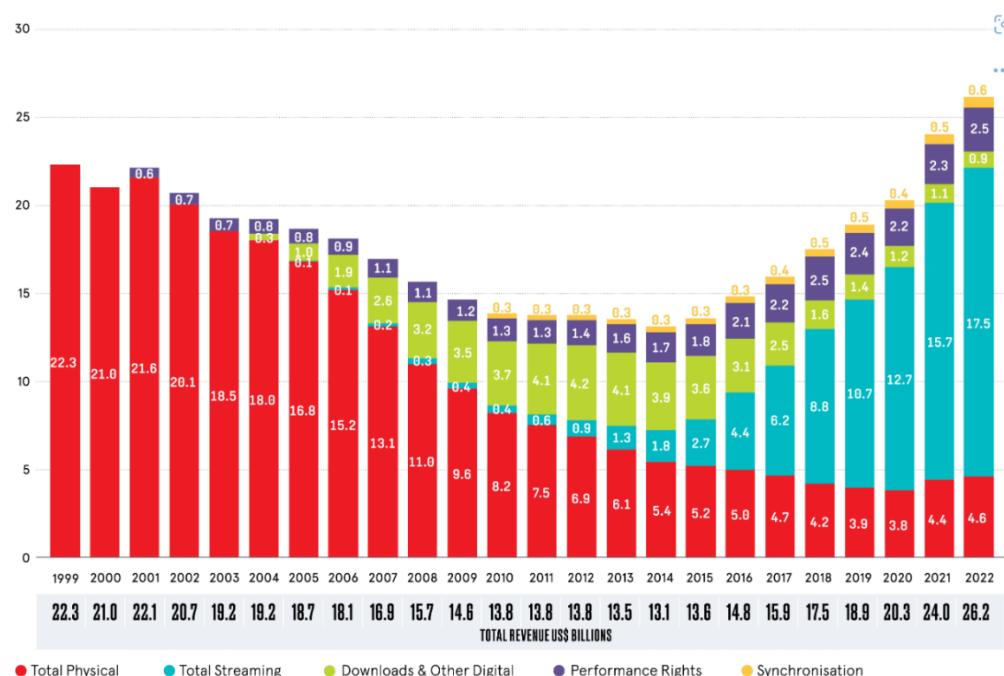


Figura 1. Evolución del mercado musical.

Actualmente existe una gran diversidad de plataformas de streaming, siendo Spotify el servicio de streaming musical dominante con 180 millones de suscriptores.

Según las estadísticas de la plataforma de transmisión de música de Midia Research, Spotify tiene la participación de mercado de transmisión de música más alta con el 30,5% del mercado (Mulligan, 2021).Le siguen en el ranking Apple Music, Tencent Music, Amazon Music y YouTube Music.

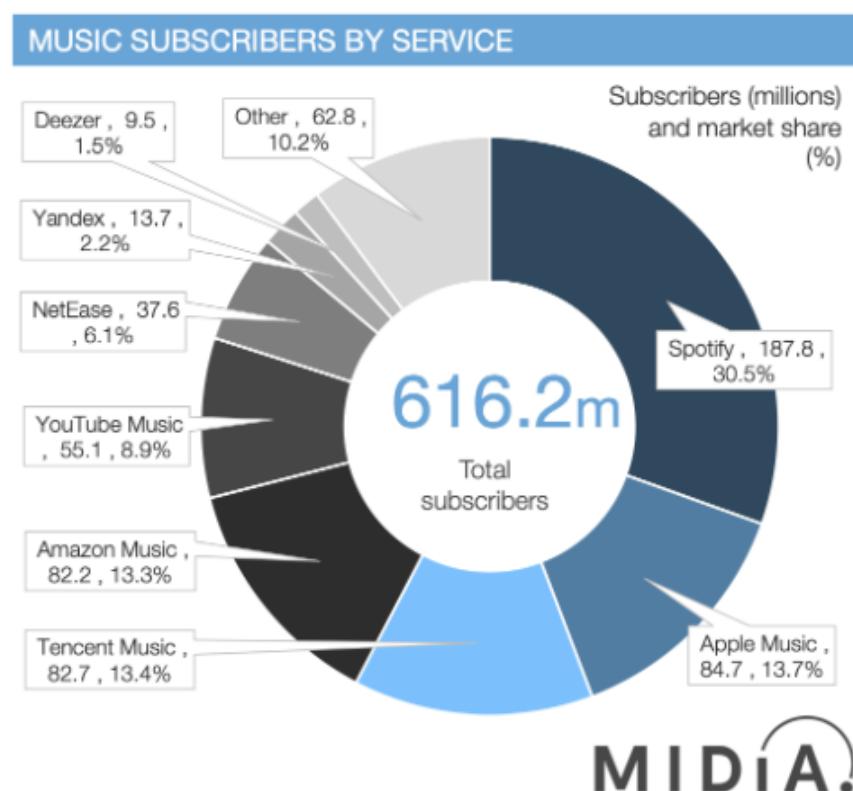


Figura 2. Cuota de mercado los servicios de streaming de música.

consumo y factores como la calidad del servicio, variedad musical y disponibilidad de actualizaciones (Nicolas-Sans & cols, 2022)

2.2 Necesidades del cliente y oportunidad de negocio

Las características que afectan a la percepción de la calidad del streaming musical son:

- Facilidad de uso, fundamento de nuestro proyecto: es la característica que más influye en la calidad, pero cabe mencionar también que dependerá en gran medida del nivel tecnológico de cada país(Del Valle, 2015).
- Conveniencia: se refiere a la capacidad de acceder y descargar la música en la que están interesados, sintonizando desde el lugar que ellos decidan ya sea en el trabajo, en su hogar, en la escuela...(Del Valle, 2015).
- Posibilidad de ser dueños de canciones: Una vez una canción ha sido descargada pueden hacer uso de ella cuantas veces como deseen (Del Valle, 2015).
- Precio: esta característica es muy importante para los consumidores, ya que tienden a optimizar sus recursos económicos. Además hay usuarios que ni siquiera contemplan pagar

A pesar de que Spotify encabeza por séptimo año consecutivo los servicios de streaming musical, el mayor crecimiento en el último año lo ha registrado YouTube Music.

Con respecto a lo que se factura por pago, según la investigación más reciente de estadísticas de transmisión de artistas de Trichordist (2020), la tarifa promedio de pago por transmisión en todas las plataformas es de aproximadamente 0,00157 euros (The Trichordist, 2020))

Un estudio de diseño exploratorio analizó las principales razones de suscripción en las diversas plataformas de streaming musical y hallaron una relación causal entre su

por escuchar música existiendo plataformas con versiones gratuitas, por lo que aquellas que quieran ganar suscriptores de pago tendrán que ofrecer ventajas adicionales importantes. Un estudio analizó las diferencias geográficas en los consumidores de pago y encontró diferencias significativas entre países como los países de América Latina y EEUU, en los que el número de usuarios totales es similar al número de usuarios de pago, y el resto de países que no siguen esa correlación. Las razones son: poder adquisitivo mayor y/o generalización de los pagos en línea.

- Catálogo amplio de canciones: Se ha demostrado que los consumidores valoran el acceso a un catálogo variado (Del Valle, 2015).
- Confianza: La confianza que inspira la plataforma en el usuario es uno de los puntos conflictivos, ya sea por el nivel de seguridad en la protección de datos o bien la seguridad en los pagos y transacciones que realicen. Por esta razón es importante desarrollar plataformas que respeten la privacidad del usuario y tengan mecanismos de seguridad de la información (Del Valle, 2015).

El sector en el que se mueven estas plataformas aún está en desarrollo, pero guardan varias similitudes.

El éxito de cada plataforma lo podemos medir con las cinco fuerzas de Porter, un modelo estratégico elaborado por el ingeniero y profesor Michael Eugene Porter de la Escuela de Negocios Harvard en el año 1979 que nos permitirá analizar el nivel de competencia de la industria musical para poder establecer una estrategia de negocio en base a los datos desarrollados previamente.

- Amenaza de nuevos competidores (nivel medio): El flujo de entrada de nuevos competidores no es muy elevado, pero nos encontramos con las barreras de negociar con los estudios de la industria musical y llegar a acuerdos que resulten beneficiosos para ambos.
- Amenaza de productos sustitutivos (nivel medio): Actualmente nos encontramos con diferentes formas en las que un usuario puede consumir música. El uso de los medios tradicionales se encuentra en declive por lo que no representan una gran amenaza, aunque siguen manteniéndose en el mercado usuarios fieles a la compra de algunos formatos antiguos como vinilos. Más que amenazas de plataformas de streaming musical podríamos hablar de las plataformas de otro tipo de formato multimedia como los vídeos que se suben a YouTube y que reúnen millones de visitas.
- Poder de negociación con los proveedores (nivel alto): El poder de negociación con los proveedores, esto es, los estudios de industria musical es muy bajo ya que el 72% de toda la música a nivel global está controlada únicamente por tres de ellos: Universal Music, Sony Music y Warner Music Group.
- Poder de negociación de los clientes (nivel bajo): Debido a que los clientes no tienen contacto directo con los estudios musicales no representan ninguna amenaza para nosotros y además los precios que se barajan son similares en todos ellos por lo que tendremos que aportar valor con otras características que los usuarios valoren, tales como la facilidad de uso y variedad mencionadas en el análisis de la demanda.
- Rivalidad entre competidores (nivel medio-alto): Nos encontramos con una rivalidad alta, con datos dominantes por parte de Spotify pero con niveles lícitos con respecto a otras plataformas.

2.3 *El nuevo proyecto: Luister*

Tipo de proyecto

Se trata de un proyecto de desarrollo de una aplicación web, Luister. Tiene tres componentes diferenciados, el primero de ellos (entorno cliente) realizado con Angular, el segundo (entorno servidor) con Django Rest Framework y PHP y el tercero (base de datos), implementado con MySQL.

¿Qué es Luister?

Luister proviene del afrikáans y significa escuchar. Es una aplicación web cuyo tema central es la música, específicamente en la música disponible a través de los servicios de streaming de música. Luister ofrece la posibilidad de ver novedades, los contenidos más populares, detalles de artistas y explorar el contenido de distintos servicios de streaming de música.

Una de las finalidades de Luister es facilitar el cambio entre plataformas, para ello los usuarios registrados podrán llevar el registro de su contenido favorito que podrán exportar en diferentes formatos de texto aceptados por páginas especializadas en traspaso de bibliotecas entre servicios. Por todo esto, se define a Luister como un buscador de contenido y una base de datos para los amantes de la música.

¿Para qué sirve?

Luister se fundamenta en cuatro funcionalidades que forman la base de la aplicación, todas relacionadas con la música. Algunas de ellas serán accesibles desde su opción propia integrada en la barra de navegación y otras accesibles al pulsar sobre el nombre del elemento.

1. Posibilidad de seguir las novedades del panorama musical en un solo lugar. Estas novedades se mostrarán en un apartado propio y ordenados en base a su fecha de lanzamiento y otros criterios definidos por las plataformas.
2. Seguimiento del contenido más popular. En este apartado se mostrará el contenido más popular ordenado en base a su popularidad definida por las plataformas.
3. Posibilidad de ver los detalles del contenido (artistas, álbumes o canciones). Debido a la naturaleza de esta funcionalidad no la podemos integrar dentro de la barra de navegación, para acceder a ella será necesario pulsar sobre el nombre del elemento. En estos detalles se podrán ver diferentes metadatos proporcionados por las plataformas, seguir a un artista o agregar canciones a nuestra lista de favoritos.
4. Biblioteca: es la funcionalidad más importante de Luister. Dentro de esta sección podrán crearse distintas listas y gestionar su contenido. Para acceder a ella será necesario contar con una cuenta de usuario. El contenido de la biblioteca podrá ser exportado en formatos de texto admitidos por servicios de traspaso de bibliotecas musicales, con el fin de facilitar la migración entre servicios.

¿Cómo funciona?

Mediante el uso de diferentes APIs obtendremos el contenido necesario para las funcionalidades definidas previamente. Estas funcionalidades varían en función del tipo de acceso que se haga a la aplicación. Los tipos de usuarios o acceso son los siguientes:

- Primer acceso (sin cookies) o nuevo usuario: se muestra contenido genérico sin ninguna personalización dentro de la página de novedades, es decir, este tipo de usuario recibe el

contenido recuperado de las APIs sin ningún tipo de filtro. Como se ha mencionado anteriormente, este tipo de usuario no tendrá acceso a la sección “Biblioteca”, pero el resto de funcionalidades estarán disponibles.

- Uso continuado sin cuenta o usuario sin cuenta: haciendo uso de las cookies y la actividad que haya realizado el usuario de manera previa, se personaliza el contenido de la sección de “Novedades”. Para este tipo de usuario se mostrará contenido personalizado, en base al contenido obtenido de las APIs, en la medida de lo posible en base a la información de su actividad. Al ser un contenido dependiente de la información almacenada en el navegador esta se perderá en cuánto estas se borren o modifiquen.
- Acceso autenticado o usuario con cuenta: el usuario tendrá acceso al resto de funcionalidades y personalización de contenido mencionadas anteriormente y, además, tendrá acceso a la sección “Biblioteca” y a las funcionalidades definidas dentro de esta sección. Al crear una cuenta e iniciar sesión, tendrá acceso a ese contenido personalizado en diferentes dispositivos.

Características requeridas al proyecto

Para cumplir con lo expresado en el apartado anterior, hemos definido los siguientes requisitos relacionados con las características de Luister:

- Relativos a la funcionalidad: para ello debe mostrar el contenido de forma correcta en todos los apartados de la aplicación. Para ello es necesario asegurarnos de que la labor de codificación se ha realizado de manera correcta y que los componentes de Luister no contengan ningún fallo que afecte a su funcionamiento.
- Luister ofrece la posibilidad de acceder y consultar el contenido de varios servicios de streaming de música y ser usada para crear una biblioteca de música sin estar ligada a una sola plataforma. Por ello, es necesario un funcionamiento de las peticiones que proporcionan este contenido a Luister de forma continua e ininterrumpida.

En el proceso de desarrollo de este proyecto se han utilizado gran parte los conocimientos adquiridos durante la realización del ciclo, como:

- Programación en entorno cliente: con lenguajes de programación como TypeScript y JavaScript mediante Angular. Además de lenguaje de marcas como HTML o hojas de estilos CSS.
- Programación en entorno servidor: donde se ha usado principalmente Python mediante Django Rest Framework. Además de PHP + Nginx para el balanceo de cargas.
- Gestión, diseño y administración de bases de datos: se ha usado MySQL. Para el diseño se ha utilizado LucidCharts.
- Despliegue: realizado mediante Docker.
- Diseño de interfaces: predominante en la parte del diseño previo a la programación. Realizado con herramientas como Draw.io o mediante programas de dibujo para el sketching.

Además del uso de Git, a través de un repositorio común en Github, que nos ha permitido gestionar el proceso de desarrollo de una manera más cómoda y eficiente.

3. DISEÑO DEL PROYECTO

Dando por hecho la viabilidad del proyecto, en este apartado se concretarán las fases necesarias para llevarlo a cabo, y cumplir con los objetivos que se establezcan, teniendo en cuenta los recursos necesarios.

3.1 Fases del proyecto

El desarrollo de este proyecto se llevará a cabo en cuatro fases: análisis, diseño, implementación y pruebas, que pasan se detallan a continuación.

3.1.1 Análisis

En esta fase se establecerán los requisitos del proyecto, dentro de estos requisitos debemos distinguir lo que debe realizar nuestro proyecto y las propiedades y características que debe tener. En el caso de Luister, tenemos dos partes claramente diferenciadas mencionadas anteriormente, por lo que diferenciamos entre las propiedades y características necesarias de cada una de las partes dentro de los diferentes requisitos.

Requisitos funcionales

Es fundamental que la aplicación web esté operativa en todo momento, independiente de la carga y que se caracterice por su estabilidad y un mantenimiento lo más sencillo posible.

En cuanto a la parte cliente, los requisitos funcionales son la compatibilidad y que la funcionalidad de nuestra aplicación sea la misma, independientemente del navegador y del dispositivo usado para acceder. Las diferentes funcionalidades de Luister deben estar disponibles de manera permanente, por lo que es prioritario asegurarnos que el acceso al contenido de las APIs se realiza de manera correcta y que los servicios usados para ello funcionan de manera correcta, ya que es la base de nuestro proyecto. Por ello, es necesario revisar los diferentes endpoints de forma periódica para verificar que su funcionamiento es correcto.

Otro de los requisitos es que nuestra aplicación web sea responsive, es decir, que se adapte al tamaño del dispositivo usado para acceder. Siendo este un requisito de vital importancia debido a que la mayor parte del acceso se realiza actualmente desde dispositivos móviles (teléfonos móviles o tablets) y afectaría negativamente a nuestro número potencial de usuarios.

Requisitos no funcionales

La aplicación web debe ser visualmente atractiva y su aspecto debe poder asociarse a la música, con el fin de llamar la atención de posibles usuarios y llamar la atención para fomentar su uso.

El contenido, a nivel visual, debe ser acorde a los objetivos definidos en el párrafo anterior, pero además deben proporcionar una información relevante y suficiente para que los potenciales usuarios vean en el uso de Luister algo atractivo y diferenciador.

La experiencia de uso dentro de nuestra aplicación web debe ser lo más amigable y cómoda posible, por ello hemos intentado reducir tiempos de carga del contenido obtenido de las APIs y de los servidores al mínimo tiempo posible.

Por último, debido a la naturaleza del proyecto y de las circunstancias de su realización, hemos debido hacer frente a ciertas limitaciones como el número de plataformas implementadas en Luister, menor del previsto inicialmente, debido a razones técnicas o bien por razones de acceso a sus correspondientes APIs, y al cambio de algunas de las funcionalidades previstas en el anteproyecto y en las primeras fases del proyecto.

3.1.2 Diseño

En esta parte del proceso hemos determinado cómo podemos satisfacer los requisitos del apartado anterior, funcionales y no funcionales.

Diseño relativo al entorno cliente

La primera parte del proceso de diseño se ha dedicado a los requisitos no funcionales, debido a su importancia para Luister por los motivos mencionados previamente, es decir, se ha centrado en el entorno cliente. Por este motivo, en esta fase hemos realizado diferentes versiones de bocetos, *sketching* y *wireframing* con el fin de obtener un diseño acorde a las funcionalidades de Luister.

Todos los componentes definidos en Angular tienen una base común en cuanto a estructura:

Las secciones de Luister son las siguientes:

- Inicio
- Novedades
- Popular
- Biblioteca
- Descubrir
- Inicio de sesión y registro
- Detalles*

El diseño de los componentes responde a los requisitos fijados en el apartado anterior. Por ello se han incorporado iconos fácilmente reconocibles dentro de nuestra aplicación web para hacer el uso lo más intuitivo posible.



Figura 3. Estructura base de las secciones.

Todas las secciones serán accesibles desde la barra de navegación salvo “Detalles”, ya que se trata de un componente al que se accede pulsando sobre el nombre de los elementos mostrados en el bloque de contenido.

Se ha usado este diseño debido a las características derivadas del uso de Angular, por lo que Luister es una aplicación de una sola página o SPA.

Uno de estos elementos comunes es la barra de navegación:



Figura 4. Barra de navegación.

Se encuentra presente en todos los componentes, con ella se realiza el acceso a las secciones definidas dentro de Luister, de esta manera solo cambiamos en la parte del contenido, sin necesidad de recargar con cada cambio de sección o de página, una de las características de las SPA. De esta manera asemejamos el diseño de Luister al usado por los distintos servicios de streaming.

El otro elemento común a todos los componentes es el footer, está presente en todos los componentes a pesar de que no lo parezca a simple vista, ya que la naturaleza de alguno de los componentes hace que no sea visible hasta que el nivel de scroll no alcanza un nivel determinado. Se ha tomado esta decisión con el fin de maximizar el área dedicada a mostrar el contenido en estas secciones.

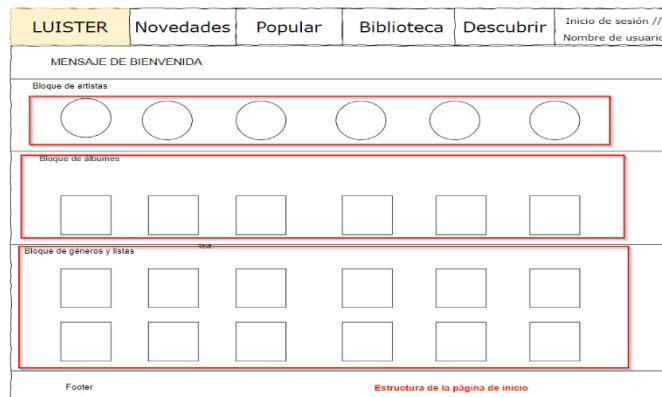


Figura 5. Estructura de "Inicio".

navegación, con la estructura mostrada en la imagen.



Figura 6. Estructura de "Novedades"

“Biblioteca”, al ser la funcionalidad principal de nuestro proyecto hemos usado una estructura con elementos propios y la hemos planteado como un componente padre, es decir, que tiene un componente hijo al que solo se puede acceder pasando por esta sección. La estructura de este componente es la siguiente:

Los componentes, detallados en el listado de secciones, de Luister parten de la base mostrada anteriormente y sobre ella se añaden una serie de elementos que pueden ser únicos para ese componente o bien compartidos por algunos de ellos, todo ello determinado por la funcionalidad de cada uno de los componentes.

En la sección de inicio, hemos planteado una estructura basada en tres bloques de contenido diferenciado. El acceso a esta sección puede hacerse pulsando sobre el logo de la barra de navegación, con la estructura mostrada en la imagen.

Un ejemplo de componentes con elementos en común, además de la base son los pertenecientes a “Novedades” y a “Popular”.

La estructura de estos componentes es la misma, un bloque con una breve explicación de lo que ofrece la sección y otro para el contenido.

En el caso de

“Biblioteca”, al ser la funcionalidad principal de nuestro proyecto hemos usado una estructura con elementos propios y la hemos planteado como un componente padre, es decir, que tiene un componente hijo al que solo se puede acceder pasando por esta sección. La estructura de este componente es la siguiente:



Figura 7. Estructura de "Biblioteca"

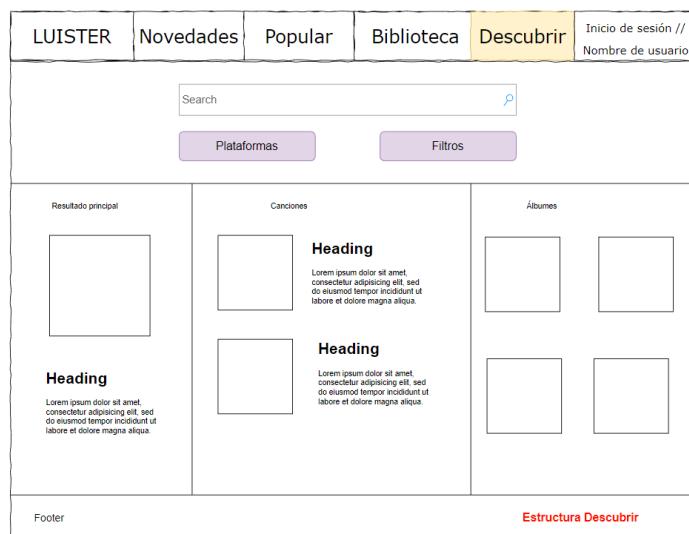


Figura 8. Estructura de "Descubrir"

con la misma distribución del contenido usada en el resto de componentes de Luister.

Relativo al backend o servidor

La siguiente parte del proceso de diseño es la relativa al servidor. Para asegurarnos que podemos cumplir los requisitos funcionales fijados para Luister hemos decidido usar una estructura con dos backends.

El primero de ellos está desarrollado con Django Rest Framework y es el que se usará como servidor principal, y otro realizado en PHP (usado durante parte del proceso de desarrollo) que quedará para casos de pruebas o desarrollo, ante sobrecarga del servidor principal o bien ante fallos o caídas del servidor principal.

Ambos tienen la misma funcionalidad, el balanceo de cargas entre los servidores se realiza mediante nginx, con el fin de que este cambio no sea perceptible para el usuario. Todo ello facilitado por el uso de Dockers.

3.1.3 Implementación

En esta parte del desarrollo pueden distinguirse 4 partes:

- Desarrollo e implementación del entorno cliente.
- Diseño e implementación de la base de datos.
- Desarrollo e implementación del entorno servidor.
- Despliegue de la aplicación.

Debe tenerse en cuenta que cada una de estas partes ha sido necesario un período inicial de instalación y configuración de las herramientas necesarias para su funcionamiento.

Desarrollo e implementación del entorno cliente.

En el entorno cliente, se hará uso del framework **Angular**, en su versión 15.2.2, el cual es un reconocido framework hecho y basado en **TypeScript**. Esta tecnología se compone del lenguaje anteriormente mencionado, así como del lenguaje de marcas **HTML**, y las hojas de estilo en cascada **CSS**.

Al usar TypeScript, Angular permite usar de forma sencilla la modularización de elementos dentro de nuestra aplicación, permite usar decoradores, también cuenta con un tipado fuerte y estricto que se ha aprovechado, especialmente, con la utilización de interfaces. Estas características han facilitado la realización del proyecto al permitir estructurar el código de una manera más sencilla y entendible.

Para trabajar con Angular, es necesario instalar previamente el gestor de paquetes npm de **Node.js**. Este proceso se hace descargando el instalable desde la página oficial. Cómo en el resto de instalaciones, se ha elegido (si ha sido posible) una versión LTS o estable.

Una instalación correcta de Node lleva acompañada una instalación del gestor de paquetes **NPM**. Después de estas instalaciones podemos proceder a instalar Angular, para lo que es necesario actualizar previamente las dependencias del gestor de paquetes. Tal y como se indica en la siguiente imagen:

```
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> npm install -g npm@latest
removed 1 package, and changed 47 packages in 3s
18 packages are looking for funding
  run 'npm fund' for details
● PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> npm -v
  9.6.5
○ PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> []
```

Figura 9. Instalación de Node

Al hacer esto, el paquete ha pasado de la versión 9.6.1 instalada en un primer momento a la versión 9.6.5. Con lo anterior, se realiza la instalación de Angular. Un detalle importante en la instalación es que, independientemente del sistema operativo usado, debe realizarse usando privilegios de administrador, ya que el comando utilizado realiza una instalación global y no se puede realizar sin este nivel de privilegios de sistema. Se instala la versión estable más reciente de **Angular CLI**, usando el comando **npm install -g @angular/cli@latest**:

```
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> npm install -g @angular/cli@latest
npm WARN deprecated @npmcli/move-file@2.0.1: This functionality has been moved to @npmcli/fs

added 240 packages in 13s

30 packages are looking for funding
  run `npm fund` for details
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> []
```

Figura 10. Instalación de Angular

Después de realizar el proceso de instalación de las herramientas necesarias para el entorno cliente, las versiones usadas son las siguientes:

- Angular CLI: 15.2.6
- Node: 18.16.0
- npm (package manager): 9.6.5

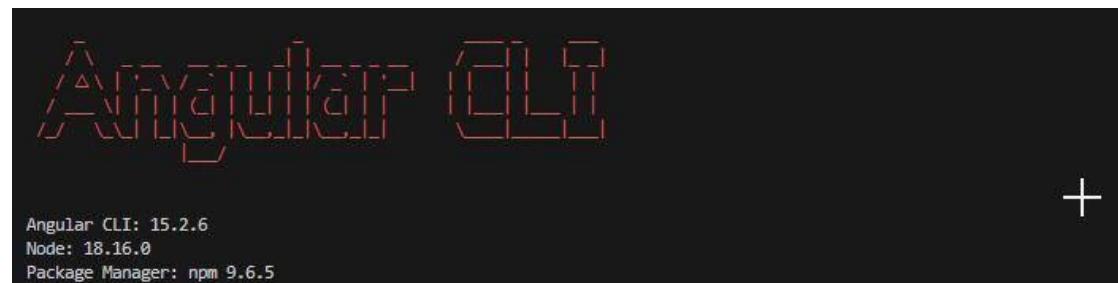


Figura 11. Versiones usadas del entorno cliente

Una vez se han instalado las herramientas necesarias se crea el proyecto, lo que se hace usando el comando **ng new <nombre-proyecto>**. Para ello se abre la terminal en el directorio deseado, se ejecuta el comando y se deben elegir una serie de opciones relativas a la creación automática de rutas y al tipo de hoja de estilos que queremos usar en el proyecto. Una vez se han definido, generará una serie de ficheros que son la estructura inicial de todo proyecto de Angular. Esto se puede ver en las siguientes imágenes:

```
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FIN_GRADO BackUp\frontend> ng new luister
? Would you like to add Angular routing? Yes
? Which stylesheet format would you like to use? CSS
CREATE luister/angular.json (2765 bytes)
CREATE luister/package.json (1038 bytes)
CREATE luister/README.md (1061 bytes)
CREATE luister/tsconfig.json (901 bytes)
CREATE luister/editorconfig (274 bytes)
CREATE luister/.gitignore (548 bytes)
CREATE luister/tsconfig.app.json (263 bytes)
CREATE luister/tsconfig.spec.json (273 bytes)
CREATE luister/vscode/extensions.json (139 bytes)
CREATE luister/vscode/launch.json (474 bytes)
CREATE luister/vscode/tasks.json (338 bytes)
CREATE luister/src/favicon.ico (948 bytes)
CREATE luister/src/index.html (203 bytes)
CREATE luister/src/main.ts (214 bytes)
CREATE luister/src/styles.css (98 bytes)
CREATE luister/src/assets/.gitkeep (0 bytes)
CREATE luister/src/app/app-routing.module.ts (245 bytes)
CREATE luister/src/app/app.module.ts (935 bytes)
CREATE luister/src/app/app.component.html (23115 bytes)
CREATE luister/src/app/app.component.spec.ts (1076 bytes)
CREATE luister/src/app/app.component.ts (211 bytes)
CREATE luister/src/app/app.component.css (0 bytes)
? Installing packages (npm)...
```

Figura 12. Creación del proyecto en Angular

Es importante mencionar que para realizar acciones de creación de elementos (componentes, módulos, servicios...) usando Angular CLI, en un proyecto de Angular es necesario situarse con la terminal deseada dentro del directorio que contiene el fichero **package.json**. Siendo esta la manera recomendada de crear este tipo de elementos, a pesar de que pueden crearse elementos de manera manual, al realizar una actualización de este fichero y de otros ficheros de nuestra aplicación de forma automática, reduciendo así la posibilidad de errores durante el proceso de creación de este tipo de elementos.

Además, para lanzar el servidor de desarrollo de Angular también debemos situarnos en este nivel, para lanzar el servidor de desarrollo se usa el siguiente comando: **ng serve**. Por defecto se utiliza el puerto 4200, aunque se puede definir el puerto deseado añadiendo la opción `--port` y el puerto deseado. Al ejecutar este comando se nos muestra lo siguiente dentro de la terminal:

Entre los directorios generados por la creación del proyecto tenemos el directorio **src**, que es el directorio en donde se encuentran los ficheros y directorios con los que se trabaja para construir nuestra aplicación y en donde al crear el proyecto se encuentran los siguientes elementos.

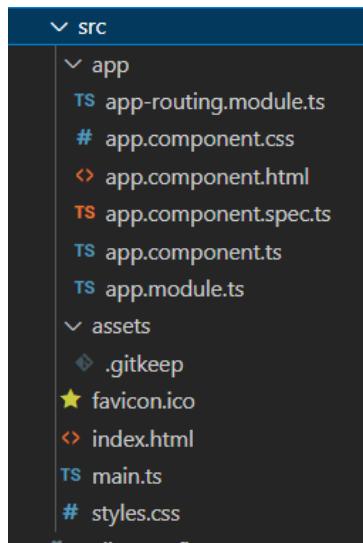


Figura 13. Estructura de ficheros inicial

En esta imagen se encuentran además otros directorios como **app**, que contendrán los distintos elementos que se vayan creando dentro de nuestra aplicación y algunos ficheros globales de la aplicación.

- **Assets**: directorio que contiene ficheros estáticos que serán accesibles por los elementos de nuestra aplicación, como logos o imágenes. Estos archivos serán usados en tiempo de ejecución.
- **Index.html**: fichero HTML raíz que se carga inicialmente al acceder a la aplicación.
- **Main.ts**: fichero encargado de la configuración inicial y de la carga del módulo principal de la aplicación.
- **Styles.css**: hoja de estilos global de la aplicación, donde se colocan los estilos generales de la aplicación.

Se parte de estos elementos para crear la aplicación, por lo que es necesario empezar a añadir los elementos mencionados en la fase de diseño. El proceso de creación se ha realizado usando Angular CLI, debido a las ventajas explicadas anteriormente.

Antes de crear elementos para nuestro proyecto es necesario mencionar la estructura de los elementos que usa Angular, se trata de una estructura jerárquica, ya que hay elementos que contienen otro tipo de elementos de menor jerarquía. El elemento con mayor jerarquía es el módulo, por lo que es recomendable crearlo en primer lugar, ya que agrupa diferentes elementos como los componentes. Todos estos elementos pueden crearse de forma manual o bien mediante variaciones del siguiente comando: **ng generate elemento ruta/nombre opciones**. Algunos ejemplos (algunos de ellos admiten abreviaciones en el tipo de elemento) del uso de este comando son:

- `ng g module nombre-módulo`. Crea un nuevo módulo.
- `ng g component nombre-componente`. Creación de un componente, por defecto crea un fichero TS, HTML, un fichero para pruebas unitarias y una hoja de estilos CSS.
- `ng g service nombre-servicio`: Creación de un nuevo servicio.

El módulo principal, el fichero **app.module.ts**, de la aplicación se crea de manera automática al generar el proyecto. Es el encargado de proporcionar el contexto de funcionamiento de la aplicación, ya que define la configuración y las dependencias necesarias para su funcionamiento. En él se realizan las importaciones necesarias de los módulos predeterminados que permiten que funcione la aplicación y los módulos que se vayan definiendo en el desarrollo.

Además, contiene la configuración de los servicios (instancias compartidas que se pueden inyectar en diferentes elementos del proyecto), que se realiza en el metadato providers, y la configuración de las rutas del proyecto, ya que contiene dentro de las importaciones el **RouterModule**.

Para utilizar de manera más eficaz la modularización se ha creado otro módulo en el que se colocarán los componentes necesarios.

```
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FINAL_GRADO BackUp\frontend\luister> ng generate module components\components --flat
? Would you like to share pseudonymous usage data about this project with the Angular Team
at Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.io/analytics. No
Global setting: enabled
Local setting: disabled
Effective status: disabled
CREATE src/app/components/components.module.ts (196 bytes)
PS C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_PROYECTO_FINAL_GRADO BackUp\frontend\luister> 
```

Figura 14. Creación de los componentes principales

Para que el funcionamiento de la aplicación sea correcto es necesario que los módulos creados se encuentren dentro del módulo principal, al realizarlo con Angular CLI esto se realiza de manera automática, ya que al crear el módulo se actualiza de manera automática con los cambios necesarios. En la imagen pueden observarse otros módulos generados de manera automática dependiendo de las opciones que hayamos elegido en el momento de creación de la aplicación, como es el caso de **AppRoutingModule**.

```
TS app.module.ts x TS navbar.components.ts TS app.component.ts nav/navbar.component.html
luister > src > app > TS app.module.ts > AppModule
  1 import { NgModule } from '@angular/core';
  2 import { BrowserModule } from '@angular/platform-browser';
  3
  4 import { AppRoutingModule } from './app-routing.module';
  5 import { AppComponent } from './app.component';
  6 import { ComponentsModule } from './components/components.module';
  7
  8 @NgModule({
  9   declarations: [
 10     AppComponent
 11   ],
 12   imports: [
 13     BrowserModule,
 14     AppRoutingModule,
 15     ComponentsModule
 16   ],
 17   providers: []
 18 })
```

Figura 15. Fichero de rutas

actualización del fichero **app.module.ts**:

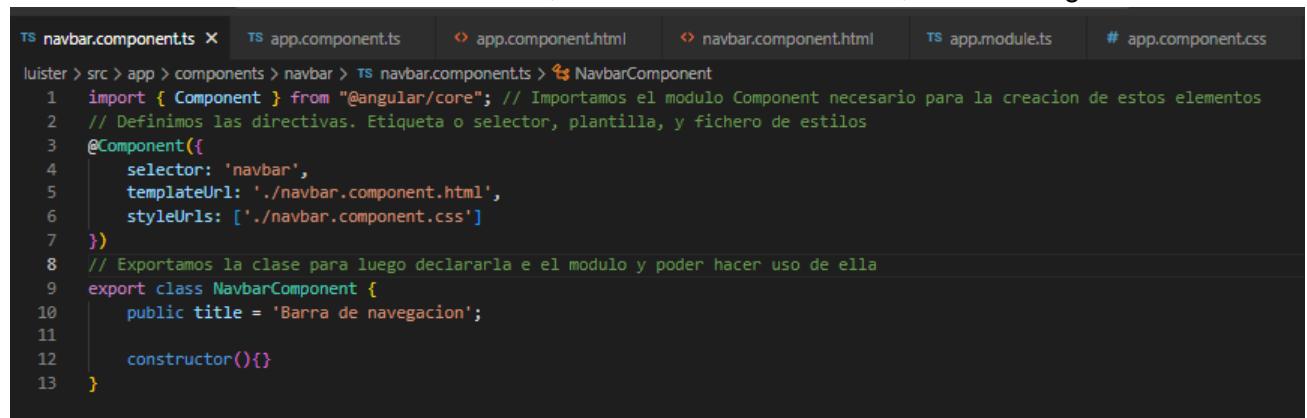
```
C:\Users\faust\OneDrive\Escritorio\devs\LUISTER_DEF\Luister\version_def\frontend\luister>ng g component components/nav/navbar
CREATE src/app/components/nav/navbar/navbar.component.html (21 bytes)
CREATE src/app/components/nav/navbar/navbar.component.spec.ts (599 bytes)
CREATE src/app/components/nav/navbar/navbar.component.ts (202 bytes)
CREATE src/app/components/nav/navbar/navbar.component.css (0 bytes)
```

Figura 16. Ejemplo de creación de un componente

Como el módulo incluirá una serie de componentes y otros elementos, estos estarán incluidos dentro de la aplicación principal al estar contenidos dentro de este módulo y de sus ficheros de forma correcta.

