UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Desarrollo de una web para la evaluación del nivel de ruido generado por los aviones en aeropuertos.

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

AUTOR/A: Ferrando Sanchis, Óscar

Tutor/a: Hernández Orallo, Enrique

CURSO ACADÉMICO: 2022/2023

Resum

Aquest treball descriu el desenvolupament d'una plataforma web per a l'avaluació del nivell de soroll generat pels avions en els aeroports. Es presenta el disseny i desenvolupament de la plataforma, que inclou els diferents models d'avions i les seues característiques acústiques, així com eines de mesura i anàlisi del soroll en temps real. S'espera que la plataforma siga de gran utilitat per a les comunitats properes als aeroports, permetent una millor comprensió i monitorització dels nivells de soroll generats per les aeronaus.

Paraules clau: web, angular, aviació, contaminació acústica, node, docker, REST API, javascript.

Resumen

Este trabajo describe el desarrollo de una plataforma web para la evaluación del nivel de ruido generado por los aviones en los aeropuertos. Se presenta el diseño y desarrollo de la plataforma, que incluye los diferentes modelos de aviones y sus características acústicas, así como herramientas de medición y análisis del ruido en tiempo real. Se espera que la plataforma sea de gran utilidad para las comunidades cercanas a los aeropuertos, permitiendo una mejor comprensión y monitoreo de los niveles de ruido generados por las aeronaves.

Palabras clave: web, angular, aviación, contaminación acústica, node, docker, REST API, javascript

Abstract

This project describes the development of a web platform for the evaluation of noise levels generated by airplanes in airports. The design and development of the platform are presented, including different airplane models and their acoustic characteristics, as well as real-time noise measurement and analysis tools. It is expected that the platform will be very useful for communities near airports, allowing for better understanding and monitoring of noise levels generated by aircrafts.

Key words: web, angular, aviation, noise polloution, node, docker, REST API, javascript

Índice general

Índice de figuras

CAPÍTULO 1
Introducción

Hoy en día no se puede negar la importancia de la contaminación. Entre sus diferentes variantes se encuentra una de un tipo a veces desconocido: la contaminación acústica. Este tipo de contaminación muchas veces no se percibe como muy problemática ya que se suele asociar contaminación a su variante más clásica, la que afecta más directamente al medio ambiente. Si se le pregunta a una persona por contaminación los más probable es que te mencione las emisiones de efecto invernadero o un vertido de petróleo en el mar. Algunos ejemplos de estos últimos tipos podrían ser la contaminación química o la contaminación térmica.

El ruido generado por los aviones puede tener un impacto significativo en la calidad de vida de las personas que viven cerca de aeropuertos. Además, también puede afectar la salud de las personas, ya que puede causar estrés, insomnio y otros problemas de salud. Es por esta razón que es importante medirlo y controlarlo.

La variante que nos incumbe en el presente trabajo es, como se ha mencionado en el párrafo anterior, la acústica. ¿Por qué no se le suele dar la importancia que merece? Posiblemente porque *a priori* no parece tan dañina como las demás. Se puede definir *grosso modo* como el exceso de ruido en el ambiente de forma continuada y que puede llegar a perturbar y afectar a la calidad de vida de la zona donde se dé.

1.1 Motivación

Hay varias posibles motivaciones para crear un proyecto web que informe sobre el ruido generado por los aviones. La primera podría ser la necesidad de brindar información a las personas que viven cerca del aeropuerto sobre el nivel de ruido que pueden esperar. Esto podría ayudar a las personas a prepararse para el ruido y tomar medidas para reducirlo, como cerrar las ventanas, mejorar los aislamientos acústicos de las edificaciones o, en el peor de los casos, utilizar medidas de protección auditiva como tapones para los oídos.

La segunda podría ser la de ayudar a las autoridades locales a monitorizar el ruido generado por los aviones en las zonas cercanas al aeropuerto. Esto podría ayudar a identificar puntos calientes donde el ruido es particularmente alto y tomar medidas para abordar el problema.

Finalmente podría haber una motivación más general de promover la sensibilización sobre el impacto del ruido en el medio ambiente y en la calidad de vida de las personas que viven cerca de un aeropuerto. Un proyecto web que brinde información sobre el ruido generado por los aviones podría ayudar a las personas a comprender mejor el impacto

del ruido y a tomar decisiones informadas sobre cómo vivir de manera más sostenible en un entorno urbano.

1.2 Objetivos

El objetivo principal de este trabajo de final de máster es el desarrollo de una aplicación web que estime la cantidad de ruido que una determinada localización recibirá ante el despegue y la aproximación para el aterrizaje de aviones en las zonas cercanas a los aeropuertos. La aplicación usará los datos de vuelo y propiedades acústicas de los aviones para calcular la cantidad de ruido generado por las aeronaves en una ubicación específica. Para ello se hará uso de una librería externa desarrollada en *Matlab* que se encargará de los cálculos dados los datos de la aeronave. El servicio estará formado por un *frontend* que se encargará de la entrada y posterior visualización de datos y un *backend* que tendrá incluida la librería y al que se le harán las peticiones para obtener los resultados. En los siguientes apartados se profundizará en los en los aspectos más técnicos del trabajo.

En resumen, nuestra aplicación web proporcionará una herramienta útil para medir y controlar el ruido generado por los aviones en una ubicación determinada, lo que puede ayudar a mejorar la calidad de vida de las personas y proteger su salud.

1.3 Estructura de la memoria

La estructura de la memoria será la siguiente:

- El en Capítulo 2 se explicará el contexto y el estado del arte (Sección 2.1) en el que se desarrolla el presente trabajo.
- A continuación (Capítulo 3), se mostrará un breve resumen de las tecnologías y metodologías que han sido utilizadas.
- Después se expondrá un análisis de distintas tecnologías más específicas para su uso en el proyecto (Capítulo 4)
- Se procederá con la explicación del proceso de planificación seguido (Capítulo 5)
- Luego se expondrá el Análisis de Requisitos (Capítulo 6) y la Arquitectura de la aplicación (Capítulo 7)
- Se continuará (Capítulo 8) con el proceso de desarrollo seguido para su implementación.
- En el capítulo 9 se mostrará el proceso de pruebas y el resultado obtenido para asegurar el correcto funcionamiento del servicio.
- En el capítulo 10 se mostrará un caso de uso en un entorno práctico, así como una comparativa con un sistema de medición real.
- Para finalizar se analizará (Capítulo 11) el cumplimiento de los distintos objetivos, relación con los estudios cursados y posibles mejoras.

CAPÍTULO 2
Contexto

Para empezar, se va a explicar el funcionamiento base del proyecto sobre el que descansará el servicio que se quiere implementar. Como se ha mencionado el proyecto se encargará de calcular el ruido generado por aviones tanto en maniobras de aproximación como de despegue. Para ello se hará uso de una librería desarrollada por el tutor de proyecto en *Matlab* que está basada en el modelo de ruido explicado en la sección 5 del artículo *Pollution and noise reduction through missed approach maneuvers based on aircraft reinjection* [1].

El desarrollo de la librería se basa en el análisis del impacto sonoro de las trayectorias de aproximación fallidas mediante el uso un método estándar para calcular los contornos de ruido alrededor de aeropuertos civiles. Se utiliza un modelo simplificado de ruido que considera las características de rendimiento y ruido de las aeronaves. El modelo se fundamenta en las trayectorias de vuelo de las aeronaves y emplea dos métricas comunes para medir el nivel de ruido: el nivel máximo de presión sonora durante el evento (*Lmax*) y el Nivel de Exposición Sonora (*SEL* por sus siglas en inglés), que representa una medida acumulativa de la energía total del sonido en el evento y es el que será usado para los cálculos del proyecto. La metodología se apoya en una base de datos de rendimiento y ruido de las aeronaves ANP, la cual proporciona métricas de ruido en función de la distancia del observador y los ajustes de potencia de la aeronave.

Para facilitar la integración con un servicio web se han generado distintos ejecutables haciendo uso del compilador de *Matlab to C*. En este caso se ha recibido un ejecutable para los sistemas *Windows*, *Linux* y *Mac*:

- WINDOWS: ANCMLibWin64.exe Compiled with DevC (gcc), the project is ANCMLib.dev
- MACARM: ANCMliMacARM Compiled in a terminal using gcc, and makeANCMLibMacARM
- MACIntel: ANCMibMacIntel Compiled in a terminal using gcc, and makeANCMLibMacIntel
- LINUX: ANCMlibLinux Compiled in a terminal using gcc, and makeANCMLibinux

El ejecutable provee de un comando de ayuda para facilitar el uso y explicar los parámetros que son necesarios.

```
1 ./ANCM_Lib_Win64 –H
```

```
oscar@oscar-VirtualBox:~/workspace/plane-noise/docker/plane-noise-server/ANCM_Lib/ANCM_Lib$ ./ANCM_Lib_Linux -H

ANCM_GetNoiseSeg HELP
   ANCM: Aircraft Noise Countour Modelling Library.
   2022 (c) Universidad Politécnica de Valencia.
 Get the noise level from an observer point given the segment of an airplane.
 Use the noise model define by ANC Doc.29, 4th edition, Vol.2
 IMPORTANT: All coordinates and distance are in FEET (ft)

INPUT PARAMETERS
  -S's1x,s1y,s1z;s2x,s2y,s2,s2z;isRolling;epsilon;P;Vseg' Segment data.
    s1x,s1y,s1z;s2x,s2y,s2,s2z; Coordinates of the two points of the segment of the route
    isRolling (1,0): the plane is in takeoff roll or landing roll
    epsilon: bank angle (in Euler angles: phi - roll)
    P: Power
    Vseg: Speed of the segment
    EXAMPLE -S'1.1,1.2,1.3;2.1,2.2,2.3;1;0;7600;150.1'

 -O'o1x,o1y,o1z' Ob: Observer location
    o1x,o1y,o1z; Coordinates of the observer
    EXAMPLE -O'9842.5,1640.4,0'

 -Fpath Path of the NPD file data in csv format.
    EXAMPLE -F./NPD_Data/NPD_data_Test_JETF.csv

 -Nmetric Noise metric (3 chars).
    It depends on the plane, but NoiseMetric can be 'EPN', 'LAm', 'PNL' and 'SEL
    EXAMPLE -NLAm

 -Mmode A char with the mode: 'A' arrival and 'D' descending
    EXAMPLE -MA

 -Vspeed Reference groundspeed for which NPD data are defined
    Normally omitted, so it is set to default value (270.05 ft/s)
    EXAMPLE -V270.05

 -A'WingMounted,TurboFan' Aircraft: information about the aircraft
    WingMounted: 1 (TRUE) -> wing-mounted engine; 0 (FALSE) -> fuselage mounted
    TurboFan: 1 (TRUE): turbo-fan propelled, 0 (FALSE): turbo-prop propelled
    EXAMPLE -A'0,1'

 -c'p,T' Atmospheric conditions to get the impedance adjustament
    Normally omitted, so it is set to default value (105.2,16)
    p: Ambient air pressure at the observer in KPa
    T: Air temperature at the observer in Celsius
    EXAMPLE -C'105.2,16'

 -Td Configure the display of trace (An 8 ubits integer)
    bit 0: Show input parameters and results
    bit 1: Show all geometric data and correction value
    bit 2: Show NPDdata
    EXAMPLE -T7
```

Figura 2.1: Salida del commando *-H*

Como se puede observar en la imagen 2.1, para el cálculo se requieren distintos parámetros.

El primer parámetro, "*-S*", hace referencia a la aeronave y está compuesto por:

- Las coordenadas de los dos puntos del segmento de la ruta. El valor de estas coordenadas está expresado en pies (ft.) y se encuentran en referencia la pista de aterrizaje.
- *isRolling*: si el avión está en despegue o aterrizaje
- epsilon: grado de giro del avión como se observa en al figura 2.2
- *power*: Ppotencia.
- VSeg: velocidad del segmento

El segundo parámetro, "*-O*", es la posición del observador respecto al aeropuerto.

El tercer parámetro, "*-Fpath*", indica el archivo de modelos de ruido que se usará. Depende de la aeronave.

El parámetro "*-NMetric*" indicará el modelo de ruido a usar. Existen varios como se muestra a continuación pero únicamente nos vamos a centrar en el de tipo SEL, como se ha mencionado anteriormente.

- EPN.
- LAm.
- PNL.
- SEL.

El parámetro "*-MMode*" puede ser de dos tipos, "*A*" (*arrival*) si se trata de una aproximación o "*D*" (*departure*) si es un despegue.

El parámetro "*-A*" consta de dos partes. La primera parte indica dónde están montados los motores. Existen dos tipos:

- *Wing mounted*: el motor se encuentra montado en las alas.
- *Fuselage mounted*: el motor se encuentra montado en el fuselaje del avión, cerca de la cola. Suele ser el caso de los aviones privados (*jets* privados).

La segunda parte del parámetro *-A* indica el tipo de motor que se usa. Para este caso existen dos tipos:

- *Turbo prop*: en el caso de ser un motor con hélice.
- *Turbo fan*: en el caso de ser un motor a reacción. Se basan en el principio de comprimir el aire y con una combustión hacen que esta se expande y salga a una velocidad mayor a la de entrada, creando con eso el empuje necesario.

En resumen, la principal diferencia entre los motores de avión de hélice y los motores a reacción es el tipo de tecnología utilizada para generar la fuerza necesaria para impulsar el avión hacia delante. Los motores de hélice utilizan pistones y hélices para generar fuerza, mientras que los motores a reacción utilizan turbinas y compresores para crear un flujo de aire comprimido que se expulsa hacia atrás para generar la fuerza necesaria.

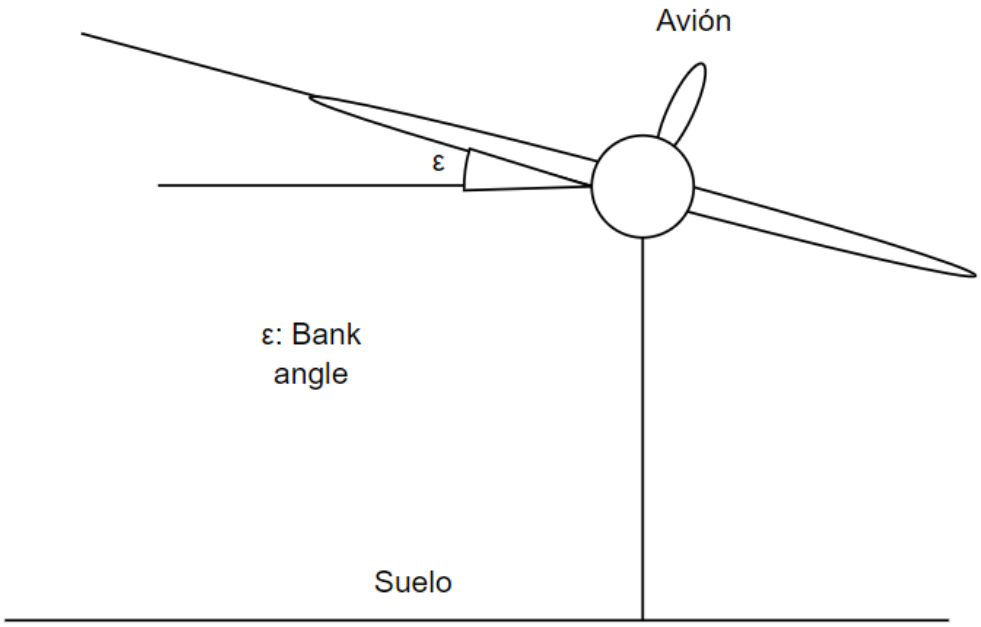

Figura 2.2: Bank angle o inclinación del ala respecto al suelo

El parámetro *"-C'p,T'"* indica las condiciones ambientales del observador. Se suele omitir para el cálculo pero en el caso de usarse, "*p*" indica la presión de aire en el observador en kilopascales (kPa) y "*T*" la temperatura del aire (también en la posición del observador) en grados Celsius (º)

El último parámetro, "*-T*", se encarga de la verbosidad de la salida del binario. Puede ser desde mostrar una salida con todos los datos posibles con la opción "*-T7*" o mostrar una salida donde únicamente se muestren los datos de entrada y el resultado del ruido calculado.

Una vez explicados todos los parámetros se puede hacer uso de la librería para obtener la aproximación del sonido. Para el siguiente caso de uso de prueba se ha usado el ejecutable de *Windows* con los siguientes valores:

- Posición aeronave: 3141.5,0,3.3.
- Posición aeronave 2: 3869.3,0,3.3.
- VSeg: 149.8962.
- Posición del observador: 9842.5,1640.4,0.
- NPD: archivos de pruebas para *jet*.
- NMEtric: SEL.
- MMode: A.
- *Power*: 7500.
- *Bank angle*: 0.
- *isRolling*: 0.

- Tipo de motor: *jet*.
- Montaje de motor: alas.

Al ejecutar el comando:

```
./ANCM_Lib_Linux -S'3141.5,0,3.3;3869.3,0,3.3;0;0;7500;149.8962' -O'9842.5,1640.4,0' -NSEL, -MA, -A'1,1' -FNPD_Data/NPD_data_Test_JETF.csv -T7
```

Muestra la salida de la imagen 2.3.

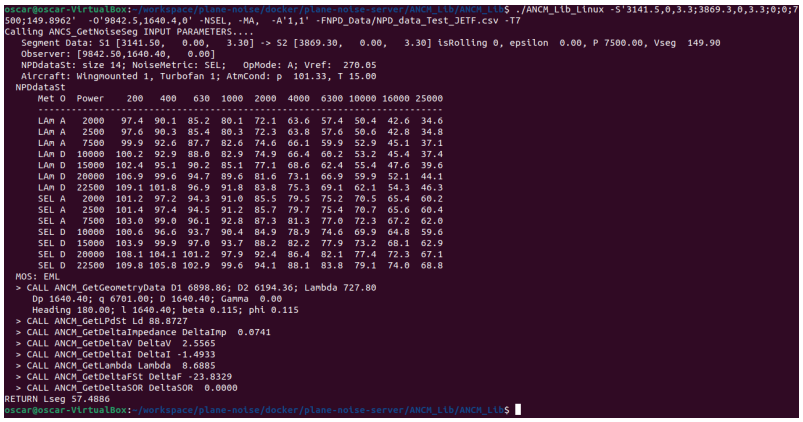

Figura 2.3: Salida de la ejecución de la librería sin filtrar el resultado

En la salida mostrada se observa que hay una gran cantidad de datos que no son necesarios para el servicio que se va a desarrollar. Pero por suerte el último parámetro *"-T"* permite limitar esto. Cambiando la salida de *"-T7"* a *"-T0"* se obtiene un resultado mucho más simple y fácil de utilizar.

```
./ANCM_Lib_Linux -S'3141.5,0,3.3;3869.3,0,3.3;0;0;7500;149.8962' -O'9842.5,1640.4,0' -NSEL, -MA, -A'1,1' -FNPD_Data/NPD_data_Test_JETF.csv -T0
```

Como se puede observar en la imagen 2.4 la salida tiene una menor cantidad de datos y se centra en el valor que se requiere. Este valor será el utilizado en capítulos posteriores y se refinará para que pueda ser usado a través de una *API* correctamente.

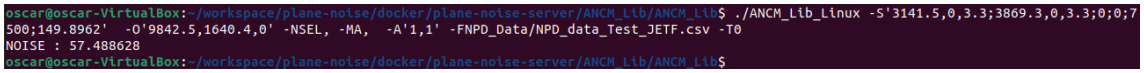

Figura 2.4: Salida de la ejecución de la librería reducida a los datos indispensables

2.1 Estado del arte

En esta segunda sección se van a presentar los distintos servicios que existen actualmente y que proveen de una funcionalidad similar a la que se quiere implementar en el proyecto. La aplicación requerirá del uso de datos de navegación para conocer la posición y de datos específicos de cada aeronave.

Se ha encontrado principalmente un servicio que ofrece datos de ruido de las aeronaves en las proximidades de los aeropuertos.

2.1.1. Webtrack

El sistema *Webtrack* es una herramienta desarrollada por la empresa pública AENA (Aeropuertos Españoles y Navegación Aérea) para la gestión de los mapas de ruido en los alrededores de los aeropuertos. Este sistema es utilizado por AENA para recopilar y analizar datos sobre el ruido generado por las aeronaves y para elaborar mapas de ruido que permitan identificar las zonas más afectadas por el ruido.

El funcionamiento del sistema *Webtrack* se basa en la integración de diferentes tecnologías y sistemas de información geográfica (GIS por sus siglas en inglés). En primer lugar, el sistema recopila datos en tiempo real sobre las operaciones aéreas a través de la información proporcionada por los radares y otros sistemas de seguimiento de vuelos. Esta información y un sistema de micrófonos y cámaras denominado SCVA (Sistema de Control Visual y Acústico) se utiliza para calcular el ruido generado por cada avión durante su trayectoria de vuelo.

A continuación, el sistema utiliza los datos de los niveles de ruido para elaborar mapas de ruido que muestran la distribución del ruido en los alrededores del aeropuerto. Estos mapas se basan en modelos matemáticos que tienen en cuenta factores como la velocidad, la altitud, la dirección del viento que pueden afectar la propagación del sonido.

Una de las principales funcionalidades de *Webtrack* es la posibilidad de acceder a los mapas de ruido a través de internet. Estos mapas están disponibles para el público y pueden ser consultados en línea a través de la página web de AENA. Además, el sistema permite a los usuarios hacer consultas específicas sobre el nivel de ruido en una determinada zona y sobre las medidas que se están llevando a cabo para reducir el impacto del ruido.

Otra funcionalidad importante del sistema *Webtrack* es la capacidad de generar informes y estadísticas sobre los niveles de ruido y sobre el impacto del ruido en la población y el medio ambiente. Estos informes son utilizados por AENA para identificar áreas problemáticas y para tomar medidas para reducir el impacto del ruido en estas zonas.