Una vez se ha creado el módulo, se crea un componente usando la terminal, en donde pueden verse los elementos creados y la

El fichero ts de cada elemento creado, en este caso **navbar.ts**, tiene la siguiente estructura:



```

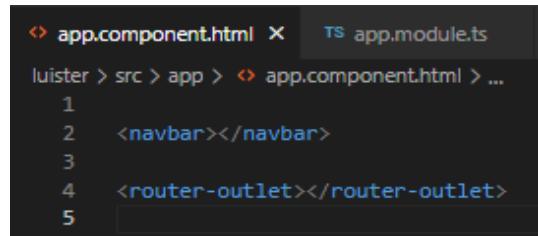
  1 import { Component } from '@angular/core'; // Importamos el modulo Component necesario para la creacion de estos elementos
  2 // Definimos las directivas. Etiqueta o selector, plantilla, y fichero de estilos
  3 @Component({
  4   selector: 'navbar',
  5   templateUrl: './navbar.component.html',
  6   styleUrls: ['./navbar.component.css']
  7 })
  8 // Exportamos la clase para luego declararla e el modulo y poder hacer uso de ella
  9 export class NavbarComponent {
10   public title = 'Barra de navegacion';
11
12   constructor(){}
13 }

```

Figura 17. Estructura inicial del fichero ts de un componente.

Esta estructura permite que cada uno de los elementos creados, en este caso un componente, puedan tener una plantilla y una hoja de estilos propia. Esto es un ejemplo de la modularización que permite el uso de Angular, lo que conlleva una encapsulación de los diferentes elementos creados dentro un proyecto de este framework. Lo que permite aislar y prevenir conflictos con otros elementos propios o externos, facilitando la reutilización y el mantenimiento del proyecto.

Con la base definida durante el proceso de diseño, los primeros elementos en crearse han sido la barra de navegación y el footer, ya que son elementos que se muestran en todos las secciones de Luister.



```

  1
  2   <navbar></navbar>
  3
  4   <router-outlet></router-outlet>
  5

```

Figura 18. Ejemplo de uso de router-outlet

Debido a la modularización es necesario añadir el componente a las páginas del sitio web en las que sea necesario. En este caso, se añadirá en la plantilla principal cuyos elementos aparecerán en todas las páginas de la aplicación, ya que se trata de un *layout* (Bloques genéricos de código). Además, en la imagen se muestra la etiqueta **<router-outlet>**, que es una directiva de Angular usada para mostrar de forma dinámica los

componentes asociados a las diferentes rutas, esto se traduce en que una vez tengamos definidas de las rutas y asociadas a componentes, el lugar donde se encuentre esta etiqueta es donde se mostrará el contenido de esos componentes.

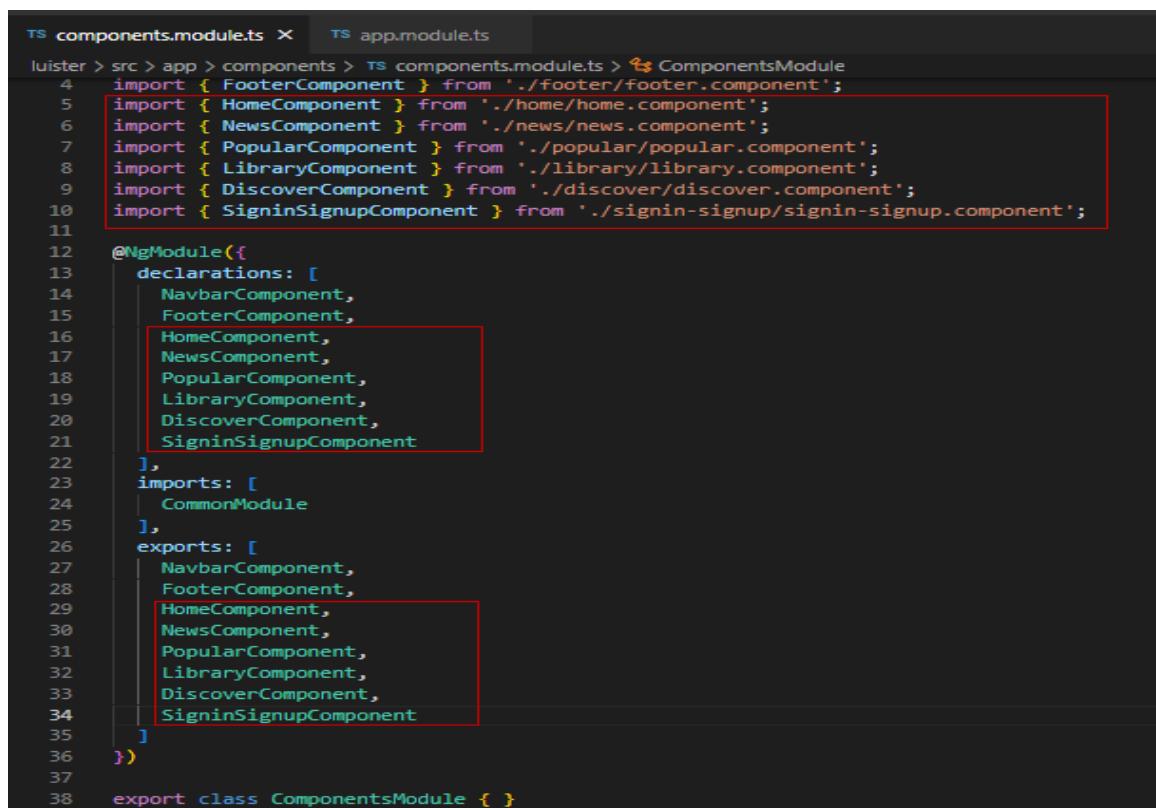
Para que esto se muestre dentro del navegador es necesario comprobar que se han incorporado los componentes deseados dentro del módulo creado anteriormente, lo que se hace al incluirlo dentro del fichero **components.module.ts**, es necesario asegurarse que el componente se encuentra tanto en *declarations* como en *exports*, ya que es necesario para que pueda ser visualizado fuera de este componente.

El proceso de creación del resto de componentes ha seguido las mismas fases que las explicadas para el componente navbar y para el footer. Estos componentes son los que, partiendo de la base explicada en la fase de diseño, van a albergar el contenido de las distintas secciones de Luister.

Ha sido en esta parte del proceso donde se han creado los componentes correspondientes a las siguientes secciones y otros elementos:

- **home**
- **news**
- **library**
- **discover**
- **singin**
- **signup**
- Otros elementos: Paneles, alertas, mensajes interactivos...

Por el modo de creación usado, realizada con Angular CLI, el fichero `components.module.ts` se actualiza automáticamente con los componentes creados.



```

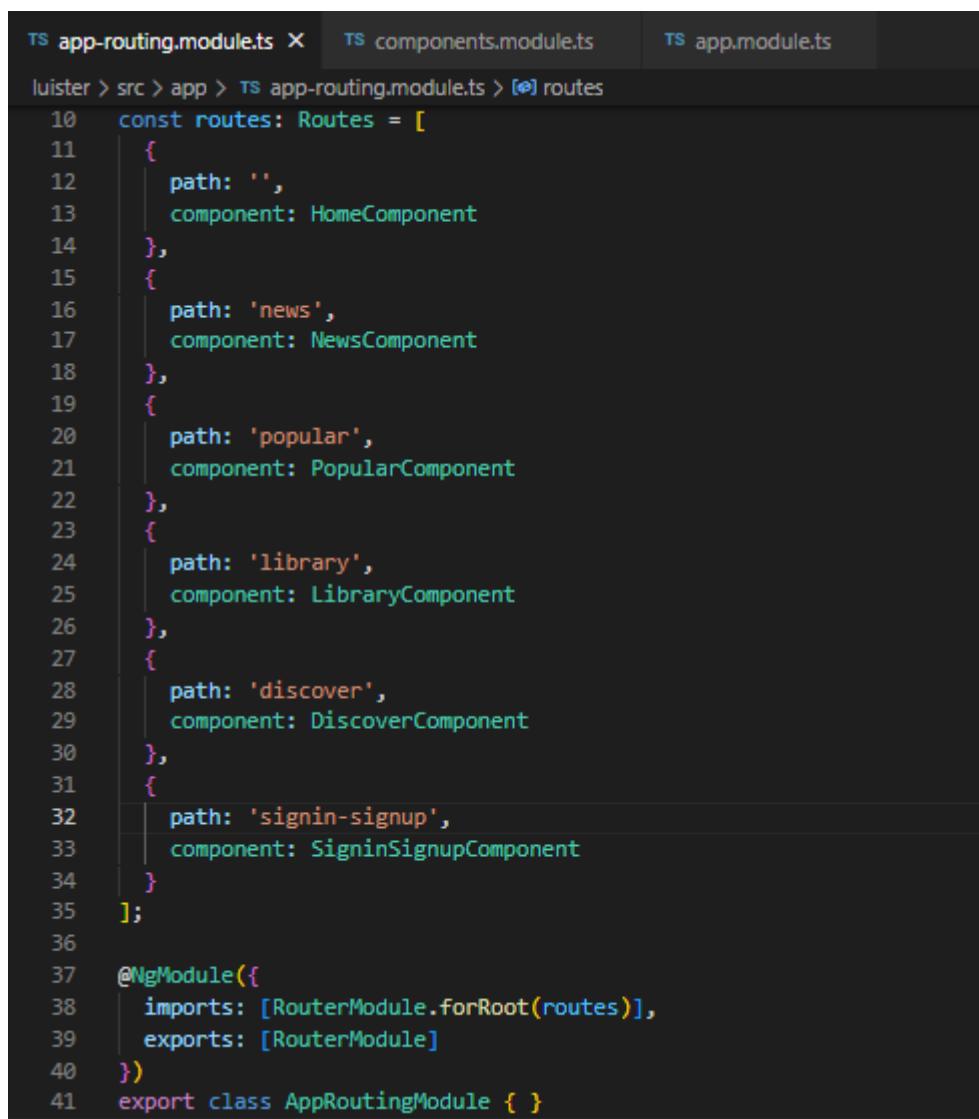
1  ts components.module.ts x  ts app.module.ts
2  luister > src > app > components > ts components.module.ts > ComponentsModule
3  4  import { FooterComponent } from './footer/footer.component';
4  5  import { HomeComponent } from './home/home.component';
5  6  import { NewsComponent } from './news/news.component';
6  7  import { PopularComponent } from './popular/popular.component';
7  8  import { LibraryComponent } from './library/library.component';
8  9  import { DiscoverComponent } from './discover/discover.component';
9 10 import { SigninSignupComponent } from './signin-signup/signin-signup.component';
10
11
12 @NgModule({
13   declarations: [
14     NavbarComponent,
15     FooterComponent,
16     HomeComponent,
17     NewsComponent,
18     PopularComponent,
19     LibraryComponent,
20     DiscoverComponent,
21     SigninSignupComponent
22   ],
23   imports: [
24     CommonModule
25   ],
26   exports: [
27     NavbarComponent,
28     FooterComponent,
29     HomeComponent,
30     NewsComponent,
31     PopularComponent,
32     LibraryComponent,
33     DiscoverComponent,
34     SigninSignupComponent
35   ]
36 })
37
38 export class ComponentsModule { }

```

Figura 19. Contenido del módulo `components`

Con los componentes creados, se procede a definir sus respectivas rutas. Esto se realiza en el fichero `app-routing.module.ts`, que se ha creado previamente al haber seleccionado la opción de usar el routing de Angular durante la creación del proyecto. Este módulo es el que permite la navegación entre los diferentes componentes o secciones de una aplicación sin tener que recargar la aplicación completa. El enrutamiento permite que se carguen solamente los componentes que se necesitan, es decir, los que coinciden con la ruta solicitada haciendo que el uso de la aplicación sea más fluido y cómodo para el usuario.

Dentro de este fichero se definen las rutas mediante el uso de la constante `routes`. Dentro de esta constante y mediante un objeto JSON por ruta, que debe contener como mínimo el PATH y el componente al que asignamos esa ruta dentro de nuestra aplicación.



```

 10  const routes: Routes = [
 11    {
 12      path: '',
 13      component: HomeComponent
 14    },
 15    {
 16      path: 'news',
 17      component: NewsComponent
 18    },
 19    {
 20      path: 'popular',
 21      component: PopularComponent
 22    },
 23    {
 24      path: 'library',
 25      component: LibraryComponent
 26    },
 27    {
 28      path: 'discover',
 29      component: DiscoverComponent
 30    },
 31    {
 32      path: 'signin-signup',
 33      component: SigninSignupComponent
 34    }
 35  ];
 36
 37 @NgModule({
 38   imports: [RouterModule.forRoot(routes)],
 39   exports: [RouterModule]
 40 })
 41 export class AppRoutingModule { }

```

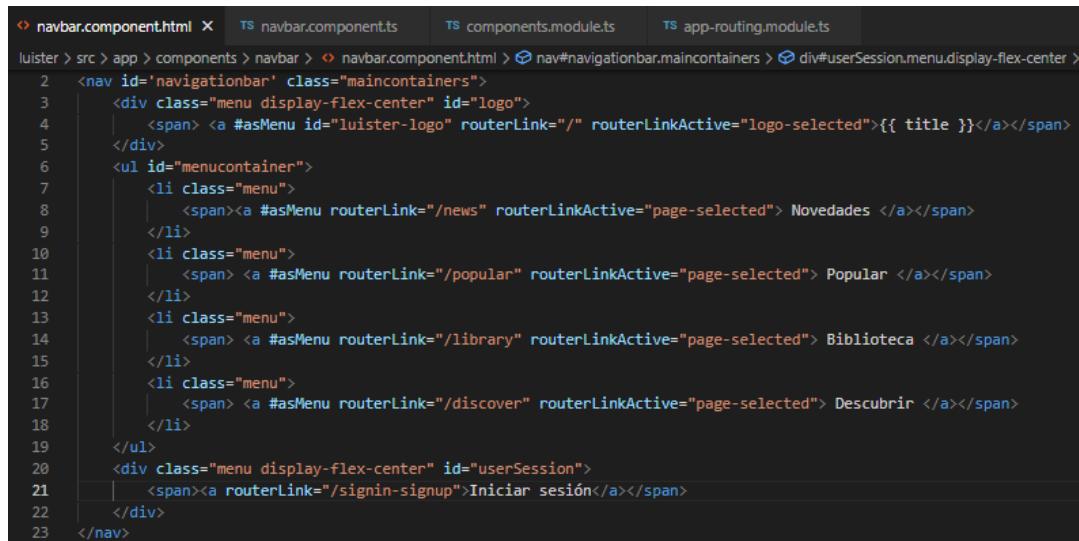
Figura 20. Ejemplo de rutas básicas.

Estas rutas se definen de manera simple y provisional para mostrar el contenido de estos componentes y se modificarán más adelante para adaptarlas a la lógica de cada uno de estos componentes. El mensaje que aparece es el mensaje por defecto que incluye Angular al crear un componente dentro de su plantilla HTML para indicar que el componente se ha creado de manera exitosa al acceder a él desde el navegador.

Se debe recalcar que por el momento las rutas solo son accesibles mediante la barra de direcciones del navegador. Por ello, se asigna cada una de las rutas definidas por el momento con su correspondiente opción

dentro de la barra de navegación, lo que se realiza usando el atributo **routerLink**, atributo propio de Angular. Este atributo tiene la misma funcionalidad que **href**, pero su utilización presenta ciertas ventajas ya que está pensado para ser usado dentro de páginas realizadas con Angular, ya que permite usar las rutas definidas dentro del fichero de rutas (rutas relativas) en lugar de rutas completas. Esto además de permitir que el uso de la página sea más fluido y rápido previene la recarga de la página y, lo más importante, es que permite integrar de manera sencilla otras funcionalidades de la gestión de rutas de Angular como la protección de rutas (AuthGuards) o la gestión de parámetros(:param).

Se complementa el uso de este atributo con la directiva **routerLinkActive**, que permite añadir clases concretas a un elemento cuando la ruta que contiene este elemento está activa con el fin de que los usuarios puedan saber de forma sencilla en que sección de la página se encuentran.



```

<nav id="navigationbar" class="maincontainers">
  <div class="menu display-flex-center" id="logo">
    <span><a #luister-logo routerLink="/" routerLinkActive="logo-selected">{{ title }}</a></span>
  </div>
  <ul id="menucontainer">
    <li class="menu">
      <span><a #asMenu routerLink="/news" routerLinkActive="page-selected"> Novedades </a></span>
    </li>
    <li class="menu">
      <span><a #asMenu routerLink="/popular" routerLinkActive="page-selected"> Popular </a></span>
    </li>
    <li class="menu">
      <span><a #asMenu routerLink="/library" routerLinkActive="page-selected"> Biblioteca </a></span>
    </li>
    <li class="menu">
      <span><a #asMenu routerLink="/discover" routerLinkActive="page-selected"> Descubrir </a></span>
    </li>
  </ul>
  <div class="menu display-flex-center" id="userSession">
    <span><a routerLink="/signin-signup">Iniciar sesión</a></span>
  </div>
</nav>

```

Figura 21. Uso de routerLink

El funcionamiento de la directiva mencionada puede verse en la siguiente imagen, al visitar la sección Biblioteca se muestra el mensaje de que nos encontramos en esta sección y dentro de la barra de navegación presenta un color distinto frente al resto de secciones disponibles.



Figura 22. Comprobación del funcionamiento de las rutas.

Una vez contamos con la estructura básica de nuestro proyecto se procede a implementar la parte funcional del proyecto. Para lo que se tiene que explicar el papel de los servicios en Angular.

Los servicios son clases cuyo uso es encapsular la lógica, la manipulación de datos, la comunicación con APIs y otras funcionalidades no relacionadas de forma directa con el apartado visual de la aplicación. Es otra forma de aplicar la modularización ya que evita que los componentes tengan que gestionar la parte lógica, permitiendo también la reutilización de código y su mantenimiento al tener la lógica centralizada en los servicios, haciendo que las aplicaciones que los usan sean más escalables y facilita su mantenimiento.

En el caso de Luister y debido a la importancia que tienen las APIs para el proyecto, los servicios juegan un papel fundamental. La creación de servicios se ha realizado mediante Angular CLI, con el siguiente comando: **ng g s ruta/nombre-servicio**.

En un primer momento de la creación de servicios debido a su reducido número los endpoints, tokens de acceso de cada uno de los servicios se encontraban dentro de cada uno de los ficheros. A medida que se ha avanzado en la incorporación de otros servicios, estos elementos se han movido a un fichero de environments, con el fin de facilitar su gestión y de proteger las claves a la hora de realizar el despliegue y en el supuesto de que el repositorio pase a ser público.

Dentro de los servicios usados se pueden distinguir 3 grupos en función del uso que tienen dentro de la aplicación:

- Servicios usados para APIs externas.

- Servicios usados para APIs internas o conexión al entorno servidor.
- Otros servicios: validación, guards, menú contextual y alertas personalizadas.

Servicios para APIs externas

El objetivo de estos servicios es permitir la conexión de nuestra aplicación con las APIs de los servicios de streaming de música con el fin de obtener el contenido necesario para Luister. A través de esta conexión se obtienen los datos necesarios de la base de datos de la plataforma, este tipo de datos cambia dependiendo del componente, por lo que dentro de los servicios se han definido una serie de funciones que permiten adaptar el tipo de información recibida por el componente.

Dentro de este grupo de servicios se procede a explicar la estructura y funcionalidad del servicio principal en cuanto a contenido proveniente de las plataformas de streaming de música. Este servicio es el encargado de gestionar y proporcionar la información que necesita la aplicación en cada uno de sus componentes o secciones.

Para poder obtener esta información el servicio debe cumplir con los requisitos de acceso que establezca la plataforma propietaria de la API. En este caso, los requisitos que establece Spotify para poder acceder al contenido son la existencia de un API key válido, (token de acceso o cadena de caracteres que actúa como identificador y validador del origen de las peticiones realizadas a la API) y su renovación después de un período de tiempo, todo ello ligado a una clave de cliente.

En el caso de Spotify, para obtener esta clave de cliente es necesario:

- Registrarse en la web de desarrolladores.
- Crear una aplicación.

Con esta clave de cliente se puede generar el token de acceso para realizar las peticiones en función de los diferentes endpoints que ofrece la plataforma.

Debido a los requisitos establecidos es necesario que cada petición que se realice tenga un token válido. Además, es importante tener en cuenta las condiciones de acceso que establezca la plataforma en cuestión, en este caso Spotify establece una serie de restricciones relativas, entre otras, al

Figura 23. Creación de una aplicación en Spotify.

número de peticiones que se puede realizar con clave de cliente, al uso que se haga del contenido obtenido y a la protección que se debe implementar de este contenido.

Para poder cumplir con los requisitos relativos al token de acceso ligados a una clave de cliente, se definen dentro del servicio los siguientes métodos para generar un token y para su renovación:

```
generateAuthToken():Promise<any>{
  let result;
  return new Promise((resolve)=>{
    if(!this.isValidAuthToken()){
      this.http.post(this.spotifyTokenReq.url, this.spotifyTokenReq.body, { headers: this.spotifyTokenReq.headers })
        .subscribe((res:any) => {
          localStorage.setItem('token_value', res.access_token);
          localStorage.setItem('token_timestamp', new Date().getTime().toString());
          resolve(res.access_token);
        })
    }else {
      result = localStorage.getItem('token_value') || '';
      resolve(result);
    }
  })
}
```

Figura 24. Generación del token de autenticación de Spotify.

```
isValidAuthToken(){
  const TTL = 3300000;
  let tokenTimeStamp = parseInt(localStorage.getItem('token_timestamp') || '0'),
  response = false;
  if(localStorage.getItem('token_value') && ((new Date().getTime() - tokenTimeStamp) <= TTL)) response = true;
  return response;
}
```

Figura 25. Validación del token de autenticación.

Una vez se establecen los métodos de conexión y se comprueba su funcionamiento se procede con la creación de los métodos que permiten la obtención de información. Se han definido varios con el fin de tener el contenido adaptado a las necesidades de los componentes de nuestro proyecto.

```

searchForData(query:string, type:string='album,artist,track', offset=0, limit=20){
  return this.generateAuthToken()
  .then((token) =>{
    return this.http.get(`${this.URL}search?q=${query}&type=${type}&offset=${offset}&limit=${limit}`, {
      headers: {
        Authorization: 'Bearer ' + token
      }
    });
  });
}

getRandomElements(element:string){
  let max=122, min=97,
  char = String.fromCharCode(Math.trunc(Math.random() * (max - min) + min)),
  offset = Math.trunc(Math.random() * 980);
  return this.searchForData(char, element, offset);
}

getAlbumsRelease(url:string=${this.URL}browse/new-releases/){
  return this.generateAuthToken()
  .then((token) =>{
    return this.http.get(url, {
      headers: {
        Authorization: 'Bearer ' + token
      }
    });
  });
}

```

Usado en discover. Permite realizar búsquedas del contenido de la base de datos de Spotify.

Usado en Inicio, para obtener contenido aleatorio.

Usado en Novedades, para obtener el contenido más reciente.

Figura 26. Ejemplo de funciones del servicio de Spotify

Con lo definido en el servicio, se muestra el contenido en los componentes de la siguiente manera:

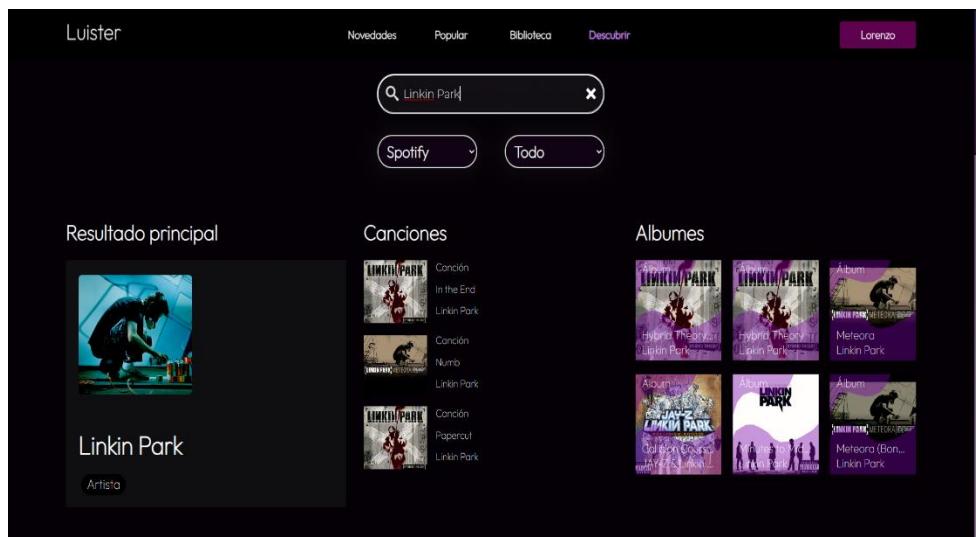


Figura 27. Funcionamiento del componente "Discover"

Otros servicios

Además de los servicios explicados anteriormente se han usado los servicios para distintas tareas, entre los que debemos destacar:

Esta captura pertenece al componente discover. Este componente usa la función **searchForData** mediante la barra de búsqueda. Al introducir un término de búsqueda se ha programado este componente para que muestre el contenido obtenido por el servicio en los bloques definidos en la fase de diseño.

- **Validations.service.ts:** con este servicio y usando las posibilidades brindadas por Angular respecto al tratamiento de formulario. Se han definido métodos para validación de formularios, validación del cumplimiento de expresiones regulares en los formularios de inicio de sesión o verificación de que se introducen contraseñas coincidentes al registrarse en Luister.

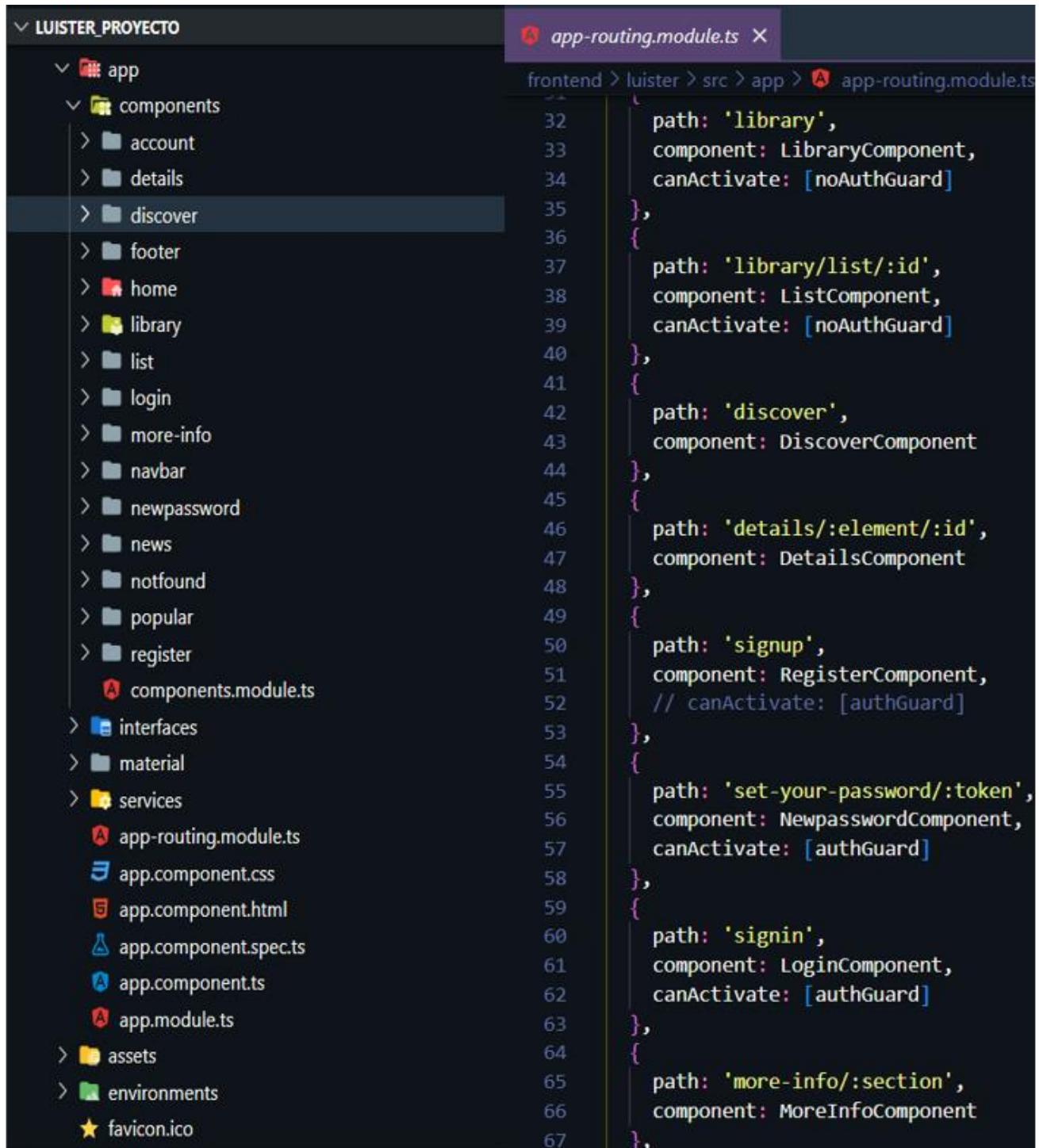
```
public validField( form: FormGroup,
  field: string): boolean | null {
  const control = form.controls[field];
  return control.errors && (control.touched || control.dirty);
}
```

- **contextMenu.ts:** para asegurar que Luister no vulnera las condiciones de uso establecidas por las plataformas, donde se especifica que el contenido obtenido se puede mostrar, pero no se puede facilitar su obtención por medio de nuestro sitio web, es decir, que no se pueda descargar el contenido obtenido. Se ha aprovechado esta restricción para incorporar una funcionalidad dentro de nuestro proyecto, al crear un menú contextual evitamos la descarga de las imágenes mediante la opción existente al hacer clic derecho y se permite que los usuarios acceder a los detalles del elemento o bien una acción personalizada en función del elemento sobre el que se ha hecho clic.
- **luister-cookie-manager.service:** para gestionar las cookies generadas por la actividad de los usuarios de una manera más eficiente se han definido diferentes funciones dentro de un servicio específico. Se ha realizado debido a la importancia de las cookies en el funcionamiento de nuestra aplicación web y ya que está desplegada, cumplir con la obligación impuesta por el RGPD de informar del uso y el tipo de cookies usadas y su aceptación.
- **luisterSweetAlert:** con el fin de no usar las alertas por defecto se ha definido un servicio que permite personalizar los avisos que se muestran al usuario después de realizar una acción dentro de nuestra aplicación.



Figura 28. Menú contextual

Para cerrar la fase de la implementación del entorno cliente, se muestra la estructura de ficheros resultante del proceso realizado y además el contenido del fichero de rutas al final de esta fase, ya que al explicarlo se ha mostrado un ejemplo simple perteneciente a una fase temprana del desarrollo. En esta fase no se habían usado las capacidades de las rutas que ofrece Angular, de ahí que se muestre su estado final.



The image shows a file explorer on the left and a code editor on the right. The file explorer is titled 'LUISTER_PROYECTO' and shows a tree structure of files and folders under 'app'. The 'discover' folder is selected. The code editor is titled 'app-routing.module.ts' and shows the following TypeScript code:

```

32   path: 'library',
33   component: LibraryComponent,
34   canActivate: [noAuthGuard]
35 },
36 {
37   path: 'library/list/:id',
38   component: ListComponent,
39   canActivate: [noAuthGuard]
40 },
41 {
42   path: 'discover',
43   component: DiscoverComponent
44 },
45 {
46   path: 'details/:element/:id',
47   component: DetailsComponent
48 },
49 {
50   path: 'signup',
51   component: RegisterComponent,
52   // canActivate: [authGuard]
53 },
54 {
55   path: 'set-your-password/:token',
56   component: NewpasswordComponent,
57   canActivate: [authGuard]
58 },
59 {
60   path: 'signin',
61   component: LoginComponent,
62   canActivate: [authGuard]
63 },
64 {
65   path: 'more-info/:section',
66   component: MoreInfoComponent
67 },

```

Figura 29. Estructura de ficheros y fichero de rutas final.

Implementación de la base de datos

En base a los requisitos fijados en la fase de diseño, se establece una determinada estructura para el almacenamiento y gestión de la información y preferencias de usuarios de la plataforma. Para ello se crea un servidor de base de datos, usando una base de datos relacional y con **MySQL** con la estructurada detallada a continuación.

Requisitos y motivos de la elección de la estructura

Tal y como se comentó anteriormente, Luister es una aplicación cuya funcionalidad está centrada en APIs y la información obtenida de ellas, sin necesidad de almacenar esta información, lo que simplifica la estructura de la base de datos del proyecto. Debido a esto, solo ha resultado necesario almacenar información referente a los usuarios de la plataforma.

Conceptos

De cada uno de los usuarios se necesita almacenar el nombre de usuario, dirección de correo electrónico, contraseña, fecha de creación de la cuenta de usuario, estado, tipo de usuario (normal o administrador).

Los usuarios mediante el uso de Luister podrán crear listas personalizadas, gestionar su contenido o eliminar las listas. Un usuario puede tener varias listas, sin embargo, una lista solo puede pertenecer a un usuario. Las listas contendrán canciones seleccionadas por los usuarios, de las cuales se almacenarán los datos relativos a su nombre, nombre del artista, nombre del álbum, fecha de inclusión y clave de búsqueda (usada para acceder a los detalles de dicha canción en la API).

Una lista puede tener varias canciones y una canción pertenecer a varias listas.

El usuario puede suscribirse a las novedades de sus artistas preferidos mediante el botón “Seguir”, que se encontrará en la sección accesible al pulsar sobre el nombre del artista, ya sea en el menú contextual o en la sección de detalles de artista.

Un usuario puede seguir varios artistas y un artista puede ser seguido por varios usuarios.

Al registrarse cada usuario cuenta con una lista predeterminada, llamada “Lista de favoritos”, en la cual se almacenan las canciones favoritas del usuario. Las cuales se añaden mediante la funcionalidad “Me gusta”, disponible de la sección detalles y en el listado de canciones de las listas creadas por el usuario, con el fin de agregarlas a “Lista de favoritos” de forma sencilla.

Luister tiene un sistema de recomendaciones predeterminado para los usuarios registrados, que puede ser desactivado en su perfil, esta configuración supone tener una tabla de ajustes de usuarios para almacenar esta información para poder mostrar o no el contenido personalizado.

Teniendo en cuenta lo detallado previamente, se ha definido el siguiente modelo E/R:

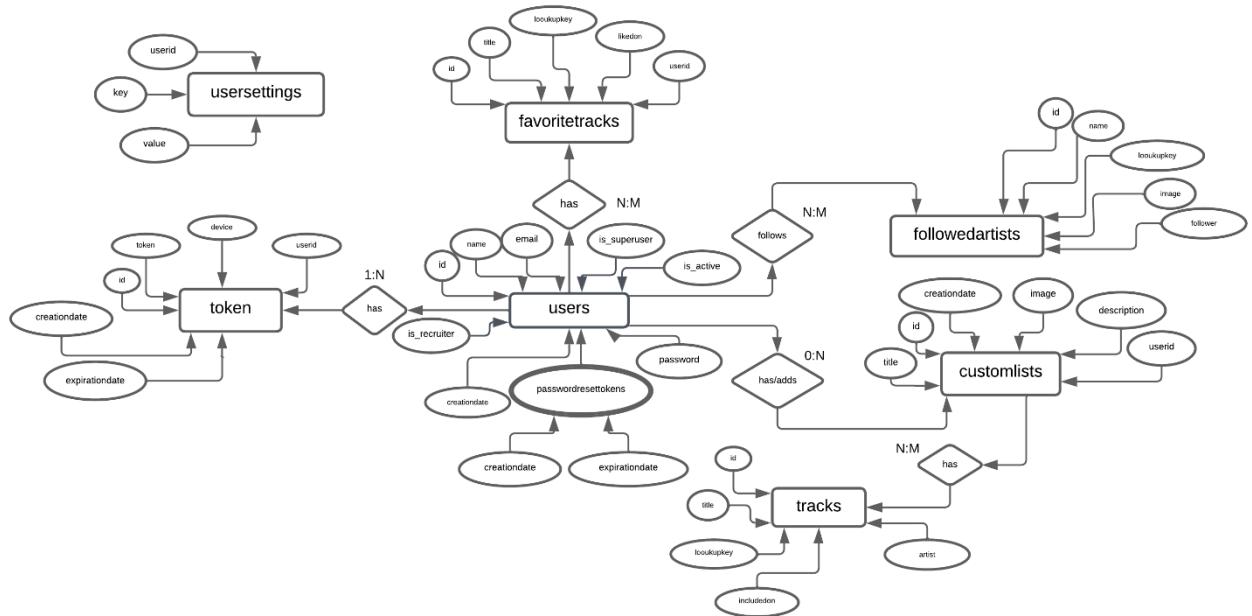


Figura 30. Modelo E/R de Luister

Implementación del entorno servidor

Django Rest Framework es una herramienta para la creación de API REST Web y está basado en el framework de desarrollo web de backend Django, el cual está escrito en Python.

Su metodología de trabajo se basa en lo siguiente: En el servidor almacena la lógica y en el cliente estaría la interfaz, por lo que, a diferencia de Django, en Django Rest no vamos a usar plantillas con CSS.

Entre sus características principales destaca la utilización de unas políticas de autenticación especiales (incluidos paquetes para OAuth1a y OAuth2), el sistema de serialización que admite orígenes de datos ORM y no ORM, el uso de vistas basadas en funciones personalizables, etc.

Para que todos los componentes de una API Web se comuniquen sin ningún problema, establece algunas reglas. Entre ellas se encuentra que utilicen el mismo tipo de dato para la comunicación de la información. Así, el estándar para devolver la información será por lo general o en formato XML o en formato JSON, este último el más usado. Además, Django Rest Framework toma como base el protocolo http y ofrece diversos métodos que puede usar el cliente para mandar peticiones: GET, POST, PUT y DELETE. Por último, define que la petición no tiene que tener estado (no tiene que depender de otra petición para poder responder) pero el servidor tiene que indicarle al cliente unos códigos de estado junto con la información a devolver, los cuales se clasifican en 4 grupos: 20X (Éxito), 30X (Redirección), 40X (No encontrado) y 50X (Error de servidor).

Se han usado las siguientes versiones:

- Django: 4.2.1
- Python: 3.11.1

Creación del proyecto

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 38 de 113
--------------------------	--------------------	------------------

A la hora de la planificación, se decide crear un proyecto llamado api y dentro del varias aplicaciones cada una con sus respectivas funcionalidades. Esto se hace con el comando: **\$django-admin startproject <nombre_proyecto>**. En este caso el comando seria \$django-admin startproject api.

En la configuración del propio proyecto se deja la configuración por defecto de sqlite, aunque ya veremos que posteriormente se cambia a mysql para facilitar la integración con el frontend.

```
 DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR + 'db.sqlite3',
    }
}
```

Figura 31. Configuración de base de datos en Django

Creamos las migraciones, creamos el super usuario y desplegamos el servidor.

Para ello usamos los siguientes comandos: **\$python manage.py makemigrations**, **\$python manage.py migrate**, **\$python manage.py createsuperuser** y **\$python manage.py runserver**, respectivamente.

Instalación de librerías

Se crea el fichero **requirements.txt** para la instalación del entorno de desarrollo del servidor de la aplicación. En este fichero se encontrarán detalladas las dependencias necesarias para el proyecto, las cuales se procederán a instalar mediante el comando **pip install -r requirements.txt**.

y registramos django rest framework en el archivo settings del proyecto, concretamente en el apartado **INSTALLED_APPS**.

Configuración de la carpeta media

Se procede a realizar las configuraciones para el tratamiento de imágenes: creación de la carpeta media, especificación de la ruta y enlace al a misma en el archivo urls.py.

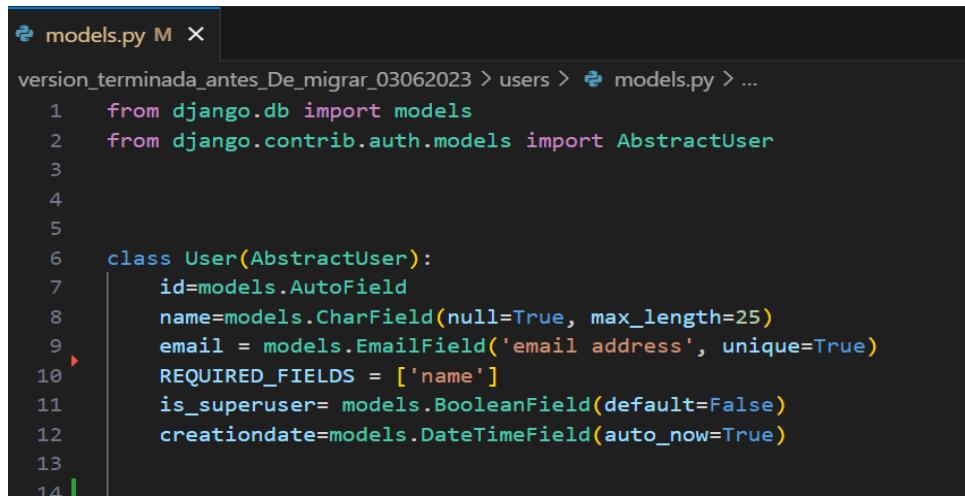
Creación de las aplicaciones

1.Users

La primera aplicación que se crea es la de Users, que será la principal, mediante el comando **\$django-admin startapp users**.

- Models.py

Lo primero que se hace en esta aplicación es crear los modelos acorde a la base de datos del frontend con los campos name, email, is_superuser y creationdate.



```

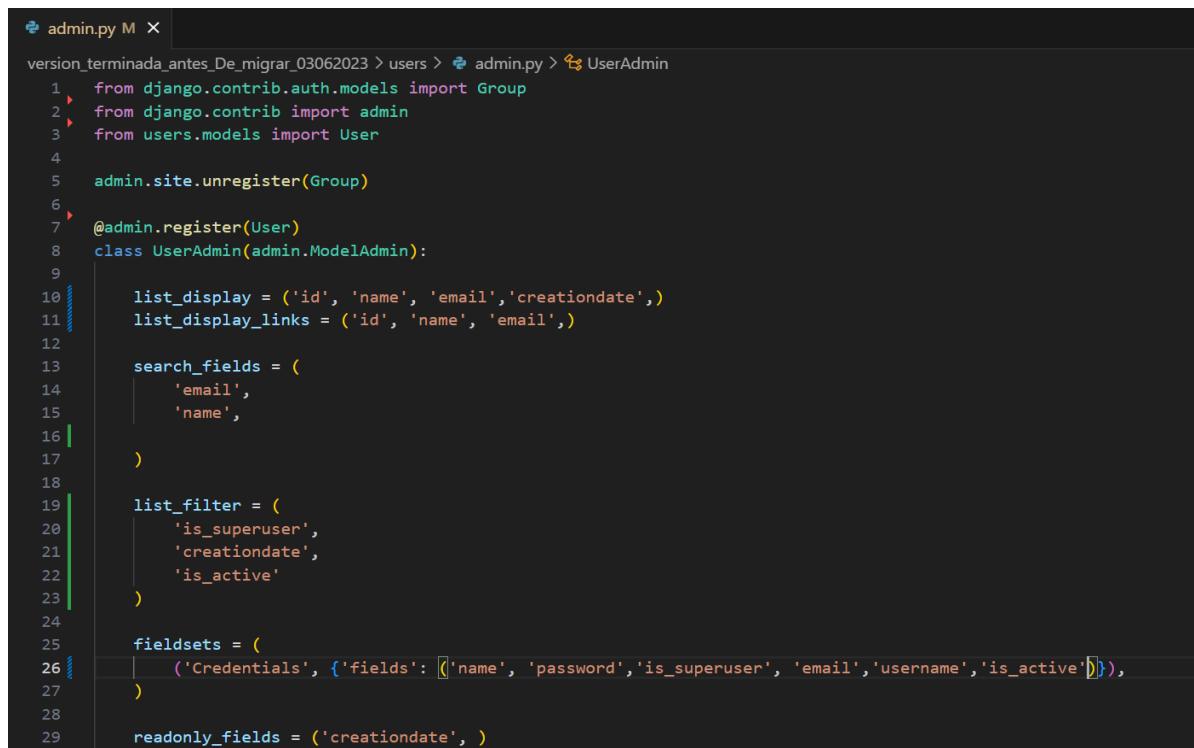
version_terminada_anter_De_migrar_03062023 > users > models.py > ...
1   from django.db import models
2   from django.contrib.auth.models import AbstractUser
3
4
5
6   class User(AbstractUser):
7       id=models.AutoField
8       name=models.CharField(null=True, max_length=25)
9       email = models.EmailField('email address', unique=True)
10      REQUIRED_FIELDS = ['name']
11      is_superuser= models.BooleanField(default=False)
12      creationdate=models.DateTimeField(auto_now=True)
13
14

```

Figura 32. Modelo de Usuario

una vez creadas las migraciones, en el archivo admin de la app seleccionamos los campos que se desean agregar en el panel de administración

- Admin.py



```

version_terminada_anter_De_migrar_03062023 > users > admin.py > UserAdmin
1   from django.contrib.auth.models import Group
2   from django.contrib import admin
3   from users.models import User
4
5   admin.site.unregister(Group)
6
7   @admin.register(User)
8   class UserAdmin(admin.ModelAdmin):
9
10      list_display = ('id', 'name', 'email','creationdate')
11      list_display_links = ('id', 'name', 'email',)
12
13      search_fields = (
14          'email',
15          'name',
16      )
17
18      list_filter = (
19          'is_superuser',
20          'creationdate',
21          'is_active'
22      )
23
24      fieldsets = (
25          ('Credentials', {'fields': ['name', 'password','is_superuser', 'email','username','is_active']}),
26      )
27
28      readonly_fields = ('creationdate', )
29

```

Figura 33. Modelo de Admin de usuarios.

A continuación, se importa el modelo y especificamos que en la primera página de visualización nos muestre los campos id, name, email y creationdate, este último en modo sólo lectura y el resto configurados para poder hacer click en ellos.

	ID	NAME	EMAIL ADDRESS	CREATIONDATE
<input type="checkbox"/>	1	alumno	alumno@alumno.com	10 de junio de 2023 a las 19:37
1 usuario				

Figura 34. Tabla de usuarios en el panel de administración.

La tabla Groups forma parte de la configuración por defecto de django rest, pero como en principio no se va a hacer uso de ella se decide ocultarla en el panel a través de la función `unregister`. Las búsquedas se podrán hacer por email y por nombre y los filtros por super usuario, usuario activo y fecha de creación.

Al acceder a los detalles de un usuario aparecen los datos especificados en fieldsets. Es importante reflejar aquí el campo `is_active` para poder banear a usuarios manualmente desde el panel de admin.

Modificar usuario

alumno@alumno.com HISTÓRICO

Credentials

Name:	alumno
Contraseña:	pbkdf2_sha256\$600000\$S7O2xTYCvHyftlR0te
<input checked="" type="checkbox"/> Is superuser	
Email address:	alumno@alumno.com
Nombre de usuario:	alumno
Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/./+/-/_	
<input checked="" type="checkbox"/> Activo	Indica si el usuario debe ser tratado como activo. Desmarque esta opción en lugar de borrar la cuenta.

Figura 35. Detalles de usuario

Como se puede observar en esta imagen, se implementó además una funcionalidad extra para conocer el histórico de cada usuario a través del botón histórico ubicado en la esquina superior derecha.

Histórico de modificaciones: alumno@alumno.com

FECHA/HORA	USUARIO	ACCIÓN
10 de junio de 2023 a las 19:37	alumno@alumno.com	Modificado Correo electrónico.
1 entrada		

Figura 36. Historial de usuario

Para ello tan sólo fue necesario ejecutar el comando `$pip install django-simple-history`.

- `Permissions.py`

Django Rest Framework tiene por defecto algunos paquetes de permisos que permiten saber por ejemplo si el usuario está autenticado, mediante la clase `IsAuthenticated`, pero además en esta aplicación se añade una restricción nueva para evitar que el administrador modifique o añada

accidentalmente datos de los usuarios desde la url y pueda hacerlo sólo desde el panel de administrador para mayor seguridad. Se creó un archivo permissions.py en el que se añadió el siguiente código:

```
from rest_framework.permissions import BasePermission

from users.models import User

class IsStandardUser(BasePermission):

    def has_permission(self, request, view):

        try:
            user = User.objects.get(
                email=request.user,
                is_superuser=False
            )
        except User.DoesNotExist:
            return False
        return True
```

Figura 37. Fichero permissions de usuario

- Serializers.py

Los serializadores de django rest son los responsables de convertir objetos en tipos de datos comprensibles para javascript y marcos front-end. Convierten instancias de modelos (consultas a la base de datos) en formato JSON (acción llamada serializar), o viceversa (deserializar).

La diferencia entre Serializer y Modelserializer es que el primero se encarga de hacer validaciones y tratamiento de datos y el segundo se asimilaría más a los datos que devuelve un formulario, aunque a diferencia de estos, los Serializers no están obligados a trabajar en la respuesta HTML y con la codificación de envío de formularios.

Por tanto, además del formato de respuesta, definen qué datos pedirá al usuario y si queremos hacer alguna acción con ellos. Así, creamos un archivo serializers.py dentro de la app users y reflejamos los datos que nos interesaban.

```
from django.contrib.auth import password_validation, authenticate

import random
from rest_framework import serializers
from rest_framework.authtoken.models import Token
from rest_framework.validators import UniqueValidator
from users.models import User
```

Figura 38. Ejemplo importaciones de serializer de usuario

En primer lugar importamos todas las clases que necesitamos, destacando el método authenticate que necesitará como parámetros obligatorios username y password, password_validation para la validación de contraseña 1 y contraseña 2 en el registro, random para generar un número aleatorio y almacenarlo en username sustituyendo el campo username por email en el login y registro, y el token que se genera cuando el usuario se loguea.

```
class UserModelSerializer(serializers.ModelSerializer):  
  
    class Meta:  
  
        model = User  
        fields = [  
            'id',  
            'email',  
            'name',  
        ]
```

Figura 39. Modelo serializer de usuario

UserModelSerializers lo usamos para poder recuperar la información de un modelo, en este caso le estamos diciendo que la información que devuelva cuando se genere una acción definida sea el id, email y name del usuario.

A partir de aquí se crean clases para cada acción específica (login, logout y registro).

En el login se opta por sustituir el username obligatorio por email, este cambio se refleja en UserLoginSerializer y además hay que especificarle en models.py que el campo username sería el equivalente al email:

```
USERNAME_FIELD='email'
```

Figura 40. Campo Username, modelo de usuario

En esta misma clase, para validar que los datos sean correctos primero autentica al usuario. En caso de ser correctas las credenciales devuelve los datos especificados en UserModelSerializer, y si no muestra el mensaje de error. Para crear el token, hace uso del método get_or_create pasándole por parámetro los datos del usuario y finalmente nos devuelve un json con los datos de usuario y el valor del token almacenado en la variable por defecto llamada access_token.

```
class UserLoginSerializer(serializers.Serializer):

    email = serializers.EmailField()
    password = serializers.CharField(min_length=8, max_length=64)

    def validate(self, data):

        user = authenticate(username=data['email'], password=data['password'])
        if not user:
            raise serializers.ValidationError('Invalid credentials')

        self.context['user'] = user
        return data

    def create(self, data):

        token, created = Token.objects.get_or_create(user=self.context['user'])
        return self.context['user'], token.key
```

Figura 41. Serializador de login

UserSignUpSerializer: Es la clase encargada del registro. Se especifica que los datos que se desean son sólo name, email (único) y las dos contraseñas las cuales tienen un mínimo de 8 caracteres.

```
class UserSignUpSerializer(serializers.Serializer):

    name=serializers.CharField(
        min_length=3,
        max_length=10)

    email = serializers.EmailField(
        validators=[UniqueValidator(queryset=User.objects.all())]
    )

    password = serializers.CharField(min_length=8, max_length=64)

    password_2 = serializers.CharField(min_length=8, max_length=64)
```

Figura 42. Serializador de registro

Dentro de esta clase se llevan a cabo las validaciones de los campos:

```

def validate(self, data):
    password1 = data['password']
    password2 = data['password_2']
    if password1 != password2:
        raise serializers.ValidationError("password and password_2 don't match")
    password_validation.validate_password(password1)
    return data

def create(self, data):
    data.pop('password_2')
    user = User.objects.create_user(**data,username=f'{random.randrange(10000000)}')
    return user

```

Figura 43. Métodos de registro

El método de validación en este caso se hace a través del método validate_password, el cual si son iguales nos devuelve los datos del modelo del serializador y si no muestra el error.

Debido a que, como se ha explicado anteriormente, se ha sustituido el campo obligatorio username por email en los datos que se piden al usuario, se tiene que añadir como parámetro de creación del usuario un valor por defecto del username (un número aleatorio convertido a string).

Necesitaremos además decoradores como action y receiver ya que decoran las vistas basadas en funciones para garantizar que reciban una instancia de Request (en sustitución de la respuesta de Django HttpRequest habitual) y les permite devolver una respuesta de tipo Response (en lugar de una respuesta Django HttpResponse), así como configurar cómo será procesada nuestra solicitud. Además, necesitaremos importar las vistas por defecto de django rest y el método de reseteo de contraseña que se explicará más adelante

- Views.py

Los ViewSet son clases similares a las vistas de Django pero añadiendo la automatización de las respuestas, métodos de intercambio (GET, POST...), los modelos que se utilizarán y como se serializará la información resultante.

```

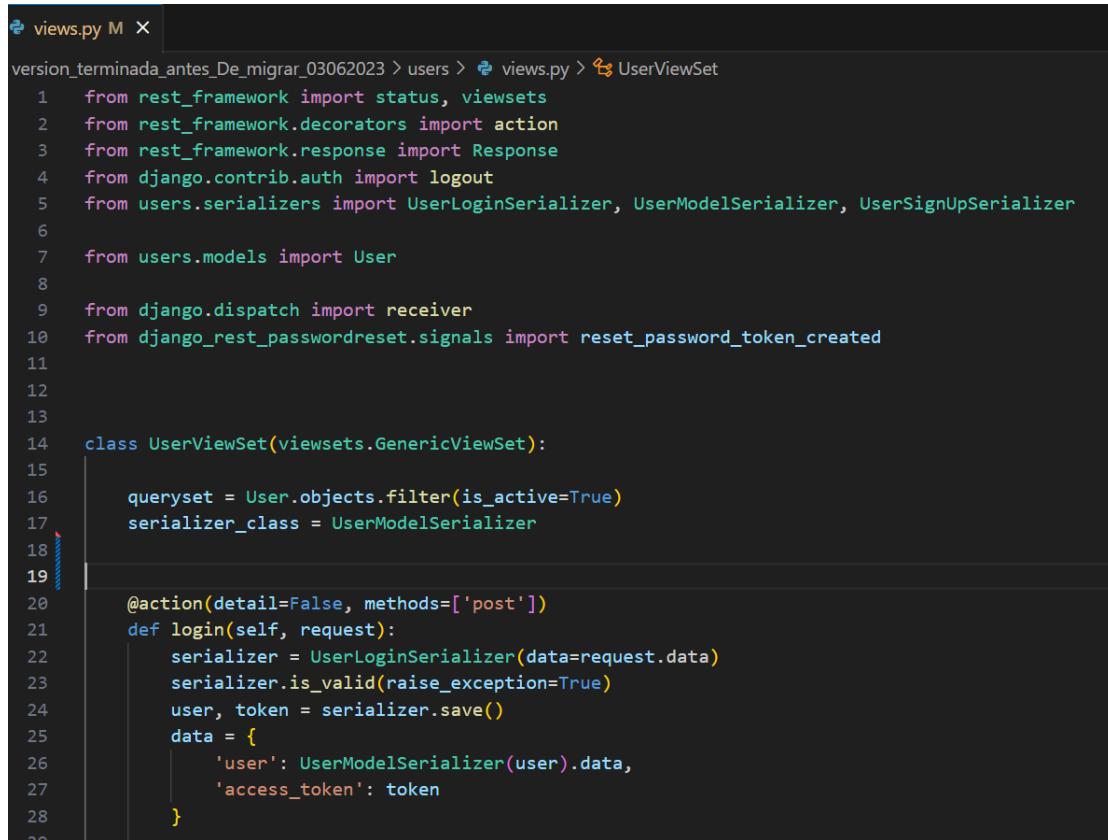
from rest_framework import status, viewsets
from rest_framework.decorators import action
from rest_framework.response import Response
from django.contrib.auth import logout
from users.serializers import UserLoginSerializer, UserModelSerializer, UserSignUpSerializer

from users.models import User

from django.dispatch import receiver
from django_rest_passwordreset.signals import reset_password_token_created

```

Figura 44. Vistas de usuario, importaciones

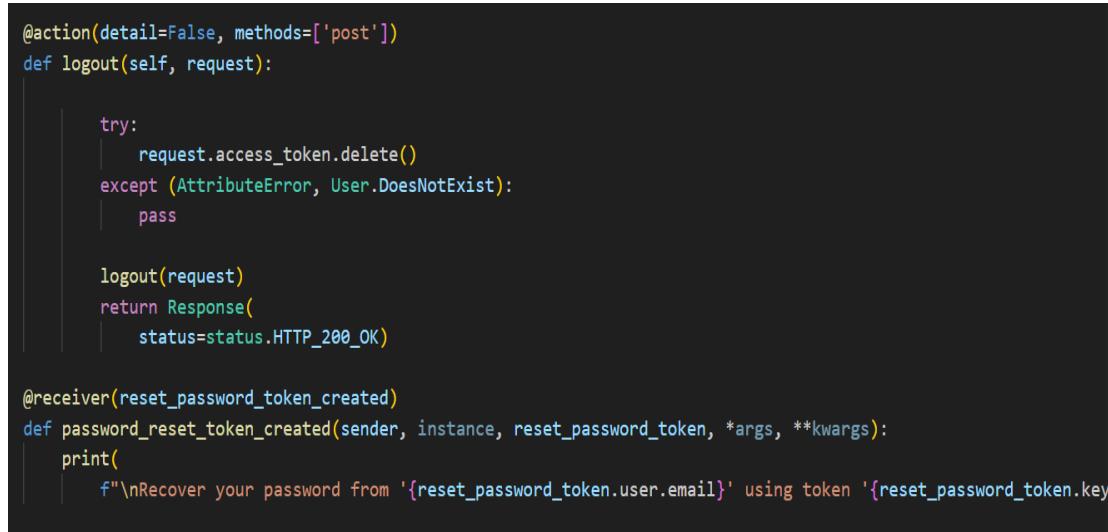


```

version_terminada_anter_De_migrar_03062023 > users > views.py > UserViewSet
  1  from rest_framework import status, viewsets
  2  from rest_framework.decorators import action
  3  from rest_framework.response import Response
  4  from django.contrib.auth import logout
  5  from users.serializers import UserLoginSerializer, UserModelSerializer, UserSignUpSerializer
  6
  7  from users.models import User
  8
  9  from django.dispatch import receiver
 10  from django_rest_passwordreset.signals import reset_password_token_created
 11
 12
 13
 14  class UserViewSet(viewsets.GenericViewSet):
 15
 16      queryset = User.objects.filter(is_active=True)
 17      serializer_class = UserModelSerializer
 18
 19
 20      @action(detail=False, methods=['post'])
 21      def login(self, request):
 22          serializer = UserLoginSerializer(data=request.data)
 23          serializer.is_valid(raise_exception=True)
 24          user, token = serializer.save()
 25          data = {
 26              'user': UserModelSerializer(user).data,
 27              'access_token': token
 28          }
 29

```

Figura 45. Vistas de usuario



```

@action(detail=False, methods=['post'])
def logout(self, request):
    try:
        request.access_token.delete()
    except (AttributeError, User.DoesNotExist):
        pass

    logout(request)
    return Response(
        status=status.HTTP_200_OK)

@receiver(reset_password_token_created)
def password_reset_token_created(sender, instance, reset_password_token, *args, **kwargs):
    print(
        f"\nRecover your password from '{reset_password_token.user.email}' using token '{reset_password_token.key}"

```

Figura 46. Decoradores de usuario

La clase UserViewSet hereda de GenericViewSet, recolecta todos los usuarios que estén activos y referencia a UserModelSerializer.

Tiene un método login que llama a UserLoginSerializer para construirlo sólo con los atributos especificados en el mismo (email y password), recibe los datos que ha introducido el usuario por método POST, si son válidos se guarda el token asociado al usuario en la tabla Tokens.

TOKEN DE AUTENTICACIÓN		
Tokens	 Añadir	 Modificar

Figura 47. Tabla tokens en admin

y se devuelven los datos en forma de diccionario junto con el código 201(created).

El registro hace referencia a UserSignupSerializer para pedir únicamente name, email, password1 y password2 y hace lo mismo que el login pero en el diccionario data sólo guarda los datos del usuario, sin token. Cuando haya comprobado que los datos son válidos los guardará en la tabla Usuarios

USERS		
Usuarios	 Añadir	 Modificar

Figura 48. Tabla usuarios en admin

El logout borra de request el atributo access_token, borra los datos de sesión y responde con un código de estado 200(ok), salvo que el usuario no exista o no tenga access_token en cuyo caso emitirá un mensaje de error.

Por último, se ha implementado la función del reseteo de la contraseña por lo que primero lo instalamos `pip install django-rest-passwordreset`, lo registramos en el archivo settings del proyecto

```
INSTALLED_APPS = [
    ...
    'django_rest_passwordreset',
    ...
]
```

Figura 49. Registro reset contraseña

e importamos `reset_password_token_created` en la cabecera del archivo.

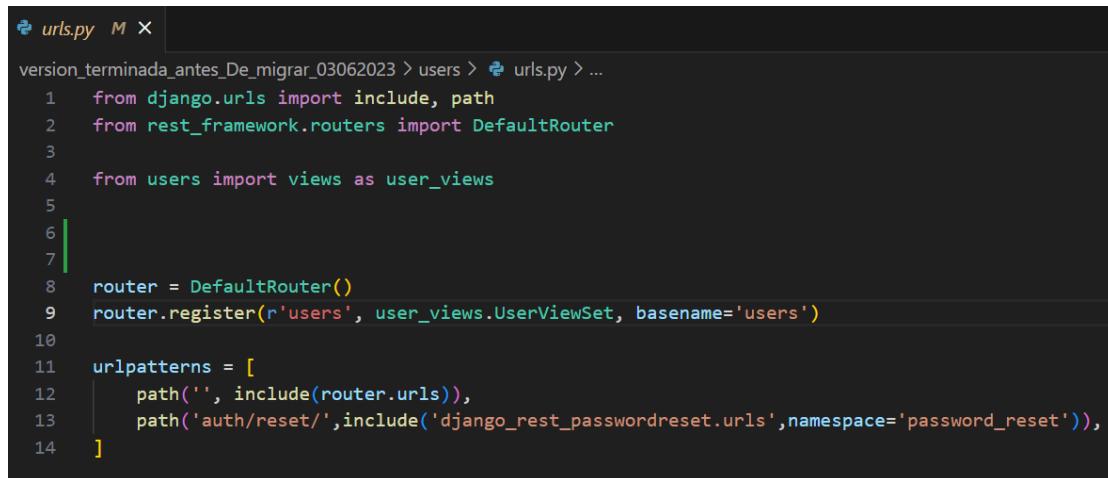
En este método el usuario introducirá su email y se creará un token de reseteo en la tabla `>Password_reset_token`

DJANGO_REST_PASSWORDRESET		
Password Reset Tokens	 Añadir	 Modificar

Figura 50. Tabla reset contraseña en admin

A continuación, le llegará un mensaje con el link al que tiene que acceder para resetear la contraseña junto con el token que se le pedirá. Cuando introduzca los datos requeridos (token y nueva contraseña) la contraseña se actualizará en tabla Usuarios.

Finalmente se crea un archivo `urls.py` donde se registra la ruta de la aplicación haciendo uso del router de django para mayor comodidad



```

urls.py M ×
version_terminada_anterior_Desarrollar_03062023 > users > urls.py > ...
1  from django.urls import include, path
2  from rest_framework.routers import DefaultRouter
3
4  from users import views as user_views
5
6
7
8  router = DefaultRouter()
9  router.register(r'users', user_views.UserViewSet, basename='users')
10
11 urlpatterns = [
12     path('', include(router.urls)),
13     path('auth/reset/', include('django_rest_passwordreset.urls', namespace='password_reset')),
14 ]

```

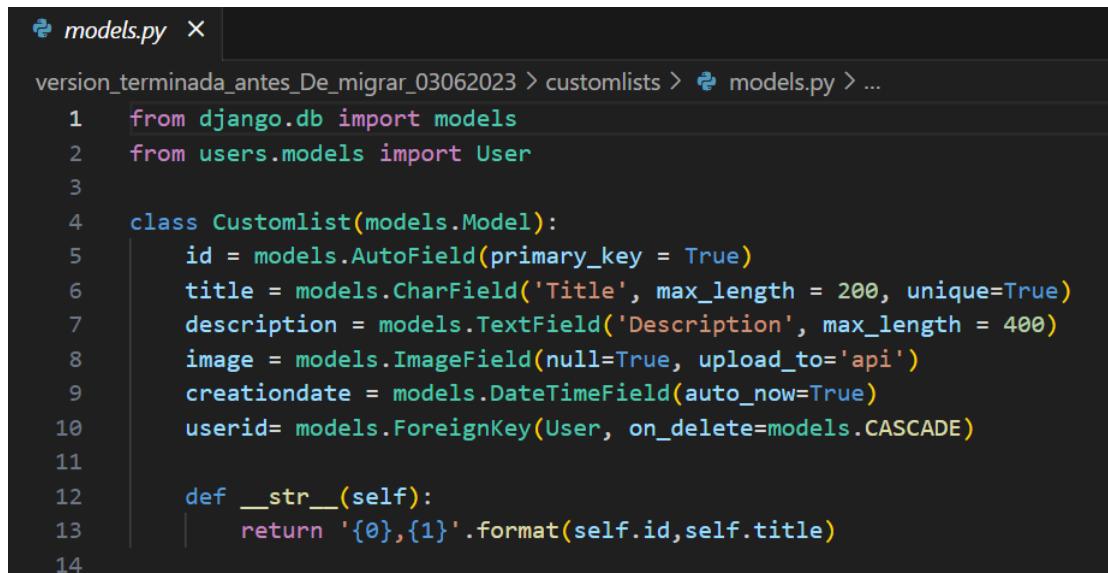
Figura 51. FicheroUrls

lo enlazamos con el archivo urls.py del proyecto y registramos la aplicación en el archivo settings.py del proyecto.

2. Customlists

- Models.py

La lista de usuarios tendrá los datos título, descripción, image(opcional), fechad e creación y la foreign key que será el id de usuario



```

models.py X
version_terminada_anterior_Desarrollar_03062023 > customlists > models.py > ...
1  from django.db import models
2  from users.models import User
3
4  class Customlist(models.Model):
5      id = models.AutoField(primary_key = True)
6      title = models.CharField('Title', max_length = 200, unique=True)
7      description = models.TextField('Description', max_length = 400)
8      image = models.ImageField(null=True, upload_to='api')
9      creationdate = models.DateTimeField(auto_now=True)
10     userid= models.ForeignKey(User, on_delete=models.CASCADE)
11
12     def __str__(self):
13         return '{0},{1}'.format(self.id,self.title)
14

```

Figura 52. Modelo de customList

- Admin.py

después de crear las migraciones registramos el modelo en el admin.py de la siguiente manera:

```

# admin.py M X
version_terminada_anter_De_migrar_03062023 > customlists > # admin.py
1  from django.contrib import admin
2
3  from customlists.models import Customlist
4
5  class CustomlistAdmin(admin.ModelAdmin):
6      list_display = ('id', 'userid')
7
8  admin.site.register(Customlist)
9
10
11

```

Figura 53. Admin de customList

- Serializers.py

El modelo serializador mostrará por pantalla todos los datos excepto el id de usuario y su serializador pedirá al usuario el título, la descripción y la imagen.

```

# serializers.py M X
version_terminada_anter_De_migrar_03062023 > customlists > # serializers.py > # Customlist
1  from rest_framework import serializers
2
3  from customlists.models import Customlist
4
5  class CustomlistModelSerializer(serializers.ModelSerializer):
6
7      class Meta:
8
9          model = Customlist
10         fields = (
11             'id',
12             'title',
13             'description',
14             'image',
15             'creationdate',
16         )
17
18

```

Figura 54. Modelo serializador de customList

La fecha de creación se rellenará gracias al método auto_now y el userid será el del usuario que esté registrado en ese momento, proporcionado por el método de los serializadores llamado CurrentUserDefault()

```
class CustomlistSerializer(serializers.Serializer):  
  
    userid = serializers.HiddenField(default=serializers.CurrentUserDefault())  
    title = serializers.CharField(max_length=250)  
    description = serializers.CharField(allow_null=True)  
    image = serializers.ImageField(max_length=None, use_url=True, allow_null=True, required=False)  
    creationdate = serializers.DateTimeField()  
  
    def create(self, data):  
  
        customlist = Customlist.objects.create(**data)  
        return customlist
```

Figura 55. Serializador de customList

Cuando se haya creado la lista del usuario, la devolverá en el formato json con los datos especificados en el modelo serializador.

```
views.py  x  
  
version_terminada_anter_De_migrar_03062023 > customlists > views.py > CustomlistViewSet  
1  from rest_framework import mixins, status, viewsets  
2  from rest_framework.response import Response  
3  
4  from rest_framework.permissions import IsAuthenticated  
5  from users.permissions import IsStandardUser  
6  
7  from customlists.serializers import (CustomlistModelSerializer, CustomlistSerializer)  
8  
9  from customlists.models import Customlist  
10
```

Figura 56. Importaciones de vistas de customList

- Views.py

El archivo views.py de esta aplicación va a necesitar la librería mixins, cuyas clases proporcionan las acciones que se utilizan para obtener el comportamiento de vista básico. Esto lo hicimos ya que tienen la ventaja de facilitar mucho el trabajo a la hora de la creación, listado, actualización y borrado en nuestra API. También va a necesitar el método IsAuthenticated de django junto con nuestro método personalizado IsStandardUser definido en el archivo permissions.py de la aplicación Users. Por último hace uso de los serializadores y de su modelo.

```
views.py X
version_terminada_anteriores_De_migrar_03062023 > customlists > views.py > CustomlistViewSet
  ↴
11  class CustomlistViewSet(mixins.ListModelMixin,
12      mixins.CreateModelMixin,
13      mixins.UpdateModelMixin,
14      mixins.DestroyModelMixin,
15      viewsets.GenericViewSet):
16
17      serializer_class = CustomlistModelSerializer
18
19      def get_permissions(self):
20          permission_classes = [IsAuthenticated, IsStandardUser]
21          return [permission() for permission in permission_classes]
22
23
24      def get_queryset(self):
25          queryset = Customlist.objects.filter(userid=self.request.user)
26          return queryset
27
28
29      def create(self, request, *args, **kwargs):
30          serializer = CustomlistSerializer(data=request.data, context={"request": self.request})
31          serializer.is_valid(raise_exception=True)
32          customlist = serializer.save()
33          data = CustomlistModelSerializer(customlist).data
```

Figura 57. Vistas de *customList*

```
return Response(data, status=status.HTTP_201_CREATED)
```

Figura 58. Respuesta de creación de customList

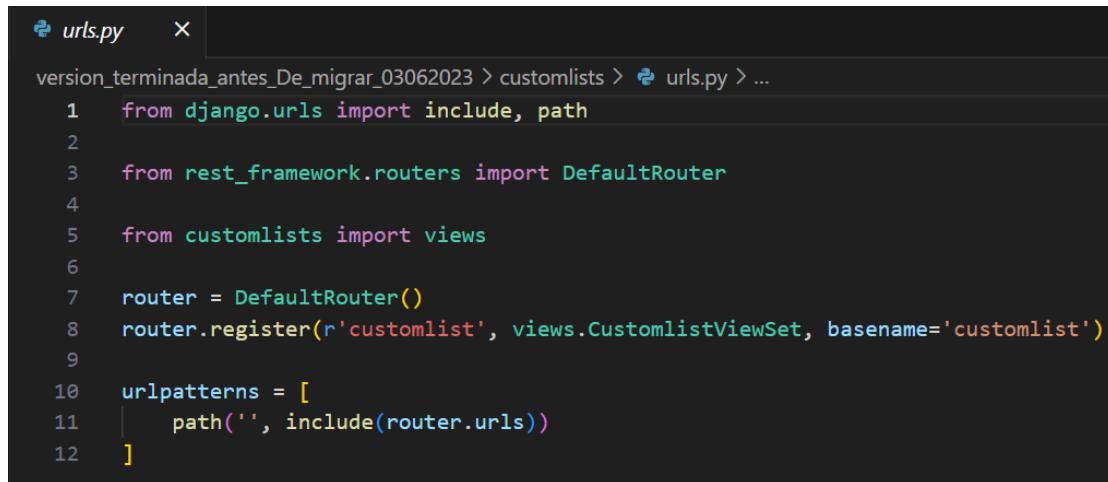
Tiene un método `get_permissions` para verificar que quien requiera una acción de listado, modificación, creación o borrado desde la url sea el usuario estándar y no el administrador, que sólo podrá realizar esta acción desde el panel de administración.