En resumen, el sistema *Webtrack* es una herramienta avanzada para la gestión de los mapas de ruido en los alrededores de los aeropuertos. Este sistema permite a AENA recopilar y analizar datos sobre los niveles de ruido generados y elaborar mapas que permitan identificar las zonas más afectadas. Además, se trata de un sistema abierto para que los usuarios puedan acceder a los mapas de ruido y hacer consultas específicas sobre el nivel de ruido en una determinada zona.

En la imagen 2.5 se puede observar el funcionamiento del sistema para el aeropuerto de Barcelona-El Prat. Al pulsar unos de los iconos grises (indican las localizaciones de los terminales de monitorización de ruido) se puede obtener los datos de ruido en ese punto. En la ella aparece una medición de 76 dB, que se encuentra por arriba del umbral recomendado por la Organización Mundial de la Salud para gozar de buena salud y bienestar de 65 dB durante el día.

2.2 Problemática

A pesar de que el sistema Webtrack es tremendamente completo y proporciona datos en tiempo real exactos del ruido generado por aviones, tiene un problema claro. El sistema hace uso de micrófonos instalados por distintas zonas alrededor del aeropuerto y ahí radia el principal problema de este sistema. La solución web que se va a desarrollar supera ese problema al no tener que instalar estos micrófonos con el consiguiente ahorro económico que esto supone. Además, en caso de que los resultados sean los esperados y

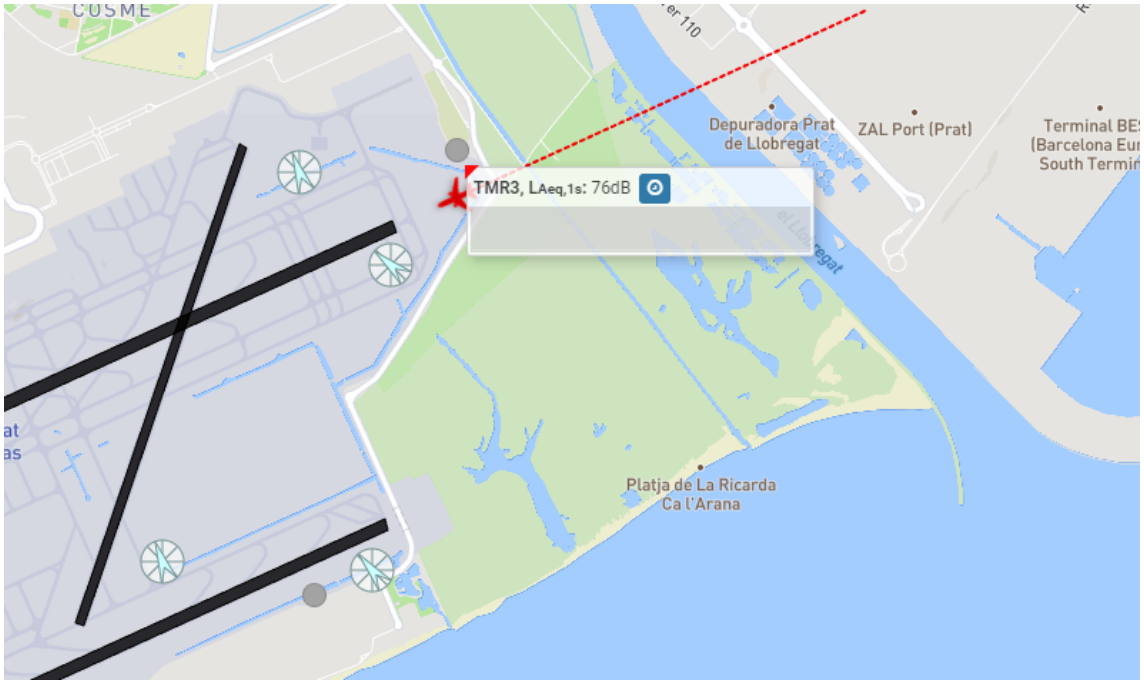

Figura 2.5: Webtrack

estos se asemejen lo suficiente a los recogidos por el sistema *Webtrack* sería posible sustituir este sistema y calcular el ruido en cualquier lugar del mundo sin la necesidad de ir a ese punto en concreto, todo de forma teórica.

CAPÍTULO 3
Metodología y tecnologías

3.1 Metodología

3.1.1. Scrum

Aunque el proyecto ha sido realizado por una persona la metodología seguida para el desarrollo ha sido Scrum [11]. Scrum es una metodología ágil [12] el la que el proyecto se divide en ciclos llamados *sprints*. Cada *sprint* tiene una duración fija, generalmente entre una y cuatro semanas. Para este proyecto se ha establecido una duración de dos semanas por *sprint*. Esto es debido a que se ha considerado que la dedicación de una semana no era suficiente para obtener un incremento potencial del producto a desarrollar.

En cada una de las iteraciones (*sprints*) se han ido realizando las siguientes fases/tareas:

1. Planificación del *sprint*: Marcaba el inicio de la iteración se se seccionaban los elementos que se querían desarrollar.

2. Reunión diaria: Aunque como se ha mencionado, el equipo estaba formado por una sola persona, se realizaba una reunión de seguimiento para comprobar diariamente el estado del desarrollo y comprobar el cumplimiento de los objetivos fijados el día anterior.

3. Revisión del *sprint*: Al finalizar el *sprint* se comprobaba el resultado, si éste había sido el esperado o no y si se habían podido cumplir ciertos objetivos fijados al inicio.

4. Reunión de retrospectiva: Al acabar la revisión se analizaba cómo había sido la forma de trabajar y si se podía mejorar el desempeño. De esta forma, identificando las fortalezas y los errores, se podía mejorar mediante la definición de acciones concretas para futuros *sprints*.

3.2 Tecnologías

3.2.1. Git

Git [26] es un sistema que se usa para el seguimiento de cambios de un proyecto, es decir, para el control de versiones. Este sistema permite llevar un registro de cualquier cambio realizado en un proyecto. Para el desarrollo de este servicio web se ha creado un proyecto de visibilidad privado en el servicio de *GitHub*.

3.2.2. JavaScript

JavaScript [25], es un lenguaje de programación de alto nivel interpretado. Es decir, que no es compilado de forma previa y se interpreta mientras que se ejecuta. Generalmente es usado en el lado del cliente. Su uso está limitado normalmente en los navegadores o lado del cliente, pero también se usa en otros contextos como puede ser el desarrollo de servidores que exponen *APIs*.

3.2.3. Node

Node.js [6] es un entorno de ejecución de código en el lado del servidor. Esto lo hace adecuado para desarrollar servicios *backend*. Al tratarse de un entorno eficiente y de alto rendimiento permite la construcción de servicios rápidos usando *JavaScript*. En este caso se utilizará para el desarrollo de un servidor que expondrá una *API*.

3.2.4. D3 charts

D3.js [21] es una biblioteca de visualización de datos en *JavaScript* que permite crear gráficos dinámicos y animados en la web. *D3.js* se basa en estándares web abiertos como *SVG*, *HTML* y *CSS*, lo que le permite aprovechar las capacidades nativas del navegador para generar gráficos de alta calidad. *D3.js* es muy flexible y potente, y se utiliza ampliamente para crear visualizaciones de datos interactivas y personalizadas en línea, desde gráficos simples hasta complejos paneles de control de datos.

3.2.5. TypeScript

TypeScript [22] es un lenguaje de programación que está basado en *JavaScript*. Una de las ventajas principales del uso de *TypeScript* es, como su nombre indica, el uso de tipos estáticos. Este tipado más fuerte permite la detección de errores en tiempo de compilación que no aparecerían si se usa *JavaScript*. Se usa principalmente para el desarrollo en el lado del cliente y es transpilado a *JavaScript* para poder ser interpretado por los navegadores.

3.2.6. Angular

Angular [7] es un *framework* para el desarrollo de aplicaciones web. *Angular* utiliza como lenguaje de programación *TypeScript*. Una de las principales ventajas de su uso radica en su arquitectura basada en componentes que permiten un mantenimiento y modularidad mejor. Para usar *Angular* por línea de comando se ha instalado la dependencia de *Angular-CLI* [5].

3.2.7. Cypress

Cypress [8] es un *framework* de pruebas de extremo a extremo para aplicaciones web que se integra bien con *Angular* y otras tecnologías de *frontend*. Permite escribir, ejecutar y depurar pruebas automatizadas para aplicaciones web modernas con una sintaxis sencilla y fácil de leer. En las últimas versiones de angular se ha eliminado que por defecto el uso *Protactor* para las pruebas de integración. *Cypress* proporciona una API sencilla para interactuar con la aplicación y se integra bien con herramientas de construcción de proyectos, como *Angular CLI*.

3.2.8. Docker

Docker [9] es una plataforma que permite a los desarrolladores empaquetar aplicaciones y sus dependencias en contenedores ligeros y portátiles. Los contenedores son entornos de ejecución que encapsulan una aplicación y sus dependencias, lo que permite que se ejecute de manera confiable y consistente en diferentes entornos. *Docker* simplifica el proceso de implementación de aplicaciones y mejora la portabilidad y escalabilidad de estas.

3.2.9. Docker Compose

Docker Compose [10] es una herramienta que se utiliza para definir y ejecutar aplicaciones *Docker* de múltiples contenedores. Permite definir todos los servicios necesarios para una aplicación en un archivo *YAML*, incluyendo la configuración de cada contenedor, los volúmenes y las redes que se necesitan para que los contenedores se comuniquen entre sí.

3.2.10. Jest

Jest [19] es una herramienta de pruebas para *JavaScript* que se utiliza para realizar pruebas unitarias y de integración en aplicaciones. Es una herramienta gratuita, de código abierto y fácil de usar.

Jest se integra fácilmente con *Node.js* y *Express*, y proporciona una amplia variedad de características útiles para probar una *API* de *Node.js*. Por ejemplo, es capaz de realizar pruebas asíncronas de forma sencilla, lo que es fundamental para probar una *API* que realiza llamadas a servicios externos o interactúa con una base de datos.

CAPÍTULO 4
Análisis de tecnologías

En el siguiente capítulo se van a proceder con una comparativa de tecnologías más específicas para el proyecto y sobre las que no se tenía un conocimiento al inicio del mismo. Para poder elegir cuales serán las candidatas a usar se realizará un análisis comparativo de las características de cada una. El capítulo se dividirá en dos secciones, una primera destinada al análisis de varios servicios para la representación de mapas en una web y una segunda sección que se encargará de comparar las distintas *API* de navegación que proporcionan datos en tiempo real de las distintas aeronaves.

4.1 Representación de mapas

4.1.1. Google Maps

Posiblemente lo primero que se le ocurre a alguien cuando piensa en un mapa es *Google Maps* [23]. La *API* de *Google Maps* es una herramienta poderosa que ofrece a los desarrolladores una amplia gama de funcionalidades para integrar en sus aplicaciones y sitios web. Esta *API* puede ser utilizada en aplicaciones *Angular* para agregar mapas, rutas y otros elementos geoespaciales.

Algunas de las funcionalidades que ofrece la *API* de *Google Maps* son por ejemplo, la visualización de mapas personalizados, la búsqueda de lugares cercanos y la creación de rutas de navegación. Además, la *API* permite a los desarrolladores crear marcadores, agregar información de ubicación y proporcionar herramientas de navegación avanzadas para los usuarios.

La *API* de *Google Maps* está disponible en varios planes de precios. El plan gratuito es adecuado para proyectos de bajo tráfico y pequeñas empresas que necesitan integrar mapas en su sitio web como podría ser el caso de este trabajo. Este plan incluye un crédito mensual gratuito de 200 $ que se puede utilizar para consumir los servicios de la *API*. Esto equivale a aproximadamente 28500 cargas de mapas al mes y por lo tanto, más que suficiente para el proyecto que se va a desarrollar.

Para proyectos de mayor envergadura, *Google Maps* ofrece el plan *Premium* de la *API*, que proporciona acceso a herramientas y servicios avanzados para la integración de mapas y rutas en aplicaciones de alto tráfico. Este plan ofrece una mayor cantidad de créditos mensuales y acceso a servicios adicionales, como el soporte técnico 24/7 y la personalización avanzada de mapas.

La integración de la *API* en aplicaciones *Angular* es relativamente sencilla. Los desarrolladores pueden utilizar las bibliotecas de la *API* de *Google Maps* para *TypeScript* e

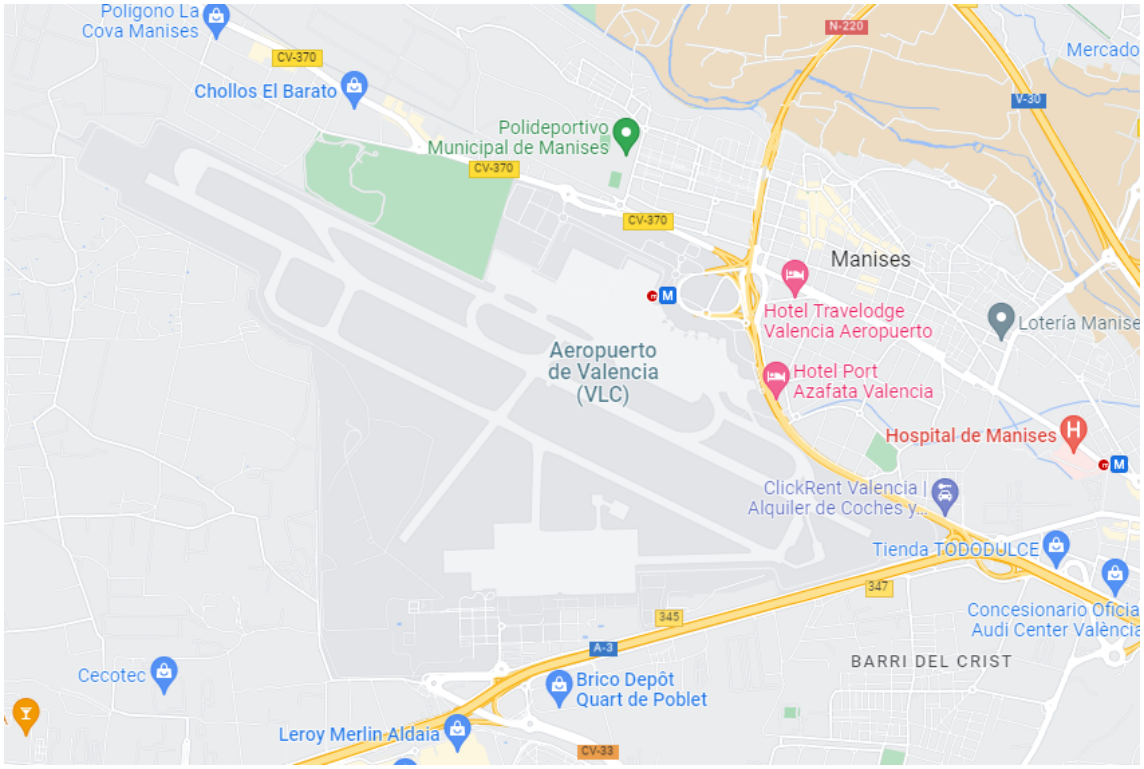

Figura 4.1: Previsualización de un mapa generado por *Google Maps*

incrustar el mapa en su aplicación. Además, es posible personalizar los estilos y el diseño de los mapas utilizando *CSS* y *HTML*.

En conclusión, se trata de una herramienta muy potente y fácil de usar para la integración de funcionalidades geoespaciales donde es posible aprovechar las muchas funcionalidades que provee para agregar mapas personalizados en las aplicaciones web.

En la imagen 4.1, se puede observar un ejemplo de uso mostrando el aeropuerto de Valencia.

4.1.2. Open Layers

La *API* de *OpenLayers* [24] es una herramienta que permite agregar mapas interactivos y funcionalidades geoespaciales avanzadas en sus aplicaciones web. Esta *API* es de código abierto y gratuita, lo que la hace una alternativa atractiva para aquellos que no quieren pagar por el uso de la *API* de *Google Maps*.

Entre las funcionalidades que ofrece la *API* de *OpenLayers* se encuentran la generación de mapas en la web y la creación de rutas de navegación. Además, esta *API* es altamente personalizable, lo que permite a los desarrolladores agregar capas personalizadas y utilizar diferentes fuentes de datos.

En cuanto a los precios, la *API* de *OpenLayers* es de código abierto y gratuita, lo que significa que los desarrolladores pueden utilizarla sin costo alguno. Además, como es de código abierto, los desarrolladores tienen acceso al código fuente, lo que les permite personalizarla según sus necesidades.

En conclusión, *OpenLayers* es una alternativa de código abierto para aquellos que desean agregar mapas interactivos en sus aplicaciones. Además, al ser gratuita, no hay ningún coste asociado a su uso.

Figura 4.2: Previsualización de un mapa generado por *OpenLayers*

En la imagen 4.2, se puede observar un ejemplo de uso de *OpenLayers* mostrando el aeropuerto de Valencia.

4.1.3. Conclusión

Después de unas cuantas pruebas de campo se ha optado por el uso de *Google Maps*. El principal motivo de esta elección es que, a pesar de no ser una alternativa completamente gratuita, tiene mayor facilidad de uso en la interacción con el mapa y el uso de marcadores, lo que la convertía en una mejor opción. Por el contrario *OpenLayers*, siendo gratuita, a supuesto un reto el poder integrarla en la aplicación. Además, la cantidad de usos sin coste que permite *GoogleMaps* es suficiente para el alcance del trabajo.

4.2 APIS de navegación

Otro de los puntos clave de la aplicación es la obtención de datos de posición, velocidad y modelo de los aviones en tiempo real. Para ello es necesario el uso de una *API* que devuelva estos datos. Se han encontrado diversas *API* que proveen estos datos y se va a proseguir con un detalle de los resultados encontrados y la elección de la que se va a usar finalmente.

4.2.1. Aviationstack

La *API* de *Aviationstack* [14] es una herramienta que proporciona datos en tiempo real sobre vuelos comerciales en todo el mundo, incluyendo información sobre aeropuertos, rutas, horarios de vuelos y más. Esta *API* es de pago y ofrece diferentes planes de suscripción según las necesidades del usuario.

Entre las funcionalidades que ofrece la *API* de *Aviationstack* se encuentran la búsqueda de información de vuelos específicos, seguimiento de vuelos en tiempo real y más.

Además, esta *API* ofrece una amplia documentación y recursos en línea para ayudar a los desarrolladores a integrarla en sus aplicaciones.

En cuanto a los precios, la *API* de *Aviationstack* ofrece diferentes planes de suscripción, desde un plan gratuito hasta planes pagos con diferentes niveles de funcionalidad y acceso a datos. Los precios varían según el plan y el volumen de solicitudes. El mayor problema es que su plan gratuito únicamente ofrece 100 peticiones al mes.

4.2.2. Flighaware

La *API* de FlightAware [15] es una herramienta que proporciona información sobre vuelos en tiempo real, incluyendo datos de posición, estado de vuelo, horarios de llegada y más. Esta *API* ofrece diferentes planes de suscripción que varían en precio y funcionalidad.

En referencia a los precios, la *API* de *FlightAware* tiene también diferentes planes de suscripción, con un plan gratuito hasta diferentes planes de pago. Ofrece un plan de 5$/mes gratuitos. Esto equivale a unas 1000 peticiones al mes.

4.2.3. OpenSky

La *API* de *OpenSky* [13] es una herramienta que proporciona información en tiempo real sobre el tráfico aéreo en todo el mundo. Esta *API* ofrece datos de posición, estado de vuelo, horarios de llegada y más.

Entre las funcionalidades que ofrece la *API* de *OpenSky* se encuentran la búsqueda de información de vuelos, seguimiento de vuelos en tiempo real, integración con sistemas de reservas y más. Además, esta *API* ofrece una amplia documentación y recursos en línea para ayudar a los desarrolladores a integrarla en sus aplicaciones. Una de las funcionalidades más interesante de esta *API* es la capacidad de filtrar para un espacio aéreo concreto, permitiendo acotar la cantidad de aeronaves a identificar al área para la que se quiere obtener el ruido. La *API* de *OpenSky* ofrece un plan gratuito que permite hasta 4000 solicitudes por día, lo que puede ser suficiente para pequeñas aplicaciones. Sin embargo, para aplicaciones más grandes, se pueden adquirir planes pagos que ofrecen un mayor número de solicitudes.

4.2.4. Cirium Flight Stats

La *API* de *Cirium FlightStats* [16] es una plataforma de datos en tiempo real y de históricos de vuelos para la industria de viajes y transporte aéreo. Proporciona información precisa y detallada sobre vuelos, aeropuertos y aerolíneas, incluyendo horarios de vuelos, retrasos, cancelaciones, rutas, itinerarios y más. Esta *API* es utilizada por aerolíneas, aeropuertos, agencias de viajes y otros proveedores de servicios de viajes para mejorar la experiencia del usuario, proporcionar información en tiempo real y mantener a los viajeros informados en todo momento. Con esta *API*, se pueden construir aplicaciones de seguimiento de vuelos, aplicaciones de reservas de viajes, paneles de control de información de vuelos, entre otras. Una de las funcionalidades que mejor se puede adaptar al trabajo es la capacidad de filtrar vuelos activos para un aeropuerto destino. De esta forma, como con *OpenSky*, es posible reducir el numeró de aviones a únicamente los estrictamente necesarios.

En cuanto al precio, la *API* de *Cirium FlightStats* ofrece una variedad de planes de suscripción para adaptarse a las necesidades de cada empresa o desarrollador. El costo depende de varios factores, como la cantidad de solicitudes de *API*, el volumen de datos

que se solicita, la frecuencia de actualización de datos, entre otros. El plan gratuito que ofrece es para un mes y a partir de entonces se paga por el consumo que se haga de la *API*.

4.2.5. OAG Fligh Status Data

La *API* de *OAG Flight Status* [17] Data proporciona datos de vuelo en tiempo real, que incluyen detalles sobre los vuelos, como la hora de salida y llegada, el número de vuelo, el aeropuerto de origen y destino, la duración del vuelo, la frecuencia de vuelo y el estado de vuelo actual, entre otros datos. La *API* recopila información de varias fuentes, incluyendo la información de los sistemas de control de tráfico aéreo, y se actualiza en tiempo real, lo que la convierte en una herramienta valiosa para aquellos en la industria de la aviación.