El método `get_queryset` filtra los usuarios por el id de usuario almacenado en `request` y el método `create` define los datos que se pedirán al usuario, los cuales estaban definidos por `CustomlistSerializer`. Si son válidos se guarda una nueva lista de usuario en la tabla `Customlists`:



Figura 59. Tabla *customList* en *admin*

y devuelve los datos que contiene CustomlistModelSerializer en un diccionario



```

urls.py  X
version_terminada_anter_De_migrar_03062023 > customlists > urls.py > ...
1  from django.urls import include, path
2
3  from rest_framework.routers import DefaultRouter
4
5  from customlists import views
6
7  router = DefaultRouter()
8  router.register(r'customlist', views.CustomlistViewSet, basename='customlist')
9
10 urlpatterns = [
11     path('', include(router.urls))
12 ]

```

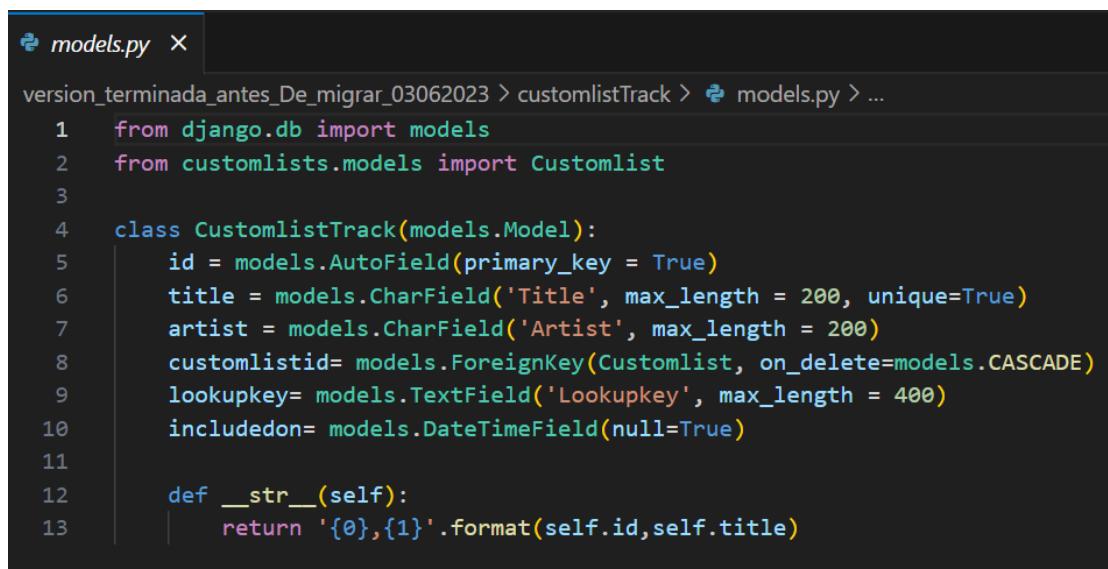
Figura 60. Fichero urls customList

Finalmente registramos en el archivo urls.py las rutas propias de la aplicación y las enlazamos con las urls del proyecto y registramos la aplicación en el archivo settings.py del proyecto

3. CustomlistTracks

- Models.py

El archivo models.py almacena los datos title, artist, lookupkey(clave de búsqueda de Spotify, el cual sirve para recoger el usuario de la base de datos y mostrar los detalles) e includedon (fecha de inclusión del elemento a la lista), seguidos del id del usuario que es la clave foránea de esta aplicación, la cual referencia a su vez a un id de usuario.



```

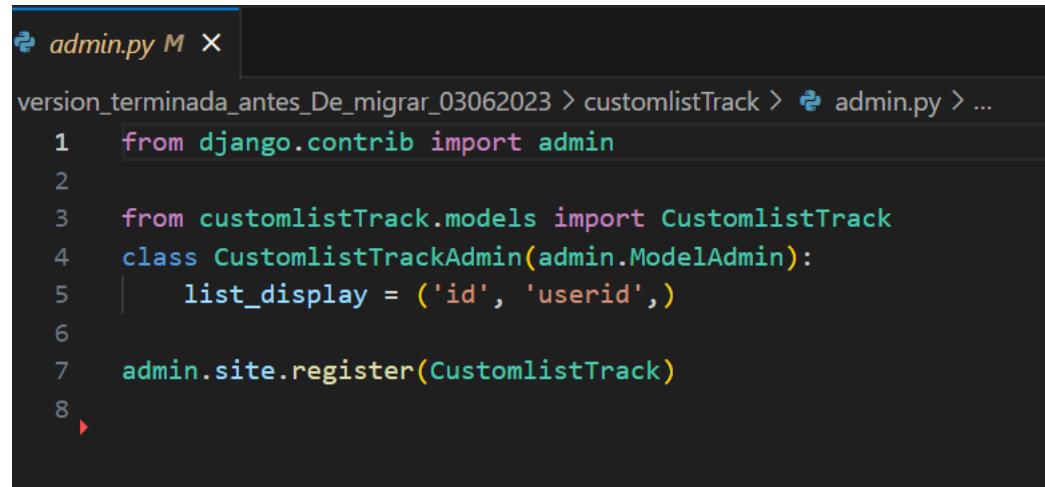
models.py  X
version_terminada_anter_De_migrar_03062023 > customlistTrack > models.py > ...
1  from django.db import models
2  from customlists.models import Customlist
3
4  class CustomlistTrack(models.Model):
5      id = models.AutoField(primary_key = True)
6      title = models.CharField('Title', max_length = 200, unique=True)
7      artist = models.CharField('Artist', max_length = 200)
8      customlistid= models.ForeignKey(Customlist, on_delete=models.CASCADE)
9      lookupkey= models.TextField('Lookupkey', max_length = 400)
10     includedon= models.DateTimeField(null=True)
11
12     def __str__(self):
13         return '{0},{1}'.format(self.id,self.title)

```

Figura 61. Modelo customListTracks

- Admin.py

Después creamos las migraciones correspondientes y lo registramos en el admin.py

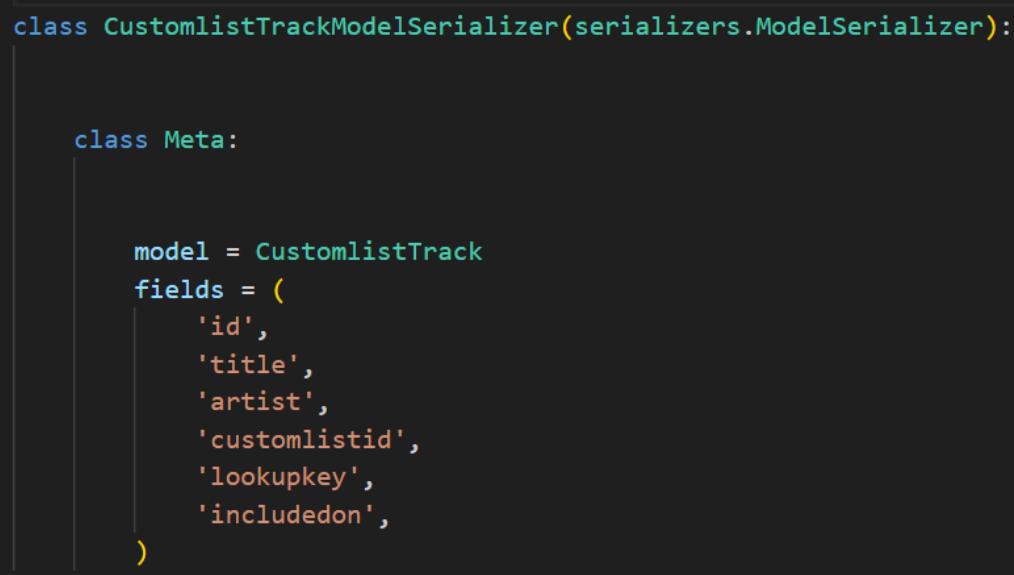


```
admin.py M X
version_terminada_anterior_De_migrar_03062023 > customlistTrack > admin.py > ...
1  from django.contrib import admin
2
3  from customlistTrack.models import CustomlistTrack
4  class CustomlistTrackAdmin(admin.ModelAdmin):
5      list_display = ('id', 'userid',)
6
7  admin.site.register(CustomlistTrack)
8
```

Figura 62. Admin customListTrack

- Serializers.py

El modelo serializador emitirá los siguientes datos:



```
class CustomlistTrackModelSerializer(serializers.ModelSerializer):
    class Meta:
        model = CustomlistTrack
        fields = (
            'id',
            'title',
            'artist',
            'customlistid',
            'lookupkey',
            'includedon',
        )
```

Figura 63. Modelo serializador customListTrack

y el serializador requerirá para la creación de una nueva CustomlistTrack el título, el artista, lookupkey e includedon. Debido a que hay almacenada una clave foránea que a su vez referencia a otra y que el necesita, es necesario crear una instancia de Customlist filtrando por el id de Customlist de los datos que requerirá y en caso de que ya esté añadida esa Customlist en ese usuario emitirá un mensaje de error. De lo contrario creará una nueva customlistTrack pasándole como parámetro la instancia del Customlist y el resto de atributos:

```

class CustomlistTrackSerializer(serializers.Serializer):

    customlistid=serializers.IntegerField()
    title = serializers.CharField(max_length=250)
    artist = serializers.CharField(allow_null=True)
    lookupkey = serializers.CharField(allow_null=True)
    includedon = serializers.DateTimeField(allow_null=True)

    def create(self, data):
        cid = Customlist.objects.get(id=data['customlistid'])
        try:
            customlist = CustomlistTrack.objects.get(customlistid=data['customlistid'],title=data['title'],artist=data['artist'],lookupkey=data['lookupkey'],includedon=data['includedon'])
        except:
            new = CustomlistTrack.objects.create(customlistid=cid,title=data['title'],artist=data['artist'],lookupkey=data['lookupkey'],includedon=data['includedon'])
            return new
        raise forms.ValidationError(u'CustomlistTrack "%s" is already added' % customlist)

```

Figura 64. Serializador customListTrack

- Views.py

Al igual que en customlist, requerirá que el usuario esté autenticado y que sea usuario estándar y emitirá un mensaje de error si hay algún error a la hora de la creación

```

class CustomlistTrackViewSet(mixins.ListModelMixin,
                             mixins.CreateModelMixin,
                             mixins.UpdateModelMixin,
                             mixins.DestroyModelMixin,
                             viewsets.GenericViewSet):

    serializer_class = CustomlistTrackModelSerializer

    def get_permissions(self):
        permission_classes = [IsAuthenticated, IsStandardUser]
        return [permission() for permission in permission_classes]

    def get_queryset(self):
        queryset = CustomlistTrack.objects.filter(customlistid=self.request.user)
        return queryset

```

Figura 65. Vistas customListTrack

```

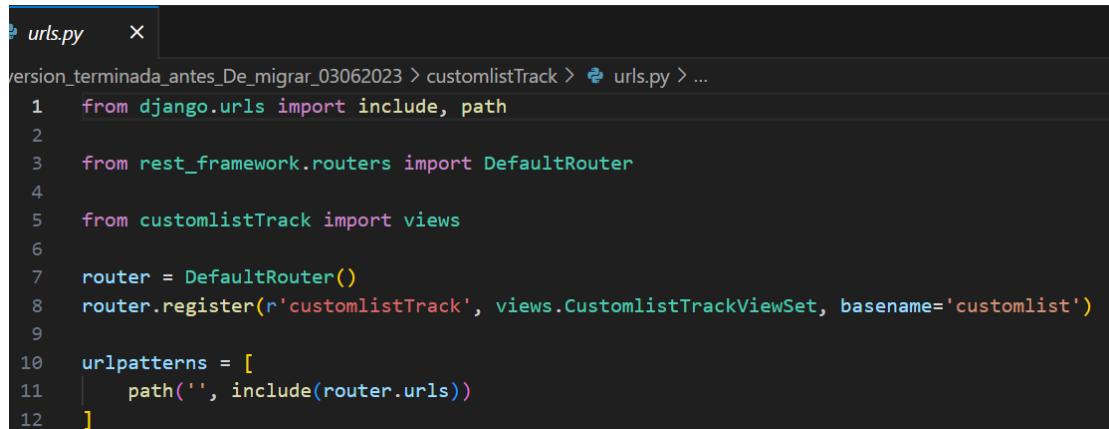
def create(self, request, *args, **kwargs):
    serializer = CustomlistTrackSerializer(data=request.data, context={"request": self.request})
    serializer.is_valid(raise_exception=True)
    customlisttrack = serializer.save()
    data = CustomlistTrackModelSerializer(customlisttrack).data
    return Response(data, status=status.HTTP_201_CREATED)

```

Figura 66. Creación customListTrack

- Urls.py

Por último se crea el archivo urls.py de la aplicación y se registran por un lado las rutas en el archivo de rutas del proyecto y por otro lado la aplicación en el archivo settings.py del proyecto.



```

urls.py  X
version_terminada_anter_De_migrar_03062023 > customlistTrack > urls.py > ...
1  from django.urls import include, path
2
3  from rest_framework.routers import DefaultRouter
4
5  from customlistTrack import views
6
7  router = DefaultRouter()
8  router.register(r'customlistTrack', views.CustomlistTrackViewSet, basename='customlist')
9
10 urlpatterns = [
11     path('', include(router.urls))
12 ]

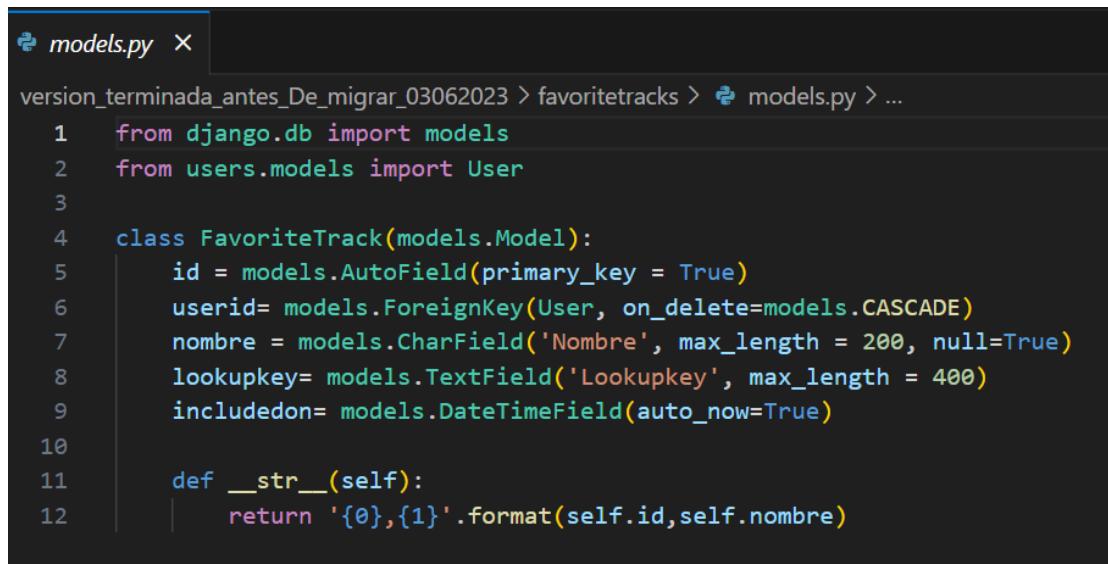
```

Figura 67.Urls customListTrack

4.FavoriteTracks

- Models.py

Esta aplicación almacena los datos nombre, lookupkey e includedon junto con la referencia al id de usuario



```

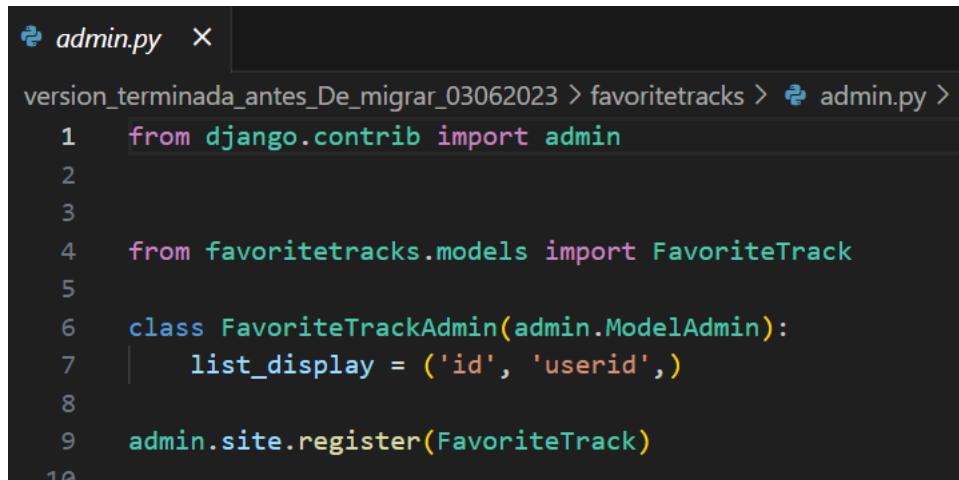
models.py  X
version_terminada_anter_De_migrar_03062023 > favoritetracks > models.py > ...
1  from django.db import models
2  from users.models import User
3
4  class FavoriteTrack(models.Model):
5      id = models.AutoField(primary_key = True)
6      userid= models.ForeignKey(User, on_delete=models.CASCADE)
7      nombre = models.CharField('Nombre', max_length = 200, null=True)
8      lookupkey= models.TextField('Lookupkey', max_length = 400)
9      includedon= models.DateTimeField(auto_now=True)
10
11     def __str__(self):
12         return '{0},{1}'.format(self.id,self.nombre)

```

Figura 68. Modelo favoriteTracks

- Admin.py

En el admin.py aparecerán las columnas id y userid



```

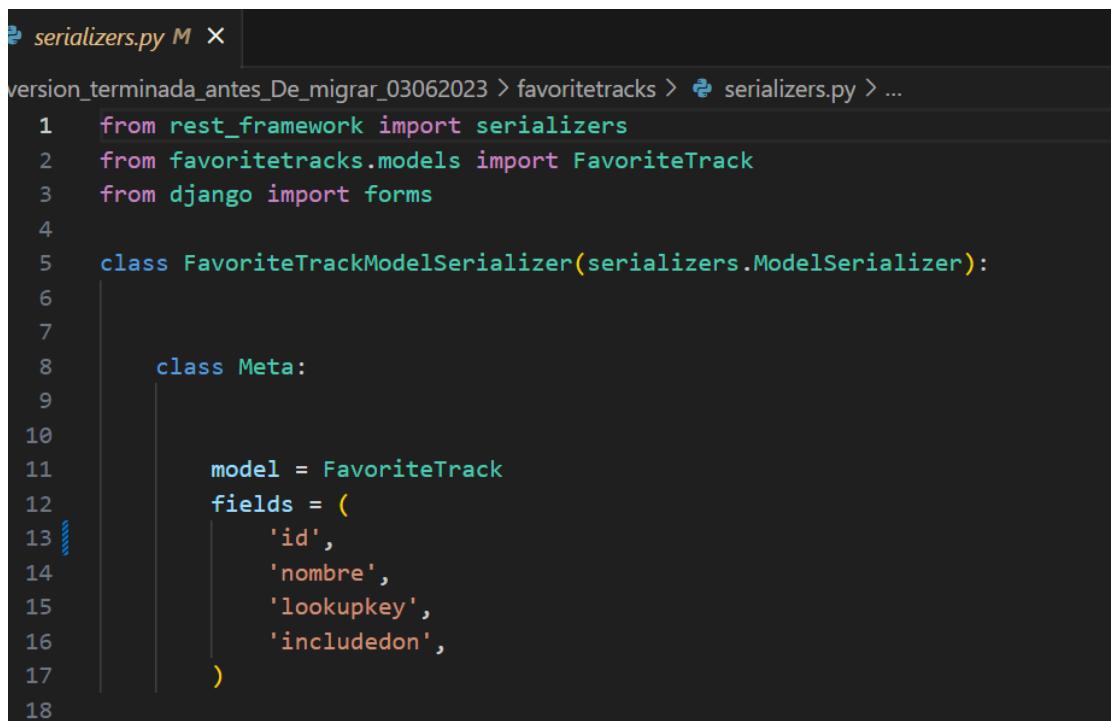
admin.py X
version_terminada_anter_De_migrar_03062023 > favoritetracks > admin.py > .
1  from django.contrib import admin
2
3
4  from favoritetracks.models import FavoriteTrack
5
6  class FavoriteTrackAdmin(admin.ModelAdmin):
7      list_display = ('id', 'userid',)
8
9  admin.site.register(FavoriteTrack)
10

```

Figura 69. Admin favoriteTracks

- Serializers.py

La clase FavoriteTrackModelSerializer devolverá todos los datos junto con la clave primaria(id)



```

serializers.py M X
version_terminada_anter_De_migrar_03062023 > favoritetracks > serializers.py > ...
1  from rest_framework import serializers
2  from favoritetracks.models import FavoriteTrack
3  from django import forms
4
5  class FavoriteTrackModelSerializer(serializers.ModelSerializer):
6
7
8      class Meta:
9
10         model = FavoriteTrack
11         fields = (
12             'id',
13             'nombre',
14             'lookupkey',
15             'includedon',
16         )
17
18

```

Figura 70. Modelo serializador favoriteTracks

- Views.py

La vista FavoriteTrackViewSet requerirá los permisos y si los datos son válidos creará una Favorite track en la tabla Favoritetracks

```

class FavoriteTrackViewSet(mixins.ListModelMixin,
                           mixins.CreateModelMixin,
                           mixins.UpdateModelMixin,
                           mixins.DestroyModelMixin,
                           viewsets.GenericViewSet):

    serializer_class = FavoriteTrackModelSerializer

    def get_permissions(self):
        permission_classes = [IsAuthenticated, IsStandardUser]
        return [permission() for permission in permission_classes]

    def get_queryset(self):
        queryset = FavoriteTrack.objects.filter(userid=self.request.user)
        return queryset

    def create(self, request, *args, **kwargs):
        serializer = FavoriteTrackSerializer(data=request.data, context={"request": self.request})
        serializer.is_valid(raise_exception=True)
        customlist = serializer.save()
        data = FavoriteTrackModelSerializer(customlist).data
        return Response(data, status=status.HTTP_201_CREATED)

```

Figura 71. Serializador favoriteTracks

- Urls.py

Por último, como siempre se registran las urls en la aplicación y en el proyecto

```

urls.py      X
version_terminada_anterior_a_03062023 > favoritetracks > urls.py > ...
1  from django.urls import include, path
2
3  from rest_framework.routers import DefaultRouter
4
5  from favoritetracks import views
6
7  router = DefaultRouter()
8  router.register(r'favoritetrack', views.FavoriteTrackViewSet, basename='favoritetrack')
9
10 urlpatterns = [
11     path('', include(router.urls))
12 ]

```

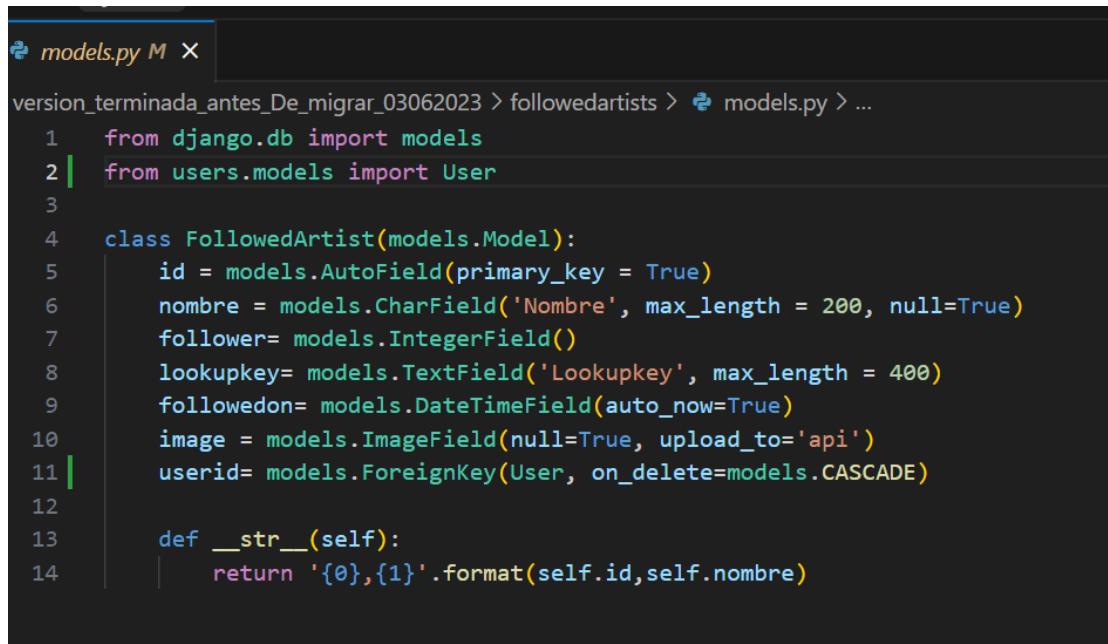
Figura 72.Urls favoriteTracks

y la aplicación en el archivo settings.py del proyecto.

2.Followedartists

- Models.py

Esta última aplicación tiene los campos nombre, follower, lookupkey,followedon,image(opcional) y userid(fk)



```

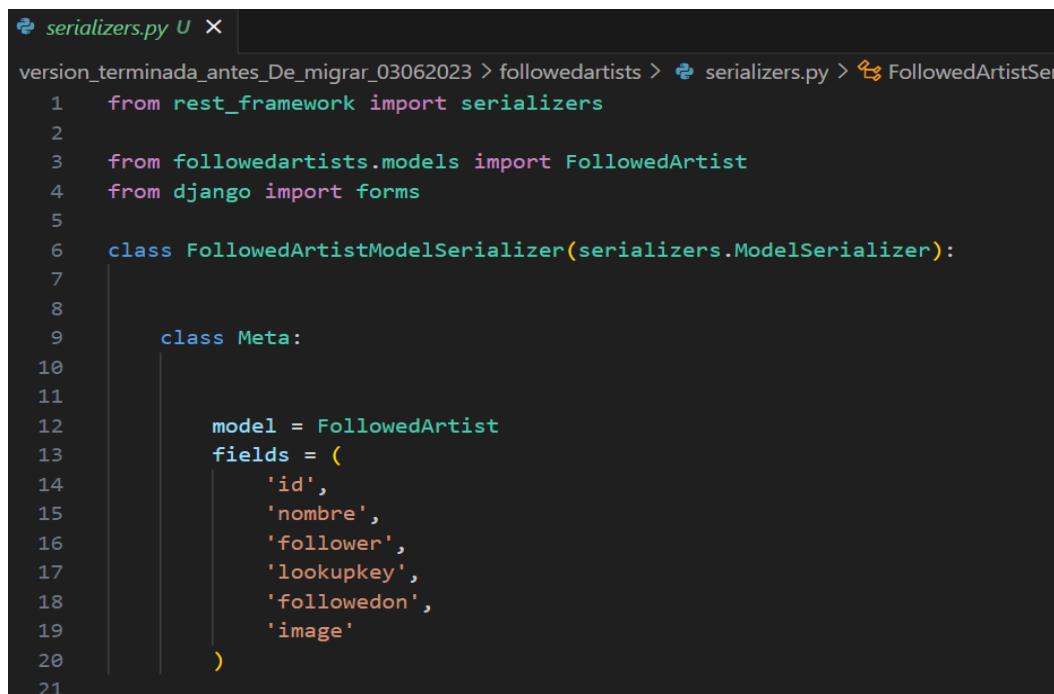
models.py M X
version_terminada_anterior_a_03062023 > followedartists > models.py > ...
1  from django.db import models
2  from users.models import User
3
4  class FollowedArtist(models.Model):
5      id = models.AutoField(primary_key = True)
6      nombre = models.CharField('Nombre', max_length = 200, null=True)
7      follower= models.IntegerField()
8      lookupkey= models.TextField('Lookupkey', max_length = 400)
9      followedon= models.DateTimeField(auto_now=True)
10     image = models.ImageField(null=True, upload_to='api')
11     userid= models.ForeignKey(User, on_delete=models.CASCADE)
12
13     def __str__(self):
14         return '{0},{1}'.format(self.id,self.nombre)

```

Figura 73. Modelo followedArtists

- Serializers.py

El modelo serializador devuelve todos los datos menos la foreign key y el serializador pedirá todos los datos menos el id que se genera sólo y el id de usuario que se generará por defecto con el id del usuario que esté logueado.



```

serializers.py U X
version_terminada_anterior_a_03062023 > followedartists > serializers.py > FollowedArtistSer
1  from rest_framework import serializers
2
3  from followedartists.models import FollowedArtist
4  from django import forms
5
6  class FollowedArtistModelSerializer(serializers.ModelSerializer):
7
8
9      class Meta:
10
11          model = FollowedArtist
12          fields = (
13              'id',
14              'nombre',
15              'follower',
16              'lookupkey',
17              'followedon',
18              'image'
19          )
20
21

```

Figura 74. Modelo serializador followedArtists

```

class FollowedArtistSerializer(serializers.Serializer):

    userid = serializers.HiddenField(default=serializers.CurrentUserDefault())
    nombre = serializers.CharField(max_length=250)
    follower = serializers.IntegerField(allow_null=True)
    lookupkey = serializers.CharField(allow_null=True)
    image = serializers.ImageField(max_length=None, use_url=True, allow_null=True, required=False)
    followedon = serializers.DateTimeField()

    def create(self, data):

        try:
            followed = FollowedArtist.objects.get(nombre=data['nombre'],userid=data['userid'])
        except:
            f = FollowedArtist.objects.create(**data)
            return f
        raise forms.ValidationError(u'FollowedArtist "%s" already following' % followed)

```

Figura 75. Serializador followedArtists

Se valida que no exista previamente el artista añadido a la lista de favoritos del usuario y si no existe devuelve todos los datos en un diccionario

- Views.py

Las vistas hacen uso de los permisos por defecto de django y de los personalizados en la aplicación users y para crear la vista hace una validación previa y después devuelve los datos con un código 200.

```

class FollowedArtistViewSet(mixins.ListModelMixin,
                            mixins.CreateModelMixin,
                            mixins.UpdateModelMixin,
                            mixins.DestroyModelMixin,
                            viewsets.GenericViewSet):

    serializer_class = FollowedArtistModelSerializer

    def get_permissions(self):
        permission_classes = [IsAuthenticated, IsStandardUser]
        return [permission() for permission in permission_classes]

    def get_queryset(self):
        queryset = FollowedArtist.objects.filter(userid=self.request.user)
        return queryset

    def create(self, request, *args, **kwargs):
        serializer = FollowedArtistSerializer(data=request.data, context={"request": self.request})
        serializer.is_valid(raise_exception=True)
        customlist = serializer.save()
        data = FollowedArtistModelSerializer(customlist).data
        return Response(data, status=status.HTTP_201_CREATED)

```

Figura 76. Vistas followedArtists

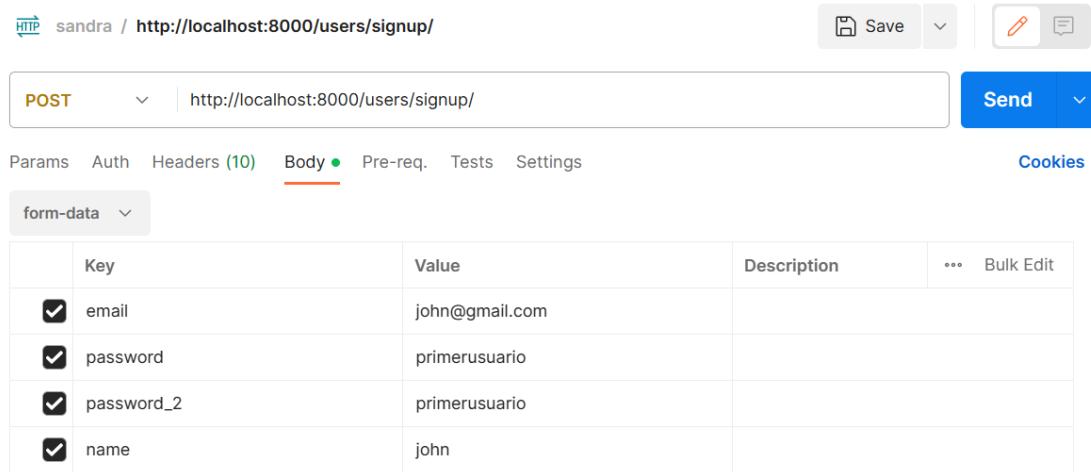
El penúltimo paso fue el registro de las urls en la aplicación y en el proyecto y por último registramos la aplicación en el archivo settings.py del proyecto

Verificación del funcionamiento

Para probar el funcionamiento de todas las urls se hace uso de la plataforma Postman, que permite introducir todas las API endpoint y ver en la consola las acciones derivadas de ellas.

Se ejecutaron las pruebas para todas las rutas especificadas en las aplicaciones, pero en este apartado sólo se reflejarán tres de ellas ya que el resto fueron realizadas en el apartado de tests.

1. Registro de usuarios

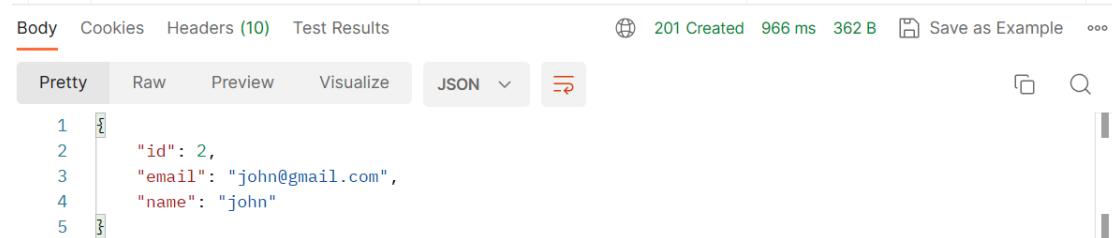


The screenshot shows the Postman interface with a POST request to `http://localhost:8000/users/signup/`. The 'Body' tab is selected, showing a table with four rows of form-data:

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> email	john@gmail.com			
<input checked="" type="checkbox"/> password	primerusuario			
<input checked="" type="checkbox"/> password_2	primerusuario			
<input checked="" type="checkbox"/> name	john			

Figura 77. Prueba registro de usuarios.

Para el registro, al acceder por POST a `http://localhost:8000/users/signup/` y pasarle los datos obligatorios email, name, password y password_2, aparecen en pantalla los datos del usuario en el formato especificado(en este caso json)



The screenshot shows the Postman interface with the JSON response of the user registration POST request. The response is a JSON object with the following structure:

```

1  {
2   "id": 2,
3   "email": "john@gmail.com",
4   "name": "john"
5 }

```

Figura 78. Resultado prueba de registro

2. Login

HTTP sandra / http://localhost:8000/users/login/

POST http://localhost:8000/users/login/

Params Auth Headers (9) Body **Pre-req.** Tests Settings Cookies

form-data

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> email	john@gmail.com			
<input checked="" type="checkbox"/> password	primerusuario			

Figura 79. Prueba login

Para el inicio de sesión a través de la url `http://localhost:8000/users/login/` se pasan por POST los campos `email` y `password`, y si las credenciales son correctas devuelve los datos de usuario y el `access_token`

Body Cookies Headers (10) Test Results

201 Created 705 ms 430 B Save as Example

Pretty Raw Preview Visualize JSON

```

1  {
2   "user": {
3     "id": 2,
4     "email": "john@gmail.com",
5     "name": "john"
6   },
7   "access_token": "76c5e25fc3903e2e45ecbc0b707983dc58fb8a3a"
8 }
```

Figura 80. Resultado login

En este momento aparece un token asociado a ese usuario en la tabla tokens que podemos visualizar en el panel de administración

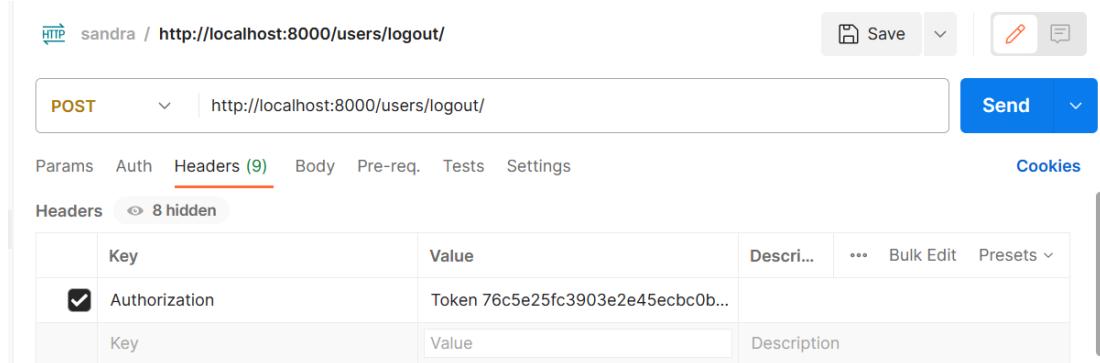
Seleccione token a modificar

CLAVE	USUARIO
76c5e25fc3903e2e45ecbc0b707983dc58fb8a3a	john@gmail.com
1 token	

Figura 81. Token login

3.Logout

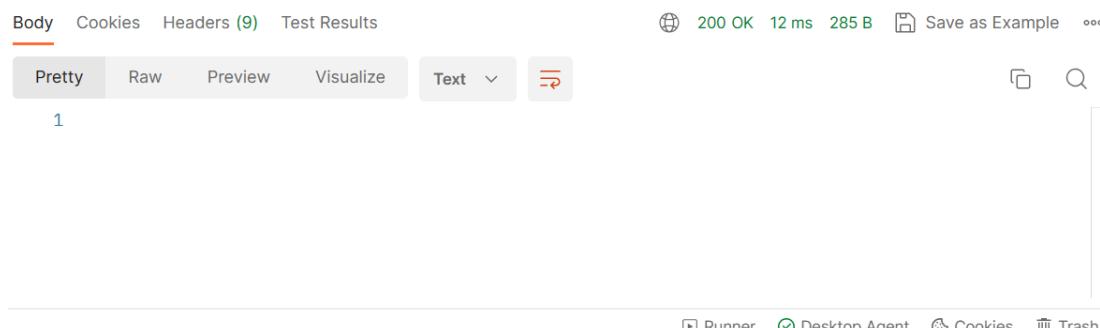
Para cerrar la sesión de usuario hay que pasarle por cabecera a `http://localhost:8000/users/logout/` la variable `Authentication` en la que especificamos Token seguido de un espacio y el token asociado al usuario que quiere cerrar sesión.



The screenshot shows the Postman interface. The URL is `http://localhost:8000/users/logout/`. The method is set to `POST`. The `Headers` tab is selected, showing a table with one row: `Authorization` with value `Token 76c5e25fc3903e2e45ecbc0b...`. Other tabs like `Params`, `Auth`, `Body`, `Pre-req.`, `Tests`, and `Settings` are visible. A `Send` button is at the top right.

Figura 82. Logout usuario

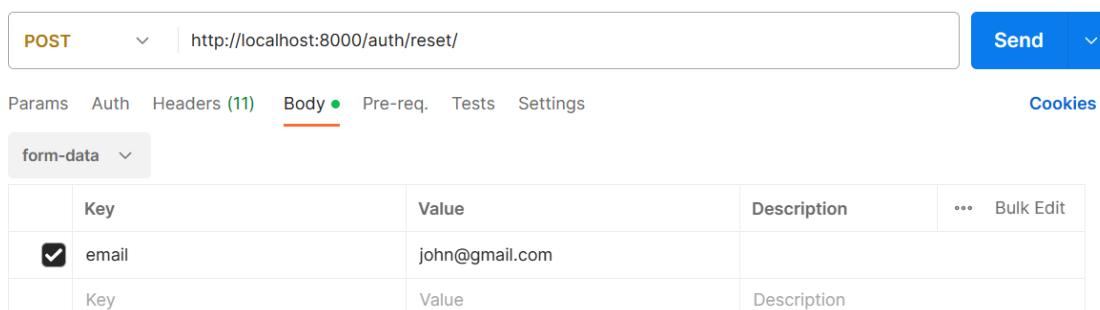
Si se introduce correctamente el token aparecerá el código 200 indicando que el usuario ha cerrado sesión, la cual se elimina en la variable `request`.



The screenshot shows the Postman interface after sending the request. The status is `200 OK`, time is `12 ms`, and size is `285 B`. The `Headers` tab is selected, showing a table with one row: `Content-Type` with value `text/html; charset=UTF-8`. Other tabs like `Body`, `Cookies`, `Test Results`, and `Text` are visible. A `Save as Example` button is at the top right.

Figura 83. Resultado logout

4. Reseteo de contraseña



The screenshot shows the Postman interface. The URL is `http://localhost:8000/auth/reset/`. The method is set to `POST`. The `Body` tab is selected, showing a `form-data` table with one row: `email` with value `john@gmail.com`. Other tabs like `Params`, `Auth`, `Headers`, `Pre-req.`, `Tests`, and `Settings` are visible. A `Send` button is at the top right.

Figura 84. Password reset

el usuario introducirá por POST en `http://localhost:8000/auth/reset/` el email de la contraseña que quiere recuperar y si el email existe aparece un mensaje en la consola con el método 200(ok)



The screenshot shows a REST API tool interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (10)', and 'Test Results'. On the right, it displays a status of '200 OK' with '30 ms' and '325 B' response times, along with a 'Save as Example' button. Below this, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The JSON response is shown in a tree view with the following content:

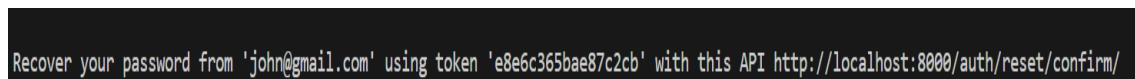
```

1
2   "status": "OK"
3

```

Figura 85. Resultado password reset

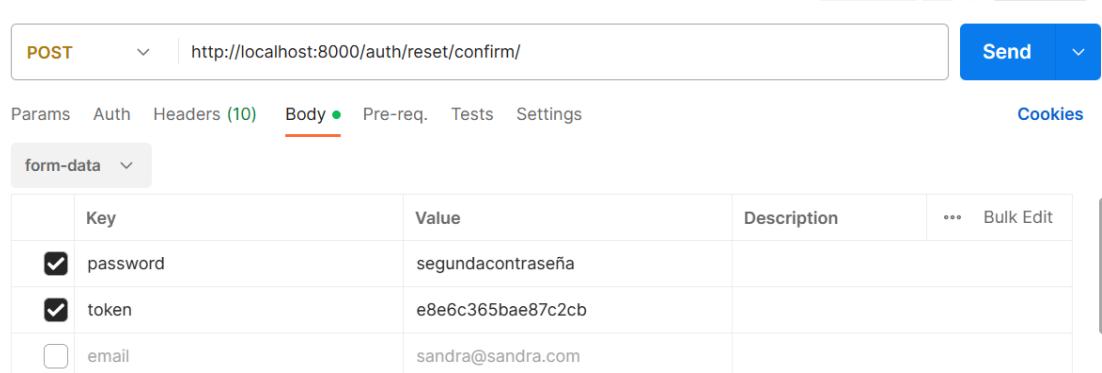
seguido del mensaje que especificamos en el método print que es el que visualizará el usuario una vez hecha la integración con la parte de frontend.



Recover your password from 'john@gmail.com' using token 'e8e6c365bae87c2cb' with this API <http://localhost:8000/auth/reset/confirm/>

Figura 86. Mensaje consola password reset

Si este usuario accede a <http://localhost:8000/auth/reset/confirm/> e introduce la nueva contraseña junto con el token que le aparece en el mensaje

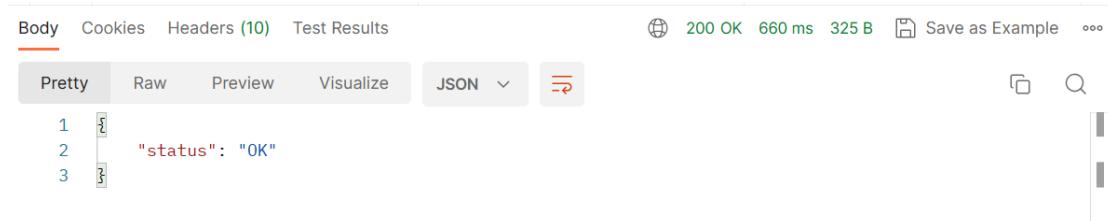


The screenshot shows a REST API tool interface. A 'POST' request is selected, and the URL is 'http://localhost:8000/auth/reset/confirm/'. Below the URL, there are tabs for 'Params', 'Auth', 'Headers (10)', 'Body' (which is selected and highlighted in green), 'Pre-req.', 'Tests', and 'Settings'. The 'Body' section is set to 'form-data' and contains the following data:

	Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/>	password	segundacontraseña			
<input checked="" type="checkbox"/>	token	e8e6c365bae87c2cb			
<input type="checkbox"/>	email	sandra@sandra.com			

Figura 87. Confirmación cambio contraseña

En caso de que la contraseña cumpla con los requisitos de validaciones y el token sea válido se efectuará el cambio de la contraseña:



The screenshot shows a REST API tool interface. At the top, there are tabs for 'Body', 'Cookies', 'Headers (10)', and 'Test Results'. On the right, it displays a status of '200 OK' with '660 ms' and '325 B' response times, along with a 'Save as Example' button. Below this, there are buttons for 'Pretty', 'Raw', 'Preview', 'Visualize', and 'JSON'. The JSON response is shown in a tree view with the following content:

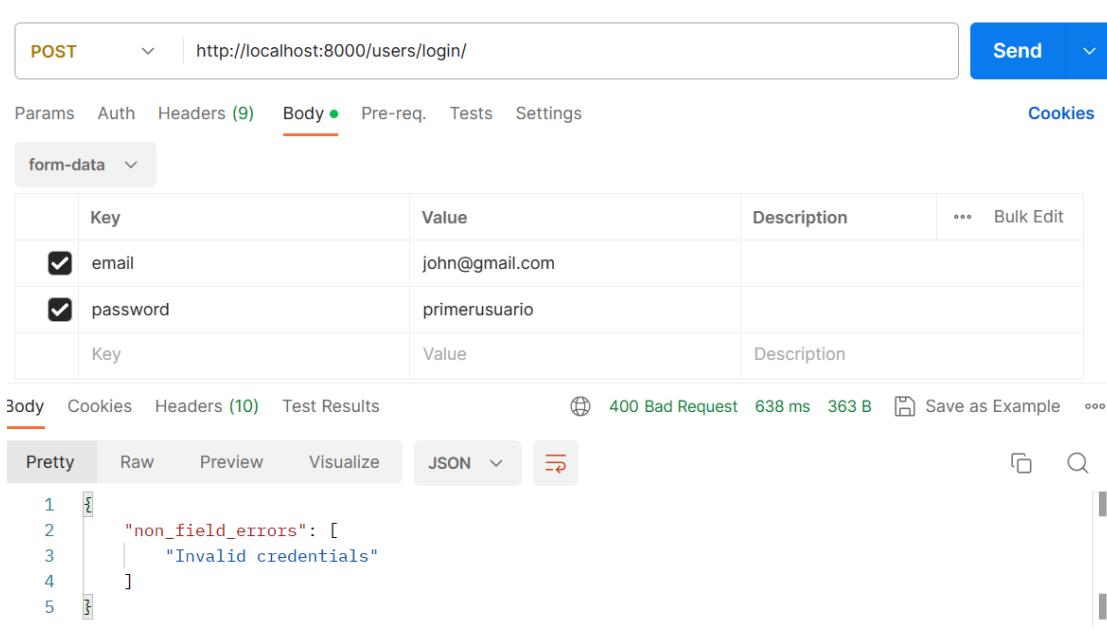
```

1
2   "status": "OK"
3

```

Figura 88. Resultado cambio contraseña

para comprobarlo, se procede a hacer login con ese usuario y la primera contraseña y aparece el error que definimos en la función login



POST <http://localhost:8000/users/login/> Send

Params Auth Headers (9) **Body** Pre-req. Tests Settings Cookies

form-data

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> email	john@gmail.com			
<input checked="" type="checkbox"/> password	primerusuario			
Key	Value	Description		

Body Cookies Headers (10) Test Results 400 Bad Request 638 ms 363 B Save as Example

Pretty Raw Preview Visualize JSON

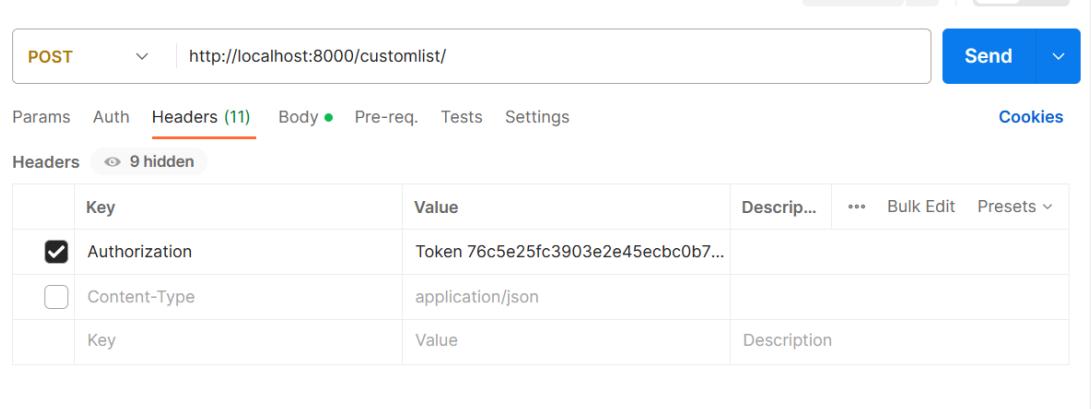
```

1  {
2   "non_field_errors": [
3     "Invalid credentials"
4   ]
5 }
```

Figura 89. Comprobación cambio contraseña

5. Creación de customlists

Para la creación de una lista de usuario a través de <http://localhost:8000/customlist/> necesitamos pasarle como cabecera la clave `authorization` junto con el token del usuario.



POST <http://localhost:8000/customlist/> Send

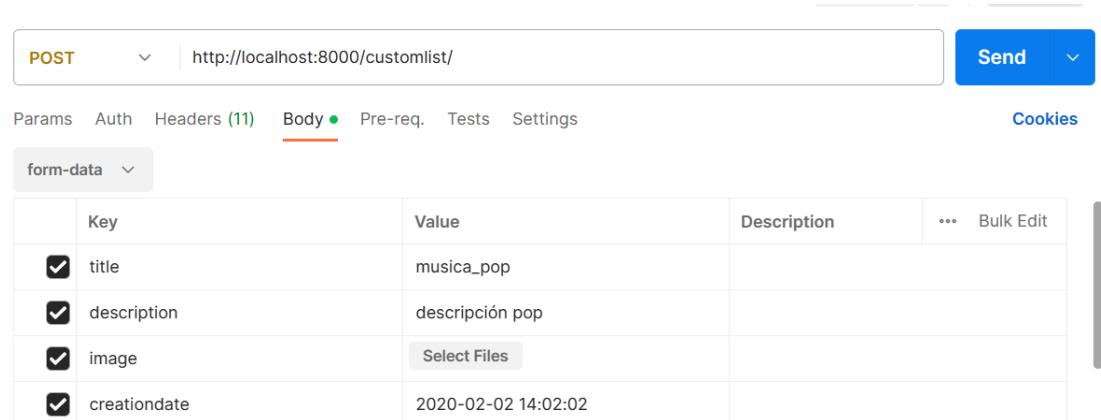
Params Auth Headers (11) Body Pre-req. Tests Settings Cookies

Headers 9 hidden

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Authorization	Token 76c5e25fc3903e2e45ecbc0b7...				
<input type="checkbox"/> Content-Type	application/json				
Key	Value	Description			

Figura 90. Creación customList

como cuerpo de la petición estarán los datos requeridos:



POST <http://localhost:8000/customlist/> Send

Params Auth Headers (11) **Body** Pre-req. Tests Settings Cookies

form-data

Key	Value	Description	...	Bulk Edit
<input checked="" type="checkbox"/> title	musica_pop			
<input checked="" type="checkbox"/> description	descripción pop			
<input checked="" type="checkbox"/> image	Select Files			
<input checked="" type="checkbox"/> creationdate	2020-02-02 14:02:02			

Figura 91. Cuerpo creación customList

y tras validar los campos se devolverán los datos en el formato elegido



Body Cookies Headers (10) Test Results

201 Created 28 ms 447 B Save as Example

Pretty Raw Preview Visualize JSON

```

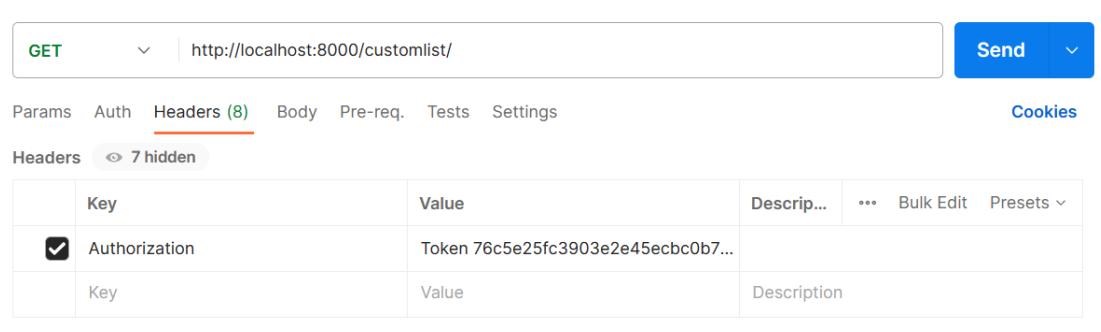
1
2   "id": 1,
3   "title": "musica_pop",
4   "description": "descripción pop",
5   "image": null,
6   "creationdate": "2023-06-11T10:03:26.298782Z"
7

```

Figura 92. Resultado creación customList

6. Customlists consulta

Con el método GET y especificando en la cabecera el token del usuario, el usuario accederá a <http://localhost:8000/customlist/>



GET <http://localhost:8000/customlist/> Send

Params Auth Headers (8) Body Pre-req. Tests Settings Cookies

Headers [7 hidden](#)

Key	Value	Description	...	Bulk Edit	Presets
<input checked="" type="checkbox"/> Authorization	Token 76c5e25fc3903e2e45ecbc0b7...				
Key	Value	Description			

Figura 93. Cabecera consulta customList

y si el token es válido devolverá los datos de todas sus listas

```

2   "count": 1,
3   "next": null,
4   "previous": null,
5   "results": [
6     {
7       "id": 1,
8       "title": "musica_pop",
9       "description": "descripción pop",
10      "image": null,
11      "creationdate": "2023-06-11T10:03:26.298782Z"
12    }
  
```

Figura 94. Resultado consulta customList

7.Borrado de Customlists

el id del customlist que se acaba de crear es 1, según se puede comprobar en el panel de administración.

Seleccione customlist a modificar

Acción: Ir seleccionados 0 de 1

CUSTOMLIST

1,musica_pop

1 customlist

Figura 95. Comprobación registro customList

para poder borrarla se hace de la siguiente manera:

DELETE Send

Params	Auth	Headers (8)	Body	Pre-req.	Tests	Settings	Cookies
		<input checked="" type="checkbox"/> Authorization Token 76c5e25fc3903e2e45ecbc0b...					

Figura 96. Borrado customList

accediendo con el método DEL a la url <http://localhost:8000/customlist/1> cuyo último parámetro es el id de la Customlist que se desea borrar, y pasándole como cabecera el token del usuario

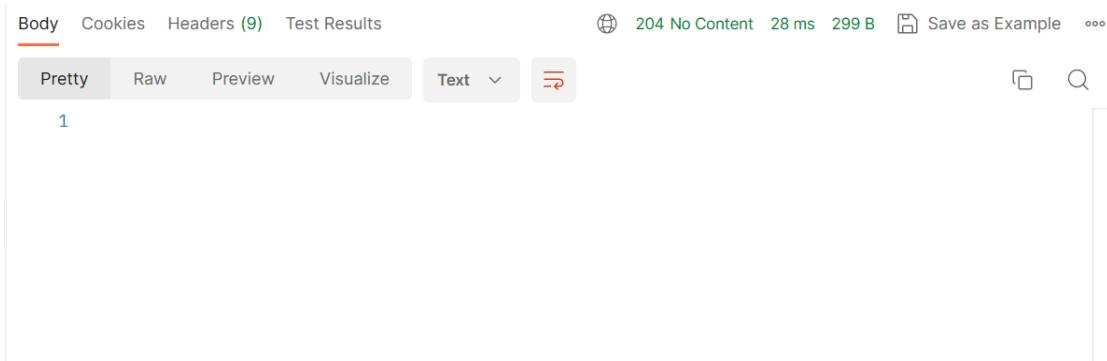


Figura 97. Resultado borrado customList

Se puede comprobar que se ha eliminado correctamente desde el panel de admin.

Seleccione customlist a modificar

0 customlists

Figura 98. Comprobación borrado customList

8.Creación de Followedartists

Para implementar la funcionalidad del botón de seguir a artistas, al acceder a <http://localhost:8000/followedartists/> pasándole como cabecera el token del usuario y como cuerpo los datos del artista que quiere seguir

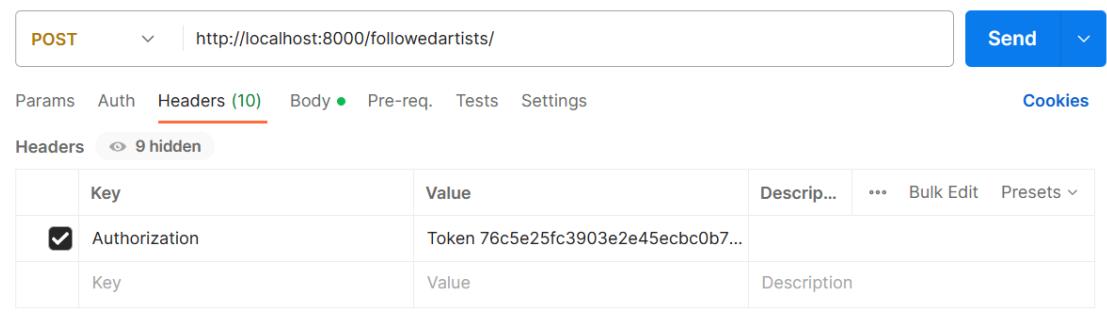


Figura 99. Cabecera creación followedArtists

Key	Value	Description	...	Bulk Edit
nombre	Crhistina Perri			
follower	2			
lookupkey	lookupkey			
followedon	2020-02-02 01:01:01			

Figura 100. Cuerpo creación followedArtists

se mostrarán todos los datos tanto obligatorios como opcionales

9. Followedartists consulta

En este caso basta con acceder a <http://localhost:8000/followedartists/> con el token del usuario y se mostrarán con éxito todos los artistas a los que sigue

Key	Value	Description	...	Bulk Edit	Presets
Authorization	Token 76c5e25fc3903e2e45ecbc0b7...				
Key	Value	Description			

Figura 101. Consulta followedArtists

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
  
```

```

{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": 1,
      "nombre": "Crhistina Perri",
      "follower": 2,
      "lookupkey": "lookupkey",
      "followedon": "2023-06-11T10:24:13.713773Z",
      "image": null
    }
  ]
}
  
```

Figura 102. Resultado consulta followedArtists

Adaptación del backend al frontend

- Peticiones CORS

Cuando se realizan peticiones de tipo REST entre orígenes y destinos existe una política de seguridad encargada de proteger este método de comunicación conocido como CORS(Cross Origin Resource Sharing) el cual se encarga de impedir el acceso o la comunicación entre terminales con diferente dominio y dirección IP. A continuación, se detallan los ajustes a realizar y sus correspondientes apartados:

1. Instalación de las cabeceras CORS en el backend, a través del comando **pip install django-cors-headers**
2. Ajustes de la aplicación `settings.py`.

Se registran las cabeceras recién instaladas en el apartado `INSTALLED_APPS`.

Después, en el apartado `MIDDLEWARE` se configura un middleware con preferencia, el cual se encargará de procesar las peticiones CORS.

```
'corsheaders.middleware.CorsMiddleware',
```

Figura 103. Registro middleware CORS

Se añaden los dominios que se quieran permitir, en nuestro caso el cliente 3000 y el servidor desplegado en Docker Luister-website

```
CORS_ORIGIN_WHITELIST = ["http://localhost:3000", "http://luister-website"]
```

Figura 104. Whitelist CORS

Se añade como orígenes o hosts permitidos, la IP del servidor y su correspondiente dominio.

```
ALLOWED_HOSTS = ['localhost', 'luister.es', '51.91.78.21']
```

Figura 105. Post permitidos CORS

En este mismo fichero, se procede a modificar la configuración por defecto de la base de datos, por la del servidor de bases de datos de producción de la aplicación. Se indica el motor a utilizar (Engine), nombre de la base de datos, el usuario, la contraseña, el host o dirección IP del servidor y el puerto de escucha.

```

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'luister',
        'USER': 'admin',
        'PASSWORD': '████████',
        'HOST': 'luister.es',
        'PORT': '3306'
    }
}

```

Figura 106. Base de datos settings mySQL

Por último, se añade un apartado para gestionar las credenciales de la sesión en las cabeceras de las peticiones CORS

```
CORS_ALLOW_CREDENTIALS = True
```

Figura 107. Credenciales de sesión CORS

- Models.py de todas las aplicaciones

Se añadió la clase meta en todos los modelos para especificarle a django que primero a la hora de hacer las migraciones de la bbdd no cree una tabla a partir del modelo definido(`managed=False`). Este parámetro va acompañado de `db_table` que sirve para indicarle que, debido a que no va a crear la tabla porque ya existe, simplemente establezca un vínculo con la tabla especificada entre comillas simples.

```

class Meta:
    managed = False
    db_table = 'users'

```

Figura 108. Clase Meta Modelos

- Models.py de customlistTrack

Dado que en esta aplicación existe una fk que a su vez referencia a otra, tuvimos que añadir `db_column='customlistid'`

```
customlistid= models.ForeignKey(Customlist, on_delete=models.CASCADE, db_column='customlistid')
```

Figura 109. dbcolumn customlistID

- Models.py de Followedartists

Se eliminó el campo follower y al userid se le cambió el nombre a follower porque es el propio follower el que actúa de fk

```
follower= models.ForeignKey(User, on_delete=models.CASCADE, db_column='follower')
```

Figura 110. FollowedArtists

- Dockerfile

Fue necesario añadir una línea con los CMD ya que es la encargada de ejecutar los comandos necesarios para, en este caso, habilitar el servicio principal del servidor(Python).

```
12
13     CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
14 |
```

Figura 111. CMD DockerFile

- Requirements.txt

A la hora de instalar django saltaba el siguiente error:

```
raise ImproperlyConfigured(
django.core.exceptions.ImproperlyConfigured: Error loading MySQLdb module.
Did you install mysqlclient?
```

Figura 112. Error instalación Django

así pues, tuvimos que añadir mysqlclient en el archivo requirements.

```
mysqlclient
```

Figura 113. Instalación mySQL en requirements.txt

Despliegue de la aplicación

Tras finalizar con la fase de programación de las diferentes aplicaciones que componen el sitio web final, se procede con su despliegue.

El despliegue es el proceso de puesta en marcha la aplicación en un entorno de producción, de forma que se vuelva accesible para los usuarios finales.

Preparación del entorno de despliegue

En este apartado se ajustará el sistema del servidor. Se instalarán todas las dependencias necesarias y se realizarán aquellas configuraciones necesarias para la puesta en marcha de la aplicación. Implica la instalación y configuración del servidor web, servidor de base de datos y demás recursos imprescindibles para el correcto funcionamiento de la aplicación.

Instalación y configuración del software para el entorno

El despliegue de la aplicación se realizará a través de la herramienta Docker, una plataforma de código abierto que permite la ejecución de aplicaciones en contenedores, los cuales son cápsulas con el código necesario para la ejecución del programa. Esta herramienta permite tener varios servidores ejecutándose en un LXC o contendor principal de forma eficiente, ya que prescinde de todos aquellos programas y elementos gráficos innecesarios para la aplicación, contenido únicamente los recursos relevantes. Esto ofrece diversas ventajas y disminuye los requisitos de hardware y software necesarios para el despliegue de la aplicación.

La implementación de Docker en el despliegue de aplicaciones, ofrece ciertas ventajas:

- **Portabilidad:** Con Docker es posible envolver una aplicación o servidor junto con todas sus dependencias en un contenedor autónomo y ligero (portabilidad, implementación y despliegue más sencillos).
- **Aislamiento:** Docker utiliza tecnologías de virtualización a nivel de sistema operativo para proporcionar un alto nivel de aislamiento entre contenedores (ejecución segura y con recursos propios de cada contenedor).
- **Escalabilidad y flexibilidad:** Docker facilita la escalabilidad horizontal de las aplicaciones al permitir la ejecución de múltiples contenedores en diferentes hosts o en un clúster de Docker (creación y destrucción flexible).
- **Reproducibilidad y consistencia:** Con Docker, se define el entorno de ejecución de una aplicación en un archivo llamado Dockerfile. Este archivo contiene todas las instrucciones necesarias para construir la imagen del contenedor, que es una instantánea de la aplicación y sus dependencias en un estado particular.
- **Rapidez y eficiencia:** Docker utiliza una arquitectura liviana y utiliza características del **kernel** (*núcleo del sistema encargado de mediar entre procesos de usuario y el hardware*) del sistema operativo subyacente para compartir recursos entre contenedores.

Instalación

Antes de instalar cualquier programa en un sistema basado en Linux, es necesario actualizar la lista de paquetes del sistema, instalar parches de seguridad y corrección de errores, y garantizar la disponibilidad de los paquetes necesarios (sincronización de dependencias).

\$ apt update (comando de instalación y verificación de paquetes y dependencias)

```
root@vps-a82e7669:/# apt update
Get:1 https://download.docker.com/linux/ubuntu jammy InRelease [48.9 kB]
Hit:2 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:3 http://nova.clouds.archive.ubuntu.com/ubuntu jammy InRelease
```

Figura 114. Actualización librería

Tras realizar esta operación se instalarán los paquetes **ca-certificates**, necesario para el uso de certificados de autoridades de certificación que demuestran la autenticidad de los sitios web y servicios en línea a través del certificado SSL/TL.

```
root@vps-a82e7669:/# apt-get install ca-certificates curl gnupg
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

Figura 115. Instalación certificados

Posteriormente, es necesario añadir las claves **GPG** (GNU Privacy Guard) de Docker para garantizar la integridad y autenticidad de los metadatos de Docker.

```
root@vps-a82e7669:/# install .m 0755 -d /etc/apt/keyrings
root@vps-a82e7669:/# curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
root@vps-a82e7669:/# chmod a+r /etc/apt/keyrings/docker.gpg
root@vps-a82e7669:/#
```

Figura 116. Instalación repositorio.

Se vincula al repositorio principal a través del protocolo HTTPS y se agrega el repositorio de Docker a la lista de fuentes de paquetes de APT (Advanced Package Tool) del sistema.

```
root@vps-a82e7669:/# echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
$(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
root@vps-a82e7669:/#
```

Figura 117. Ajuste repositorio

Con este comando se agrega la configuración del repositorio de Docker al fichero docker.list en el sistema, lo que permite que el administrador de paquetes **APT** acceda a las dependencias de Docker desde la URL proporcionada.

Finalizados los ajustes y configuraciones previa, se procede primero a actualizar y sincronizar los paquetes del sistema con el comando \$ sudo apt update.

Se da comienzo a la instalación de **Docker Engine**, el componente principal de Docker y mediante el cual se realizará la construcción, ejecución y administración de los contenedores a partir de imágenes de Docker. Se instala **containerd**, el componente subyacente de Docker Engine que proporciona una interfaz de bajo nivel para la administración y gestión de servidores; y **Docker compose**, que permite definir la configuración de una aplicación multi contenedor a través de un fichero YAML.

```
root@vps-a82e7669:/# apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  docker-ce-rootless-extras libltdl7 libslirp0 pigz slirp4netns
Suggested packages:
  aufs-tools cgroupfs-mount libcgroup-lite
The following NEWER packages will be installed:
  containerd.io docker-buildx-plugin docker-ce docker-ce-rootless-extras docker-compose-plugin libltdl7 libslirp0 pigz slirp4netns
  0 upgraded, 10 newly installed, 0 to remove and 0 not upgraded.
Need to get 111 MB of archives.
After this operation, 402 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
```

Figura 118. Instalación Docker

Como se puede observar en la imagen posterior, la herramienta ya se encuentra instalada.

```
root@vps-a82e7669:/# docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Figura 119. Test Docker

Transferencia de los archivos de la aplicación al servidor

El siguiente paso a llevar a cabo consiste en realizar una copia de todos los ficheros referentes a la aplicación en el servidor de despliegue. Existen diversas formas de realizar esta labor (scp, ftp, sftp). En este caso, ya se encuentran los ficheros en el servidor debido a que en éste, se encuentra clonado el repositorio de la aplicación.

```
root@luister:/projects/luister# tree -L 1
.
└── README.md
   └── backend
      └── frontend
         └── luister-db
```

Figura 120. Ficheros aplicación

Empaquetado de recursos

En esta fase se agruparán las distintas aplicaciones a desplegar en formatos coherentes, portátiles y transferibles a diferentes entornos para su despliegue. Se crearán los ficheros **Dockerfile**, a partir de los cuales se crearán las imágenes de cuya ejecución se originarán los contenedores de aplicaciones.

Creación de fichero Dockerfile para la aplicación cliente

Los archivos **Dockerfile** son ficheros de texto utilizados para configurar una imagen de Docker. A partir de las imágenes, se crearán los servidores en los que se encontrarán en ejecución las aplicaciones y servicios. Es un componente fundamental en el despliegue de aplicaciones en Docker, y consta de un conjunto de instrucciones que se detallarán posteriormente.

Se recomienda crear estos ficheros en la raíz o directorio padre/madre del proyecto o aplicación a desplegar.

```
root@luister:/tmp# ll luister/ | grep -ie docker
-rw-r--r--  1 root root    79 May 28 19:09 .dockerignore
-rw-r--r--  1 root root  237 May 28 19:10 Dockerfile
root@luister:/tmp#
```

Figura 121. Ficheros Docker

Se definen las directivas y configuraciones necesarias para el correcto funcionamiento del contenedor de la aplicación cliente.

```
FROM node:14-alpine as build-step
WORKDIR /app/website
COPY package.json .
RUN npm install
COPY . .
RUN npm run build --prod
FROM nginx:latest
COPY --from=build-step /app/website/dist/luister /usr/share/nginx/html
COPY nginx.conf /etc/nginx/
COPY default.conf /etc/nginx/conf.d/
COPY ./certs/* /usr/share/nginx/certificates/
CMD ["nginx", "-g", "daemon off;"]
```

Figura 122. Fichero DockerFile

A continuación, un breve detalle de cada una de las instrucciones:

- **FROM:** Especifica la imagen base a partir de la cual se construirá el sistema. En este caso, se hará uso de una imagen de **Node**, ya que la aplicación está desarrollada con el Framework Angular, el cual hace uso de **node** y el gestor de paquetes **npm**.
- **WORKDIR:** Especifica el directorio de trabajo de la aplicación. En este caso, se hará uso de la ruta **/app/website** creada para la aplicación cliente.
- **COPY:** Copia ficheros y directorios desde el sistema de archivos local al contenedor. En este caso, se copia el fichero **package.json** de la aplicación de angular, al directorio del contenedor.
- **RUN:** Con esta instrucción, se ejecuta un comando en dentro del contenedor durante el proceso de creación de la imagen. En la cuarta instrucción, se encarga de ejecutar el comando **npm install**, mediante el cual se sincroniza las dependencias de la aplicación y se instalan los recursos necesarios.
- En la quinta instrucción, nuevamente se hace uso del **COPY** para copiar todos los ficheros del proyecto, al directorio de trabajo.
- En la sexta instrucción, se realiza el empaquetado de la aplicación mediante la sentencia **RUN npm run build --prod**. El resultado de este comando es un directorio de nombre **/dist/** que vendría a ser la compilación de la aplicación. El despliegue de la aplicación se realizará mediante este último recurso.
- **En la séptima sentencia** se hace uso de la instrucción **FROM**, para indicar que se hará uso de una imagen adicional para ejecutar las instrucciones posteriores. En este caso, se hace uso de la imagen de **nginx**, un servidor web ligero de alto rendimiento, proxy inverso y servidor de correo electrónico, ya que será la encargada de poner en marcha la aplicación cliente.
- **En la octava sentencia**, se hace uso de la instrucción **COPY**, para copiar los ficheros generados en la imagen anterior, resultado de la compilación, en el directorio de **nginx** reservado para los recursos, HTML, CSS, JS y demás, necesarios para el correcto funcionamiento de la aplicación web cliente.
- **En la novena sentencia**, se copia el fichero personalizado de configuración de **nginx** al directorio de la aplicación, para que ésta funcione con las configuraciones establecidas en dicho fichero y con los ajustes por defecto. Más adelante se mostrarán los detalles designados en este fichero de configuración.
- **En la décima sentencia**, se copian los certificados SSL de la aplicación, a la ruta del contenedor, desde la cual se hará uso de dichos ficheros. Éstos son necesarios para poder acceder al servidor mediante el protocolo HTTPS.
- **CMD:** Mediante esta instrucción, se especifica el comando a ejecutarse al iniciarse el contenedor. En este caso, se ejecuta el comando **nginx** para poner en marcha el servidor.

Antes dar inicio a la construcción de la imagen, se crearán todos aquellos recursos y ficheros de los que hará uso a la hora de construir el contenedor.

Se crea el fichero **nginx.conf** y **default.conf**, los cuales se copiarán al directorio de configuración de **nginx** para aplicar la configuración deseada al contenedor, lo cual incluye el proxy inverso para evitar el bloqueo, por **CORS (Cross-Origin Resource Sharing)**, de las peticiones de la aplicación cliente a las **APIs** externas a las que necesita acceder.

La configuración del fichero **default.conf** se encuentra incluida en una directiva **server**, ya que este fichero luego sería incluido en el archivo principal de configuración de nginx **/etc/nginx/nginx.conf**.

Estructura del fichero nginx.conf

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 75 de 113
--------------------------	--------------------	------------------

```

user    nginx;
worker_processes  auto;

error_log  /var/log/nginx/website.log  notice;
pid      /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/website.log  main;

    sendfile      on;
    #tcp_nopush    on;

    keepalive_timeout  65;
    #gzip  on;
    include /etc/nginx/conf.d/*.conf;
}

```

Figura 123. Fichero NginX

- **user nginx:** Se especifica que el usuario encargado de ejecutar el proceso principal de NGINX es el usuario del mismo nombre.
- **worker_processes auto:** Número de procesos de trabajo definidos para manejar las peticiones. El valor auto permite que NGINX ajuste el número de procesos en función de la cantidad de núcleos de la unidad central de procesamiento.
- **error_log /var/log/nginx/website.log notice:** Establece la ruta y el nivel de registro de errores de NGINX.
- **pid /var/run/nginx.pid:** Especifica la ruta donde se almacenará el archivo PID (identificador de proceso) de NGINX.
- **events:** Ajustes referente a los eventos en NGINX, como las conexiones de los clientes.
- **worker_connections 1024:** Número de conexiones simultáneas que un proceso de trabajo puede manejar.
- **http:** Define las configuraciones principales para el protocolo HTTP.
- **include /etc/nginx/mime.types:** Incluye el archivo de tipos MIME predefinidos en la configuración de NGINX. Este archivo proporciona mapeos entre extensiones de archivo y tipos MIME.
- **default_type application/octet-stream:** Establece el tipo MIME predeterminado para las respuestas sin tipo MIME especificado.
- **log_format main:** Define un formato de registro personalizado llamado "main". Esta directiva especifica cómo se registrará la información en los registros de acceso.
- **access_log /var/log/nginx/website.log main:** Configura el registro de acceso de NGINX y especifica que se utilizará el formato "main". Los registros de acceso se guardarán en el archivo /var/log/nginx/website.log.
- **sendfile on:** Activa la funcionalidad de envío de archivos de NGINX, lo que permite la transmisión eficiente de archivos estáticos.
- **keepalive_timeout 65:** Establece el tiempo máximo que una conexión se mantendrá abierta para permitir solicitudes posteriores (keep-alive).
- **include /etc/nginx/conf.d/*.conf:** Incluye archivos de configuración adicionales ubicados en el directorio /etc/nginx/conf.d/. Ofrece modularidad y la organización de la configuración en archivos separados.

Estructura del fichero default.conf

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 76 de 113
--------------------------	--------------------	------------------

En la primera línea posterior a la directiva anteriormente indicada, se definen los puertos por los que el servidor se encontrará a la escucha. Se definen los puertos **80** para el acceso mediante el protocolo **HTTP**, y el puerto **443** para el acceso mediante el protocolo **HTTPS**, el cual requiere de los certificados SSL anteriormente mencionados en la construcción de la imagen, y de los cuales se hablará a continuación.