Dado que no ofrece los datos básicos que se requieren para la implementación e la aplicación, es decir, los datos de posición de la aeronaves en tiempo real, se descarta su uso.

4.2.6. AeroDataBox

La *API* de *Aerodatabox* [18] es una plataforma de datos de aviación en tiempo real que proporciona información detallada sobre aeropuertos, aerolíneas, entre otros datos relacionados con la aviación. Con esta *API*, los desarrolladores pueden acceder a información actualizada sobre la industria de la aviación y utilizarla para mejorar la experiencia del usuario, construir aplicaciones de seguimiento de vuelos entre otras aplicaciones.

Esta *API* tiene diferentes planes de suscripción, que van desde un plan gratuito hasta planes de suscripción de pago que ofrecen diferentes niveles de acceso y características adicionales. Los planes de suscripción de pago comienzan en 5$ por mes y ofrecen características adicionales, como acceso a datos históricos, datos de vuelos en tiempo real, y mayor cantidad de peticiones. El plan gratuito por otra parte únicamente ofrece 100 peticiones al mes.

4.2.7. Resultados

En la siguiente tabla se puede observar una tabla comparativa de los datos necesarios que se pueden obtener a través de estas *API*. En esta tabla se puede ver que casi todas las *API* proporcionan información sobre la longitud, latitud, modelo de avión, altitud y velocidad de los aviones, excepto *OAG Flight Status*. Todas estas *API* son de pago, excepto la *OpenSky REST API* que aunque tiene una versión de pago la cantidad de 4000 peticiones gratuitas al día es suficiente para el alcance del trabajo. Después de analizar los resultados se ha optado por el uso de *The OpenSky Network* [2], que es la encargada de gestionar la *API* de *OpenSky*. El uso de datos de *OpenSky* obliga a citar la publicación anterior en caso de desarrollar una página web como es el caso. Se pueden obtener todos los datos necesarios y además es gratuita (permite 4000 llamadas a la *API* al día). Además, permite filtrar por espacio aéreo permitiendo seleccionar un rectángulo en cielo y de esa forma acotar las aeronaves a un aeropuerto, muy conveniente para el desarrollo del trabajo. De las otras opciones la más sencilla de usar y la que mejores datos proporcionaba era *AviationStack* pero se han tenido problemas para la recepción de datos en vivo de los aviones. Ese campo de la respuesta siempre venía con valor *null*. Se intentó contactar con el soporte para poder resolverlo, pero la respuesta fue que no se daba soporte al plan

gratuito. Además, la versión gratuita únicamente permitía 100 peticiones al mes y para aumentar este número a 10000 había que pagar 50$ mensuales.

API	OpenSky	AeroDataBox	AviationSky	FlightAware	Cirium	OAG
Longitud	Sí	Sí	Sí	Sí	Sí	No
Latitud	Sí	Sí	Sí	Sí	Sí	No
Altitud	Sí	No	Sí	Sí	Sí	No
Modelo	Indirectamente	No	Sí	Sí	Sí	Sí
Velocidad	Sí	No	Sí	Sí	Sí	No
Aeropuerto origen	No	Sí	Sí	Sí	Sí	Sí
Aeropuerto destino	No	Sí	Sí	Sí	Sí	Sí
De pago	No	Sí	Sí	Sí	Sí	Sí

Tabla 4.1: Tabla comparativa *APIs* de navegación

CAPÍTULO 5
Planificación

Para la realización del proyecto se ha requerido de una planificación. En esta sección se detalla una aproximación del tiempo y dinero que se va a invertir en cada una de las fases del proyecto. Dado que el proyecto a ha sido llevado a cabo por una persona toda la carga de trabajo recaerá sobre la misma persona.

El proyecto se va a dividir en diferentes fases:

- Preparación
  - 1 semana.
  - Preparar el entorno de desarrollo y la máquina virtual en la que se desarrollará el proyecto.
- Análisis de requisitos
  - 1 semana.
  - Tener un entendimiento claro del problema que se va a resolver y establecer de forma clara el alcance, restricciones y requisitos que van a tener.
- Análisis de las distintas tecnologías
  - 1 semana.
  - Dado que no se tiene un conocimiento específico sobre varias de las tecnologías que se van a utilizar, es necesario destinar parte del tiempo a la realización de pruebas para establecer cuáles serán las seleccionadas para su uso en el prototipo final.
- Arquitectura
  - 3 días.
  - Hay tener una idea clara de la forma que va a tener el proyecto y el diseño de la base sobre la que se va a construir el producto.
- Desarrollo de un primer prototipo
  - 6 semanas.
  - Implementación de la primera prueba de concepto funcional.
- Prototipo final
  - 3 semanas.

    - Segunda iteración que tiene como objetivo el refinamiento de la prueba de concepto para la obtención de un producto más profesional y completo.

- Pruebas unitarias
    - 1 semana.
    - Se van a implementar las pruebas necesarias para asegurar el correcto funcionamiento del prototipo y comprobar que cumple con los requisitos descritos.
- Pruebas de servicio
    - 3 días.
    - Se va a dedicar parte del tiempo a realizar pruebas sobre el prototipo de forma empírica para comprobar su funcionamiento y usabilidad.
- Redacción de la memoria
    - 4 semanas.
    - Tiempo invertido en la redacción y comprobación del documento de memoria que se tiene que entregar.
- Imágenes y diagramas
    - 1 semana.
    - Obtención y creación de las distintas imágenes y diagramas que se van a usar durante la memoria.

El presupuesto destinado a la realización del trabajo serían las horas que se han presupuestado. Dado que se ha estimado una duración estimada de 18 semanas, el número de horas sería de 1080. A un precio de 15€/hora, da un total de 16.200€ de gasto. Como el proyecto ha sido desarrollado en el contexto de un trabajo de final de máster por el alumno, el coste real efectivo es de 0€.

CAPÍTULO 6
Análisis de requisitos

En el transcurso de este capítulo, se detallarán los distintos requisitos del sistema, tanto los funcionales como los no funcionales, siguiendo las pautas establecidas por el estándar *IEEE 830* [28]. Este estándar proporciona una serie de prácticas y pasos para una adecuada especificación de requisitos, los cuales se presentarán a continuación.

Propósito

El servicio web que se he desarrollar tiene como función principal visualización del ruido que generan las aeronaves en relación con un punto de observación determinado. De esta forma se puede mejorarla calidad de vida de una zona afectada por este tipo de contaminación acústica. Para darle un nombre comercial se ha optado por *Plane-Noise*, que de forma simple y concisa identifica el propósito general del servicio web.

Alcance

Se debe desarrollar y diseñar el servicio web *Plane-Noise*. La herramienta proporcionará una visualización del ruido generado en las proximidades de un aeropuerto en relación a un punto específico donde se ubicará un observador.

6.1 Descripción general

En esta primera sección se presentan de manera general las principales funcionalidades del sistema.

Perspectiva del producto

Plane-Noise, tiene como característica poder observar de una forma visual el ruido generado tanto en tiempo real mediante gráficas como en un instante determinado. El diseño del sistema *Plane-Noise* permitirá su funcionamiento tanto en entornos web como en dispositivos móviles, con el objetivo de brindar una experiencia rápida, eficiente y fácil de usar.

Funciones del producto

- Visualización del ruido generados por los aviones en la proximidad del aeropuerto.
- Validación de los datos enviados al servidor.
- Facilidad de uso.

Restricciones

Se restringe el alcance del proyecto para tener una visión clara a los siguientes puntos:

- La herramienta se podrá usar desde un navegador web y dispositivos móviles.
- Tecnologías: *Docker*, *Node* y *Angular 14*.

Suposiciones y dependencias

Se requiere que también se tengan en cuenta los siguientes requisitos:

- Es necesaria la disponibilidad y correcto funcionamiento tanto del servidor como de la *API* de navegación de *OpenSky* para que la aplicación web funcione.
- El producto está diseñado para ser utilizado en todos sus aspectos por usuarios sin conocimientos técnicos, por lo tanto, se enfoca en ofrecer una experiencia de uso intuitiva y sencilla.

6.2 Requisitos específicos

6.2.1. Interfaces

Interfaces de usuario

Se debe poder observar de forma gráfica el ruido generado por las aeronaves.

Requisitos de software

Se requiere del desarrollo de una *API* que permita el cálculo del ruido y para ello es necesario que esta se integre con la librería de *Matlab*.

Interfaces de comunicación

Para administrar las solicitudes a los servicios web y el envío de datos, se utilizará el protocolo *HTTP*. En el caso de la transmisión de datos sensibles, se empleará su versión más segura, *HTTPS*.

6.2.2. Funcionales

Son los encargados de mostrar los distintos comportamientos del sistema que se va a desarrollar. Para el servicio se han identificado los siguientes:

- Seleccionar aeropuerto: Mediante un selector se podrán seleccionar los distintos aeropuertos de España para los que se quieren obtener las coordenadas.
- Transformar las coordenadas a puntos *x,y,z* cartesianos: Dado que la *API* de navegación proporciona las coordenadas en latitud/longitud se tendrá que convertir al formato específico que requiere la librería.
- Obtener las coordenadas de un rectángulo para acotar la superficie de búsqueda de la *API* a una zona concreta: Cuando la distancia en muy grande el cálculo de ruido deja de tener sentido ya que es 0. Se deberá acotar la búsqueda a un área pequeña.
- Centrar mapa en el aeropuerto seleccionado: Para facilitar la usabilidad, el mapa se tendrá que centrar una vez se seleccione un aeropuerto.
- Marcar puntos en el mapa para calcular el ruido: Se podrá marcar en el mapa un punto para el que se calculará el ruido.

- Prevenir puntos de observador erróneos: Si se intenta seleccionar un punto muy alejado del aeropuerto para el que el ruido no es relevante no se marcará en el mapa.
- Calcular la acumulación de decibelios de los distintos aviones. Dado que puede haber varios aviones en maniobras de despegue o aterrizaje se tendrá que calcular el cómputo de *dB* de todos ellos.
- Mostrar una gráfica con el ruido actual para permitir la observación y evolución en tiempo real.
- Mostrar en una tabla todas las aeronaves que se encuentren en el espacio aéreo seleccionado. Para cada aeronave se mostrarán los siguientes datos:
  - *Callsign*
  - Latitud.
  - Longitud.
  - Altitud.
  - Velocidad.
  - Modelo del avión.
  - Tipo de motor.
  - Montaje de motor.
  - Distancia al observador.
  - Ruido generado.

Requisitos de rendimiento

- El tiempo de ejecución completo del cálculo ha de ser inferior a 6 segundos.
- El dibujado de la gráfica deberá ser en tiempo real con actualización cada 1 segundo.

6.2.3. Atributos

Disponibilidad

A nivel de usuario se establecerá un acuerdo de nivel de servicio (SLA) que garantice una disponibilidad superior al 99 %. En situaciones en las que el servicio no esté disponible, se responderá con un estado HTTP 503 (Servicio no disponible).

Portabilidad

Es necesario que el cliente se pueda integrar con otras aplicaciones. Además, la interfaz se adaptará de forma automática a la resolución y tamaño de las pantallas en las que se ejecute.

Verificación

Con el fin de garantizar la calidad y el adecuado funcionamiento del sistema, se llevarán a cabo pruebas automáticas como parte del proceso de desarrollo.

CAPÍTULO 7
Arquitectura

Teniendo ya los requisitos funcionales listos en el siguiente apartado se va a explicar el diseño y arquitectura que tendrá la aplicación.

La arquitectura se ha basado en un sistema de componentes. Aparecen principalmente dos componentes, un servidor y un cliente.

La parte del servidor será la encargada de recibir los datos del *frontend* web y, a partir de estos, se devolverá la aproximación del ruido generado por la aeronave que ha calculado la librería de cálculo del ruido *noise_library* integrada en el servidor.

La segunda parte consistirá en el cliente web desarrollado en *Angular* que, haciendo uso de la *API* de aviación mencionadas anteriormente, recabará los datos y los preparará para enviarlos en el formato que requiere la librería *Matlab*. El ruido generado por la aeronave se mostrará de forma visual en el cliente web .

En la imagen 7.1 se puede observar la forma de comunicación anteriormente mencionada donde se muestra el servidor de la aplicación y el cliente. Como se puede apreciar el servidor aparece rodeado de una capa adicional de *Docker* que facilita la portabilidad de este. Este apartado se explicará con mayor profundidad durante la sección 8, Desarrollo. Además, también se puede ver cómo será la *API* de *OpenSKy* la que proveerá de los datos de navegación y posición al cliente. La comunicación entre el cliente y el servidor se llevará a cabo mediando una *API* expuesta por parte del servidor a la que se le pasarán ciertos parámetros que requiere la librería *nosie_library* de *Matlab*.

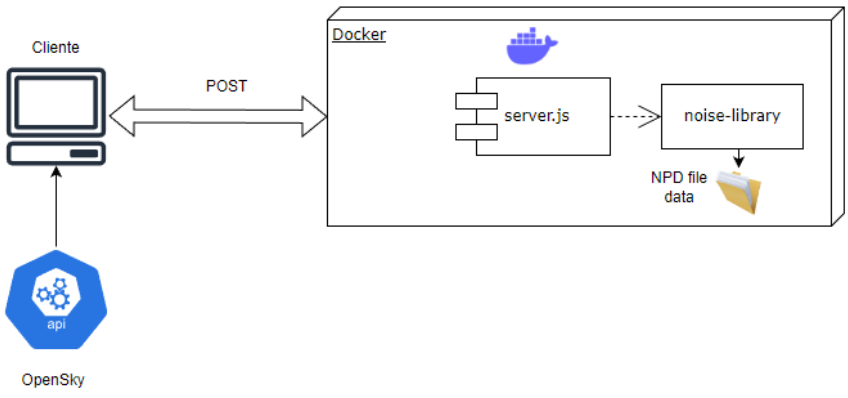

Figura 7.1: Arquitectura

CAPÍTULO 8
Desarrollo

En el siguiente apartado se va a exponer el desarrollo de la aplicación, el servidor y sus diferentes fases. Dado que se trata de un servicio web que muestra datos, primero se ha optado por realizar una prueba de concepto (*POC* por sus siglas en inglés). De esta forma se podía conseguir un prototipo funcional con los requisitos mínimos de uso y en una segunda iteración, una vez estuviese el POC terminado, se le podría añadir y mejorar el apartado visual mediante el uso de un mapa interactivo de *Google Maps* para mostrar la posición del observador y mejorar la usabilidad.

8.1 Servidor

El primer paso del desarrollo del proyecto consiste en la implementación de un servidor que sea capaz de usar la librería que se ha proporcionado para el cálculo de ruido.

El servidor tendrá una arquitectura como la mostrada de la imagen 8.1 que se irá explicando a lo largo del apartado, así como su funcionamiento.

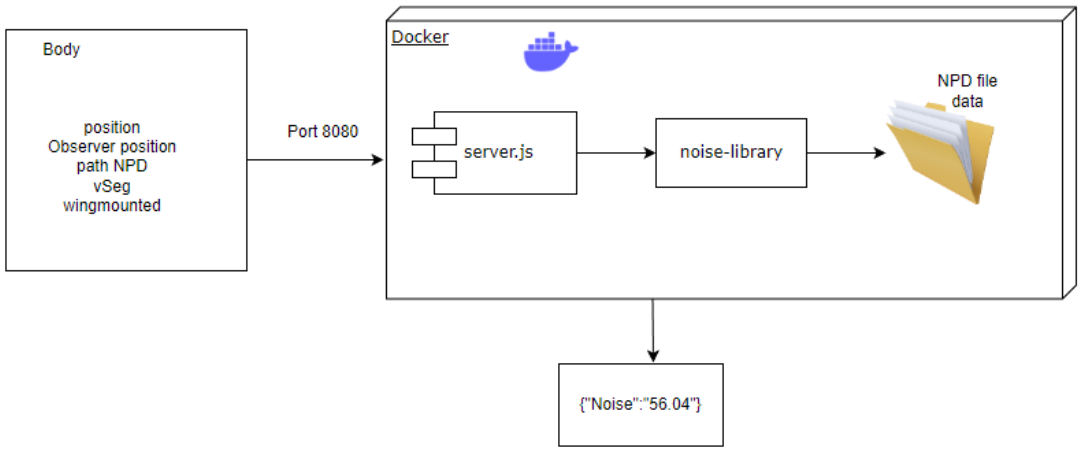

Figura 8.1: Arquitectura servidor

El primero obstáculo del desarrollo del servidor aparece porque no existe una *API* que se pueda usar para interactuar con la librería directamente, sino que al tratarse de un ejecutable compilado para diferentes entornos (*Linux*, *MacOS* o *Windows*), únicamente se puede usar por la línea de comandos.

La solución a este problema se puede solventar encapsulando el servidor dentro de una API que se encargue de hacer una llamada interna a la librería. Para esto se ha usado una de las tecnologías mencionadas en el apartado 3.2, *Node*.

Mediante el uso de *Node* con *Express* se ha creado un *API REST* que se encarga de recibir los datos que obtendrá el cliente 8.2. Esta *API* tiene un servicio *REST POST* llamado */api/calculate-noise* al que se le pasarán los siguientes datos en el cuerpo (*body*) de la petición:

- Coordenadas del avión.
- Coordenadas del avión al final del segmento.
- *isRolling* = 0
- *epsilon*(*bank angle*) = 0
- *Power*.
- *vSeg*.
- Posición del observador.
- *wingMounted*.
- *turbofan*.
- *NPD_ID* del avión para el modelo de sonido.

Dado que uno de los parámetros de entrada del ejecutable es un archivo *CSV* con los *ANP data* de las aeronaves, es necesario saber el NPD_ID del avión que se relaciona con ese archivo. Para obtener estos archivos se ha usado la página web `https://www.aircraftnoisemodel.org/` y se han descargado todos los modelos de ruido de todas las aeronaves de la base de datos. Estos datos venían juntos en un mismo *CSV* completo que ha tenido que ser troceado con los datos pertenecientes a cada aeronave. Esto se ha hecho mediante un *script* en *Python* que se puede ver en el apéndice B y donde se explica su funcionamiento.

En el siguiente bloque se puede ver la forma en la que ha implementado la servicio *POST* en el archivo *server.js*:

```
app.post('/api/calculate-noise', (req, res) => {
  body = req.body
  isErrorBodyResponse = isErrorBody(body)
  if (isErrorBodyResponse[0]) {
    console.log(isErrorBodyResponse[1])
    res.status(400).send({ error: isErrorBodyResponse[1] });
    return;
  }
  isRolling = '0'
  epsilon = '0'
  sParam = body.planeCoords1 + ';' + body.planeCoords2 + ';' + isRolling + ';'
      + epsilon + ';' + body.power + ';' + body.vSeg
  oParam = body.observerCoords
  aData = body.wingMounted + "," + body.turboFan
  command = "./ANCM_Lib_Linux -S'" + sParam + "'  -O'" + oParam + "' -N" + body
      .nMetric + ", -M" + body.mMode + ",  -A'" + aData + "' -FNPD_Data/" +
      body.npd + ".csv -T0"
  console.log(command)
  exec(command, (err, stdout, stderr) => {
    if (err) {
```

```
18        // Error handle
19        res.status(400).send({ error: err.message });
20        return;
21      }
22      splitted = stdout.split(":")
23      console.log(splitted)
24      calculatedNoise = splitted[1].trim()
25      console.log("Noise calculated for plane is: '" + calculatedNoise + "'")
26      res.send({ noise: calculatedNoise });
27    });
28  });
```

Para comunicarse con el ejecutable dado que el proyecto se ha desarrollado en *Linux* se ha hecho uso de la función *exec*. El ejecutable se encuentra dentro de la misma carpeta donde está el archivo *server.js*.

Como se ha mencionado se hace uso de la función *exec* y ésta, si se usa de manera incorrecta, puede presentar varios riesgos. Algunos de estos riesgos pueden ser:

- Inyección de comandos: si se permite que los usuarios ingresen argumentos directamente en la función *exec*, puede ser vulnerable a ataques de inyección de comandos, lo que podría permitir a los atacantes ejecutar comandos maliciosos en el sistema.
- Seguridad del sistema: si se ejecutan comandos peligrosos sin restricciones, puede comprometer la seguridad del sistema. Por ejemplo, si se ejecuta un comando para eliminar archivos del sistema, podría eliminar archivos importantes y causar daños irreparables.
- Rendimiento: la función *exec* puede ser costosa en términos de rendimiento, ya que puede ser necesario cargar todo el entorno del sistema operativo cada vez que se ejecuta un comando.

Para mitigar estos riesgos, se recomienda tener precaución al utilizar la función *exec* en *Node.js*. Se debe validar cuidadosamente cualquier entrada de usuario y asegurarse de que solo se ejecuten comandos seguros y necesarios. En el apartado de Pruebas (capítulo 9), se muestra cómo se validan los campos de entrada para aceptar únicamente caracteres o números en cierto formato y así minimizar el riego que se pueda materializar alguna de los casos anteriormente descritos. Aunque la aplicación web no hace uso de datos críticos o personales, cosa que minimiza la exposición ante una brecha de seguridad, no por ello hay que no tenerlo en cuenta durante el desarrollo ya que la seguridad es uno de los pilares de cualquier servicio.

Como segunda parte del desarrollo del servidor se ha encapsulado el mismo dentro de un contenedor de *Docker*. En el apartado anterior se ha mencionado que la librería se ha compilado para distintos entornos. Esto implica una gran dependencia o acoplamiento del entorno donde se ha va a ejecutar el servidor. Si se hace uso del ejecutable *Windows* únicamente se podrá ejecutar el servidor en un entorno *Windows* y así sucesivamente con las demás compilaciones. Para solventar este problema se ha hecho uso de *Docker*. De esta forma se consigue aislar el servidor en un entorno completamente portable y en el que únicamente se depende del tipo de compilación del ejecutable para la creación del contenedor de *Docker*. Una vez creado este se puede mover libremente y se convierte en totalmente migrable. Únicamente se requiere que la maquina donde vaya a estar funcionando el servidor tenga *Docker* instalado.