```
server {
    listen 80;
    listen 443 ssl;
    server_name luister.com;

    ssl_certificate /usr/share/nginx/certificates/certificate.pem;
    ssl_certificate_key /usr/share/nginx/certificates/key.pem;

    location /api1 {
        rewrite ^/api1/(.*)$ /$1 break;
        proxy_pass https://accounts.spotify.com;
        proxy_set_header Host accounts.spotify.com;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location /api2 {
        rewrite ^/api2/(.*)$ /$1 break;
        proxy_pass https://www.deezer.com/uk/;
        proxy_set_header Host www.deezer.com;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location / {
        root /usr/share/nginx/html;
        index index.html;
        try_files $uri $uri/ /index.html;
    }
}
```

Figura 124. Fichero de configuración de proxy

En las siguientes líneas, se definen los directorios en los que se encuentra el certificado y la clave SSL necesario para el acceso al sitio web mediante el protocolo HTTPS.

Posteriormente, se definirá en la directiva **location** un conjunto de valores que vendrán a determinar el comportamiento de la aplicación en esta ruta. Mediante esta misma directiva, es posible configurar el proxy inverso. Algunas de las directivas que se pueden definir en este apartado son: **proxy_pass**, **try_files**, **rewrite**, **alias**, entre otros.

A continuación, se especificará las diferentes rutas y la configuración a aplicarse en cada una.

En la localización **/api1** y **/api1/** se definirá la instrucción **rewrite** para que, en el proxy inverso, se elimine el prefijo de la ruta, de tal forma que no se acceda a un recurso inexistente al realizar la redirección.

Se incluye la instrucción **proxy_pass** para indicar la URL de la API a la que se realizará la redirección al acceder a la ruta especificada. Al ser un proxy inverso, es necesario definir las cabeceras e indicar el host y definir la dirección IP de origen.

El siguiente paso en la creación del fichero **Dockerfile** consiste en añadir al directorio **certs/** el cual se encuentra en la raíz del proyecto, los correspondientes certificados SSL para evitar un error en la creación del contenedor al tratar de acceder a ficheros inexistentes.

```
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/frontend/luister# ls -lshrt certs/
total 8.0K
4.0K -rw----- 1 root root 1.7K Jun 1 23:31 key.pem
4.0K -rw-r--r-- 1 root root 1.4K Jun 1 23:31 certificate.pem
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/frontend/luister#
```

Figura 125. Certificados

```
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/frontend/luister# ls
node_modules
dist
tmp
logs
*.log
npm-debug.log*
.dockerignore
.git
.gitignore
karma.conf.js
protractor.conf.js
e2e/
.env
.env.*
.env.local
.env.development
.env.test
.env.staging
.env.production
.vscode
.idea
*.sublime-project
*.sublime-workspace
*.swp
*~
.DS_Store
```

Figura 126. DockerIgnore

Se define un fichero **dockerignore** para la aplicación, de manera que se proceda a ignorar los módulos y elementos innecesarios en la construcción de la imagen.

Con todos los recursos necesarios preparados, se procede con la creación de la imagen de la aplicación cliente. Para ello es necesario situarse en el directorio de la aplicación. Lugar en el que se encuentra el fichero **Dockerfile** y ejecutar la sentencia de construcción.

docker build --no-cache --progress=plain -t luister-website-image .

El comando docker build se encarga de generar la imagen, con la opción –no-cache se indica para evitar que se haga uso de la cache para cargar capas de imágenes en construcciones anteriores. Por otro lado, el parámetro –progress=plain se incluye de manera que se pueda ver los pasos que conforman la construcción de la imagen, lo cual resulta útil a la hora de observar posibles fallos en el proceso. Adicionalmente, resulta necesaria indicar el nombre de la imagen mediante el parámetro -t, el cual junto a la sentencia docker build, es obligatorio. Y, por último, se especifica la dirección

del fichero a partir del cual se construye la imagen. Como en este caso se encuentra en el mismo directorio, se especifica un punto (“.”).

```
root@luister:/home/ubuntu/producción/LUISTER_PROYECTO/frontend/luister# docker build --no-cache --progress=plain -t luister-website-image .
#1 [internal] load .dockerignore
#1 transferring context: 2B done
#1 DONE 0.0s

#2 [internal] load build definition from Dockerfile
#2 transferring dockerfile: 392B 0.0s done
#2 DONE 0.0s

#3 [internal] load metadata for docker.io/library/node:14-alpine
#3 DONE 0.0s
```

Figura 127. Construcción de imagen de sitio web

Se podrá comprobar que la imagen se creó con la sentencia Docker images

```
root@luister:/home/ubuntu/producción/LUISTER_PROYECTO/frontend/luister# docker images | grep website
luister-website-image    latest    2f6a1607d645    9 minutes ago   152MB
root@luister:/home/ubuntu/producción/LUISTER_PROYECTO/frontend/luister#
```

Figura 128. Listado imagen

Creada la imagen, se procederá a ejecutar el contenedor de Docker. Pero antes, se creará una red para conectar futuros contenedores.

docker network create luister-network

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO# docker network create luister-network
74fb65fafaada503ba4dd3b2f1d9742669f4d58a9b3d53aa5a5bb8a9068a980f
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO# docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
93e671e399e    bridge    bridge      local
e339b4fef7a9   host      host       local
74fb65fafaad   luister-network  bridge      local
7f8f34a2feef   none      null      local
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO#
```

Figura 129. Imagen contenedor

Creada la red, se procede a poner en marcha el contenedor en la red especificada.

docker run -d --name luister-website -p 80:80 -p 443:443 --network luister-network luister-website-image

Mediante el comando Docker run, se pone en marcha el contenedor; el parámetro -d indica que éste se encontrará en ejecución en segundo plano; con -name se especifica el nombre del contenedor para usos posteriores; -p se utiliza para definir los puertos de escucha y locales del servidor, en este caso, el 80 y el 443; con --network se especifica la red a la que formará parte el contenedor, y, por último, se especifica el nombre de la imagen a partir de la cual se construye el contenedor.

El servidor de la aplicación cliente se encuentra en ejecución.

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/frontend/luister# docker run -d --name luister-website -p 80:80 -p 443:443 --network luister-network luister-website-image
f31fb85c691e866f77f43d58a39aeba468e34c7a0baccfb38fdd4e121c13b414
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/frontend/luister# docker ps -a | grep website
f31fb85c691e  luister-website-image  "/docker-entrypoint..."  13 seconds ago  Up 12 seconds  0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp, 0.0.0.0:443->443/tcp  luister-website
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/frontend/luister#
```

Figura 130. Ejecución contenedor

Creación del fichero **Dockerfile** para el servidor de bases de datos.

La aplicación web Luister dispone de una API propia o servidor **backend** encargado de ofrecer información referente a los usuarios de la plataforma, a través de su base de datos. Con el fin de que la aplicación pueda funcionar de forma correcta, será necesario crear el contenedor para la base de datos.

Se procede con la creación del fichero **Dockerfile** para el servidor de bases de datos.

- **FROM**: En esta sentencia se indica que la imagen se creará a partir de una imagen predefinida de MySQL en su versión 8.0.
- **ENV**: Se define en este caso, una variable de entorno la cual contiene la contraseña del usuario root.
- **COPY**: A través de esta sentencia, se copia el fichero de inicialización de la base de datos al directorio **/docker-entrypoint-initdb.d/**.
- **VOLUME**: Se crea un volumen para la ruta **/var/lib/mysql/** de manera que la información en dicho directorio, referente a la estructura y contenido de la base de datos, se conserve en un directorio del sistema de archivos local del servidor o host, con el fin de preservar los datos de la base de datos.
- **CMD**: Se ejecuta el comando **mysqld** para iniciar el servicio.

```
FROM mysql:8.0
ENV MYSQL_ROOT_PASSWORD=████████
COPY luister.sql /docker-entrypoint-initdb.d/
VOLUME /var/lib/mysql
CMD ["mysqld"]
```

Figura 131. Fichero Dockerfile BBDD

Antes de crear el contenedor, es necesario comprobar que todos los recursos que utilizará se encuentran disponibles. En este caso, que el fichero SQL referente a la inicialización de la base de datos, se encuentra disponible. Adicionalmente, se creará el directorio en el que se vinculará el volumen del contenedor de base de datos para el respaldo y persistencia de datos.

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/luister-db# ls -lhrt
total 16K
drwxr-xr-x 2 root root 4.0K Jun  3 07:35 data-backup
-rw-r--r-- 1 root root 134 Jun  3 07:35 Dockerfile
-rw-r--r-- 1 root root 4.7K Jun  3 14:45 luister.sql
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/luister-db#
```

Figura 132. Ficheros BBDD

Finalizada la verificación, se procede con la creación de la imagen para la base de datos de la aplicación.

```
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/luister-db# docker build --no-cache -t luister-db-image .
[+] Building 22.0s (4/6)
--> [internal] load build definition from Dockerfile
--> => transferring dockerfile: 173B
--> [internal] load .dockerignore
--> => transferring context: 2B
```

Figura 133. Construcción de imagen

Como se puede observar, la imagen fue creada de forma exitosa.

```
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/luister-db# docker images | grep db
luister-db-image      latest      7f78e492001c  About a minute ago  565MB
root@luister:/home/ubuntu/prod/LUISTER_PROYECTO/luister-db#
```

Figura 134. Listado imagen

A continuación, se pone en marcha el contenedor. En este caso, será necesario definir, mediante el parámetro `-v`, el directorio vinculado al volumen anteriormente especificado para almacenar la información de la base de datos en dicho directorio, así como incluir al contenedor a la red creada para la aplicación.

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/luister-db# docker run -d -p 3306:3306 --name luister-db -v $(pwd)/data-backup:/var/lib/mysql --network luister-network luister-db-image
```

Figura 135. Ejecución contenedor

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/luister-db# docker ps | grep db
a9f5695ac8d luister-db-image      "docker-entrypoint.s..." 2 hours ago      Up 2 hours      0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 33060/tcp
0
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/luister-db#
```

Figura 136. Vista contenedor

Para iniciar la base de datos, será necesario acceder al contenedor y realizar las configuraciones pertinentes, y ejecutar el script SQL anteriormente copiado, el cuál posee la estructura e información de la base de datos.

```
mysql> show tables;
+-----+
| Tables_in_luister |
+-----+
| customlists |
| customlisttracks |
| favoritetracks |
| followedartists |
| passwordresettokens |
| sessions |
| users |
| usersettings |
+-----+
8 rows in set (0.01 sec)
```

Figura 137. BBDD

Como se puede observar en la imagen superior, la base de datos se encuentra operativa y el servidor, listo para ofrecer la información solicitada.

Creación de fichero Dockerfile para la API interna.

Con los contenedores de la base de datos y de la aplicación clientes en ejecución, se procede con el despliegue de la API interna encargada de acceder a la base de datos de la aplicación principal. Se accede al directorio de la API para realizar los ajustes previos a la creación del fichero **Dockerfile** y la imagen para el contenedor.

Se crea el fichero **Dockerfile** especificando lo siguiente:

- **FROM:** Se parte de una imagen de Python en su versión 3 para la creación de la imagen ya que se trata de una aplicación desarrollada en Django Rest Framework, una herramienta de desarrollo cuyo lenguaje de programación base es Python.
- **ENV:** Se configura la variable **PYTHONUNBUFFERED** a 1. Esta variable se utiliza para configurar el comportamiento de almacenamiento en búfer de la salida estándar y la salida de error estándar en Python.
- **WORKDIR:** Se establece como directorio raíz o directorio de trabajo para la aplicación, a la ruta **/app/webapi/**.
- **COPY:** Se copia el fichero de requerimientos del sistema de archivos del servidor, al directorio principal de programa. Este fichero contiene el conjunto de dependencias a instalar para la correcta ejecución y funcionamiento del programa. Resulta útil a la hora de desarrollar aplicaciones con una gran cantidad de dependencias a instalarse.
- **RUN:** Con la siguiente sentencia, se procede a instalar todas las dependencias definidas en el fichero de requerimientos.
- **COPY:** En esta sentencia se copian todos los ficheros y directorios de la aplicación, al directorio de trabajo del contenedor. De esta forma, se podrá hacer uso de dichos ficheros a la hora de desplegar y ejecutar el programa.
- **CMD:** se encargade habilitar el servicio en la dirección IP especificada y a la escucha en el puerto que se proceda a indicar.

```
FROM python:3

ENV PYTHONUNBUFFERED 1

WORKDIR /app/webapi/

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

COPY . .

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

Figura 138. DockerFile

Definida la estructura del fichero Dockerfile, habrá que garantizar que aquellos recursos a los que accederá la imagen a la hora de generarse, se encuentran disponible y configurados de forma adecuada.

La imagen hace uso del fichero **requirements**, el cual debe encontrarse en la misma ruta que el fichero **Dockerfile** para evitar posibles errores en el proceso de construcción. En el fichero, deben encontrarse aquellas dependencias necesarias para la correcta construcción de la aplicación.

```
django<3
Markdown==3.1.1
django-filter==2.0.0
django-rest-framework==3.9.4
django-rest-passwordreset
django-cors-headers
mysqlclient
psycopg2>=2.7,<3.0
django-ckeditor==5.9.0
Pillow==9.5.0
```

Figura 139. Requirements.txt

- **Django<3**: es la biblioteca principal de Django cuyas versiones a instalar deben ser inferiores a la 3.
 - **Markdown=3.1.1**: Es una biblioteca para trabajar con el lenguaje de marcado **Markdown** en Python.
 - **Django-filter=2.0.0**: Es una biblioteca que proporciona funcionalidad de filtrado avanzado para las consultas en Django.
 - **Django-restframework=3.9.4**: Es una biblioteca que extiende las capacidades de Django para construir **APIs** web.
 - **Django-rest-passwordreset**: Es una biblioteca que proporciona funcionalidad para restablecer contraseñas en Django a través de una API REST.
 - **Django-cors-headers**: Es una biblioteca que proporciona encabezados CORS (Cross-Origin Resource Sharing) a

Una vez se encuentren preparados los recursos necesarios para la construcción de la imagen, se procede con su creación.

```
[root@luister /home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister# docker build --no-cache -t luister-webapi-image .
[+] Building 22.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> [internal] load .dockerignore
=> [internal] load history
=> [internal] load metadata for docker.io/library/python:3
=> [5/5] FROM docker.io/library/python:3@sha256:3a619e3c96fd4c5fc5e1998fd4dc6bf1403eb90c4c6409c70d7e80b9468df7df
=> [internal] load build context
=> [internal] load build context: 32.30kB
CACHED [internal] NOHTTP /app/webapi/
=> [3/5] COPY requirements.txt
=> [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY
=> exporting to image
=> exporting layers
=> writing image sha256:f87569870ac67959051b734340e4e20b6f21cb93d5f84e5222423bad2b2aef6
=> naming to docker.io/library/luister-webapi-image
root@luister /home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister# ]
```

Figura 140. Construcción imagen API

Cómo se puede observar la imagen se encuentra disponible para su ejecución:

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister# docker images
REPOSITORY          TAG      IMAGE ID      CREATED             SIZE
luister-webapi-image  latest   f87569870ac6  About a minute ago  986MB
luister-db-image     latest   d7039cc16d91  2 hours ago       565MB
luister-website-image latest   fde06aa47a40  26 hours ago      152MB
luister-altwebapi-image latest   36b658307d30  26 hours ago      486MB
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister#
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister#
```

Figura 141. Vista contenedor

Es posible ver un listado o una tabla con las imágenes creadas, mediante el comando de Docker **docker images**.

Continuando con el despliegue de la API, se procede a crear un contenedor a partir de la imagen anteriormente generada. Se ejecuta un contenedor en modo **deamon** (-d), exponiendo el puerto 8000 en el host local y enlazándolo con el puerto 8000 del contenedor (-p 8000:8000).

El contenedor se identificará con el nombre **luister-webapi** (--name luister-webapi) y se conectará a la red llamada **luister-network** (--network luister-network) y se basará en la imagen **luister-webapi-image**.

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister# docker run -d -p 8000:8000 --name luister-webapi --network luister-network luister-webapi-image
5dd1ecf6b616634ece058e596b0920797188b2ce437efc4d2211971c093b3
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister#
```

Figura 142. Ejecución contenedor

Es posible que en el proceso de ejecución del servidor se hayan producido errores, los cuales pueden ser visualizados con el comando: **docker logs <nombre-del-contenedor>**.

```
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister# docker logs luister-webapi
Watching for file changes with StatReloader
Performing system checks...
System check identified no issues (0 silenced).
June 05, 2023 - 20:37:05
Django version 2.2.28, using settings 'api.settings'
Starting development server at http://0.0.0.0:8000/
Quit the server with CONTROL-C.
root@luister:/home/ubuntu/produccion/LUISTER_PROYECTO/backend/luister#
```

Figura 143. Logs contenedor

En este caso, la ejecución del servidor se llevó a cabo de forma correcta.

Cabe la posibilidad de que los contenedores se encuentren ejecutándose sin presentar fallo visible o inconveniente alguno, sin embargo, es recomendable verificar que los servicios se encuentran funcionando de forma correcta accediendo al contenedor o consumiendo el servicio que ofrece.

Mediante un cliente encargado de realizar peticiones a **APIs**, se procede verificar la funcionalidad de inicio de sesión de la plataforma.

3.1.4 Pruebas

Conexión y acceso a los servicios de las APIs externas

Pruebas de conexión satisfactoria, solicitud y obtención de la información requerida.

Prueba de ajuste de las respuestas de las aplicaciones externas a los modelos de datos de la aplicación local.

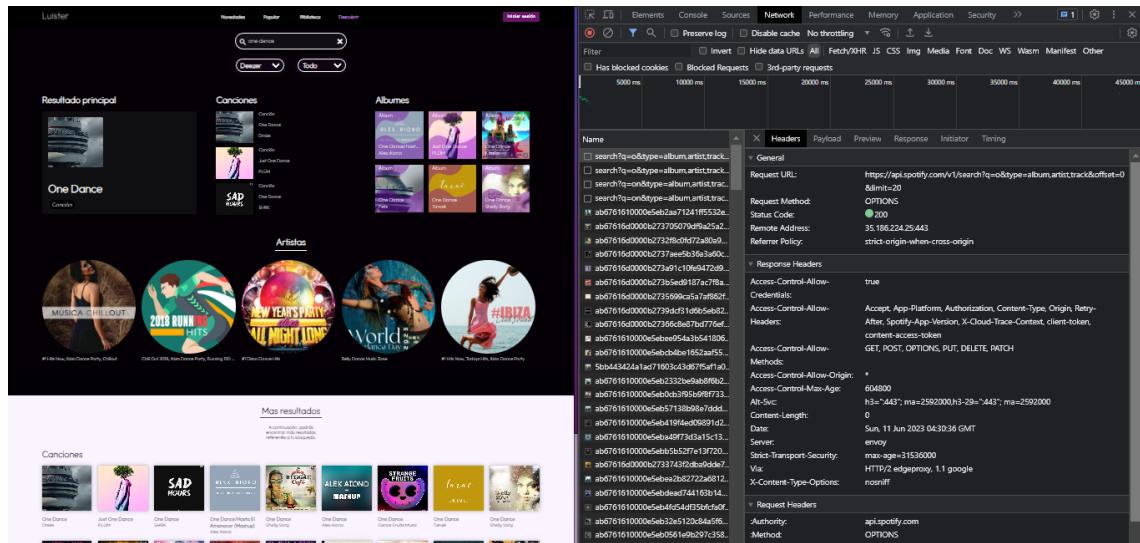


Figura 144. Prueba API

Conexión y acceso a los servicios de la API interna

Funcionalidad de inicio de sesión y creación de cookies.

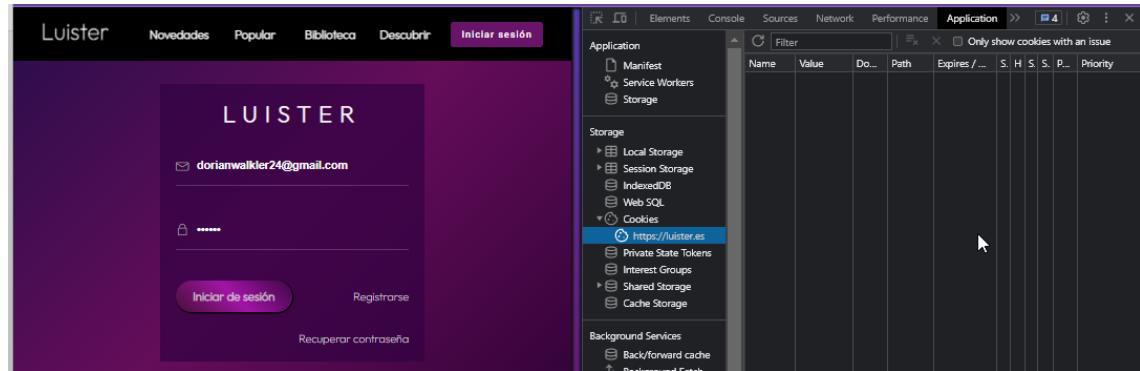


Figura 145. Prueba login

Se procede a iniciar sesión

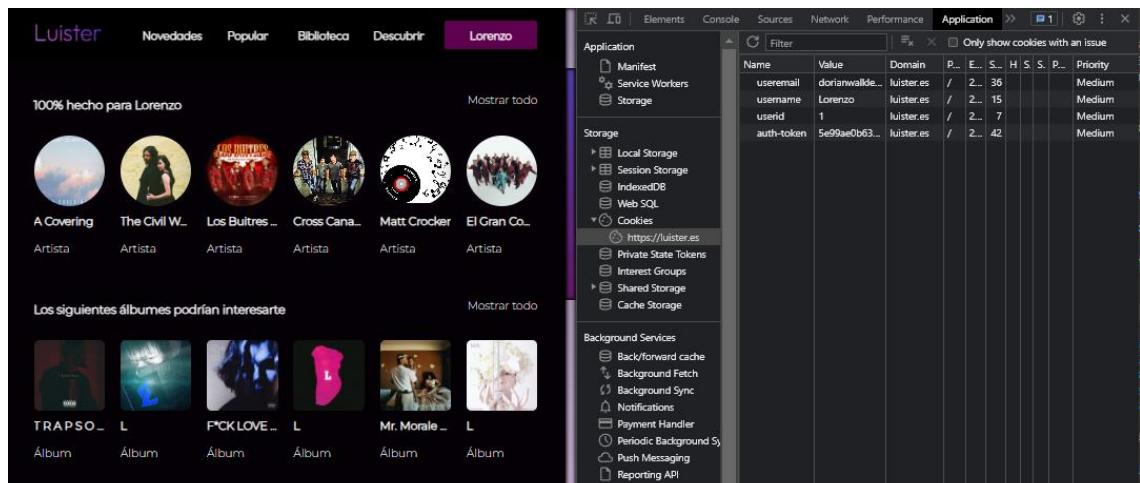


Figura 146. Resultado login

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 84 de 113
--------------------------	--------------------	------------------

Funcionalidad de registro de usuario y designación o restauración de contraseña.

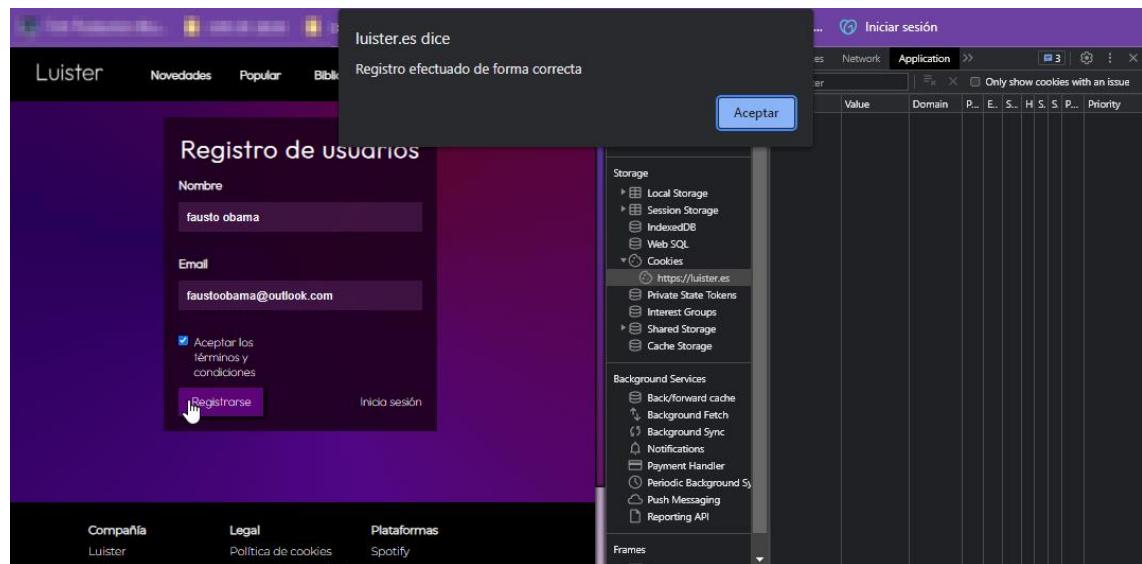


Figura 147. Prueba Registro

Designación de contraseña a través del enlace enviado al correo

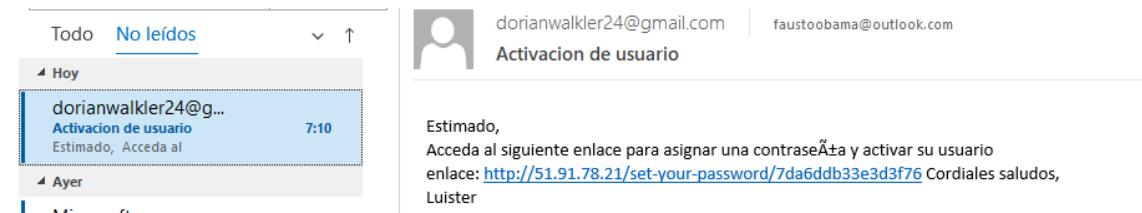


Figura 148. Enlace registro

Cuenta activada

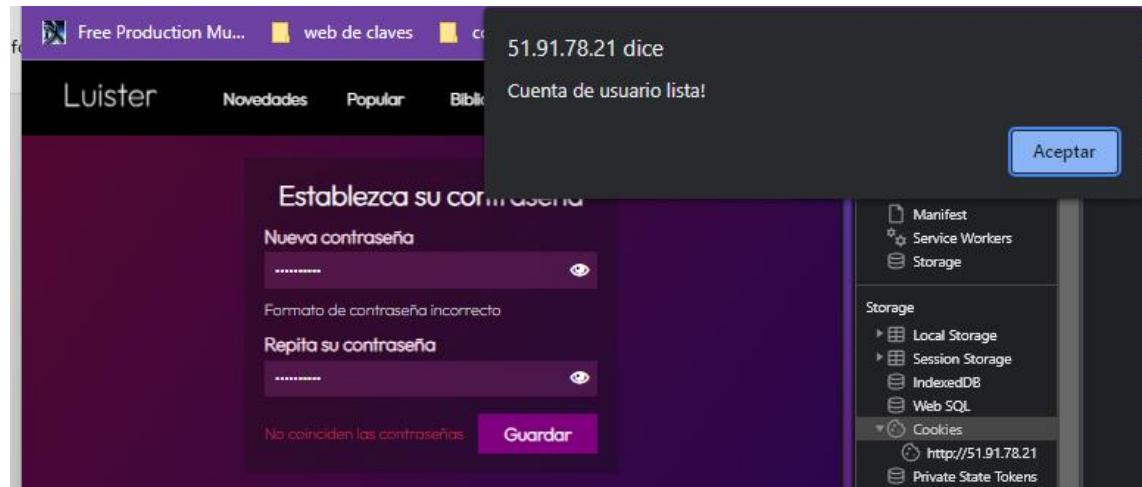


Figura 149. Registro correcto

Funcionalidad de cierre de sesión.

Se accede con el usuario recién registrado

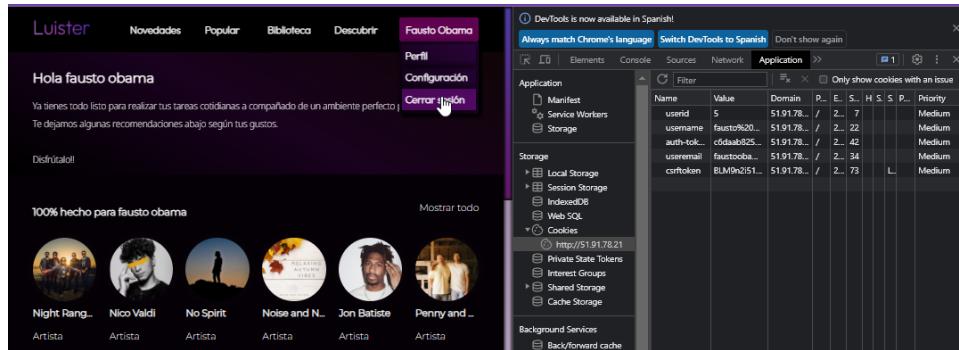


Figura 150. Prueba cierre sesión

Cierre de sesión satisfactorio y eliminación de cookies

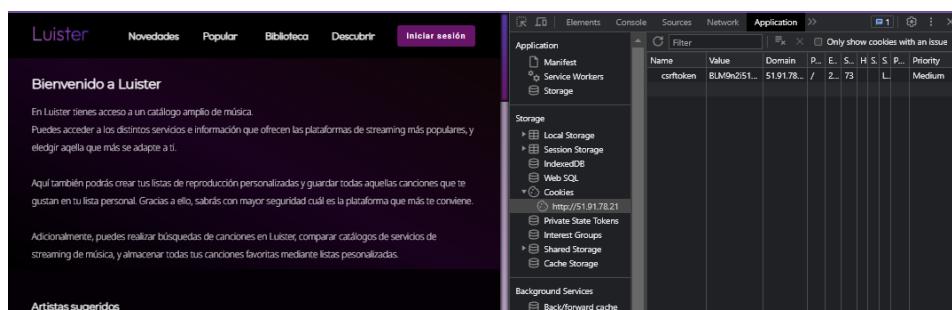


Figura 151. Prueba cookies

Funcionalidad de gestión de listas personalizadas: Creación, modificación y eliminación.

En el apartado de biblioteca, disponible solo para usuarios autenticados, se procede a crear una nueva lista.

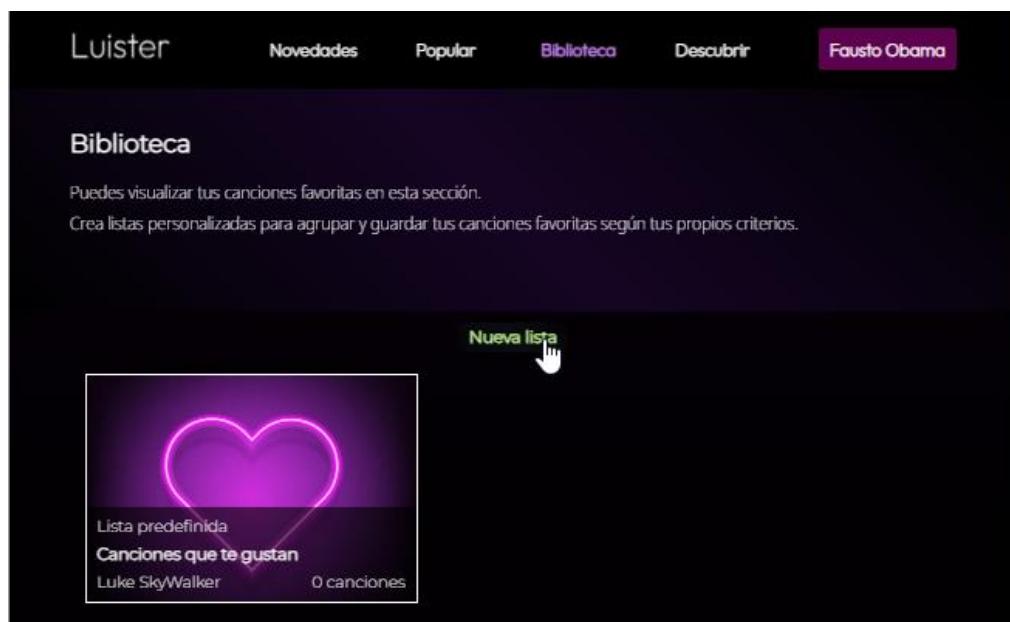


Figura 152. Creación lista

Se definen los datos que se consideren necesarios

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 86 de 113
--------------------------	--------------------	------------------

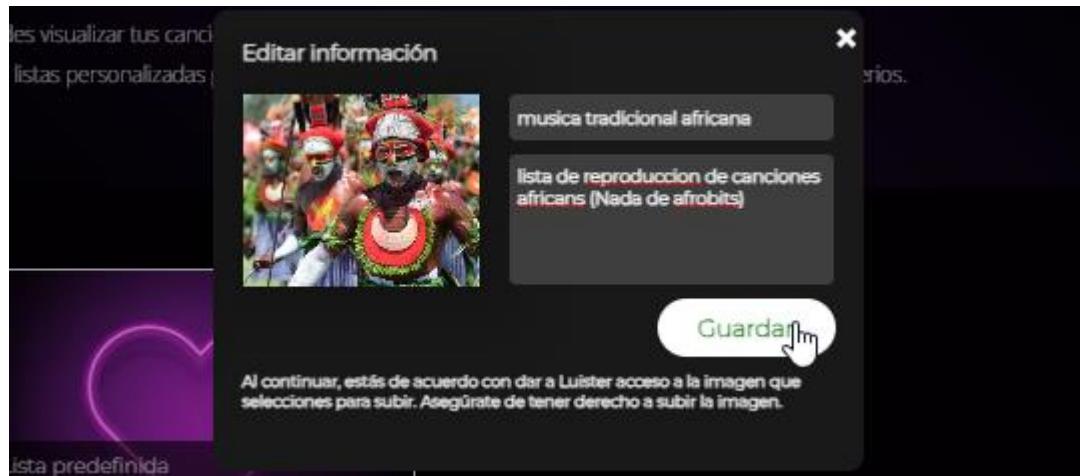


Figura 153. Guardado lista

Como se puede observar, la lista se encuentra creada de forma correcta



Figura 154. Visualización lista

Funcionalidad de adición y eliminación de pistas a listas personalizadas o a favoritos.

Mediante el menú contextual, se procede a añadir canciones a las listas personalizadas. Se añade la canción Karolina de Awilo a la lista de música tradicional africana.

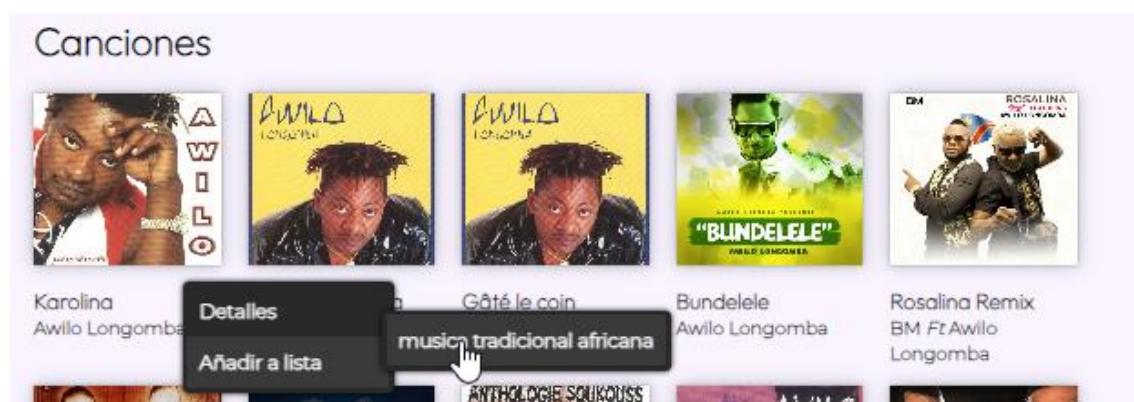


Figura 155. Prueba menú contextual

Como se puede observar, tanto en el apartado de la biblioteca como en el detalle de listas, se refleja el reciente cambio.



Figura 156. Visualización Pista añadida

Detalle de lista



Figura 157. Visualización detalle lista

Funcionalidad de seguimiento de artistas y eliminación de la lista de artistas seguidos.

Mediante el menú contextual o la sección de detalles de artista, se procede a seguir a aquellos artistas cuyo contenido resulta interesante para el usuario.

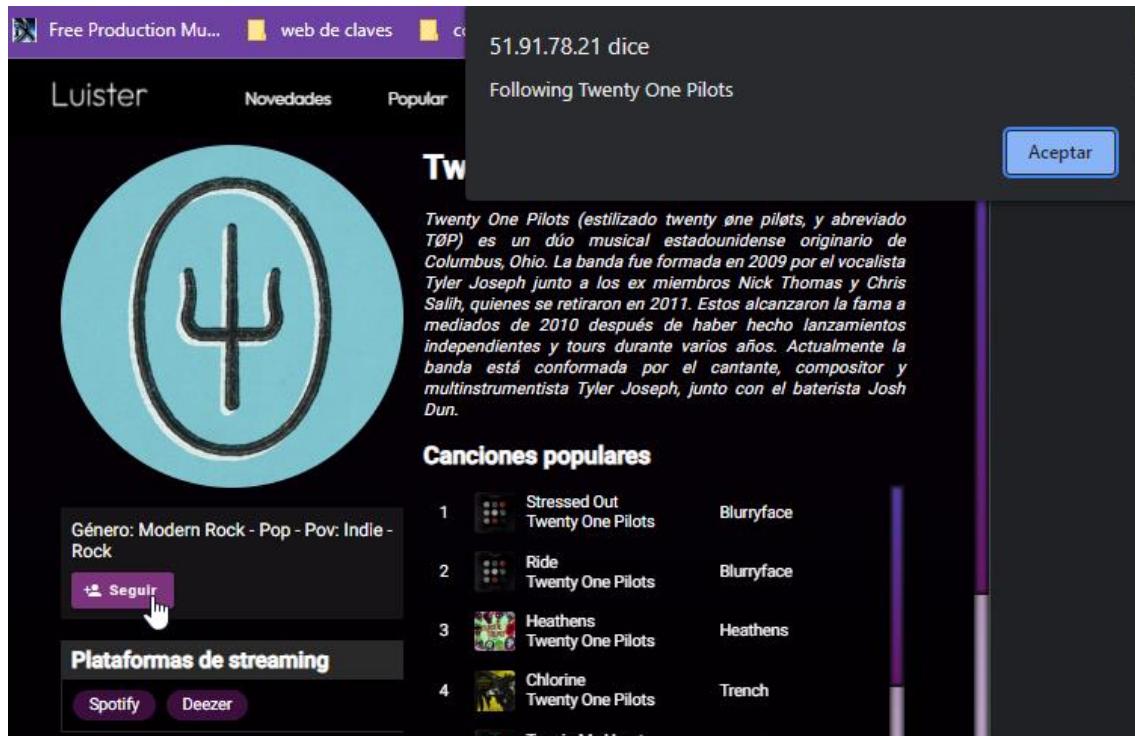


Figura 158. Prueba seguir artista

Se observa que el artista fue añadido a la lista de artistas seguidos del usuario

	name	image	follower	lookupkey
▶	Twenty One Pilots	https://i.scdn.co/image/ab6761610000e5eb196...	5	sfy:3YQKmKGau1PzlVlkL1iodx
▶	YOASOBI	https://i.scdn.co/image/ab6761610000e5ebfbe...	1	sfy:64tJ2EAv1R6UaZqc4iOCyj
▶	Linkin Park	https://i.scdn.co/image/ab6761610000e5eb811...	1	sfy:6XyY86QOPPrYVGvF9ch6wz
*	NULL	NULL	NULL	NULL

Figura 159. Vista BBDD artista

Se procede a dejar de seguir al artista

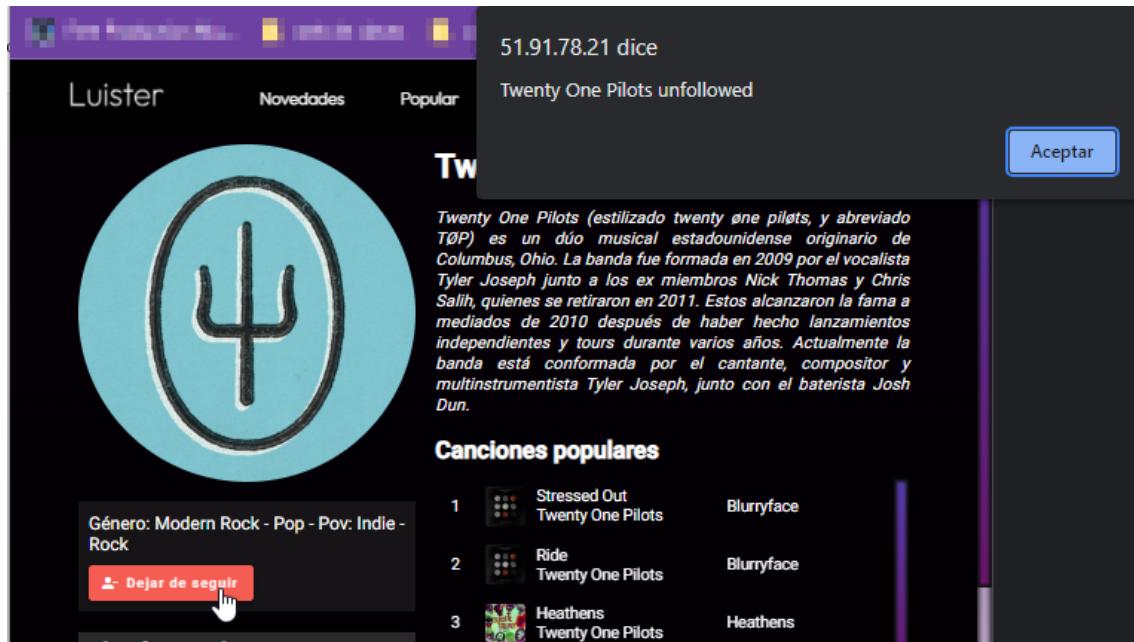


Figura 160. Dejar se seguir artista

Más funcionalidades de la aplicación

Impresión correcta de la información en las distintas plantillas de los componentes de la aplicación, mediante el acceso las propiedades en función del modelo de datos definido.

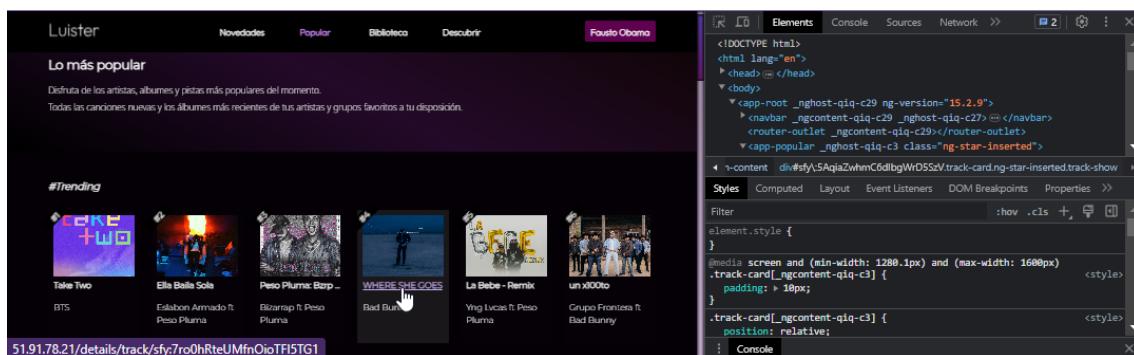


Figura 161. Impresión datos API

Prueba del servicio de inserción del menú contextual personalizado.

En función del tipo de elemento

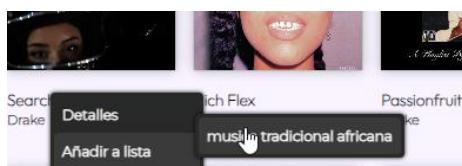


Figura 162. Menú contextual custom

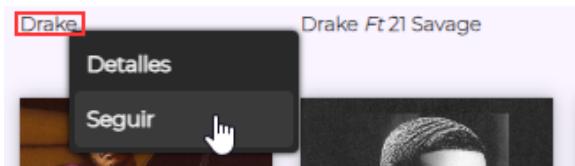


Figura 163. Menú contextual custom 2

Listas personalizadas

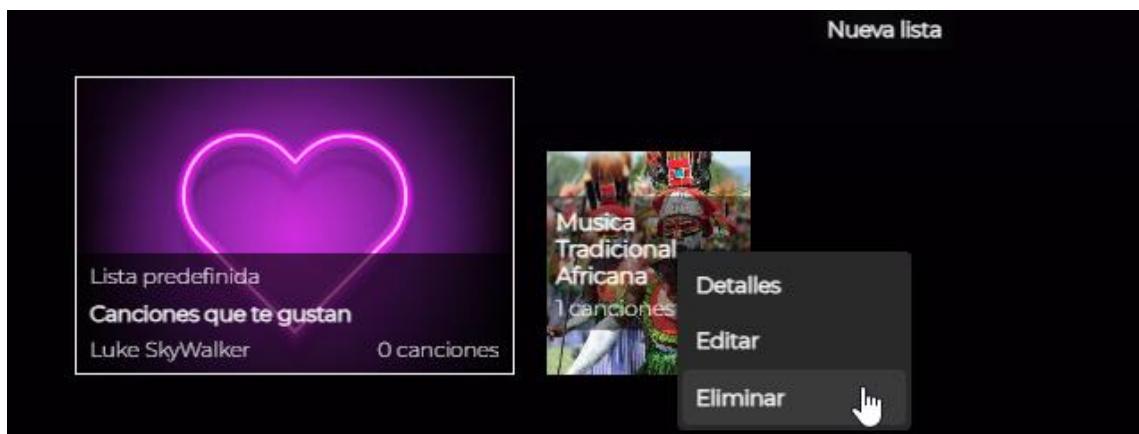


Figura 164. Menú contextual 3

Revisión del funcionamiento del servicio de administración de cookies.

Como se pudo observar en la sección de pruebas del inicio de sesión, el administrador de cookies se encarga de la creación de, edición y eliminación de las cookies cuando sea necesario. Acción que está siendo ejecutada de forma satisfactoria, de lo contrario, acciones como el obtener información de páginas concretas o iniciar sesión, no se llevarían a cabo ya que dependen de esta funcionalidad.

Funcionalidad del servicio de validación de campos de formulario.

Al ingresar valores incorrectos en los campos de los formularios, observamos que estos restringen cualquier tipo de acción hasta no haber sido corregidos.



Figura 165. Prueba registro

Rendimiento del servicio de restricción de acceso a rutas protegidas.

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 91 de 113
--------------------------	--------------------	------------------

En el diagrama de casos de uso de la aplicación, se especifican las acciones y recursos a los que tienen acceso los diferentes actores o usuarios.

Los usuarios autenticados podrán acceder a la aplicación sin restricciones, mientras los usuarios anónimos o no autenticados, tendrán limitado el acceso a apartados como la creación de listas, adición de pistas a listas, acceso a la biblioteca, seguimiento de usuarios, cierre de sesión, entre otros. Por ende, se implementaron mecanismos con el fin de cumplir con las restricciones.

- Bloqueo al acceder a la sección de biblioteca.

Mediante el guarda de rutas de Angular (GUARD), se redirige al usuario al apartado de inicio de sesión.

- Bloqueo al intentar seguir a artistas

Restricción por parte de la aplicación

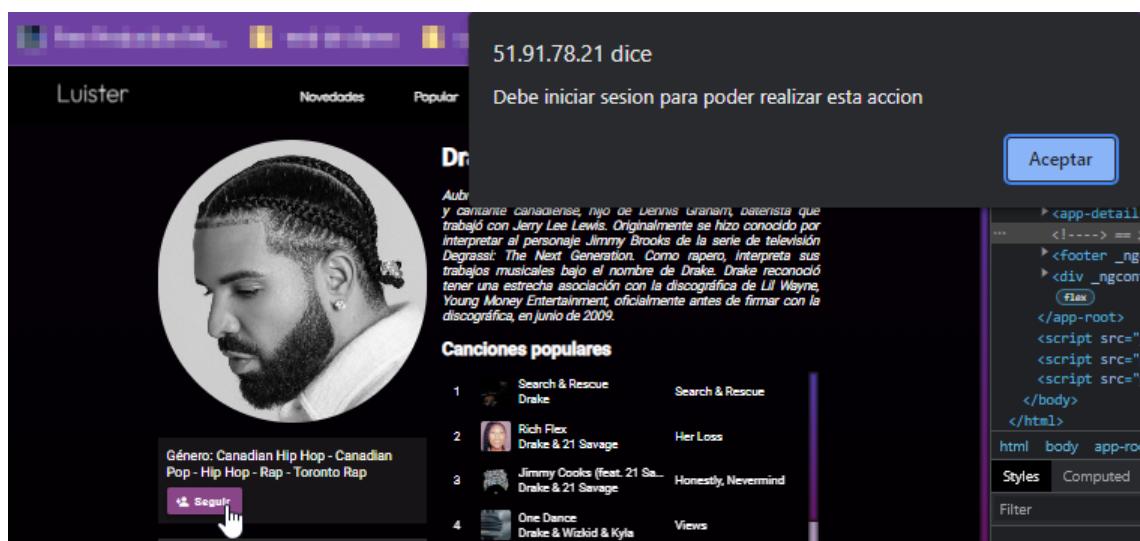


Figura 166. Restricción acceso

- Restricción del menú contextual a la hora de realizar cualquier acción no autorizada por el usuario prescindiendo de la renderización de dichos campos antes de su inserción.

Como se puede observar, al no estar registrado, solo se muestra en el menú contextual, aquellas acciones permitidas para el tipo de usuario.



Figura 167. Restricción en menú contextual

Funcionalidad de reproducción de pistas.

Se procede a verificar que en la sección de detalles se reproducen las pistas de forma correcta y no se crean bucles de reproducción al clicar repetidamente sobre el botón de reproducción.



Figura 168. Prueba de previsualización

3.2 Objetivos a conseguir

Una vez finalizado el proyecto y siendo conscientes de la posibilidad de mejora que existe. Hemos definido una serie de objetivos con diferentes períodos temporales.

- Conseguir una cifra de 100 usuarios activos (usuarios registrados) en los seis primeros meses.
- A medio plazo, incorporar nuevas plataformas en Luister (nuevas APIs o desarrollo de APIs propias).
- Mejorar el posicionamiento del sitio web. A través de un creciente número de usuarios y mejoras dentro de la aplicación. El objetivo es que el SEO de Luister mejore en el medio plazo.
- Conseguir una cifra de visitas superior a 300 en los seis primeros meses.

3.3 Previsión de los recursos materiales y humanos necesarios

Para la realización de Luister, se ha usado **Angular**. Esta elección viene justificada por ser un framework que se ajusta a las necesidades que planteaba el diseño del proyecto en su parte de cliente. Para poder usar Angular y poder aprovechar las funcionalidades que ofrece este framework, los miembros encargados de la parte cliente han realizado un curso previo, lo que unido a los conocimientos previos de **HTML, CSS y Javascript** (totalmente aplicables a TypeScript), han permitido la realización del proyecto en su parte cliente.

En cuanto al entorno servidor, también ha sido necesario un período de formación previo para la realización del servidor, ya que usa **Django API Rest** para complementar la formación del grado en Python. Durante este período se ha usado un servidor alternativo con la misma funcionalidad con el fin de poder comprobar el correcto funcionamiento de las secciones dependientes de un servidor, este servidor realizado en **PHP** ha quedado como una alternativa para futuras pruebas.

Con el fin de minimizar el coste de los recursos financieros utilizados, siempre que ha sido posible, se ha optado por usar herramientas de carácter gratuito o bien herramientas cuyos límites de uso gratuito adecuados al proyecto. Además, todo el trabajo se ha realizado usando los equipos propios de los integrantes del grupo.

Todos los integrantes han usado Visual Studio Code para la parte concerniente al desarrollo del proyecto y programación propiamente dicha, tanto en la parte cliente como para el entorno servidor.

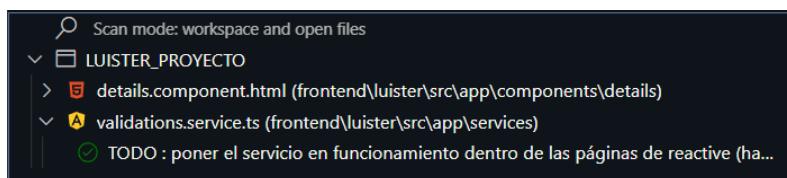


Figura 169. Todo Tree

Dentro de VS Code se ha usado la extensión **Todo Tree**, con el fin de poder establecer las diferentes tareas pendientes o puntos conflictivos dentro del código y poder facilitar la realización de tareas entre los distintos componentes del grupo, ya

que esta extensión permite, usando los comentarios del código, ver estas anotaciones de forma esquemática y de forma sencilla.

Otro recurso usado ha sido **Postman**, con el fin de realizar las diferentes pruebas de peticiones. Con esta herramienta se han ido probando los diferentes *endpoints* proporcionados por las plataformas con el fin de obtener la respuesta necesaria para nuestro proyecto, además de realizar las distintas pruebas de conexión a los servidores.

Como la intención del grupo es desplegar Luister y hacerlo de una manera correcta, se ha previsto el uso de una serie de recursos como un servidor, DNS y la compra de un dominio y una dirección de correo, que usaremos para mandar los correos de registro.

En cuanto a los recursos humanos necesarios, para llegar al estado actual de Luister, se han realizado las tareas necesarias para este proyecto entre los tres miembros, para lo que se han invertido una media de 16 horas semanales.

Para realizar mejoras en Luister, sería necesario incorporar más plataformas a través de nuevas APIs, con el fin de proporcionar a los usuarios una mayor cantidad de contenido. Estas mejoras podrían realizarse contando con más medios económicos y humanos, con los que se conseguiría acceso a nuevas APIs o bien desarrollar APIs propias para proveer de contenido a Luister.

3.4 Presupuesto económico.

Al tratarse de un proyecto de CFGS se ha diseñado desde un principio para no necesitar de un gran presupuesto para llegar al estado actual de funcionamiento. Por ello, se tienen en cuenta los gastos mínimos para realizar el despliegue y el resto de fases necesarias para el proyecto. (78.8)

- Coste servidor: 5.8 €/mes
- Lucidchart: 39.9 €, por superar el número de elementos permitido en la versión gratuita.
- Dominio y dirección de correo: 9 euros.
- VPS para copias de seguridad: 14€ /mes.
- Políticas de cookies (incluye política de privacidad): 10 euros.

En el caso de plantear Luister como un proyecto real además de los gastos listados anteriormente es necesario contemplar los gastos relacionados con los recursos humanos usados durante la realización del proyecto, es decir, las horas invertidas. Esto nos da como resultado la siguiente estimación:

- Salarios: 1200 € (3 desarrolladores) x 3 meses = 3600 €.

Con esta estimación del coste de los recursos humanos y los gastos detallados anteriormente, el coste de un proyecto como Luister, en el caso de tratarse de un proyecto real sería de 3678.8 €

4. PLANIFICACIÓN DE LA EJECUCIÓN DEL PROYECTO

Se detallan en este apartado, los objetivos a cumplir, las tareas a llevar a cabo, los plazos, así como responsabilidades relacionadas al proyecto, con el fin de organizar y estructurar todas las actividades necesarias para lograr los resultados establecidos y deseados.

Tareas: distribución de tareas y definición del calendario de trabajo.

Fechas: 13 de marzo - 15 de marzo.

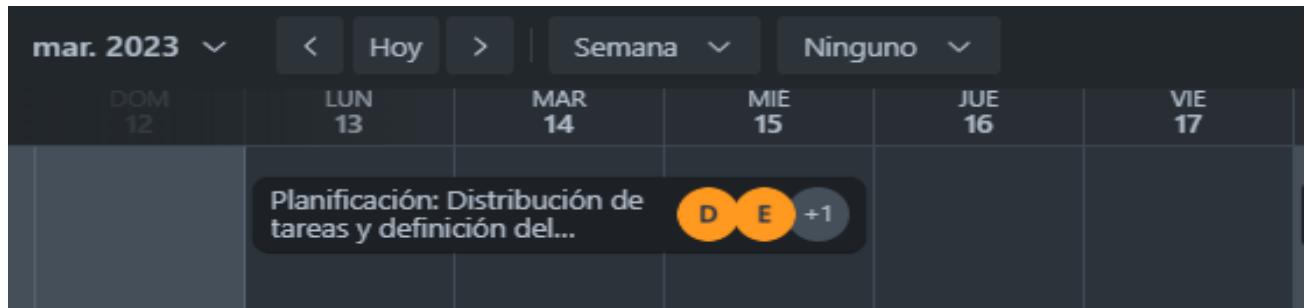


Figura 170. Planificación de tareas

4.1 Fase de Análisis

Fechas: 13 de marzo - 15 de marzo.

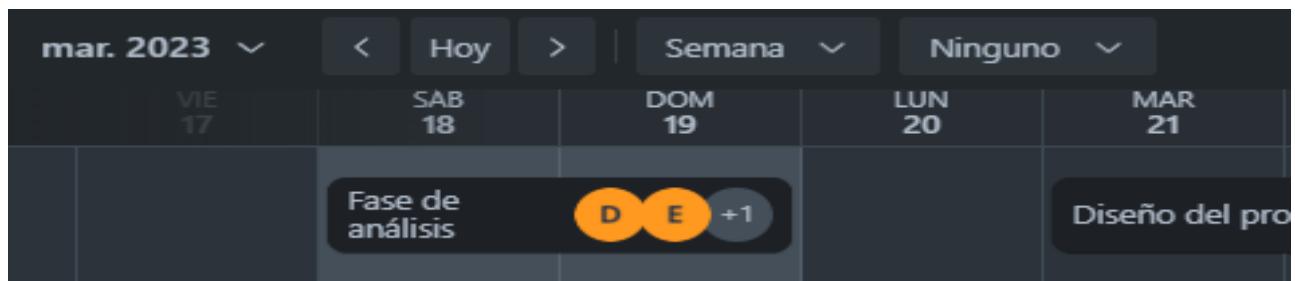


Figura 171. Fase de análisis

Tareas y actividades

- Búsqueda de información del mercado del streaming musical.
- Investigación de las plataformas musicales existentes.
- Búsqueda de proyectos parecidos a Luister.

4.2 Fase de diseño

Se definen y planifican las características, requisitos y funcionalidades del proyecto.

Fechas: 21 de marzo - 26 de marzo.

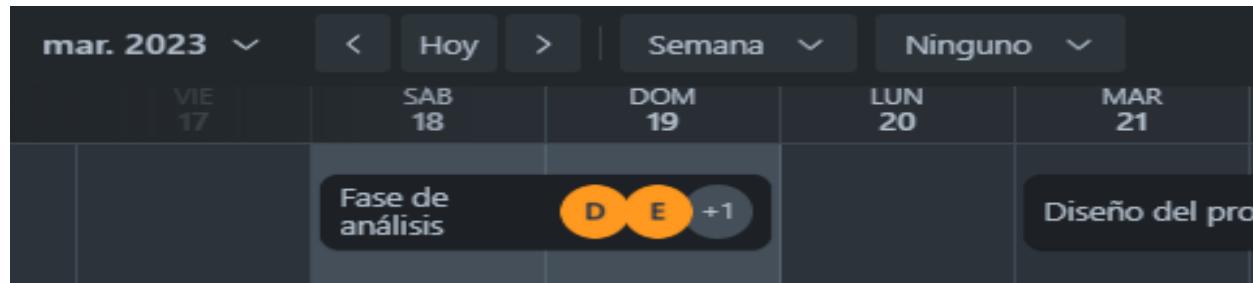


Figura 172. Fase de diseño

Tareas y actividades

- Creación de diagramas de casos de uso, estado y actividad.
- Modelo entidad – relación de la base de datos.
- Diseño de prototipos de la interfaz web: Sketching, Wireframing, prototipo final y designación de bloques genéricos de código o layouts.
- Definición de los colores y fuentes de la interfaz de la aplicación.

4.3 Fase de Implementación

Fechas: Del 15 de abril - 5 de junio.

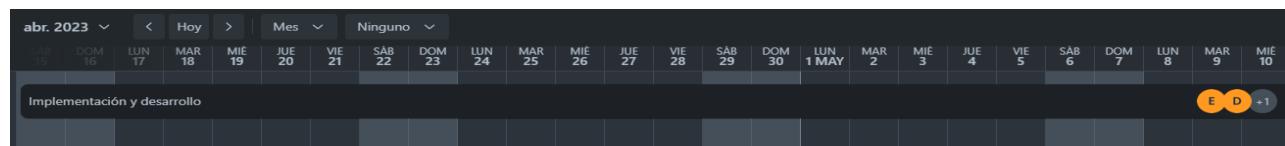


Figura 173. Fase de implementacion s1



Figura 174. Fase de implementación s2

Tareas y actividades

Entorno cliente

- Preparación del entorno de desarrollo: Instalación de Node y Angular.
- Creación de la aplicación: Luister.
- Creación de los componentes básicos del sitio web: Home, News, Popular, Library, Discover, Navbar y Footer.
- Creación del módulo de enrutamiento y rutas para cada componente.
- Creación de servicios de peticiones REST.
- Creación de modelos de respuestas para las peticiones REST.
- Creación del servicio de administración de cookies.
- Creación del servicio de validación de campos de formularios.
- Creación del servicio de impresión del menú contextual.
- Creación del servicio de restricción de acceso a rutas protegidas de la aplicación.

- Creación del servicio de estructuración del contenido HTML de los componentes de la aplicación.
- Creación de los estilos y maquetación del apartado visual de la aplicación.

Entorno servidor

- Preparación del entorno de desarrollo: Instalación de Django Rest Framework.
- Ajuste del fichero principal de configuración de dependencias y herramientas de la aplicación.
- Creación de las aplicaciones y rutas de acceso a los servicios de la aplicación.
- Creación de modelos de datos de las aplicaciones que conforma la API.
- Creación de los conversores de objetos o en representaciones serializadas como JSON, XML entre otros (Serializers).
- Creación del módulo de permisos de usuarios.
- Creación de vistas encargadas de brindar los servicios desarrollados en el programa, en función de la ruta de acceso, el método y los permisos del usuario.

Despliegue de la aplicación

- Instalación de Docker.
- Subida de los recursos, ficheros y directorios del proyecto al servidor de despliegue.
- Empaquetado de las aplicaciones que conforman el proyecto y creación del fichero Dockerfile referente a las imágenes mediante las cuales se ejecutarán los contenedores.
- Creación de redes Docker para las aplicaciones del proyecto.
- Creación de los directorios de almacenamiento de volúmenes para la persistencia de datos, y los ficheros (logs) de registro de accesos y errores.
- Despliegue del producto final: Creación y ejecución de los contenedores y servicios que componen la aplicación.

4.4 Fase de pruebas

De forma previa y durante la realización de las pruebas se ha realizado un período de documentación del proyecto, con el fin de recapitular las actividades llevadas a cabo y asentar los conocimientos adquiridos durante la realización del proyecto.

Documentación del proyecto

Fechas: 25 de mayo - 11 de junio

Tareas y actividades

- Documentación inicial del proyecto.
- Documentación de la fase de análisis y diseño del proyecto.
- Documentación de la fase de implementación y desarrollo.
- Documentación de la fase de pruebas.
- Documentación del despliegue y definición de métodos preventivos, correctivos y monitorización de la aplicación.

Fase de pruebas

Fechas: 5 de junio - 11 de junio.

I.E.S. Juan de la Cierva	Curso: 2022 / 2023	Página 98 de 113
--------------------------	--------------------	------------------

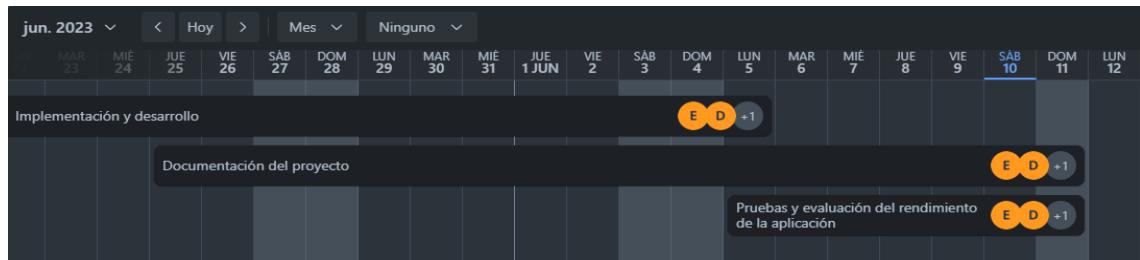


Figura 175. Fase de pruebas

Pruebas a realizar

Conexión y acceso a los servicios de las APIs externas

- Pruebas de conexión satisfactoria, solicitud y obtención de la información requerida.
- Prueba de ajuste de las respuestas de las aplicaciones externas a los modelos de datos de la aplicación local.

Conexión y acceso a los servicios de la API interna

- Funcionalidad de inicio de sesión y creación de cookies.
- Funcionalidad de registro de usuario y designación o restauración de contraseña.
- Funcionalidad de cierre de sesión.
- Funcionalidad de gestión de listas personalizadas: Creación, modificación y eliminación.
- Funcionalidad de adición y eliminación de pistas a listas personalizadas o a favoritos.
- Funcionalidad de seguimiento.

Más funcionalidades de la aplicación

- Impresión correcta de la información en las distintas plantillas de los componentes de la aplicación, mediante el acceso las propiedades en función del modelo de datos definido.
- Prueba del servicio de inserción del menú contextual personalizado.
- Revisión del funcionamiento del servicio de administración de cookies.
- Funcionalidad del servicio de validación de campos de formulario.
- Rendimiento del servicio de restricción de acceso a rutas protegidas.
- Funcionalidad de reproducción de pistas.

5. DEFINICIÓN DE PROCEDIMIENTOS DE CONTROL Y EVALUACIÓN

Medidas preventivas, correctivas y de mantenimiento.

Tras finalizar con el despliegue de las aplicaciones y servidores, es recomendable llevar un seguimiento del estado del sistema. Monitorizar aspectos como la cantidad de almacenamiento restante en el servidor, la carga de trabajo del procesador, entre otros, es una tarea recomendable para garantizar el correcto funcionamiento del servidor. Por ende, en este apartado se realizarán ajustes con el fin de prevenir y corregir posibles contingencias en el servidor, ya sea por hardware o software.

Revisión de espacio en disco

Es importante monitorizar el espacio en disco debido a que el rendimiento del sistema depende de la cantidad de espacio disponible. Si el espacio en disco se encuentra completamente ocupado, puede llegar a generar lentitud en las aplicaciones y un funcionamiento incorrecto del servidor al no disponer de almacenamiento suficiente para el intercambio de archivos o creación de ficheros temporales.

Otra de las consecuencias de no disponer de espacio en disco podría ser la incapacidad de modificar y guardar archivos, configuraciones del sistema o de realizar actualizaciones, lo cual puede tener consecuencias fatales en el sistema.

Implementación

Se crea un script de **BASH** en cargado de realizar la tarea. Se especifica la **ruta o volumen** cuyo espacio en disco se desea monitorizar, el **umbral o límite** de espacio que, al ser superado, generará la alerta.

```
#!/bin/bash
EMAIL="info@luister.es"
# DISK SPACE VARS
VOLUME="/"
DISK_LIMIT_PCENT=20
FREE_DISK_SPACE=$(df -h --output=pcent $VOLUME | awk 'NR==2 {print $1}' | tr -d '%')
# DISK SPACE
if [ $FREE_DISK_SPACE -lt $DISK_LIMIT_PCENT ]
then
  echo "Free disk space is less than ${DISK_LIMIT_PCENT}% on volume ${VOLUME} @$(hostname) - value: ${FREE_DISK_SPACE}%" \
    | mail -s "[ $(hostname) ] Alerta de uso de memoria" $EMAIL
fi
```

Figura 176. Scripts 1

El programa se encargará de obtener información sobre el porcentaje de espacio disponible en el volumen principal. Si éste es inferior al límite establecido previamente, se procede a enviar un correo a la dirección de correo electrónico del administrador del sitio informando de la potencial incidencia, de manera que éste proceda con su revisión y las medidas necesarias para su corrección.

Revisión de carga de trabajo del procesador y uso de memoria

La revisión de la carga del procesador permite identificar los procesos y aplicaciones propensos a consumir un gran porcentaje de recursos de CPU. Identificarlos facilita la gestión y optimización de dichos programas y, por ende, del sistema.

Permite identificar programas pesados que se encuentren causando problemas en el sistema o ralentizándolo. La detección de picos y subidas inusuales de la carga de trabajo de la unidad central de procesamiento es un indicio de aplicaciones perjudiciales para el sistema.

Implementación

Se modifica el script de monitorización de espacio en disco y se añade un condicional adicional para revisar al mismo tiempo, la carga del procesador y el uso de memoria en el sistema.

```
#CPU LOAD VARS
CPU_MAX_AV=50
CPU_LOAD=$(uptime | awk '{print $8}' | tr -d ',')
#MEMORY USAGE VARS
MEM_MAXUSG=80
TOTAL_MEM=$(free -m | awk 'NR==2 {print $2}')
AVAIL_MEM=$(free -m | awk 'NR==2 {print $7}')
MEM_USAGE=$((TOTAL_MEM - AVAIL_MEM))
MEM_USAGE_PCT=$(echo "scale=2; $MEM_USAGE / $TOTAL_MEM * 100" | bc)

# CPU LOAD AVERAGE
if [ ${CPU_MAX_AV} -lt ${CPU_LOAD%.} ]
then
  echo "Processor LOAD on @$(hostname) higher than expected - VALUE: ${CPU_LOAD}" \
    | mail -s "[ $(hostname) ] Alerta de uso de memoria" $EMAIL
fi

# MEM USAGE
if [ ${MEM_USAGE_PCT%.} -gt $MEM_MAXUSG ]
then
  echo "Lack of available memory in @$(hostname) - USAGE: ${MEM_USAGE_PCT}%" FREE: ${AVAIL_MEM}MB of ${TOTAL_MEM}MB" \
    | mail -s "[ $(hostname) ] Alerta de uso de memoria" $EMAIL
fi
```

Figura 177. Scripts 2

A continuación, se añade al **CRONTAB** el script de BASH de manera que este se ejecute periódicamente. En este caso, se ejecutará la revisión de almacenamiento en el volumen principal del servidor, uso de memoria del sistema y carga de trabajo del procesador, cada 4 horas.

Mediante el comando **crontab -e**, se accede al fichero de la herramienta del mismo nombre. En ésta se define las tareas a ejecutarse y el periodo de tiempo o intervalos de ejecución.

Se configura la ejecución del script de revisión de memoria, almacenamiento y carga de trabajo.

```
root@luister:/luister/luister# crontab -e
0 */4 * * * /opt/luister/.scripts/syshealt.sh
```

Figura 178. Crontab

El primer campo indica que se ejecutará cada segundo “0” y el segundo apartado, que se ejecutará cada 4 horas.

Renovación automática de certificados

Los certificados SSL se encargan de cifrar el intercambio de información entre los usuarios finales y la aplicación web, garantizando la protección de la información compartida contra posibles ataques de interceptación. De esta manera, se asegura la protección de la información confidencial de los usuarios.

Otro motivo que avala la renovación de los certificados son los indicadores visuales. Un sitio o aplicación web carente de certificados válidos muestra advertencias de seguridad al visitante, lo que genera desconfianza en el sitio web.

Por otro lado, las organizaciones deben cumplir con regulaciones y estándares de seguridad, como el Reglamento General de Protección de Datos (RGPD) en la Unión Europea. Estos requisitos a menudo incluyen la implementación de medidas de seguridad, como el uso de certificados SSL válidos. No renovar los certificados SSL puede resultar en incumplimiento de los requisitos legales y regulatorios.

Por último, con esta acción se evita la interrupción del servicio, asegurando que el sitio siga siendo funcional y accesible para los clientes.

Implementación

Se procede a crear el script encargado de renovar los certificados y enviarlos a los directorios de las aplicaciones.

La renovación de los certificados de la aplicación se realiza a través de **certbot**, una herramienta encargada de generar certificados y renovarlos. Se define el script de manera que éste se ejecute la sentencia **certbot renewal** cada 80 días; 8 días antes de que caduquen los certificados vigentes.

Finalizada la renovación, se notificará al administrador del sistema y la aplicación del éxito en la operación, o de un posible error en el proceso, de forma que éste proceda con las correspondientes acciones.

```
#!/bin/bash
CERTBOT_DIR="/etc/letsencrypt/live/luister.es/"
certbot renew
if [[ $? -eq 0 ]]; then
    cp -L "${CERTBOT_DIR}/*.*.pem" /projects/luister/frontend/luister/certs/
    cp -L "${CERTBOT_DIR}/*.*.pem" /projects/luister/backend/luister/certs/
    cp -L "${CERTBOT_DIR}/*.*.pem" /projects/luister/backend/phpluister/certs/
else
    echo "Certificates renewal failed" >> /tmp/failure.log
fi
```

Figura 179. Script renovación certs

Se añade el script al **CRONTAB** de manera que este se ejecute 5 días antes de que caduquen los certificados.

```
# system memory, cpu load and disk space check
0 */4 * * * /opt/luister/.scripts/syshealt.sh
# letsencrypt certs renewal
0 0 */80 * * /opt/luister/.scripts/certsrenewal.sh
```

Figura 180. Configuración Crontab

Actualización periódica de las dependencias del sistema

Las actualizaciones de dependencias incluyen mejoras en la estabilidad y el rendimiento del sistema. Contienen correcciones de fallos o irregularidades de versiones anteriores, optimizaciones y funcionalidades nuevas, garantizando un sistema más estable y con mejor rendimiento.

Incluyen correcciones de seguridad y actualizaciones en métodos de prevención de ataques en función de vulnerabilidades detectadas en versiones anteriores.

Por último, la compatibilidad y el soporte técnico vendrían a ser otros de los motivos que justifican la actualización periódica.

Las nuevas tecnologías se encuentran en constante evolución, por lo que prescindir de la actualización periódica de dependencias propicia a que éste resulte incompatible con las tecnologías del momento, se vuelva obsoleto y pierda su funcionalidad.

Implementación

Se crea un script encargado de actualizar la lista de paquetes disponibles en los repositorios configurados en el sistema, actualizar los paquetes instalados a sus versiones más recientes, eliminar los paquetes que fueron instalados como dependencias y que ya no son necesarios por ningún otro paquete instalado en el sistema y, por último, limpia la caché de paquetes descargados en el sistema.

```
#!/bin/bash
apt update
apt upgrade -y
# apt dist-upgrade -y # en caso de requerir actualizar el SO
apt autoremove -y
apt clean
```

Figura 181. Actualización dependencias

Se ejecuta el script para verificar que funciona de forma correcta.

```
root@luister:/# /opt/luister/.scripts/dpkgupdate.sh
Hit:1 https://download.docker.com/linux/ubuntu jammy InRelease
Hit:2 http://security.ubuntu.com/ubuntu jammy-security InRelease
Hit:3 http://nova.clouds.archive.ubuntu.com/ubuntu jammy InRelease
Hit:4 http://nova.clouds.archive.ubuntu.com/ubuntu jammy-updates InRelease
Hit:5 http://nova.clouds.archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... 8%
```

Figura 182. Prueba de funcionamiento

Se procede a agregar el script al **CRONTAB**, de manera que se proceda a actualizar el listado de dependencias disponibles y a actualizar el conjunto de paquetes del sistema, los días 2 de cada mes a primera hora del día.

```
# system memory, cpu load and disk space check
0 */4 * * * /opt/luister/.scripts/syshealt.sh
# letsencrypt certs renewal
0 0 */80 * * /opt/luister/.scripts/certsrenewal.sh
# weekly backup
0 0 * * 0 /opt/luister/.scripts/luisterbackup.sh
# monthly dpkg update
0 0 2 * * /opt/luister/.scripts/dpkgupdate.sh
```

Figura 183. Configuración Crontab 2

Copia de seguridad del sistema

Hacer una copia de seguridad (**BACKUP**) del servidor es una buena práctica para garantizar la persistencia de los datos y una forma de asegurar los ficheros y aplicaciones importantes del servidor ante alguna incidencia inesperada.

Implementación

Se crea un script encargado de comprimir el directorio principal con los ficheros de la aplicación y subirlos al servidor de respaldo para su posterior descarga en caso de que se presente alguna contingencia en el servidor de origen.