Para implementar la solución con *Docker* se requiere de un fichero *Dockerfile*. Este fichero se puede ver en el siguiente bloque de código:

```
FROM node\:alpine

LABEL Maintainer="Oscar Ferrando"
      Description="Noise Plane Calculator server"

RUN mkdir -p /usr/src/plane-noise-server

COPY . /usr/src/plane-noise-server

WORKDIR /usr/src/plane-noise-server

RUN apk add zip
RUN apk add --no-cache --upgrade bash
RUN apk add gcompat

RUN chmod +x prepare.sh
RUN ./prepare.sh

WORKDIR /usr/src/plane-noise-server/ANCM_Lib/ANCM_Lib

RUN ls -l
RUN npm install

CMD [ "node","server.js"]
```

El archivo *Dockerfile* se encarga de:

- Hacer uso de *node:alpine*, una versión ligera de un entorno *Linux* ideal para la ejecución de servidores desarrollados en *node*.
- Copiar todos los archivos y carpetas necesarias, (*server.js* y la carpeta *ANCM_Lib* que contiene el ejecutable).
- Ejecutar el fichero *prepare.sh* que se encarga de, una vez dentro del contenedor, mover los archivos de configuración y servidor dentro de la carpeta del ejecutable para tener un acceso más sencillo.
- Marcar como *WORKDIR* la carpeta del ejecutable dentro del contenedor.
- Ejecutar *npm install* para instalar la dependencia de *express* y poder correr el servidor.
- Y por último se hace la llamada a *CMD node server.js* para empezar iniciar el servidor.

Además, para facilitar y automatizar el despliegue y orquestación tanto de servidor como del cliente se ha creado un archivo *docker-compose*. En lo referente al servidor, este archivo *docker-compose* contiene las variables de entorno necesarias, así como la redirección de puertos para que desde el anfitrión se pueda acceder al contenedor de forma libre.

```
services:
  plane-noise-server:
      build:
        context: ./plane-noise-server
      image: plane-noise-server
      restart: always
      ports:
        - "8080:8080"
      container_name: plane-noise-server
```

```
oscar@oscar-VirtualBox:~/workspace/plane-noise/docker$ docker ps
CONTAINER ID   IMAGE                 COMMAND                  CREATED          STATUS                    PORTS                                       NAMES
e050295cb6a8   plane-noise-server    "docker-entrypoint.s…"   2 minutes ago    Up 2 minutes (healthy)    0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   plane-noise-server
oscar@oscar-VirtualBox:~/workspace/plane-noise/docker$
```

Figura 8.2: Salida del comando *docker ps*

```
10        hostname: plane-noise-server
11        working_dir: /usr/src/plane-noise-server/ANCM_Lib/ANCM_Lib
12        environment:
13          - NODE_PATH=/usr/local/lib/node_modules/
14        healthcheck:
15            test: curl --fail --request GET --silent http://localhost/healthCheck
                  || exit 1
16            interval: 60s
17            timeout: 30s
18            retries: 10
```

Como se puede observar el *build* apunta a la carpeta del servidor y la redirección de puertos es del 8080 del anfitrión al 8080 del contenedor. Además de un sistema de *health-check* para comprobar la "salud" de servidor y si este se encuentra caído por cualquier motivo marcarlo como tal.

Con todo esto ya se tiene un servidor que expone una *API REST* a la que se le pueden hacer peticiones. Para iniciarlo simplemente hay que ejecutar las siguientes líneas en un terminal:

```
1 docker compose stop
2 yes | docker system prune
3 docker compose build --no-cache plane-noise-server
4 docker compose up -d plane-noise-server
```

Luego ejecutando el comando *docker ps* se puede observar el servidor funcionando en el puerto 8080 local y redirigiendo las peticiones al puerto 8080 del contenedor como se puede observar en al imagen 8.2.

También se puede comprobar el funcionamiento de este haciendo uso de cualquier servicio que permita llamadas *REST* como puede ser *curl* o *Postman* (ver A). En la imagen 8.3, se puede observar el resultado de 56.48 dB obtenido para cierta entrada de datos al hacer una llamada *POST* a la *API /api/calculate-noise*.

8.2 Cliente

En este apartado se abordará el desarrollo del cliente, una vez terminado el desarrollo de la parte del servidor, en dos fases como se ha indicado en apartados anteriores. El primer paso es abordar el diseño de las clases que se van a utilizar en la implementación. Dado que va a usar el *framework* de *Angular* con *TypeScript*, es posible el tipado de las clases, lo que le añade facilidad y seguridad a la implantación.

En la imagen 8.4 se pueden apreciar las clases que se van a usar, así como las propiedades de cada una que son necesarias para el correcto funcionamiento del servicio:

Vemos que aparecen 5 clases en total:

- Aircraft_data: Esta clase se encarga de tener los datos de todos los aviones de la base de datos de *ANP*. Dado que la *API* de *OpenSky* no provee de los datos relativos a los motores y tampoco del fichero *NPD* que se tiene que usar para enviarle a la *API* era necesario la existencia de esta clase de mapeo que posee estos 5 campos:
  - *ACFT_ID*: abreviatura del modelo del avión.

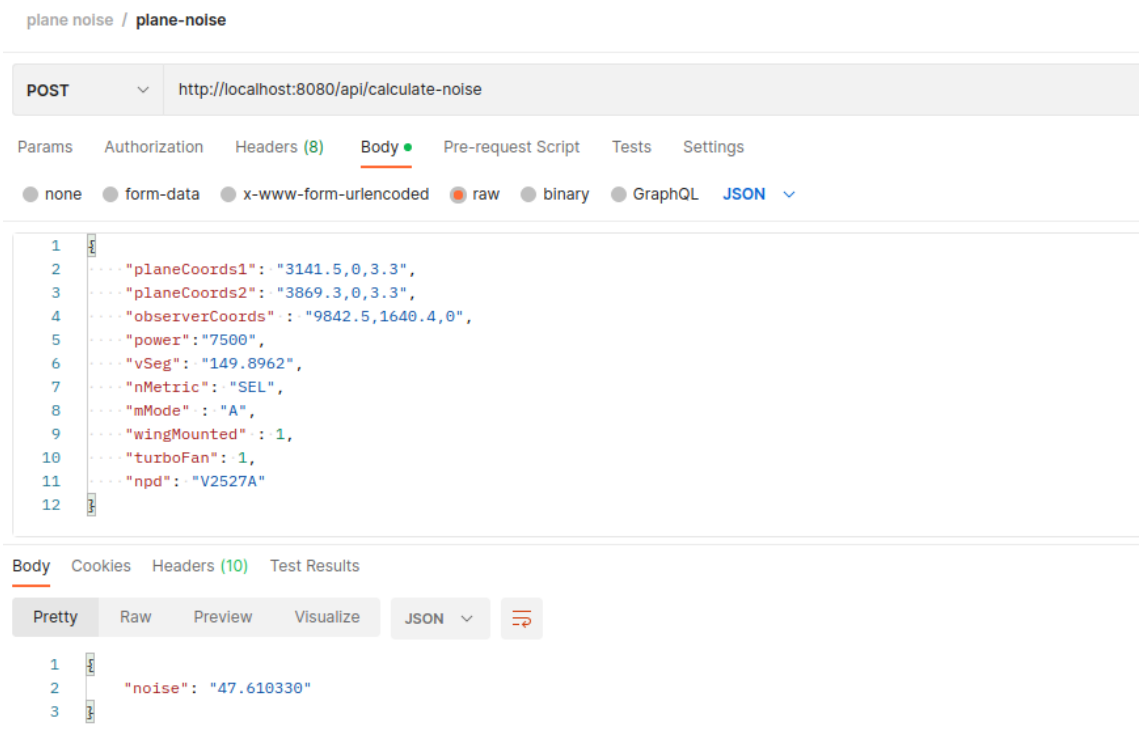

Figura 8.3: Resultado de llamar a la *API* usando *Postman*

- *description*: contiene el nombre completo del modelo del avión.
- *engineType*: tipo de motor.
- *engineMounted*: dónde se encuentra montado el motor.
- *NPD_ID*: nombre del fichero NPD.

- *Plane*: Es la clase principal y mapea el estado completo de un avión. Tiene los siguientes campos:
  - *callsign*: nombre en clave del avión.
  - *IACO*: número hexadecimal que identifica de forma univoca una aeronave.
  - *Longitude*: longitud de la primera parte del segmento.
  - *Longitude2*: longitud de la segunda parte del segmento.
  - *Latitude*: latitud de la primera parte del segmento.
  - *Latitude2*: latitud de la primera parte del segmento.
  - *Altitude*: altitud de la primera parte del segmento.
  - *Altitude2*: altitud de la primera parte del segmento.
  - *Velocity*: velocidad en un punto.
  - *TrueTrack*: dirección en grados.
  - *IATA*: aeropuerto de destino.
  - *WingMounted*: puede ser "1" o "0" según el motor se encuentre en alas o fuselaje.
  - *Turbofan*: puede ser "1" o "0" según el motor sea de hélice o a reacción.
  - *Power*: potencia.
  - *vSeg*: velocidad del segmento.

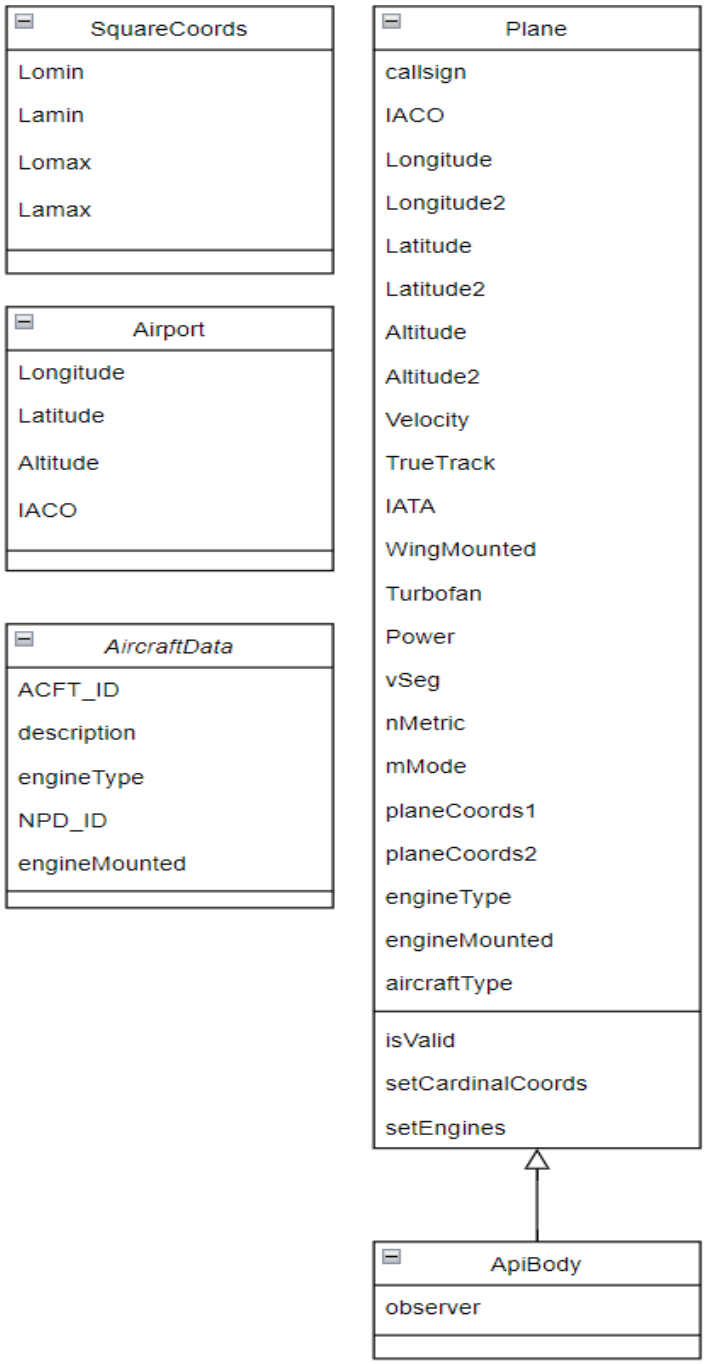

Figura 8.4: Digrama de clases

  - *nMetric*: será siempre "*SEL*".
  - *mMode*: "A" si se trata de una aproximación o "D" si es un despegue.
  - *planeCoords1*: coordenadas en formato *x,y,z*.
  - *planeCoords2*: coordenadas en formato *x,y,z* de la segunda parte del segmento.
  - *engineType*: "*Jet*" o "*Prop*".
  - *engineMounted*: "*Wing*" o "*Fuselage*"
  - *aircraftType*: modelo del avión.

- *Airport*: Una clase sencilla que únicamente posee 5 campos para gestionar la posición y nombre de los distintos aeropuertos sobre los que se van a realizar las mediciones:

  - *Longitude*: longitud donde se encuentra el aeropuerto.
  - *Latitude*: latitud donde se encuentra el aeropuerto.
  - *Altitude*: altitud a la que se encuentra el aeropuerto.
  - *Name*: nombre del aeropuerto.
  - *IACO*: número hexadecimal que identifica de forma univoca el aeropuerto.

- *API_body*: cuerpo que se le enviará a la *API* del servidor. Esta clase extiende de *plane* y le añade la propiedad de la posición del observador necesaria

  - *observer*: coordenadas en formato *x,y,z*.

- *Square_coords*: se encarga de remarcar el cuadrado sobre el que se va a realizar la petición al *API* de aviación. De esta forma se acota la búsqueda y evita el tener que filtrar los aviones:

  - *Lomin*: longitud inferior.
  - *Lamin*: latitud inferior.
  - *Lomax*: longitud superior.
  - *Lamax*: latitud superior.

Como se ha mencionado anteriormente se ha usado *Angular* para generar la aplicación. Este *framework* mediante una simple orden, en este caso *ng new plane-noise*, permite la creación de un esqueleto vacío, pero funcional de una aplicación web con una estructura como la mostrada en la imagen 8.5.

En la imagen 8.5 se observa la forma en la que el *framework* genera el proyecto donde la carpeta *e2e* tendrá las pruebas de integración (ver 9) y la carpeta principal de trabajo será *src*(*source*) donde se implementará todo el código de la aplicación.

Una vez que el proyecto se ha generado se prosigue con el diseño del distintos componentes y servicios que formarán parte de esta primera iteración y que tendrán como objetivo el primer prototipo funcional del proyecto.

En la imagen 8.6 se puede observar el diagrama de componentes, el de servicios y la forma en la que interactuarán.

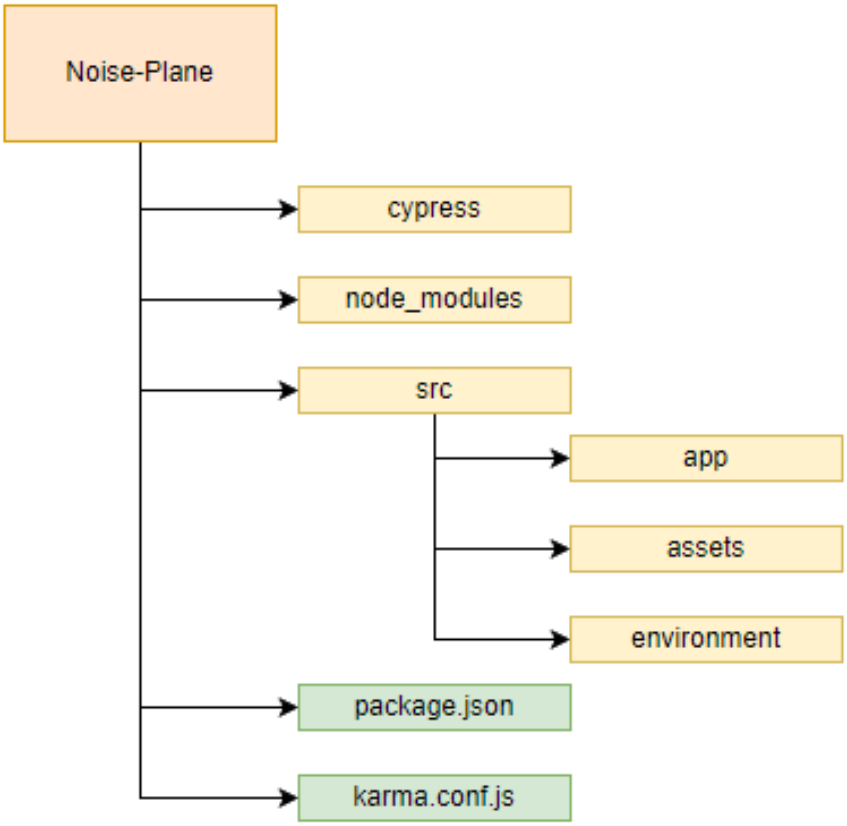

Figura 8.5: Estructura base de un proyecto *Angular*

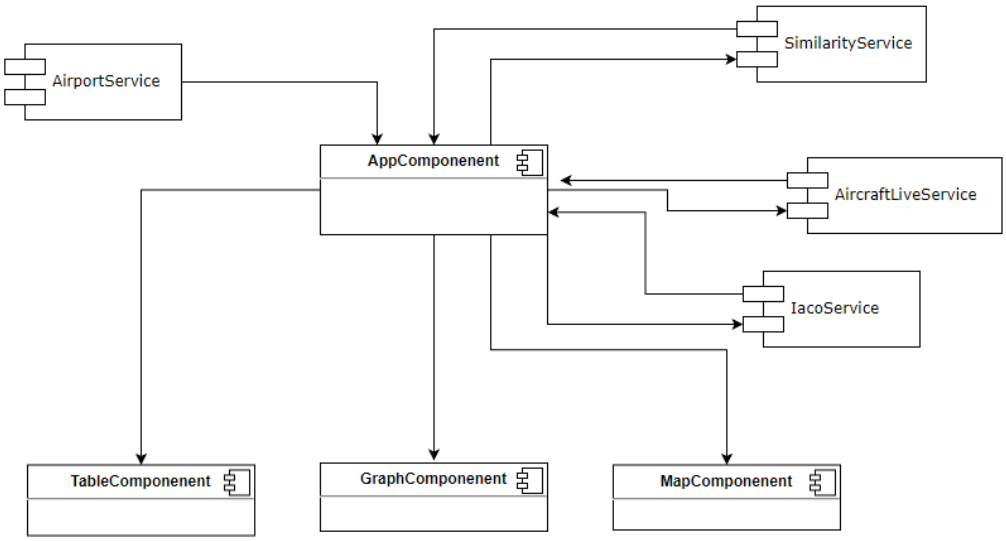

Figura 8.6: Diagrama de componentes y servicios

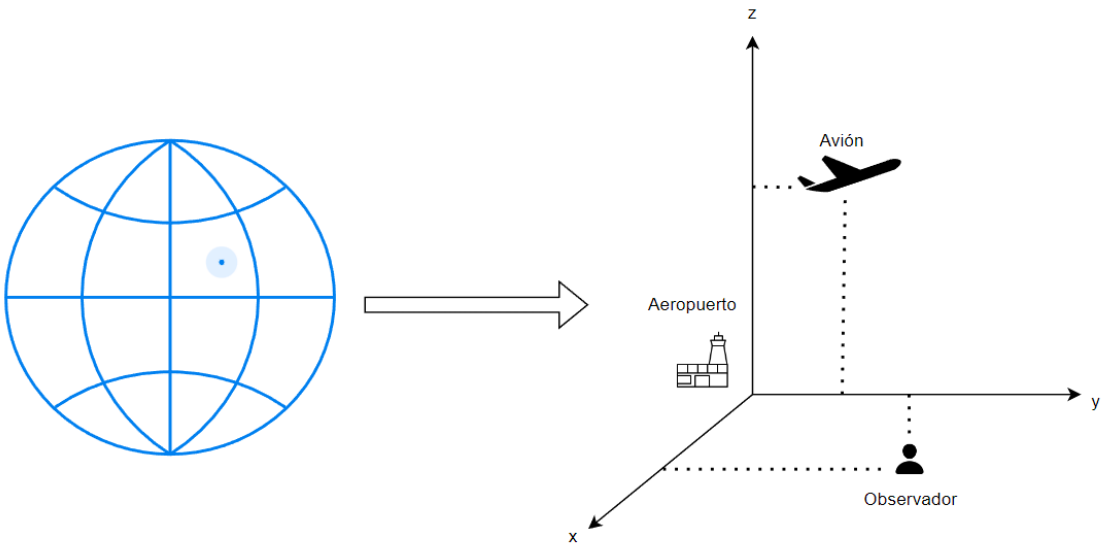

Figura 8.7: Transformar coordenadas polares a cartesianas

8.2.1. Clases

Plane

Es modelo principal del servicio a desarrollar. La clase de encarga de modelar un avión y todos los datos relacionados con este que son necesarios para el uso del servidor y por lo tanto para poder calcular el ruido generado por ellos aeropuertos.

Otra de sus funciones principales es la de obtener las coordenadas en sistema latitud/longitud (grados polares) y convertirlo a un sistema *x,y,z* cartesiano, que es el que hace uso el servidor para el cálculo del ruido. Si el cálculo se tuviese que realizar para una superficie muy grande y la distancia entre los dos objetos fuese elevada este sistema de conversión tendría una pérdida de precisión debido a la curvatura de la tierra. Al pasar una superficie curva a un plano esta se ensancharía en los bordes de la misma forma que pasa en las representaciones planas de un mapa del mundo donde la proporción del tamaño de los países que se alejan del ecuador queda distorsionada.

Para entender de una forma visual la transformación requerida entre un punto con coordenadas polares y un sistema cartesiano se pueden ver la imagen 8.7.