```
#!/bin/bash

DATE=$(date +%Y%m%d)
HOUR=$(date +%H%M)
TARGET="/projects/luister/"
BACKUPFILE="/projects/backup/luister_${DATE}_${HOUR}.tar.gz"

tar -cvzf ${BACKUPFILE} ${TARGET}
if [[ -f ${BACKUPFILE} ]]
then
    scp ${BACKUPFILE} "root@217.160.114.54:/luister/backup/"
else
    echo "Failed perform backup" >> /tmp/error.log
fi
```

Figura 184. Script respaldo

Se procede a ejecutar el script para verificar que se genera la copia de seguridad de forma correcta, y se envía al servidor de respaldo encargado de almacenarlas.

```
root@luister:/# /opt/luister/.scripts/luisterbackup.sh
tar: Removing leading '/' from member names
/projects/luister/
/projects/luister/Implementacion_frontend_inicio-componentes-navbar-footer.docx
/projects/luister/.gitignore
/projects/luister/backend/
/projects/luister/backend/phpluister/
/projects/luister/backend/phpluister/favtracks.php
/projects/luister/backend/phpluister/addtracktolist.php
```

Figura 185. Prueba respaldo

Como se observa a continuación, se realizó la copia de seguridad de los ficheros de forma correcta.

```
/projects/luister/luister-db ./data-backup/bb_buffer_pool
/projects/luister/luister-db ./data-backup/public_key.pem
luister_20230607_1814.tar.gz
root@luister:#
root@luister:#
100% 30MB 11.0MB/s 00:02
```

Figura 186. Confirmación respaldo

Se accede al servidor remoto para verificar que, en el directorio de dedicado a almacenar las copias de seguridad, se encuentra el fichero recién generado.

```
root@luister:~# ssh root@217.160.114.54
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-72-generic x86_64)

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

Last login: Wed Jun  7 18:28:26 2023 from 51.91.78.21
root@ubuntu:~# ls -lhrt /luister/backup/
total 31M
-rw-r--r-- 1 root root 31M Jun  7 18:14 luister_20230607_1814.tar.gz
root@ubuntu:~#
root@ubuntu:~#
```

Figura 187. Servidor de respaldo

Verificada la efectividad del programa, se procede a añadirlo al **CRONTAB** de forma que se ejecute la copia de seguridad de los ficheros principales de la aplicación cada semana.

```
# system memory, cpu load and disk space check
0 */4 * * * /opt/luister/.scripts/syshealt.sh
# letsencrypt certs renewal
0 0 */80 * * /opt/luister/.scripts/certsrenewal.sh
# weekly backup
0 0 * * 0 /opt/luister/.scripts/luisterbackup.sh
```

Figura 188. Configuración Crontab respaldo

Se configura la ejecución del respaldo semanal de la aplicación, los domingos a primera hora del día (00:00:00).

Monitorización de ficheros logs de aplicaciones y sistemas

Para finalizar con el apartado de medidas preventivas, correctivas y monitorización del sistema, es necesario hablar de los logs.

Un fichero log o fichero de registro es un archivo que registra y almacena eventos, actividades o mensajes generados por un sistema, una aplicación, un servicio o un dispositivo.

Los ficheros log se utilizan para llevar un registro de actividades y proporcionar información útil para el análisis, el seguimiento de problemas y el diagnóstico de errores.

El sistema y la aplicación desplegada disponen de registro a los cuales recurrir en caso de requerir información sobre algún evento concreto, o para el estudio, análisis y gestión de alguna incidencia.

A continuación, se puede observar un registro de log reportando un error a la hora de acceder a un recurso del sitio web eliminado de forma intencional para el ejemplo.

```
root@luisster:~# tail -f /app/website/logs/website.log
2023/06/07 19:19:56 [warn] 27#27: *75 upstream server temporarily disabled while SSL handshaking to upstream, client: 90.161.1.50, server: luisster.es, request: "GET /api2/search/album?q=a&offset=0&limit=22 HTTP/1.1", upstream: "https://96.16.248.152:443/search/album?q=a&offset=0&limit=22", host: "luisster.es", referer: "https://luisster.es/luisster.es"
90.161.1.50 - - [07/Jun/2023:19:19:56 +0000] "GET /api2/search/album?q=a&offset=0&limit=22 HTTP/1.1" 502 559 "https://luisster.es/discover" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36" "-"
2023/06/07 19:19:58 [error] 27#27: *75 no live upstreams while connecting to upstream, client: 90.161.1.50, server: luisster.es, request: "GET /api2/search/album?q=a&offset=0&limit=22", host: "luisster.es", referer: "https://luisster.es/discover"
90.161.1.50 - - [07/Jun/2023:19:19:58 +0000] "GET /api2/search/album?q=a&offset=0&limit=22 HTTP/1.1" 502 559 "https://luisster.es/discover" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36" "-"
2023/06/07 19:19:59 [error] 27#27: *75 no live upstreams while connecting to upstream, client: 90.161.1.50, server: luisster.es, request: "GET /api2/search/album?q=aaa&offset=0&limit=22 HTTP/1.1", upstream: "https://www.deezer.com/search/album?q=aaa&offset=0&limit=22", host: "luisster.es", referer: "https://luisster.es/discover"
90.161.1.50 - - [07/Jun/2023:19:19:59 +0000] "GET /api2/search/album?q=aaa&offset=0&limit=22 HTTP/1.1" 502 559 "https://luisster.es/discover" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36" "-"
2023/06/07 19:19:59 [error] 27#27: *75 no live upstreams while connecting to upstream, client: 90.161.1.50, server: luisster.es, request: "GET /api2/search/album?q=aaa&offset=0&limit=22 HTTP/1.1", upstream: "https://www.deezer.com/search/album?q=aaa&offset=0&limit=22", host: "luisster.es", referer: "https://luisster.es/discover"
90.161.1.50 - - [07/Jun/2023:19:19:59 +0000] "GET /api2/search/album?q=aaa&offset=0&limit=22 HTTP/1.1" 502 559 "https://luisster.es/discover" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36" "-"
2023/06/07 19:20:00 [error] 27#27: *75 no live upstreams while connecting to upstream, client: 90.161.1.50, server: luisster.es, request: "GET /api2/search/album?q=aaaa&offset=0&limit=22 HTTP/1.1", upstream: "https://www.deezer.com/search/album?q=aaaa&offset=0&limit=22", host: "luisster.es", referer: "https://luisster.es/discover"
90.161.1.50 - - [07/Jun/2023:19:20:00 +0000] "GET /api2/search/album?q=aaaa&offset=0&limit=22 HTTP/1.1" 502 559 "https://luisster.es/discover" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/114.0.0.0 Safari/537.36" "-"
```

Figura 189. Visualización Logs

Resultan útiles para rastrear las peticiones, observar los accesos, los errores y a partir de su estudio, poder trabajar en nuevas mejoras y actualizaciones para la aplicación

6. FUENTES

- El Pingüino Tech. (2022). Cómo Usar CRONTAB en UBUNTU para AUTOMATIZAR TAREAS. youtube.com. <https://www.youtube.com/watch?v=d2Q0NiyVO5M>
- Net Faculty. (2014). Administración de Logs en Linux. youtube.com. <https://www.youtube.com/watch?v=fPKjbqFQypg>
- nicobytes. (2023). Guardianes como Funciones en Angular. youtube.com. https://www.youtube.com/watch?v=3aJ8f_hKBPQ
- Leifer Mendez. (2021). PROTEGIENDO RUTAS en Angular 11 (GUARDS) ¿Cómo proteger rutas en angular 11? // CURSO ANGULAR. youtube.com. <https://www.youtube.com/watch?v=Q3Hj9ItAdUM>
- Leifer Mendez. (2021). ¿Cómo PASAR (emitir) DATOS entre un COMPONENTE A OTRO EN ANGULAR con SERVICIO? // CURSO ANGULAR. youtube.com. <https://www.youtube.com/watch?v=HTivuXwS2-Y>
- Domini Code. (2022). Enlace de datos bidireccional / Two-way data binding - 28 Días aprendiendo Angular. youtube.com. <https://www.youtube.com/watch?v=EuFh5s5lYIU>
- midudev. (2021). MEJORA tu código JAVASCRIPT con PARÁMETROS nombrados en las funciones (Mejores prácticas). youtube.com. <https://www.youtube.com/watch?v=jmxZrlHPRDg>
- Leifer Mendez. (2022). HostListener ¿Cómo usarlos? y volverte un PRO / JUGANDO con ANGULAR. youtube.com. <https://www.youtube.com/watch?v=zGzmSoaQnsY>
- pildorasinformaticas. (2022). Curso Angular. Servicios. Vídeo 21. youtube.com. <https://www.youtube.com/watch?v=W1nQXZwXKb0>
- Domini Code. (2022). ¿Qué es una directiva? nglf, ngFor - 28 Días aprendiendo Angular #6. youtube.com. https://www.youtube.com/watch?v=ZUX_K4jR-9Q
- Runebook. (s.f.). Django REST Framework 3.14 [Español]. Runebook. https://runebook.dev/es/docs/django_rest_framework/
- Encode OSS Ltd. (2011). Django REST Framework. Django REST Framework. <https://www.django-rest-framework.org/>
- Developer.pe. (2020). Curso Django REST. youtube. https://www.youtube.com/watch?v=MMFB2Eoeuk&list=PLMbRqrU_kvbRI4PgSzgbh8XPEwC1RNj8F
- Interactive Programmers Community. (2000). FORO - Django. lawebdelprogramador. <https://www.lawebdelprogramador.com/foros/Django/index1.html>
- Hardik Patel. (2018). 10 | Nested serializer for Create and update data in Django Rest Framework. youtube. <https://www.youtube.com/watch?v=EyMFf9O6E60>
- Mozilla Foundation. (2022). Framework Web Django (Python). developer.mozilla.org. <https://developer.mozilla.org/es/docs/Learn/Server-side/Django>
- MuhammedAli. (2022). [https://adamtheautomator.com/django-docker/#:~:text=Building%20Docker%20Compose%20File%20to%20Run%20Django%20and,data%20for%20your%20API%20model.%20...%20M%C3%A1s%20elementos. adamtheautomator.](https://adamtheautomator.com/django-docker/#:~:text=Building%20Docker%20Compose%20File%20to%20Run%20Django%20and,data%20for%20your%20API%20model.%20...%20M%C3%A1s%20elementos.)

<https://adamtheautomator.com/django-docker/#:~:text=Building%20Docker%20Compose%20File%20to%20Run%20Django%20and,database%20for%20your%20API%20model.%20...%20M%C3%A1s%20elementos>

Jeff Atwood, Joel Spolsky. (2014). Questions tagged [django-rest-framework]. stackoverflow. <https://stackoverflow.com/questions/tagged/django-rest-framework>

CASCUDO, T. (2014). "Tecnología de la música digital y musicología histórica: reflexiones desde el presente". En X. Aviñoa (Ed), Tecnología y creación musical (pp 301-322). Lleida: Milenio

ABEILLÉ, C. (2013). Go with the flow: cambios en la distribución y consumo de la música en la era digital. Letra. Imagen. Sonido. L.I.S., 10 (IV), 120-130.

Crossan, M. & Wilkinson, M. & Perry, M. & Hunter, T. & Smith, T.. (2001). Napster and MP3: Redefining the Music Industry. Ivey Publishing, I, pp. 0-22

Francesco Braga. (2023). IFPI's Global Music Report 2023. IFPI. <https://globalmusicreport.ifpi.org/>

Mark Mulligan. (2022). Music subscriber market shares Q2 2022. Midia Research, I, pp. 1

The trichordist. (2020). 2019-2020 Streaming Price Bible : YouTube is STILL The #1 Problem To Solve. The trichordist, I, pp. 1

Nicolas-Sans, R. & Bustos, J.. (2022). Las plataformas de streaming musical y su influencia en redes sociales: estudio comparativo Spotify y Amazon Music en España. Researchgate, I, pp. 1-20

Del Valle, A.. (2015). COMPORTAMIENTO DEL ACTUAL CONSUMIDOR DE MÚSICA REFERENTE A LOS SERVICIOS MUSICALES MÁS POPULARES Y SU UTILIZACIÓN COMO HERRAMIENTAS PUBLICITARIAS. [repositorio.comillas.edu](https://repositorio.comillas.edu/jspui/bitstream/11531/4140/1/TFG001140.pdf).

<https://repositorio.comillas.edu/jspui/bitstream/11531/4140/1/TFG001140.pdf>

7. ANEXOS

Diagrama de casos de uso

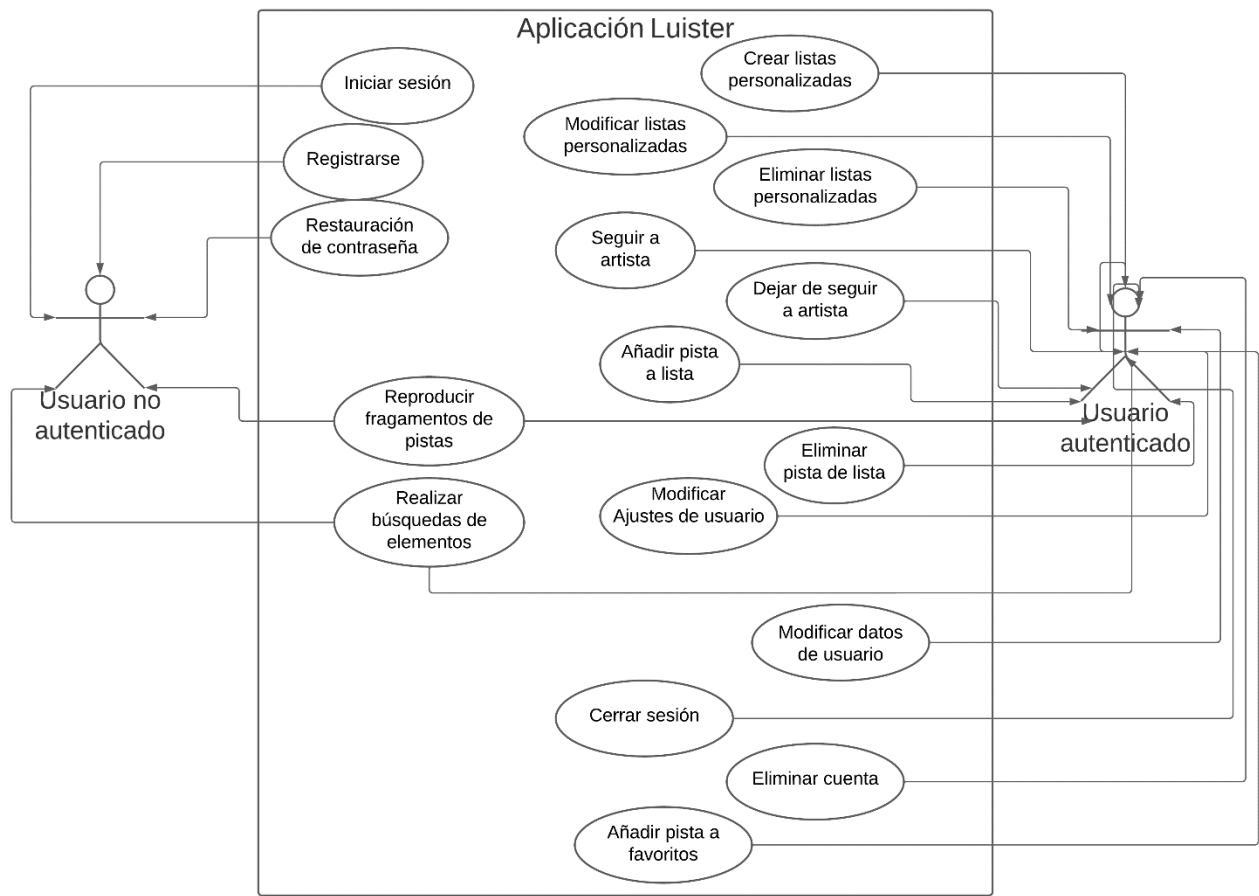


Figura 190. Diagrama Casos de uso

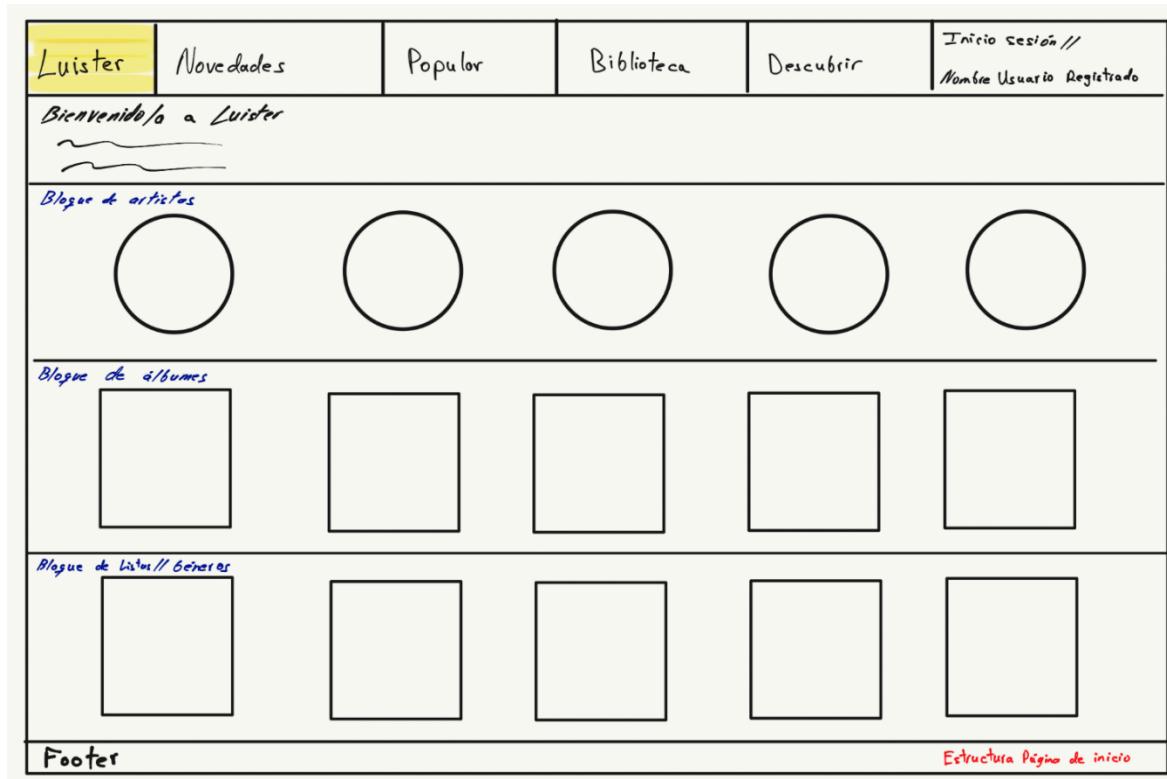


Figura 191. Sketching Inicio

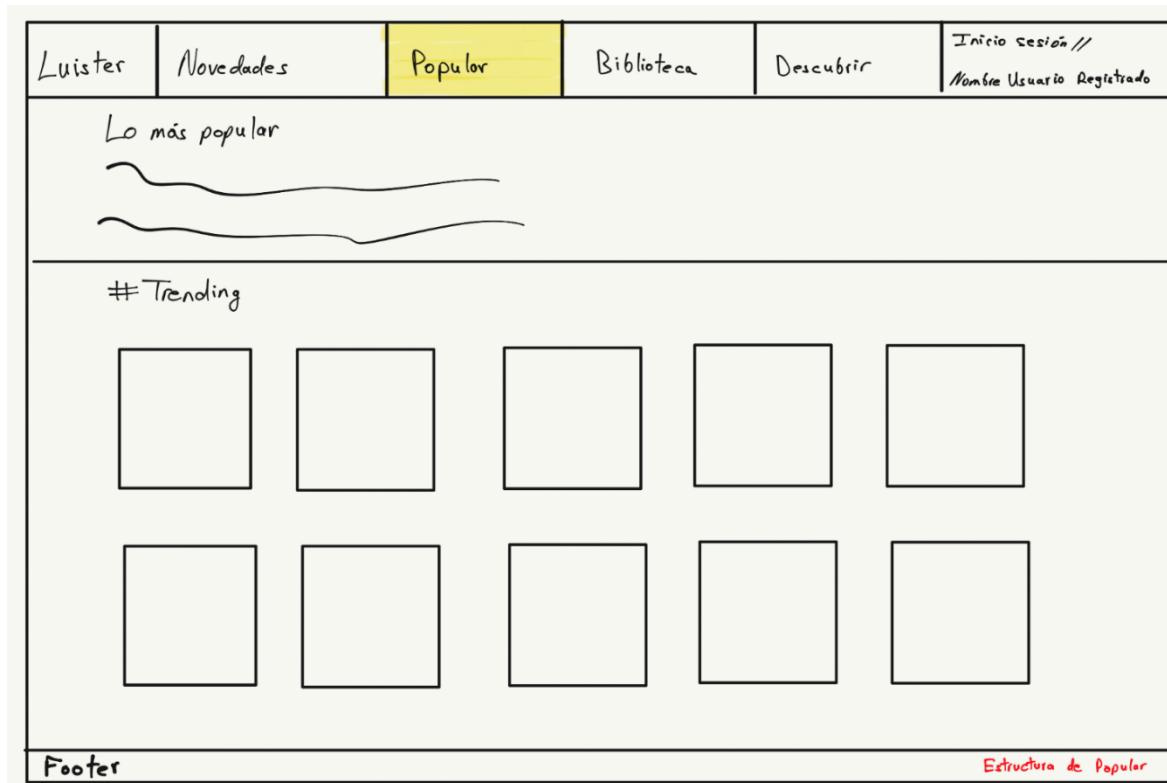


Figura 192. Sketching Popular

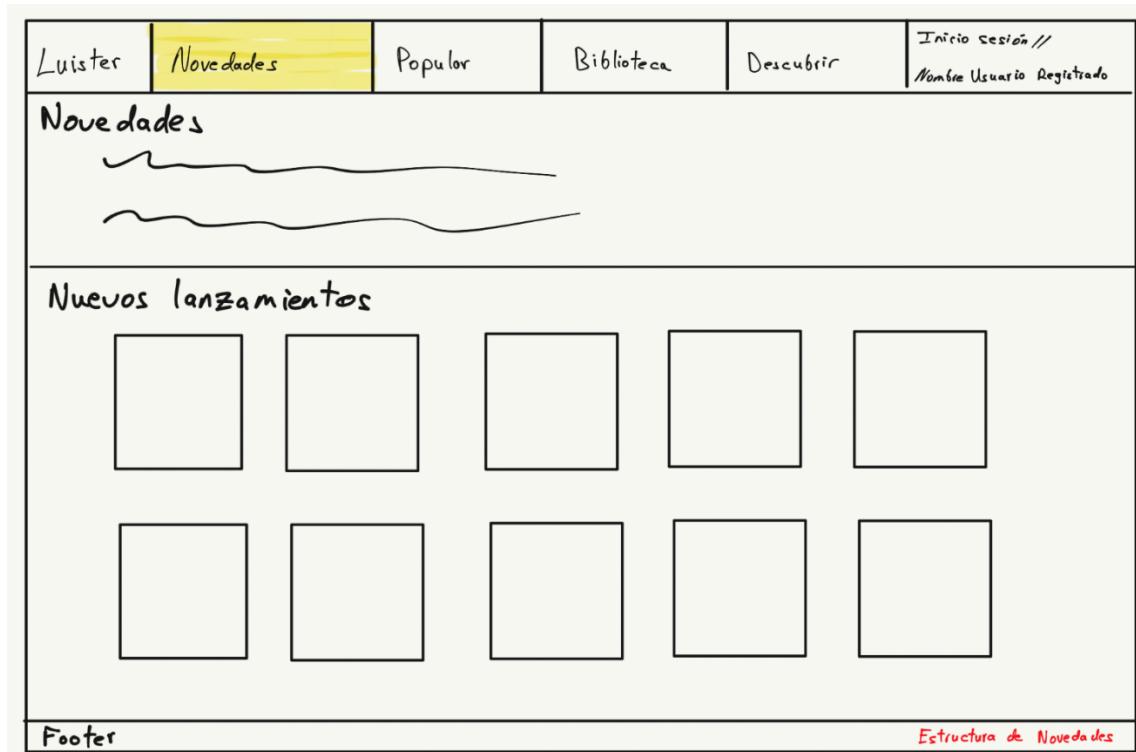


Figura 193. Sketching Novedades

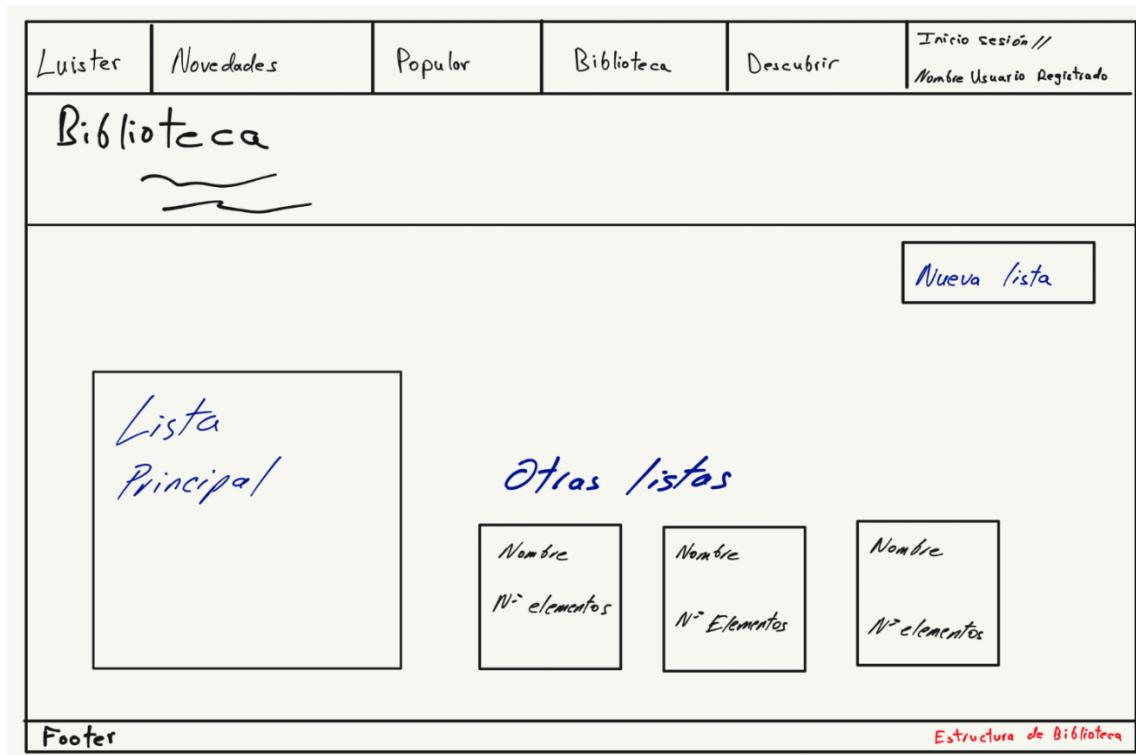


Figura 194. Sketching Biblioteca

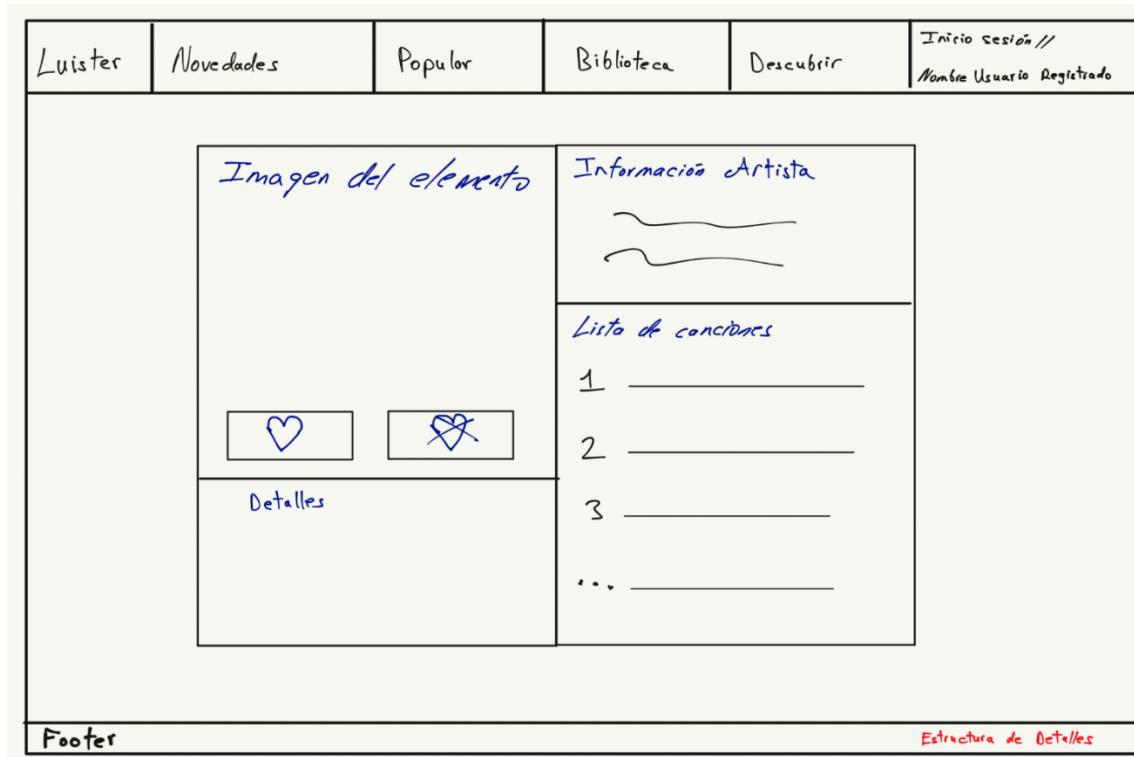
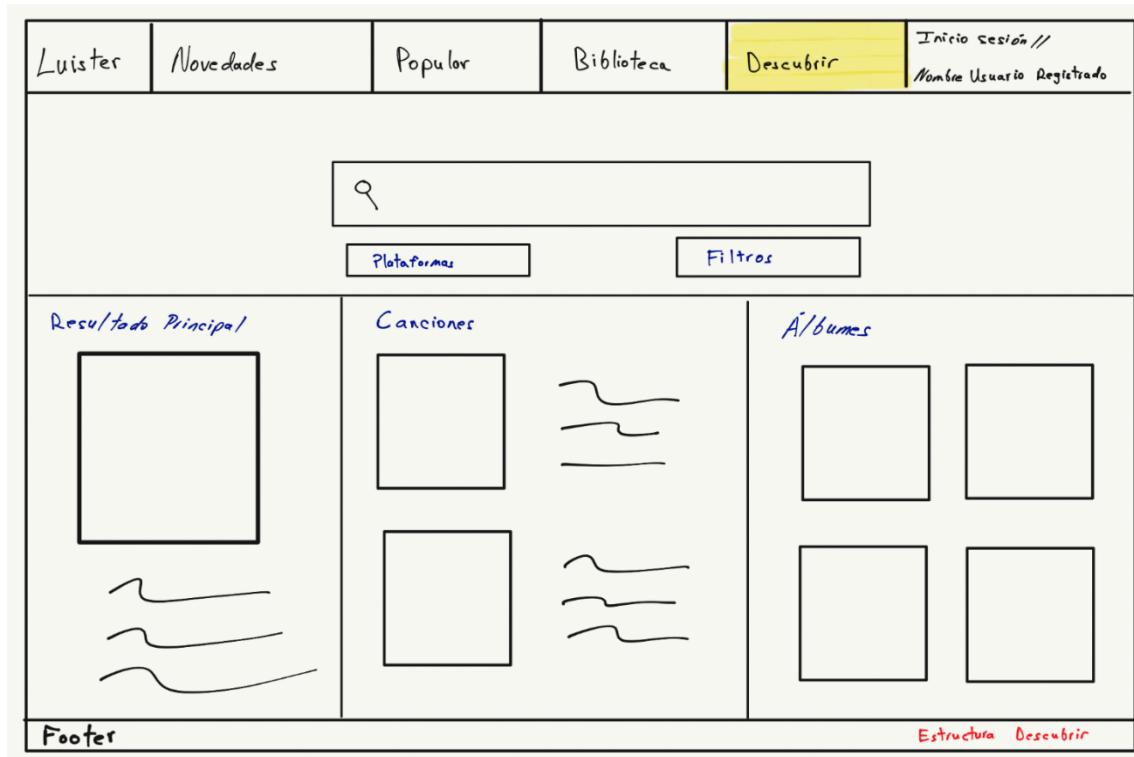


Figura 195. Sketching Detalles

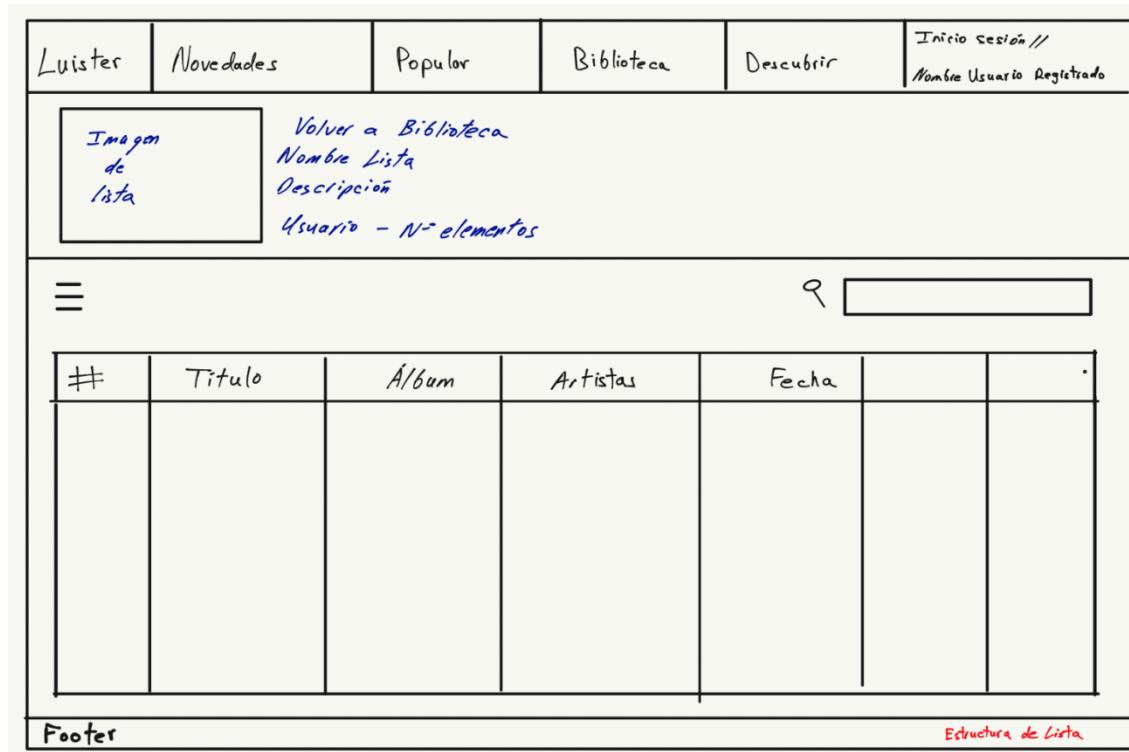


Figura 196. Sketching Lista

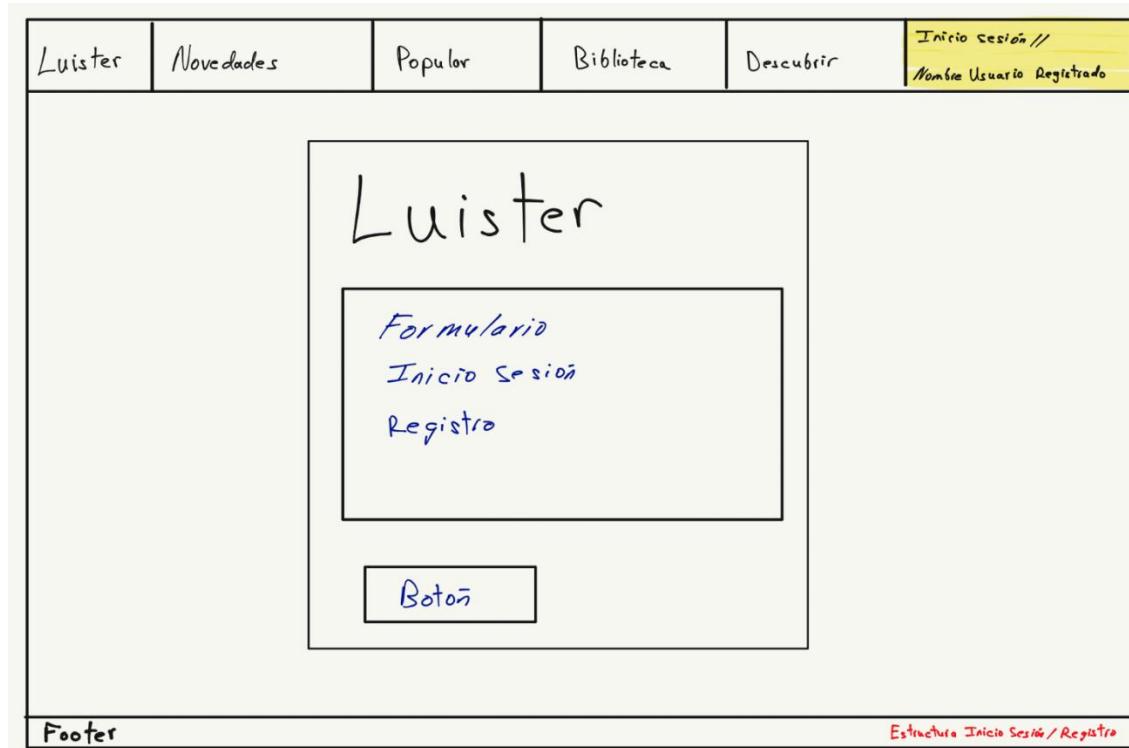


Figura 197. Sketching Login