La *API* de navegación proporcionan los datos de posición, altura y velocidad en unidades distintas a las que requiere la librería *Matlab*. En este caso de la velocidad se proporciona en metros/segundo y la altura en metros pero la librería requiere esos datos en pies (*ft.*) por que es necesaria la transformación a esa unidad.

Como se acaba de mencionar la librería requiere de un sistema de posición cardinal en 3 dimensiones. X e Y marcarán el plano y Z la altura. Para la creación de este sistema se requiere de un punto de referencia que en nuestro caso es la posición de aeropuerto sobre el que se van a realizar las mediciones. Por lo tanto, el aeropuerto será la posición 0,0,0 como se puede observar en la imagen 8.7. A partir de ahí con otra posición, como por ejemplo la del avión, se puede calcular su posición relativa a la de aeropuerto. Para ello es necesario realizar los siguientes cálculos:

1. Primero hay que convertir a decimal la longitud y latitud. En nuestro caso no aplica ya que la propia *API* ya los proporciona de esta forma. En caso contrario habría

que tener en cuenta los signos ya que, para la latitud NORTE sería positivo y SUR negativo y para longitud ESTE positivo y OESTE negativo.

2. Hay que restar la posición de aeropuerto al objeto del que queremos calcular la posición relativa.

3. Una vez realizado este cálculo hay que convertir los grados a metros, como un grado equivale a 111139 metros se multiplican los valores obtenidos en el punto anterior por el valor de conversión.

4. La *API* requiere el uso de unidades en pies(ft.) por lo tanto hay que volver a multiplicar por el valor de transformación de metros a pies que es de 3,28084. Con este cálculo ya se tendría el valor X e Y del sistema cartesiano.

5. Para la Z, únicamente hay que usar la altura. Como la *API* de navegación la proporciona en metros hay que multiplicar su valor por 3,28084 para obtenerla en pies.

El siguiente bloque de código permite ver cómo se ha implementado el algoritmo anterior en *TypeScript*. Además, también asigna el valor en el formato de escritura correcto que para este caso es X, Y, Z al atributo *planeCoords*.

```
setCardinalCoords(airportLatitude: number, airportLongitude: number) {
        let gradToMeters = 111139;
        let meterToFeet = 3.28084;
        let newLat = (this.latitude - airportLatitude) * gradToMeters *
            meterToFeet;
        let newLon = (this.longitude - airportLongitude) * gradToMeters *
            meterToFeet;
        let newLat2 = (this.latitude2 - airportLatitude) * gradToMeters *
            meterToFeet;
        let newLon2 = (this.longitude2 - airportLongitude) * gradToMeters *
            meterToFeet;
        let newAltitude = this.altitude * meterToFeet;
        let newAltitude2;
        if (this.altitude2) {
            newAltitude2 = this.altitude2 * meterToFeet;
        } else {
            newAltitude2 = this.altitude;
        }
        this.planeCoords1 = newLat + "," + newLon + "," + newAltitude
        this.planeCoords2 = newLat2 + "," + newLon2 + "," + newAltitude2
    }
```

Por ejemplo, para el aeropuerto de Alicante-Elche donde:

- Las coordenadas del aeropuerto son 38.282169 -0.558156
- Las coordenadas del avión son 38.2926 -0.6326
- La altura del avión en pies es de 297.18 ft.

Haciendo las siguientes dos operación, la primera para X e Y y la segunda para Z. Da como resultado, en sistema cartesiano: (3803.447985882455,-27144.461879121463,975.0000312)

$$((38{,}2926, -0{,}6326) - (38{,}282169, -0{,}558156)) * 111139 * 3{,}28084 = (447985882455, -27144{,}461879121463)$$

$$297{,}18 * 3{,}28084 = 975{,}0000312$$

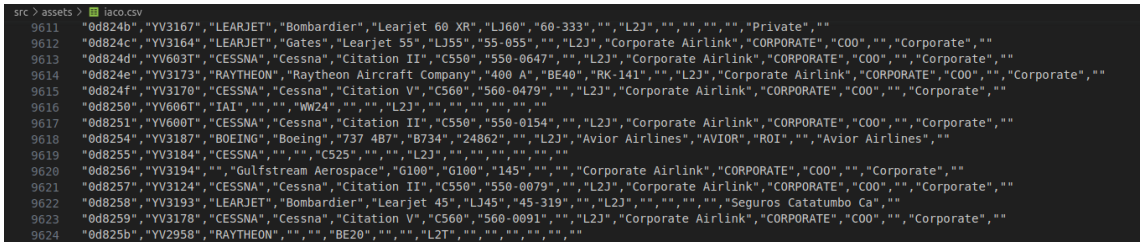

Figura 8.8: Datos IACO

8.2.2. Servicios

Primero se va a explicar el uso e implementación de los distintos servicios y luego cómo se han integrado y de qué forma los usan los distintos componentes de la aplicación web.

AiroportService

El primer servicio necesario es el *airoportService*. Este servicio es el encargado de leer de un fichero *JSON* que contiene el nombre y coordenadas de los distintos aeropuertos y transformarlo al objeto *airport*. Este fichero *JSON* se encuentra en la carpeta *assets* del proyecto, que es accesible desde el código por defecto. Para obtener el fichero se ha creado un método *getAirports()* que simplemente realiza una llamada *HTTP* a una ruta interna. En este caso a *assets* como se ha indicado anteriormente. De la siguiente forma:

```
getAirports(): Observable<any> {
    return this.http.get(this.assetsPath + 'airports.json')
}
```

De esta forma se consigue un *observable* que contiene los datos de los distintos aeropuertos.

IacoService

El segundo servicio necesario es el llamado *IacoService*. Una de las limitaciones de la *API* de *OpenSky* es, como se ha mencionado, que no devuelve el modelo de avión. Para ello ha sido necesaria la implementación de este servicio, ya que de forma indirecta se encarga de obtener el modelo de avión. La *API* de *OpenSky* devuelve el *IACO24* de la aeronave. Esto es un número hexadecimal de 6 caracteres que identifica el objeto. Para obtener el modelo de avión se ha requerido de una base de datos que existe en *OpenSky*. Esta base de datos posee los códigos *IACO* de todas las aeronaves registradas hasta la fecha de abril de 2023. La web permite descargar el fichero en formato *CSV* con los datos. Este fichero se ha incorporado en el proyecto y tiene una estructura como la mostrada en la imagen 8.8.

Para el caso que concierne únicamente son necesarios los campos de la primera columna y la quinta que indican el *IACO* y el modelo del avión. Para ello se ha creado un método *getModel(iaco:string)* al que se le pasa un *string* con el *IACO* y devuelve el modelo de la aeronave que coincide con ese *IACO*. En el código posterior se puede observar cómo se carga el fichero *CSV* de *assets*, se convierte y se hace la comparación del primer campo (*row[0]*) con el *IACO* que se pasa como parámetro, en caso de encontrarlo se devuelve el campo de la columna en quinta posición(*row[4]*)

```
getCsvData(iaco: string): Observable<string> {
    let formPath = './assets/';
    const url = formPath + 'iaco.csv';
    return this.http.get(url, { responseType: 'text' }).pipe(
      map(data => {
        const csvData = this.parseCsvData(data);
```

```
        let parsedIaco = '"' + iaco + '"'
        const result = csvData.find(row => row[0] == parsedIaco);
        return result ? result[4] : '';
      })
    );
  }
  private parseCsvData(data: string): string[][] {
    const rows = data.split('\n');
    const csvData = [];
    for (let i = 0; i < rows.length; i++) {
      const row = rows[i].split(',');
      csvData.push(row);
    }
    return csvData;
  }
```

SimilarityService

El tercer servicio que ha sido necesario implementar se ha llamado *SimilaritySercice*. El servicio de *IacoService* permitía obtener el modelo de avión, pero el modelo de avión que requiere el servidor no tiene la misma nomenclatura. Dada la gran cantidad de modelos de avión que existen la búsqueda y conversión de los distintos modelos no era un trabaja factible así que se ha optado por una aproximación.

Primero se ha generado un fichero *JSON* con los siguientes datos:

- *ACFT_ID*: abreviatura del modelo de avión.
- *Description*: nombre completo del modelo de avión.
- *engineType*: tipo de motor, *jet* o *prop*.
- *NPD_ID*: necesario para el fichero de datos que requiere el servidor para calcular el ruido.
- *engineMounted*: el motor está montado en alas o fuselaje.

Para generar este ficho *JSON* se ha necesitado acceder a la página de *ANP* que posee una base de datos completa de todos los modelos de avión. Con los datos que no posee la *API* de *Opensky*, que son el *engineType* y *engineMounted*, además de *NPD_ID*. Para ello se ha descargado la base de datos entera llamada *Aircraft.csv* y se ha mapeado para obtener el *JSON* con los campos necesarios mediante un *script* (ver B).

Una vez se tiene el fichero *JSON* se necesita encontrar la pareja que coincida entre el modelo de avión que provee el *IacoService* a través de *OpenSky* y el modelo de avión que necesita el servidor. Para ello se han analizado dos posibles aproximaciones que tengan en cuenta cuan similares son las dos palabras.

La primera aproximación es más tradicional y calcula el número letras que son iguales en las dos palabras de la siguiente forma:

```
similarity(word1: string, word2: string) {
        let similarLetters = 0;
        for (let i = 0; i < word1.length; i++) {
            if (word1[i] === word2[i]) {
                similarLetters++;
            }
        }

        // Calcular el porcentaje de letras iguales
        const similarity = similarLetters / word1.length;
```

```
11
12          // Devolver el porcentaje de letras iguales
13          return similarity;
14      }
```

La segunda aproximación que se estudia en la llamada Distancia de Levenshtein [29], también conocida como distancia de edición. La Distancia de Levenshtein, es una métrica que cuantifica la diferencia entre dos cadenas de caracteres. Se calcula como el número mínimo de operaciones de edición requeridas para transformar una cadena en otra.

Se permiten tres operaciones distintas y cada operación tiene un coste de 1:

- Inserción de un carácter.
- Eliminación de un carácter.
- Sustitución de un carácter por otro.

Por ejemplo, la distancia de Levenshtein entre las palabras 'casa' y 'calle' es 3, ya que se necesitan tres operaciones de edición para transformar la palabra 'casa' en 'calle': la eliminación de la letra 's', la sustitución de la letra 's' por 'l' y la inserción de la letra 'e'.

Este algoritmo se ha implementado de la siguiente forma:

```
similarityLevenshtein(word1:any, word2:any) {
        let longer = word1;
        let shorter = word2;
        if (word1.length < word2.length) {
          longer = word2;
          shorter = word1;
        }
        var longerLength = longer.length;
        if (longerLength == 0) {
          return 1.0;
        }
        return (longerLength - this.editDistance(longer, shorter)) / parseFloat
            (longerLength);
}
```

```
editDistance(word1:any, word2:any) {
        word1 = word1.toLowerCase();
        word2 = word2.toLowerCase();
        var amount = new Array();
        for (var i = 0; i <= word1.length; i++) {
          var lastValue = i;
          for (var j = 0; j <= word2.length; j++) {
            if (i == 0)
              amount[j] = j;
            else {
              if (j > 0) {
                var newValue = amount[j - 1];
                if (word1.charAt(i - 1) != word2.charAt(j - 1))
                  newValue = Math.min(Math.min(newValue, lastValue),
                    amount[j]) + 1;
                amount[j - 1] = lastValue;
                lastValue = newValue;
              }
            }
          }
          if (i > 0)
            amount[word2.length] = lastValue;
        }
```

Figura 8.9: Resultados algoritmos de comparación de textos

```
24          return amount[word2.length];
25      }
```

Después de varias pruebas se ha llegado a la conclusión que la distancia de Levenshtein se comportaba mejor que la aproximación más simple de caracteres iguales. Como se puede observar en la imagen 8.9 ante ambos algoritmos la distancia de Levenshtein reporta una parecido del 0.51 entre la entrada A320-232 y la descripción de Airbus A320 que aparece en el *JSON* mientras que el formato estándar únicamente un 0.17 y además, no devuelve el objeto correcto mientras que con Levenshtein sí.

AircraftLiveService

El servicio sobre el que se apoya todo el proyecto se ha llamado *AircraftLiveService*. Este servicio es el encargado de obtener los datos en tiempo real de la situación de las distintas aeronaves de una zona concreta.

Existen básicamente dos formas de obtener datos en tiempo real. El primero es el uso de un servicio de mensajes del tipo *pub/sub*, es decir, publicación/suscripción. Y el segundo es hacer repetidamente llamadas a una *API* para ir obteniendo datos de forma periódica. La imagen 8.10 muestra claramente el funcionamiento de las dos aproximaciones. En la versión de publicación suscripción es cliente el que recibe los datos por parte del servidor. El cliente está subscrito a un publicador y este de forma periódica va enviando mensajes/eventos de datos al cliente (suscriptor) que está suscrito a un tópico. Por otra parte en una llamada a una API es el cliente quien inicia la transacción al pedir explícitamente los datos al servidor a través de una petición *REST*.

Dado que la *API* de *OpenSky* no tiene una versión de *pub/sub* se ha optado por la segunda aproximación, la de una petición REST cada cierto tiempo. Una de las funcionalidades de esta *API* es que permite acotar el espacio aéreo en un área rectangular. Esto es ideal para el trabajo ya que ahorra tener que filtrar las distintas aeronaves por distancia o aeropuerto de origen/destino. La aproximación que se ha seguido para la implementación es acotar un área alrededor del aeropuerto sobre el que se va a obtener la medición. Para ello se ha creado un método que dada la latitud y longitud de un aeropuerto calcula un área cuadrada de 20 km en diagonal situando el aeropuerto en el centro de dicha diagonal. El método encargado de hacer ese cálculo tiene la siguiente forma:

```
1  getSquareCoordinates(): SquareCoords {
2      let lmin = this.calculateVertexCoordinates(this.selectedAirport.latitude,
           this.selectedAirport.longitude, 225, 10000);
3      let lmax = this.calculateVertexCoordinates(this.selectedAirport.latitude,
           this.selectedAirport.longitude, 45, 10000);
4      let squareCoords: SquareCoords = new SquareCoords();
5      squareCoords.lamin = lmin[0];
6      squareCoords.lomin = lmin[1];
7      squareCoords.lamax = lmax[0];
8      squareCoords.lomax = lmax[1];
9      return squareCoords;
10 }
```

```
1  calculateVertexCoordinates(latitud: number, longitud: number, direccion:
       number, distancia: number) {
```

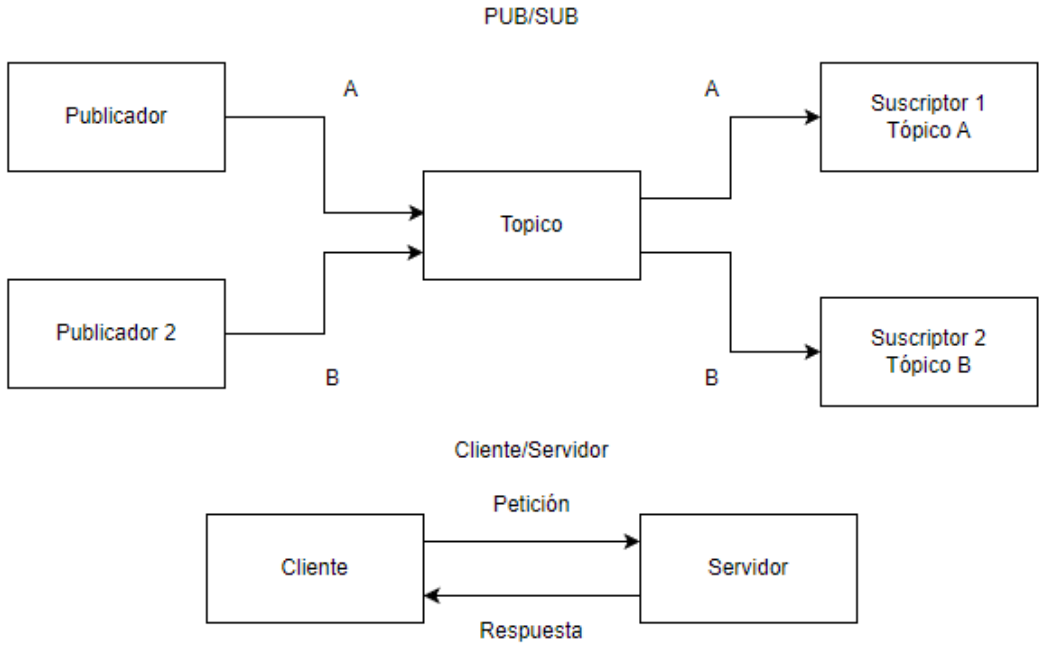

Figura 8.10: Pub/Sub vs llamada a API

```
    const radioTierra = 6371; // Radio de la Tierra en km
    const distanciaKm = distancia / 1000;
    const latitudRad = this.toRadians(latitud);
    const longitudRad = this.toRadians(longitud);
    const direccionRad = this.toRadians(direccion);

    const nuevaLatitud = Math.asin(Math.sin(latitudRad) * Math.cos(distanciaKm
        / radioTierra) +
      Math.cos(latitudRad) * Math.sin(distanciaKm / radioTierra) * Math.cos(
          direccionRad));
    const nuevaLongitud = longitudRad + Math.atan2(Math.sin(direccionRad) *
        Math.sin(distanciaKm / radioTierra) * Math.cos(latitudRad),
      Math.cos(distanciaKm / radioTierra) - Math.sin(latitudRad) * Math.sin(
          nuevaLatitud));

    return [this.toDegrees(nuevaLatitud), this.toDegrees(nuevaLongitud)];
  }
```

En el fragmento de código anterior se puede observar el código encargado que dada la longitud y latitud de un aeropuerto obtiene la *lmin* que equivaldría al punto inferior izquierdo de un cuadrado y la *lmax* que sería el superior derecho. Para obtener estas coordenadas se usa el método *calculateVertexCoordinates* donde dada una posición, un ángulo, en este caso: 45º para la superior derecha y 225º para inferior izquierda obtiene esos dos puntos. La imagen 8.11 muestra la aproximación visual de lo que se consigue con esos cálculos.

Una vez obtenida el área se puede hacer la llamada a la *API* de *OpenSky* haciendo una llamada *REST GET* a la siguiente dirección. Además, es necesario añadir la cabecera de autorización con un *token* que se obtiene a partir del usuario y la contraseña de *OpenSky*, *Authorization*. Sin esta cabecera la petición de la *API* no estaría autenticada y únicamente permitiría 400 peticiones diarias.

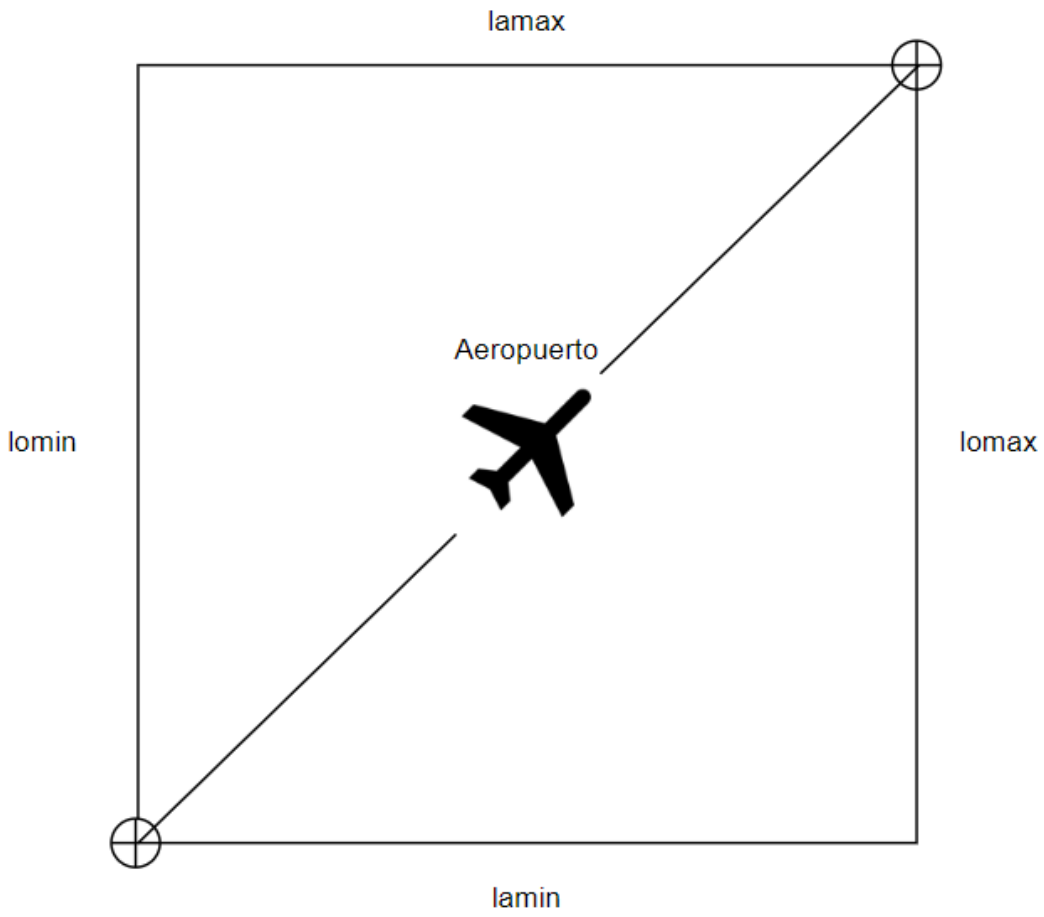

Figura 8.11: Cálculo de área a partir de las coordenadas de un aeropuerto

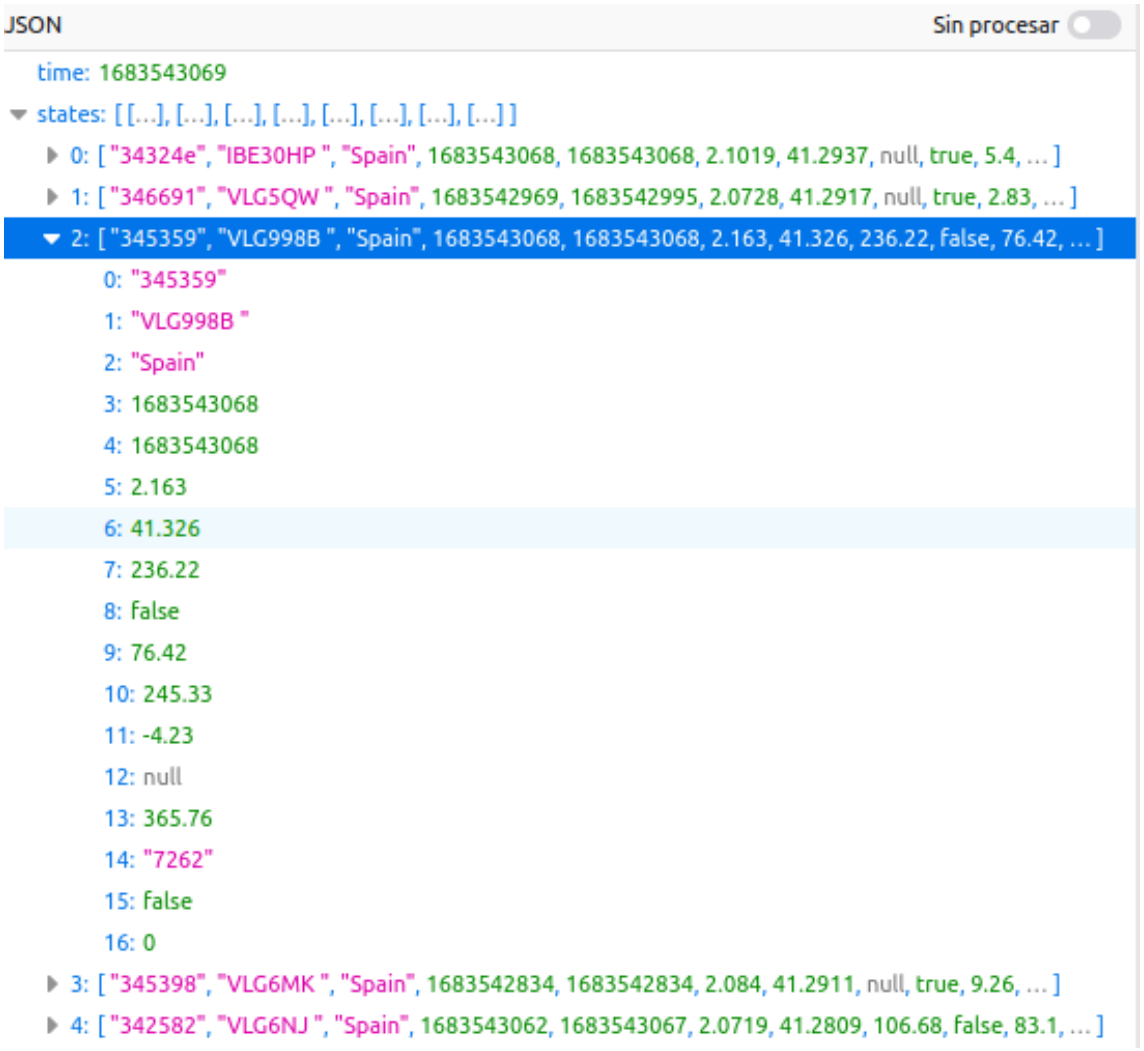

Figura 8.12: Resultado de la llamada a la *API* de *OpenSky*

```
https://opensky-network.org/api/states/all?lamin=' + LAMIN + '&lomin=' + LOMIN
    + '&lamax=' + LAMAX + '&lomax=' + LOMAX
```

Una vez realizada esta llamada, se obtienen todos los *states* (como la *API* llama a cada objeto) pero filtrados usando las variables calculadas anteriormente de *lamin*, *lomin*, *lamax* y *lomax* para acotarlos a un espacio aéreo concreto. Un ejemplo del resultado de esa llamada podría ser como el que se ve en la imagen 8.12. En este caso se trata del aeropuerto Madrid-Barajas a las 12:14h del 29 de abril de 2023.

8.2.3. Componentes

En la siguiente sección se va a proceder a exponer los distintos componentes desarrollados y la función que desempeña cada uno en la representación y obtención de los datos.

TableComponent

Para dotar de visualización es necesario mostrar las distintas aeronaves que sobrevuelan el espacio delimitado que se calcula como se ha explicado en la sección anterior. Para ello se ha hecho uso de una simple tabla que muestra los distintos datos de las aeronaves con los siguientes datos:

- Nombre (*callsign*): Código que identifica el vuelo. Normalmente es una combinación de 6 cifras como DLH12N.
- Modelo de avión.
- Latitud.
- Longitud.
- Altitud.
- Velocidad.
- Tipo de motor: Si se trata de *jet* o hélice.
- Montaje de motor: En las alas o fuselaje.
- Distancia al observador.
- Ruido generado.

Para ello se ha optado por el uso de *HTML* puro en lugar de usar alguna de las librerías existentes como puede ser *AngularMaterial*, dado que de esta forma se obtiene mayor flexibilidad en los estilos y en el funcionamiento de la tabla. El código para la implementación es relativamente simple donde se establecen las columnas y los nombre de estas y se itera sobre un *array* de aviones que se han rellenado con los datos obtenidos anteriormente.

Uno de los campos a mostrar en la tabla para enriquecer la información y el análisis de resultados es la distancia al observador. Como se ha mencionado en el apartado de *Plane*, ahora se posee la posición de los objetos en un sistema cartesiano, tanto el observador como el avión.

Para realizar este cálculo es necesario hacer uso del teorema de Pitágoras aplicado a 3 dimensiones de la siguiente forma:

$$d(P1, P2) = \sqrt{(x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$$

Este algoritmo se ha implementado en *TypeScript* como se observa a continuación:

```
setDistanceToObserver(observer: string) {
        //console.log("antes de set distancte to observer ", this.callsign,
            observer)
        let xObserver: number = +observer.split(',')[0]
        let yObserver: number = +observer.split(',')[1]
        let zObserver: number = +observer.split(',')[2]

        let xPlane = +this.planeCoords1.split(',')[0]
        let yPlane = +this.planeCoords1.split(',')[1]
        let zPlane = +this.planeCoords1.split(',')[2]
        const distancia = Math.sqrt(Math.pow(xObserver - xPlane, 2) + Math.pow(
            yObserver - yPlane, 2) + Math.pow(zObserver - zPlane, 2));
        this.observerDistance = distancia.toString();
        return distancia;
    }
```

Donde primero se extraen los datos x,y,z del *string* de posición y luego se aplica la fórmula anterior para obtener la distancia y asignarla al atributo *observerDistance* que se mostrará en la tabla.

El cálculo de un ejemplo concreto podría ser:

- Las coordenadas del avión: 40.4829 -3.5455 647.7
- Las coordenadas observador: 40.504 -3.561 592
- La distancia es 2953.16m.

Una vez obtenida la distancia al observador ya se puede representar la tabla por completo con todos los datos y se vería como en la imagen 8.13.

Aircraft data

Name	Aircraft Type	Latitude	Longitude	Altitude	Velocity	Engine Type	Engine Mounted	Observer Distance(m)	Noise
VLG941B	Airbus A320 214	41.302	2.0927	114.3	73.23	Jet	Wing	931.78	67.334277 dB
VLG6CK	Boeing 737-800 / CFM56-7B26	41.2377	2.0704	685.8	103.59	Jet	Wing	7211.84	35.975411 dB
N797CP	Gulfstream GIV-SP / TAY 611-8	41.3009	2.0896	60.96	33.38	Jet	Fuselage	573.20	44.9618 dB
DLH14K	Airbus A330-343 / RR Trent 772B	41.3227	2.1534	312.42	73.44	Jet	Wing	8045.73	32.077507 dB

Figura 8.13: Ejemplo de visualización del *TableComponent*

GraphComponent

Parte del alcance del proyecto es la obtención de datos en tiempo real. Para ello se ha creado un componente específico que se encargará de mostrar de una forma visual el ruido que se está calculando en tiempo real en una gráfica lineal usando la librería *d3.js* como se ha especificado en la sección 3.2 de Tecnologías.

Al componente *GraphComponent* se le pasan datos de ruido usando el decorador *@Input* de *Angular*. Estos datos tienen el siguiente formato:

- *Date*: marca la fecha que dibujará el eje *X*.
- *Value*: tiene el valor que se ha obtenido para ese instante de tiempo.

De esta forma se van enviado datos a la gráfica cada *X* segundos (en este caso se ha elegido cada 5 segundos) y esta va redibujando la gráfica con los nuevos datos.

La imagen 8.14 muestra un cálculo que empieza en unos 45 dB, y llega a unos 75 dB volviendo a bajar. El valor de 45db se considera el valor normal que hay dentro de un hogar mientras que el valor de 75dB para poner en contexto sería el equivalente al tráfico pesado o una aspiradora en funcionamiento.

AppComponent

El componente principal es el *AppComponent*. Este componente se encarga de la gestión de los datos de entradas para el cálculo del ruido y de aglutinar a los demás componentes y está formado por:

- Logo.

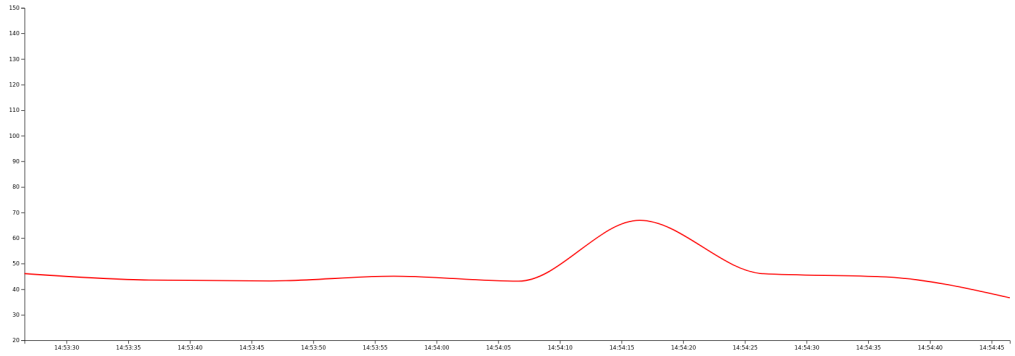

Figura 8.14: Ejemplo de visualización del *GraphComponent*

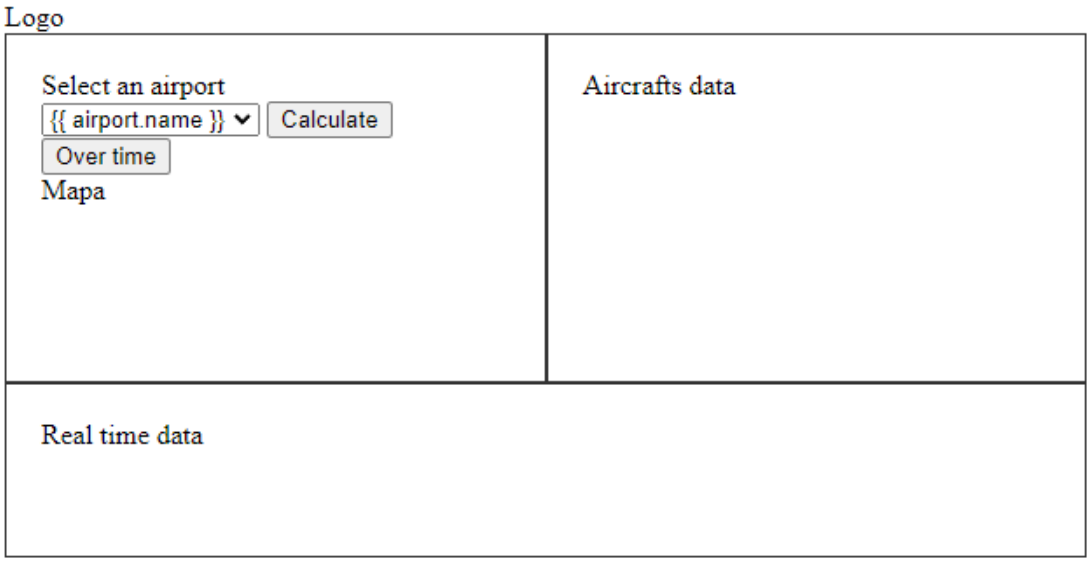

Figura 8.15: Diseño de la matriz del *AppComponent*

- Una entrada de texto para especificar la posición del observador.
- Un desplegable con los distintos aeropuertos.
- Un botón para el cálculo del estado actual.
- Otro botón para hacer una medición en tiempo real a través del tiempo.
- *TableComponent*.
- *GraphComponent*.

El componente principal estará dividido en una matriz formada por dos filas, dos columnas en la primera fila y una en la segunda. Por lo tanto un esquemático de cómo estarán los distintos componentes organizados sería como el que se observa en al imagen 8.15.

El logotipo de la aplicación ha sido generado usando un servicio de generación gratuito. Se ha tenido en cuenta que el propósito de la aplicación une dos mundos, la aviación y la salud al proveer de datos relativos a la contaminación acústica. El resultado se puede ver en la imagen 8.16.

Para la gestión de este componente son necesarios los siguientes atributos de clase:

Figura 8.16: Logotipo de la página web

- *CoordsObserver*: gestiona las coordenadas del observador.
- *Noise*: ruido generado.
- *Planes*: *array* que contiene los aviones en ese momento.
- *Airports*: *array* de los distintos aeropuertos.
- *Aircraft_Data*: contiene los datos de motor de los aviones.
- *SelectedAiroport*: gestiona el aeropuerto sobre el que se van a realizar los cálculos.
- *Interval*: gestiona la duración de la captura en tiempo real.
- *ActiveContinious*: marca si encuentra en captura en tiempo real o no.
- *noises*: *array* con los ruidos generados por las distintas aeronaves.

Dado que es necesario calcular la *vSeg* (velocidad del segmento) se ha optado por realizar una doble medición. Primero se obtiene los datos para un punto concreto en el tiempo y pasados 4 segundos se realiza una segunda medición para obtener y calcular la velocidad del segmento. Esto se consigue mediante la función *setTimeout* se *JavaScript* donde se le pasa el código a ejecutar y el retraso en milisegundos para esperar a ejecutar ese código.

Como puede haber varias aeronaves sobrevolando el espacio delimitado, cada una de ellas generará un ruido diferente. La unidad de medida del ruido usada son los decibelios (dB). Esta unidad de media es logarítmica así que su suma no se puede realizar de forma aritmética, es decir, si tenemos dos fuentes de ruido y cada una está generando 60 dB, el ruido generado no será de 120 dB sino de 63.01 dB. La fórmula usada para la suma de decibelios es la siguiente.

$$dbTotal = 10 * \log 10(10^{db1/10} + 10^{db2/10} + \cdots + 10^{dbn/10})$$

Y se ha implementado de la siguiente forma:

```
calculateDBSum(decibelios: number[]): number {
    const sumaLogaritmica = decibelios.reduce((acumulador, valorActual) =>
        acumulador + Math.pow(10, valorActual / 10), 0);
    const promedioLogaritmico = 10 * Math.log10(sumaLogaritmica);
    return promedioLogaritmico;
}
```

Por ello usando esta aproximación se obtiene el resultado de 63.01 dB mencionado anteriormente.

$$dbTotal = 10 * \log 10(10^{60/10} + 10^{60/10}) = 63{,}01dB$$

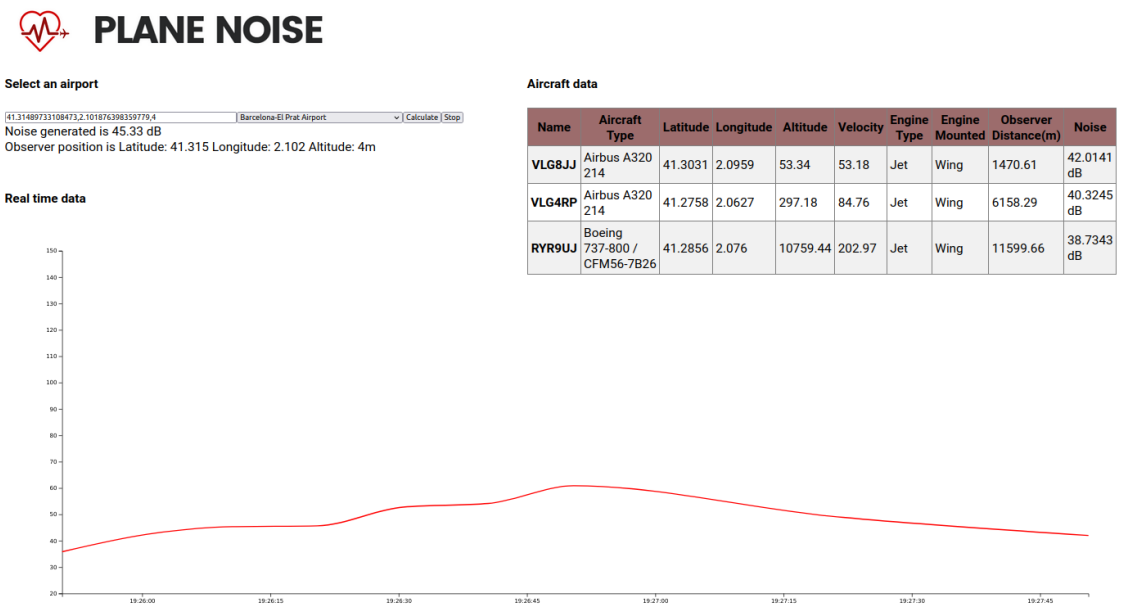

Figura 8.17: Captura de datos en tiempo real

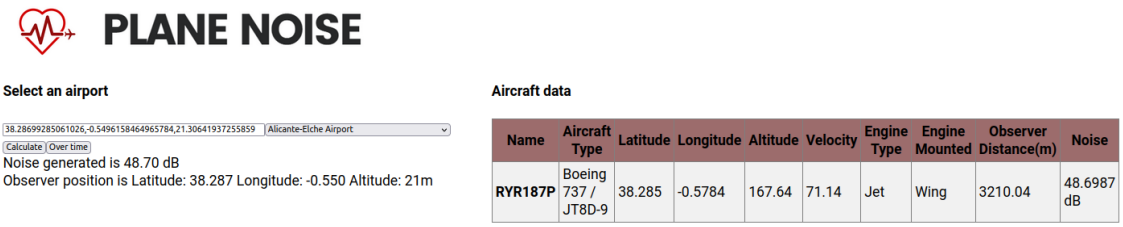

Figura 8.18: Captura de datos en un instante

Con todos los componentes en funcionamiento se concluye la primera parte del desarrollo. De esta forma ya se tiene un primer prototipo funcional de la aplicación que representa los datos de para un instante actual y para un rango de tiempo. Un ejemplo de ejecución podría ser como el que se ve en la imagen 8.17 mientras que un ejemplo para un instante concreto podría ser el observado en al imagen 8.18.

Donde se puede observar varios aviones con sus datos y los ruidos que están generando en tiempo real. Además, la gráfica permite mostrar la variación en el tiempo de los datos recopilados para un mejor análisis.

8.2.4. Mapa

Como se ha mencionado la segunda iteración del proyecto sería la encargada de mejorar el apartado visual para tener una mejor usabilidad. En el primer prototipo, la introducción de los datos del observador se realizaba de forma manual y con un formato específico. Es decir, se tendría que escribir algo del siguiente estilo si se quisiese obtener una medición para un lugar cercano al aeropuerto de Valencia:

- - 39.491,-0.473056,75

Como se puede ver, este formato de (latitud, longitud, altura) no es fácil de usar para un usuario poco experimentado. Además, es posible equivocarse en las unidades y usar

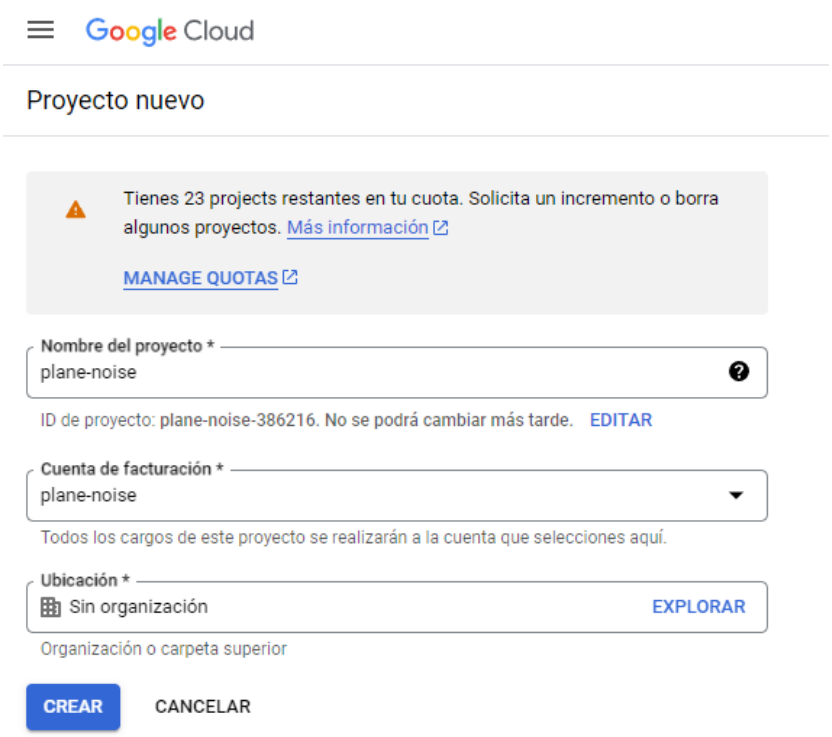

Figura 8.19: Creación de un proyecto con *Google Cloud Platform*

la altura en metros cuando se requiere el uso en pies o fallar en el formato de los grados cardinales.

Para solventar este problema se ha decido hacer uso de la *API* de *Google Maps*. Esta *API* como se ha mencionado en apartados anteriores permite dibujar un mapa e interactuar con él. Para el proyecto interesan tres funcionalidades especialmente:

- La capacidad de centrar el mapa en un punto.
- Dibujar un área.
- Seleccionar un punto en el mapa.

El primer paso para usar el mapa de la *API* de *GoogleMaps* es crear un proyecto el *Google Cloud Platform*, como se muestra en la imagen 8.19, y asociarlo a una cuenta de facturación previamente creada. Esta cuenta de facturación es necesaria ya que la *API* tiene un coste asociado por uso. Esto obliga a tener que indicar un método de pago en el caso de que se supere el importe gratuito de 200€ mensuales de carga de mapas.

Una vez habilitada la cuenta de facturación ya es posible conseguir el el *API_KEY* necesario para autenticar las llamadas contra la *API* de *Google Maps*. Para poder incrustar un mapa en la aplicación *Angular* es necesario incorporar el siguiente *script* en el *head* del archivo *index.html*, donde se sustituirá *YOUR_API_KEY* por la clave correspondiente:

```
<script src='https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY'></script
    >
```

Una vez añadida la directiva anterior ya es posible incrustar un mapa en la página web. Para hacer esto únicamente hay que poner en el *app.component.html* la siguiente directiva, *'<google-maps></google-maps>'* y ya sería posible ver un mapa por defecto como se observa en la imagen 8.20.

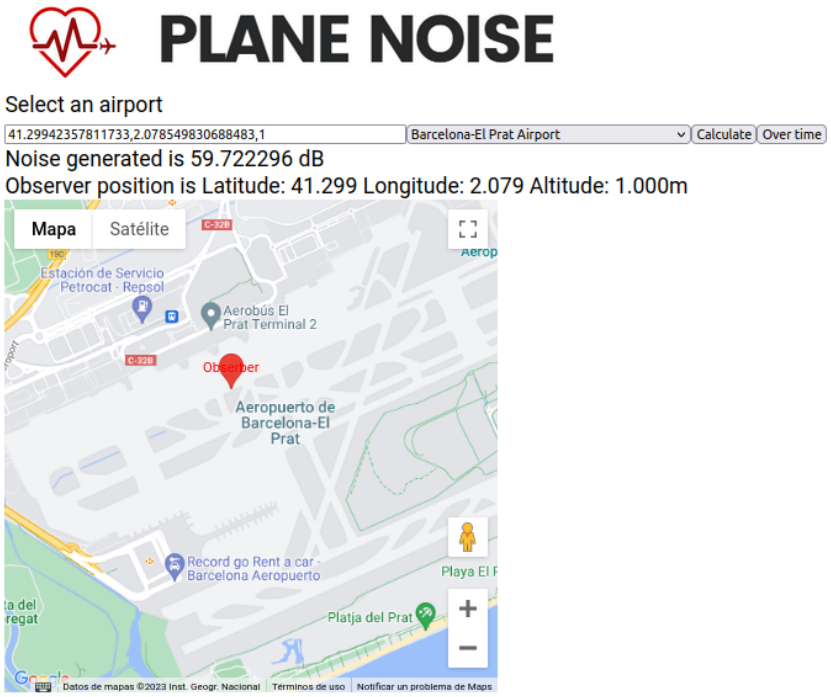

Figura 8.20: Mapa de *GoogleMaps* incrustado en el cliente con un observador

Una vez conseguido esto ya es posible implementar las tres funcionalidades mencionadas anteriormente.

Seleccionar punto de observación

El principal motivo por que que se substituye el uso de un *input* y se pasa a un mapa es la facilidad de uso. Si se está familiarizado con el uso de *Google Maps* se conocerá la posibilidad de añadir marcadores en el mapa. De la misma forma si se hace uso de la *API* también es posible el uso de estos marcadores.

Para implementar esta funcionalidad es necesario añadir, la siguiente etiqueta *HTML* dentro de la etiqueta <*google-maps*>.

```
<map-marker #markerElem *ngFor="let marker of markers" [position]="marker.
    position" [label]="marker.label"
        [title]="marker.title" [options]="marker.options">
</map-marker>
```

De esta forma y capturando el evento llamado *mapClick*, que posee los atributos, *event.latLng.lat()* y *event.latLng.lng()*, es posible generar un marcador añadiéndolo al *array* de *markers*. Otra de las ventajas del uso de *Google Maps* es que permite obtener también la altura correcta del observador. En la primera aproximación, la que usaba un *input* para escribir las tres variables de posición, no se tenía un conocimiento claro de la altura a la que estaba situada el observador. Para algunos aeropuertos como el de Barcelona esto no es muy relevante ya que están prácticamente al nivel del mar. Pero para otros casos, como el del aeropuerto de Madrid, empieza a tener un factor significativo al estar aproximadamente a 600 metros sobre el nivel del mar. Por suerte *Google Maps* posee también una *API* llamada *ElevationService* que tiene un método *getElevationForLocations()* el cual a partir de una posición, en este caso la del observador, devuelve la altura a la que se encuentra ese punto. De esta forma se obtienen de una forma sencilla e intuitiva la tres variables de posición requeridas para el observador como se puede observar en la imagen 8.21.

Centrar mapa

Figura 8.21: Altitud del observador en el aeropuerto de Valencia

Con el fin de facilitar la navegación en el mapa y evitar tener que buscar manualmente la localización de los aeropuertos, es posible centrar el mapa en un punto específico una vez que se ha seleccionado un aeropuerto de la lista desplegable o se ha cambiado a otro aeropuerto, ya que se cuenta con las coordenadas de estos últimos.

Esta funcionalidad es posible gracias a la existencia de la propiedad *'center'* dentro del componente *Google Maps*. Si se añade *'[center] = "center"'* a la etiqueta de *google-maps* se está enlazando la propiedad *'center'* con la variable *'center'* que gestiona el *AppComponent* y su modificación permite cambiar el centro del mapa.

Por lo tanto cuando se ejecute el método *airportChanged()*, que se lanza al realizar un cambio en el desplegable de aeropuertos, habría que añadir las siguientes líneas para conseguir el centrado del mapa en el aeropuerto seleccionado, donde *selectedAirport* posee los datos de latitud y longitud necesarios para centrar el mapa.

```
this.center = {
      lat: this.selectedAirport.latitude,
      lng: this.selectedAirport.longitude
}
```

Dibujar área

El último punto que se quiere implementar es el marcado de un área en el mapa que se encargará de mostrar de una forma visual la zona para la que se están obteniendo los vuelos. Como se ha mencionado, una de las funcionalidades de la *API* de *OpenSky* es el filtrado de vuelos para un espacio aéreo reducido. Para implementar esta funcionalidad es necesario añadir, de la misma forma que con al propiedad *'center'*, la siguiente etiqueta *HTML* dentro de la etiqueta <*google-maps*>.

```
<map-polyline [path]="vertices"></map-polyline>
```

A esta etiqueta se le pasan una serie de puntos en orden en el mapa y éste internamente se encarga de ir dibujando una linea uniendo estos puntos. Para generar el cuadrado es necesaria la creación de cinco puntos distintos, siendo el primero y el último iguales debido a que el último tramo del área llegará al inicio. Para ello se puede reutilizar la función que calculaba las *SquareCoords*, 8.11, cambiando los ángulos. A los ángulos usa-

Figura 8.22: Espacio aéreo sobre el que se van a obtener los aviones

dos anteriormente (45º y 225º) se le añadiría el cálculo de 135º y 315º. De esta forma sería posible obtener los puntos de la esquina inferior derecha y superior izquierda faltantes del cuadrado. El resultado sería como el mostrado en la imagen 8.22.

CAPÍTULO 9
Pruebas

En este capítulo se va a profundizar en las pruebas realizados sobre tanto la aplicación como el servidor mostrando la importancia de estos en el desarrollo de software.

En el libro Clean Code de Robert C. Martin [3], se hace énfasis en la importancia de escribir pruebas unitarias y de integración en el proceso de desarrollo de software. El autor destaca que la realización de pruebas es fundamental para garantizar la calidad del código y evitar errores y problemas en el funcionamiento de la aplicación.

En el caso de la aplicación que estamos desarrollando, que se encarga de calcular el ruido generado por los aviones en las proximidades de los aeropuertos, la realización de pruebas es esencial para asegurar que tanto servidor como cliente *front* funcionen correctamente y muestren que se han cumplido los requisitos enunciados en el capítulo 6.

En el mundo del *testing* existen muchos tipos de pruebas, pero para el proyecto desarrollado se ha optado por su sencillez y efectividad por la implementación de pruebas unitarias y de integración *end-to-end* (e2e).

9.1 Servidor

En primer lugar, se va a profundizar en las pruebas que debemos realizar en el servidor. El servidor está desarrollado en *Node.js* y se encarga de proporcionar una *API* para que el *frontend* pueda obtener a los datos necesarios para realizar los cálculos de ruido. En este caso, debemos asegurarnos de que la *API* funciona correctamente y de que los datos que devuelve son los esperados.

Para ello, es necesario realizar pruebas unitarias en cada uno de los *endpoints* de la *API* para asegurarnos de que los datos se están procesando correctamente y de que tanto las validaciones necesarias como los datos de respuesta son correctos.

En los primeros capítulos se ha mencionado una de las mayores problemáticas a las que se podía enfrentar el servidor no era la devolución correcta de los datos sino el uso del comando *exec*. La función *exec* de *childProcess* permite a *JavaScript* la ejecución de programas externos, como en el caso que nos incumbe.

Teniendo esto en cuenta hay que tener cuidado con su uso, ya que una incorrecta gestión de los parámetros de entrada podría llevar un agujero de seguridad importante donde se podría obtener el nivel más alto de brecha de seguridad, *RCE* o *Remote Code Execution* [27]

Por suerte existe una forma de prevenir esto realmente sencilla. Dado que los parámetros de entrada de nuestra *API* siguen un formato específico, es posible comprobar cada uno de ellos, ver si cumple dicho formato y en caso contrario devolver un error.

Como se ha mencionado en el apartado de Tecnologías 3.2 se va a usar el *framework Jest* para la realización de las pruebas sobre el servidor.

Se ha desarrollado una clase llamada *body.validator.js*. Esta clase expone y exporta un método llamado *isErrorBody* al que se le pasa el cuerpo enviado a la *API* como parámetro y se encargar de validad cada uno de los atributos. El valor de respuesta está formado por un *array* donde el primer valor indica si ha habido un error o no y el segundo valor es el mensaje de error con cada una de las propiedades que no han cumplido con el formato requerido por la *API*.

Para cada uno de los parámetros de la *API* el método se encarga de realizar las siguientes comprobaciones:

- *wingMouted* únicamente puede ser '1' o '0'.
- *turboFan* únicamente puede ser '1' o '0'.
- *mMode* únicamente puede ser 'A' o 'D'.
- *nMetric* únicamente puede ser 'EPM', 'LAm', PNL o'SEL'.
- *power* únicamente puede ser un número entero.
- *vSeg* únicamente puede ser un número entero o flotante.
- *planeCoords1* ha de tener 3 valores (x,y,z) separados por comas y estos valores únicamente pueden ser números enteros o flotantes.
- *planeCoords2* ha de tener 3 valores (x,y,z) separados por comas y estos valores únicamente pueden ser números enteros o flotantes.
- *npd* no puede ser una cadena vacía ni superior a 6 caracteres.

En caso de no cumplirse se mostraría un mensaje de error explicando cuál ha sido el problema de validación.

Un fragmento de la implementación de un caso de prueba sería así:

```
1 it("should return 400 incorrect turboFan", async () => {
2         let bodyNoise = generateBody()
3         bodyNoise.turboFan = "a"
4         const response = await request(baseURL).post("/api/calculate-noise").
              send(bodyNoise);
5         expect(response.statusCode).toBe(400);
6         const parsedResponse = JSON.parse(response.text)
7         expect(parsedResponse.error).toBe("The following properties have the
              wrong format: 'turboFan can only be 0 or 1'");
8 });
```

Donde se puede observar la forma de validar las entradas y los mensajes de error que se mostrarían. Para este caso se substituye el valor de *turboFan* por 'a', siendo este un valor no admitido.

Al realizar pruebas de forma práctica, un ejemplo podría ser que el parámetro *wingMounted* tenga un valor de 2 y *mMode* tenga un valor de 'C', ambos erróneos. La respuesta de salida antes esta entrada de datos sería un 400 *Bad Request* y se vería como se muestra en la imagen 9.1.

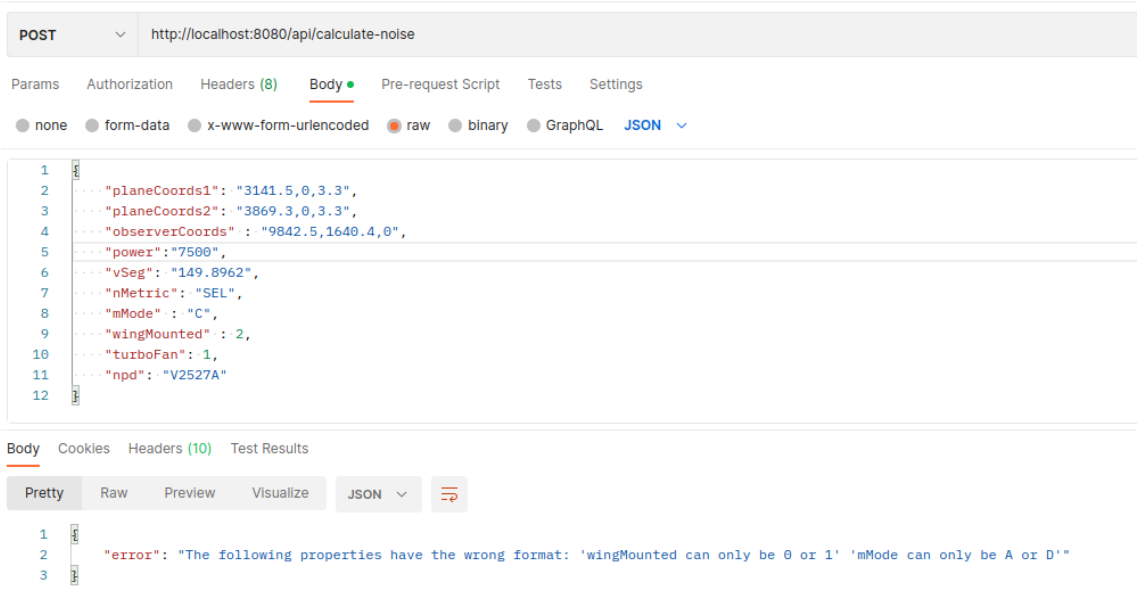

Figura 9.1: Salida *Postman* mostrando un error de validación

```
"error": "The following properties have the wrong format: 'wingMounted can only be 0 or 1'' mMode can only be A or D'"
```

Por lo tanto, es fácilmente observable dónde se ha cometido el error.

Una vez se ha comprobado de forma empírica el funcionamiento se procede a la creación de las pruebas unitarias encargados de asegurar el comportamiento de forma automatizada. Para ello se genera una clase *server.test.js* que contendrá cada una de las pruebas. Por facilidad se ha diseñado para que todos los test tengan el mismo formato:

- Se genera un *body* correcto.
- Se modifica una propiedad del cuerpo para hacerla incorrecta.
- Se envía la petición con el nuevo cuerpo.
- Se comprueba la respuesta.

Para poder ejecutar estas pruebas es necesario situarse en la carpeta del servidor y

- Poner en marcha el servidor con la orden '*npm server.js*'.
- Ejecutar las pruebas con la orden '*npm run test*'.

Esto da un resultado como el mostrado en la imagen 9.2, que demuestra que todos los test han pasado correctamente, asegurado el correcto funcionamiento del servidor de acuerdo con las especificaciones.

9.2 Cliente

En cuanto al *frontend* o cliente, es importante asegurarse de que está consumiendo correctamente la *API* proporcionada por el servidor y de que está mostrando los resultados

```
oscar@oscar-VirtualBox:~/workspace/plane-noise/docker/plane-noise-server$ npm run test

> plane-noise@0.0.0 test
> jest

 PASS  ./server.test.js
  Test /api/calculate-noise
    ✓ should return 200 (18 ms)
    ✓ should return 400 incorrect power (6 ms)
    ✓ should return 400 incorrect vSeg (2 ms)
    ✓ should return 400 incorrect nMetric (2 ms)
    ✓ should return 400 incorrect nMetric (2 ms)
    ✓ should return 400 incorrect wingMounted (2 ms)
    ✓ should return 400 incorrect turboFan (4 ms)
    ✓ should return 400 incorrect planeCoords1 (6 ms)
    ✓ should return 400 incorrect planeCoords1 not x,y,z (2 ms)
    ✓ should return 400 incorrect planeCoords2 (1 ms)
    ✓ should return 400 incorrect planeCoords2 not x,y,z (1 ms)
    ✓ should return 400 incorrect npd (1 ms)

Test Suites: 1 passed, 1 total
Tests:       12 passed, 12 total
Snapshots:   0 total
Time:        0.321 s, estimated 1 s
Ran all test suites.
```

Figura 9.2: Salida de la ejecución de pruebas unitarias usando *Jest*

de forma adecuada. En este caso, es necesario realizar pruebas sobre los servicios *Angular* del cliente, certificando que efectivamente los datos de respuesta y la lógica es correcta.

El principal problema en las pruebas unitarias del cliente es que dado que la mayor cantidad de datos, los de posición y relativos a los aviones, son proporcionados a través de una *API* externa. Esto dificulta el proceso de pruebas al no tratarse de un servicio propio y no tener control sobre él.

El primer paso que se ha llevado a cabo ha sido la configuración de un entorno de pruebas sin interfaz. Este paso ha sido realizado debido a que, prácticamente en el 100 % de los casos, las pruebas automatizadas son ejecutadas en un entorno de integración continua sin interfaz como podría ser *Jenkins* o *Azure DevOps*. Para ello se ha usado la tecnología *Chrome Headless* [20], que permite esta simulación de un entorno sin interfaz gráfica.

Para poder configurar este entorno sin interfaz hay que realizar una serie de pasos en el proyecto y en el entorno:

- Asegurar que existe una instalación de *Chromium* y que existe la variable de entorno *CHROMIUM_BIN*.
- Añadir en el fichero *karma.conf.js* un atributo llamado *customLaunchers*.

Este atributo *customLaunchers* tendrá la siguiente forma:

```
customLaunchers: {
      ChromiumHeadless: {
        base: 'Chromium',
        flags: [
          '--no-sandbox',
          '--headless',
          '--disable-gpu',
          '--remote-debugging-port=9222',
        ]
      }
},
```

Una vez se han realizado los pasos anteriores ya es posible la ejecución sin interfaz mediante la siguiente orden. Esta orden permita la finalización de la ejecución con el *–watch=false*, y usa el navegador personalizado que se ha descrito en *customLaunchers*.

```
ng test --watch=false --browsers=ChromiumHeadless
```

Pruebas unitarias sobre servicios

Las pruebas unitarias del cliente se han centrado especialmente en los servicios. Algunos de los servicios hacen uso de la dependencia *HttpClient* de *Angular*. Para poder ejecutar test sobre servicios que usan esta dependencia es necesario realizar cierta configuración sobre los archivos de *spec.ts*. Para ello es necesario añadir a la sección *imports* el *HttpClientTestingModule* debido a que no se puede usar la librería directamente y hay que usar una simulación de esta.

```
TestBed.configureTestingModule({
            providers: [AirportsService],
            imports: [HttpClientTestingModule]
});
```

Algún ejemplo de las pruebas unitarias realizadas podría ser el siguiente:

```
it('Should return airports from Http Get call.', () => {
        service.getAirports()
          .subscribe({
            next: (response) => {
              expect(response).toBeTruthy();
              expect(response.length).toBeGreaterThan(1);
            }
          });

        const mockHttp = httpTestingController.expectOne(AirportsService.
            API_ENDPOINT);
        const httpRequest = mockHttp.request;

        expect(httpRequest.method).toEqual("GET");

        mockHttp.flush(AIRPORT_RESPONSE);
});
```

En este fragmento de código se puede observar cómo se hace la llamada al método *getAirports()* del *AirportService*. Este método devuelve un *Observable* y debido a eso es necesario hacer uso de la función *subscribe()*. Internamente se comprueba que la respuesta tiene contenido y para finalizar es necesario cerrar la conexión (haciendo uso de *flush()*) ya que durante la ejecución de pruebas esta no es cerrada automáticamente y puede dar lugar a errores y filtraciones (*leaks*) de memoria.

Una vez se ha terminado la implementación de los test es posible ejecutarlos con la orden *npm run test-once*, si se ha creado un *script* nuevo en el package.json con la siguiente forma:

```
"test-once": "ng test --watch=false --browsers=ChromiumHeadless"
```

Dando como resultado la ejecución que se puede ver en la imagen 9.3 y donde también se observa que no se ha hecho uso de interfaz gráfica.

Pruebas interfaz

Debido al funcionamiento del propio servicio web, mostrar datos, las pruebas de interfaz tienen cierta importancia y permiten comprobar el correcto funcionamiento y carga de todos los componentes, así como el flujo de la aplicación.

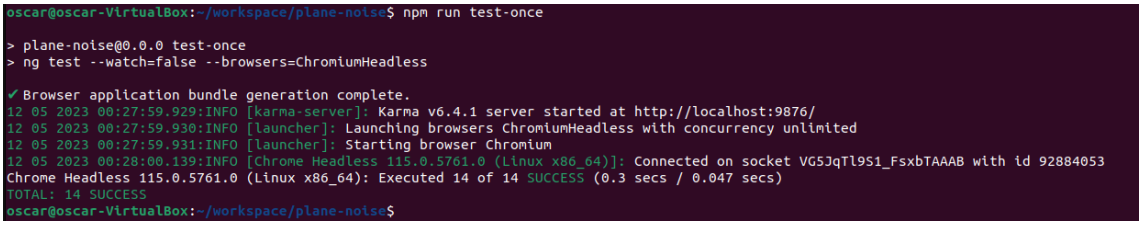

Figura 9.3: Salida de la ejecución de pruebas unitarios sobre los servicios

Para la implementación de este tipo de pruebas se ha hecho uso del framework *Cypress*. Este *framework* permite hacer una carga del servicio web e interacturar de forma gráfica como si de un humano se tratase. Es decir, permite rellenar campos, hacer selecciones y pulsar botones. Ésto permite crear una prueba de flujo y asegurar que todos los componentes están presente u ocultos cuando sea necesario.

Cypress provee de una interfaz gráfica para facilitar el uso del *framework*. Mediante el uso de esta interfaz y una clase en el proyecto se pueden diseñar las pruebas.

En este caso se ha implementado una pequeña prueba que consiste en lo siguientes pasos:

- Visitar la web en la dirección base.
- Seleccionar de la lista el aeropuerto, Adolfo Suárez Madrid-Barajas.
- Rellenar el campo de posición del observador con una posición cercana al aeropuerto.
- Comprobar que el mapa de *Google Maps* aparece.
- Pulsar el botón de cálculo de ruido para un instante.
- Comprobar que existe el componente de la tabla.
- Comprobar que las cabeceras del componente de la tabla son correctas

Y se ha implementado de la siguiente forma:

```
1 it('Check all components are up', function () {
2     cy.visit('/')
3     cy.get('select').select('Adolfo Suarez Madrid-Barajas Airport')
4     cy.get('input').type('40.50362984302942,-3.5594843034919244,590.70849609375
          ')
5     cy.get('google-map').first().should('exist');
6     cy.get('button').first().click()
7     cy.get('table').should('exist');
8     cy.get('table').contains('th', 'Latitude');
9  })
```

Si se ejecuta esta prueba es posible ver el resultado de su ejecución en directo y comprobar como cada una de las acciones anteriormente especificadas se van realizando sobre la web. La interfaz gráfica de *Cypress* muestra la ejecución como aparece en la imagen 9.4.

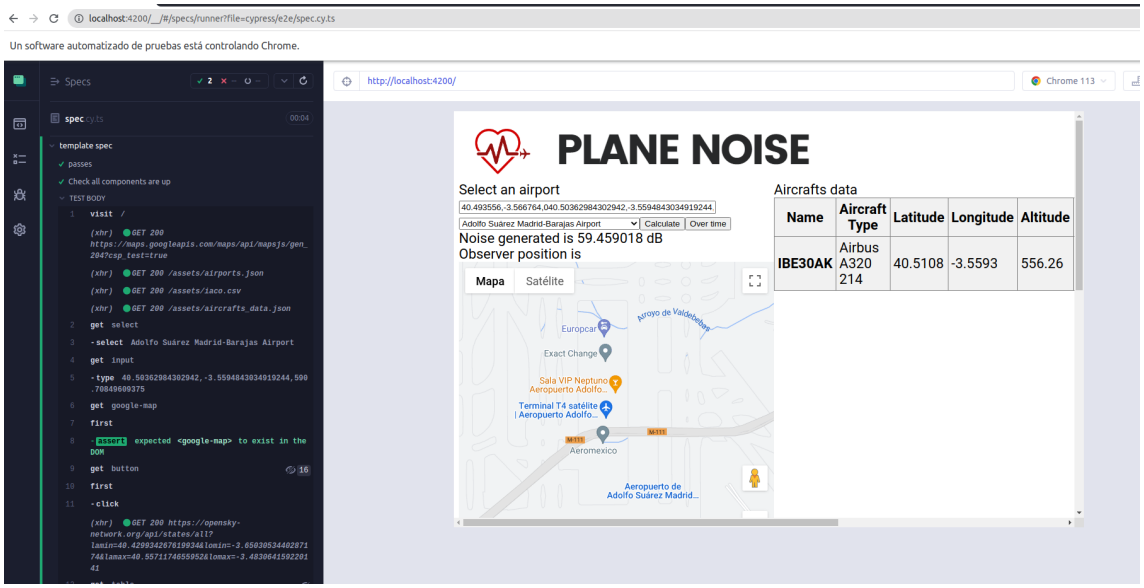

Figura 9.4: Salida ejecución de las pruebas sobre la interfaz usando *Cypress*

CAPÍTULO 10
Ejemplo de uso

En este capítulo se va a mostrar un caso de uso con los dos modos de funcionamiento de la aplicación web, instante y captura continua. Dado que el servicio no está desplegado para su acceso público hay que iniciar la aplicación en local. Para ello se ejecuta la siguiente orden desde línea de comandos mostrando la salida de la imagen 10.1.

```
npm run start
```

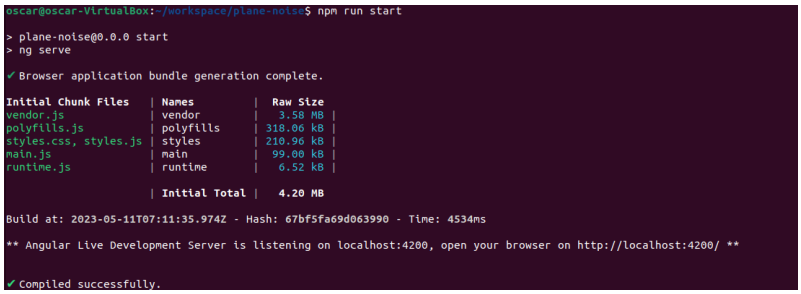

Figura 10.1: Salida ejecución del comando *npm run start* para inicar la aplicación en local

Una vez la aplicación está levantada se puede navegar a la dirección estándar de *Angular* en el puerto 4200 donde nos mostrará la página del servicio web desarrollado.

En el menú desplegable se seleccionará el aeropuerto sobre que el se van a realizar las mediciones. Luego se indicará la posición del observador y ya se podrá pulsar sobre el botón de *Calculate* o *Over time* dependiendo si se quiere captura el ruino en un instante o si se quiere capturar a través del tiempo.

Si se elige por ejemplo el aeropuerto de "Alicante-Elche" para un instante en el tiempo se puede obtener un resultado como el mostrado en la imagen 10.2 mostrado.

Si por el contrario se usa la opción de "*Over time*" para un cálculo continuo de datos se obtiene el siguiente resultado mostrado en la imagen 10.4, donde se aprecia la evolución del ruido al que está expuesto el observador a los largo de 2 minutos.

10.1 Comparativa con Webtrack

El principal rival al que se enfrenta el trabajo es al sistema actualmente desplegado de micrófonos, *Webtrack*. Este sistema como se ha mencionado tiene una precisión perfecta pero un coste asociado debido a la necesidad de instalar estos sistemas de medición.

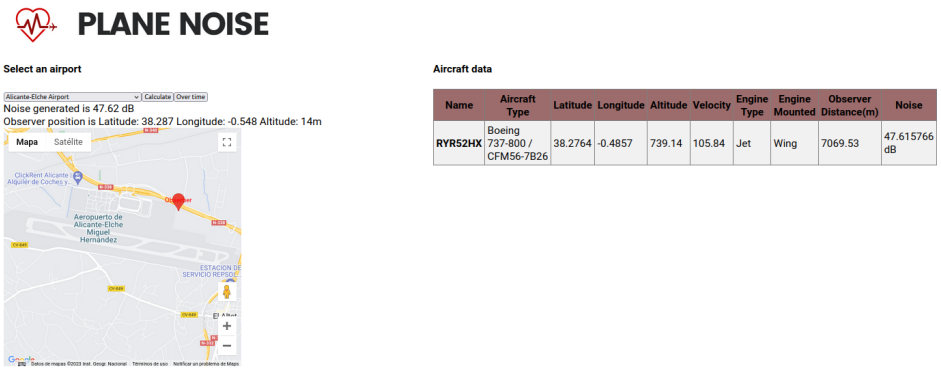

Figura 10.2: Salida cálculo del ruido para un instante concreto

Figura 10.3: Ruido calculado por *Webtrack* en la posición del observador

Para la realización de esta prueba es necesario acceder a `https://webtrak.emsbk.com/bcn` para acceder a los micrófonos instalados en las proximidades del aeropuerto de Barcelona.

Dado que hay un pequeño retraso en los datos que llegan de la *API* hay que empezar a capturar datos con cierta antelación para poder hacer la comprobación en ambos servicios.

El sistema *Webtrack* muestra la aproximación de una aeronave con código *TAP41XC* modelo *Airbus A320* y un ruido generado de 75 dB como se puede observar en la imagen 10.3.

Por otro lado el sistema *Plane Noise* muestra un resultado de 78.83 dB como el mostrado en la imagen 10.4.

De los datos anteriores se puede extraer que el sistema es muy exacto incluso sin tener en cuenta las condiciones atmosféricas ya que, el error se en encuentra en un 1.2 %. Estos resultados son mejores de lo esperado en un principio debido que al tratarse de un sistema teórico el margen de error se presuponía en el rango de un 15 %.

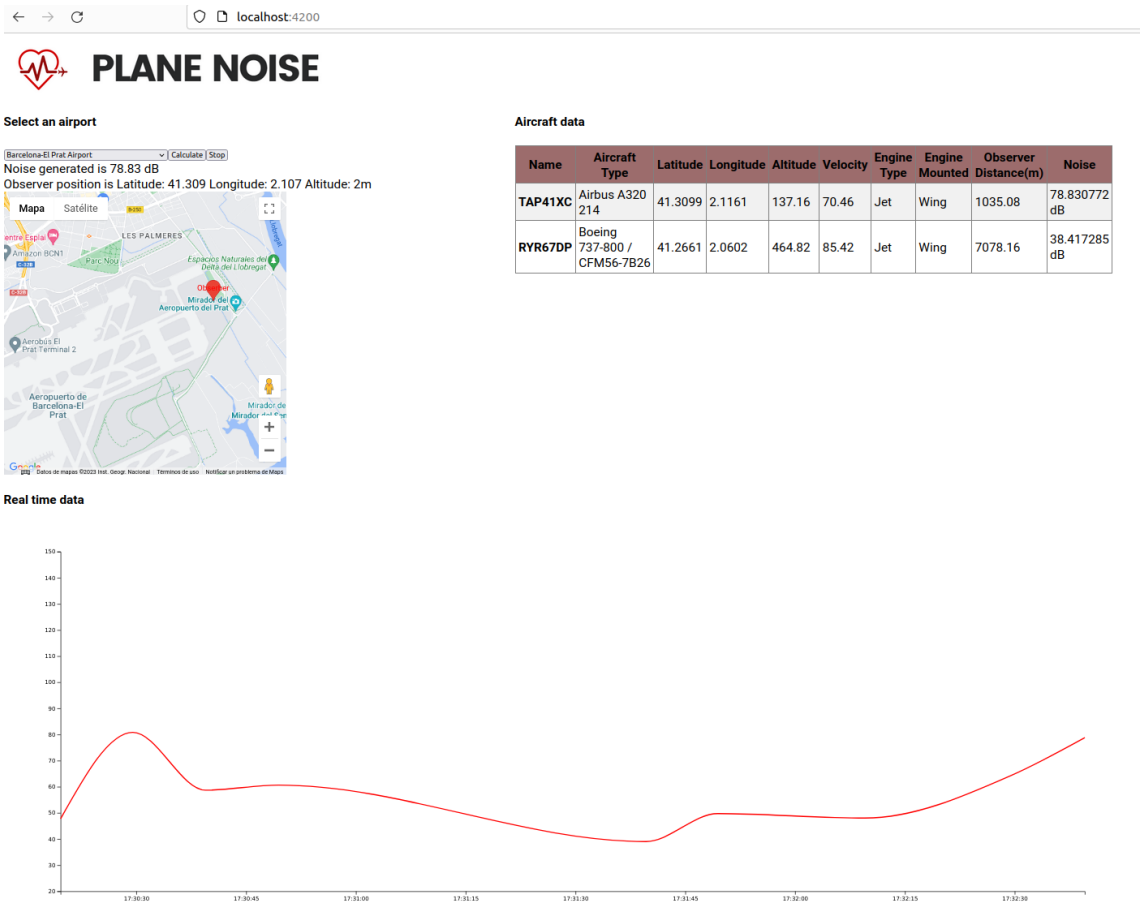

Figura 10.4: Ruido calculado por *Plane Noise* en la posición del observador

CAPÍTULO 11
Conclusiones

El desarrollo de este proyecto ha sido desafiante, pero al mismo tiempo satisfactorio. Aunque parte de las tecnologías utilizadas eran conocidas debido a experiencia profesional que se poseía en su uso otras han sido completamente nuevas y ha tocado realizar la curva de aprendizaje para poder usarlas.

En este trabajo se ha observado el proceso completo desde el análisis hasta los casos de prueba. El apartado de Pruebas (Capítulo 9), se muestra de una forma clara cómo se han conseguido todos los requisitos inicialmente propuestos en el apartado de Objetivos (Sección 1.2), mejorando en algunos aspectos. Además, se ha conseguido una aplicación estable, segura, rápida y fácil de usar para un usuario sin conocimiento técnico.

A pesar del éxito obtenido el trabajo no ha estado exento de problemas durante el desarrollo. Estos problemas han estado centrados principalmente en el uso de *API* externas ya que no se tiene control sobre ellas y se depende, si hay problemas, del soporte que se pueda obtener de las empresas u organizaciones que gestión estos servicios. En algunas de estas *API* no se ha obtenido respuesta del soporte para problemas que se tenía con su uso y con los datos que se obtenían. Como curiosidad durante el desarrollo del trabajo se encontró una vulnerabilidad grave en uno de los servicios, que ya ha sido comunicada, y permitía el acceso a datos con un usuario no autorizado.

Para cerrar se puede decir que se ha conseguido desarrollar un prototipo funcional de web que permite la visualización del ruino generado en aeropuertos e integrar la librería proporcionada en forma de un servidor que permite peticiones *REST*.

11.1 Relación con estudios cursados

El trabajo tiene relación con el Máster Universitario en Ingeniería Informática cursado por el alumno. Un punto crítico ha sido la gestión y Planificación de Proyectos. Un trabajo de esta envergadura requiere de una planificación previa concisa y clara como la que se ha expuesto en el apartado Planificación (Capítulo 5). El máster provee de unas competencias como "Capacidad para analizar las necesidades de información que se plantean en un entorno y llevar a cabo en todas sus etapas el proceso de construcción de un sistema de información." [30] que se ha puesto a prueba durante el trabajo ya que componen el ciclo de vida completo de este. Otro punto para tener en cuenta es el relacionado con la seguridad del software. Como se menciona se hace uso de llamadas sensibles a ser explotadas debido al nivel de autorización que poseen. Pero gracias al conocimiento obtenido durante el desarrollo del Máster en Ingeniería Informática se ha podido solventar y conseguir una solución segura en este aspecto. Además, como parte del máster se realizó en

el extranjero mediante el Programa Erasmus, también se obtuvieron conocimiento en la rama de *testing* avanzado que se han puesto en uso en la realización del trabajo.

11.2 Trabajo futuro

A pesar de que el trabajo ya cumple con el alcance requerido en un principio, se han identificado ciertos puntos de mejora que podrían llevarse a cabo en futuras iteraciones:

- Añadir soporte para varios idiomas en la web.
- Mejorar la seguridad del servidor mediante el uso de un *reverse proxy* que permitiese únicamente el uso de llamadas *HTTPS*.
- Mostrar alamas si se sobrepasan ciertos niveles de ruidos no recomendados.
- Guardar un histórico de ruidos generados para permitir el análisis y poder proveer de propuestas de mejora.
- Dado que ciertos componentes tienen un coste de uso asociado, como puede ser *Google Maps*, si se sobrepasa cierto nivel de peticiones mensuales, se podría crear una versión de pago del servidor y que su uso por fuentes externas tuviese el mismo paradigma, pago por uso.
- Siguiendo la línea anterior, se podría proveer de funcionalidad de pago (como la mencionada de tener un histórico de datos) a usuarios.
- Desplegar el servicio en un entorno real para su uso.
- Dado que la librería tiene un parámetro para condiciones atmosféricas sería interesante añadir el uso de una *API* de tiempo para poder mejorar la exactitud de las mediciones.

Bibliografía

[1] Rafael Casado, Aurelio Bermúdez, Enrique Hernández-Orallo, Pablo Boronat, Miguel Pérez-Francisco, Carlos T. Calafate. Pollution and noise reduction through missed approach maneuvers based on aircraft reinjection. *Elsevier Transportation Research Part D: Transport and Environment*, Volumen 114, enero, 2023.

[2] Matthias Schäfer, Martin Strohmeier, Vincent Lenders, Ivan Martinovic and Matthias Wilhelm. Bringing Up OpenSky: A Large-scale ADS-B Sensor Network for Research. *In Proceedings of the 13th IEEE/ACM International Symposium on Information Processing in Sensor Networks (IPSN)*, págs. 83-94, abril, 2014.

[3] Robert C.Martin. *Clean Code*. Prentice Hall, 2008.

[4] Webtrack Aeropuerto Barcelona. Consultado el 15 de diciembre de 2022 https://www.aeropuertobarcelona-elprat.com/cast/webtrak-aeropuerto-de-barcelona.htm.

[5] Angular CLI. Consultado el 10 de diciembre de 2022 https://angular.io/cli.

[6] Node. Consultado el 12 de diciembre de 2022 https://nodejs.org/es.

[7] Angular. Consultado el 10 de diciembre de 2022 https://angular.io.

[8] Cypress. Consultado el 20 de diciembre de 2022 https://www.cypress.io/.

[9] Docker. Consultado el 29 de diciembre de 2022 https://www.docker.com/.

[10] Docker Compose. Consultado el 30 de diciembre de 2022 https://docs.docker.com/compose/.

[11] Scrum. Consultado el 4 de diciembre de 2022 https://www.atlassian.com/es/agile/scrum.

[12] Metodología ágil. Consultado el 6 de diciembre de 2022 https://www.redhat.com/es/devops/what-is-agile-methodology.

[13] OpenSky. Consultado el 8 de enero de 2023 https://openskynetwork.github.io/opensky-api/rest.html.

[14] AviationStack. Consultado el 8 de enero de 2023 https://aviationstack.com/.

[15] FlightAware. Consultado el 8 de enero de 2023 https://es.flightaware.com/.

[16] Cirium Flight Stats. Consultado el 8 de enero de 2023 https://www.cirium.com/.

[17] OAG Flight Status. Consultado el 8 de enero de 2023 https://www.oag.com/.

[18] AeroDataBox. Consultado el 8 de enero de 2023 https://aerodatabox.com/.

[19] Jest. Consultado el 8 de enero de 2023 https://jestjs.io/es-ES/.

[20] Chrome Headless. Consultado el 21 de diciembre de 2022 https://developer.chrome.com/blog/headless-chrome/.

[21] D3Js. Consultado el 20 de enero de 2023 https://d3js.org/.

[22] Chrome Headless. Consultado el 7 de febrero de 2023 https://www.typescriptlang.org/.

[23] Google Maps. Consultado el 14 de marzo de 2023 https://developers.google.com/maps?hl=es-419.

[24] OpenLayers. Consultado el 16 de marzo de 2023 https://openlayers.org/.

[25] JavaScript. Consultado el 6 de abril de 2023 https://developer.mozilla.org/es/docs/Web/JavaScript.

[26] Git. Consultado el 29 de noviembre de 2022 https://git-scm.com/.

[27] Code Injection. Consultado el 10 de diciembre de 2022 https://owasp.org/www-community/attacks/Code_Injection.

[28] IEEE 930-1998. Consultado el 29 de diciembre de 2022 https://ieeexplore.ieee.org/document/720574.

[29] Distancia de Levenshtein. Consultado el 3 de marzo de 2023 https://es.wikipedia.org/wiki/Distancia_de_Levenshtein.

[30] Competencias MUIINF. Consultado el 4 de mayo de 2023 https://muiinf.webs.upv.es/plan-de-estudios/competencias/.

APÉNDICE A
Entorno de programación y pruebas

En esta sección se introducirán las dos herramientas principales que se han usado para el desarrollo técnico del trabajo.

A.1 Visual Studio Code

Visual Studio Code(también conocido como VS Code) es un editor de código creado y desarrollado por *Microsoft*. Es muy popular entre los desarrolladores debido a su gran cantidad de características y extensiones disponibles que lo hacen altamente personalizable y adaptable a las necesidades individuales.

Dado que el proyecto ha sido desarrollado en su mayoría usando tanto *JavaScript*, en la parte del servidor, como *TypeScript*, *HTML* y *CSS* en la parte del cliente se ha optado por el uso de este IDE.

A.2 Postman

Postman es una herramienta que permite a los desarrolladores probar, documentar y compartir sus API de una forma eficiente y sencilla. Esta aplicación provee de una interfaz gráfica de usuario (GUI) para crear, enviar y recibir solicitudes *HTTP* y *HTTPS* a través de una *API*.

APÉNDICE B
Scripts

En esta sección se introducirán distintos *scripts* usados para la modificación y obtención de datos que si bien no tiene una relación directa con el trabajo son fundamentales para facilitar el procesado de datos llevado a cabo. Si bien no son necesarias en el núcleo del trabajo dan un contexto sobre cómo se ha realizado ese trabajo y como se han solucionado ciertos problemas a lo largo del desarrollo.

Como se ha mencionado durante el desarrollo el servidor hace uso en uno de sus parámetros de una ruta a un fichero que contiene el modelo de ruido de una aeronave en específico. Para obtener este fichero ha sido necesario acceder a la web de con un usuario autenticado. Para poder acceder a la web es necesario que la empresa que la gestiona apruebe la creación del usuario que usualmente tarda una semana, pero en este caso concreto se demoró durante casi 1 mes la espera.

Para la descarga de este fichero existían dos posibilidades o la descarga de cada uno de los ficheros de forma individual, con el consiguiente esfuerzo innecesario ya que se trataba de 155 ficheros distintos o la página web también permitía descargarlos todos a la vez, pero en un único CSV.

Se ha optado por la segunda opción y se ha desarrollado el siguiente *script* que se encarga de recorrer el CSV y, dado que se distinguen por el valor de la primera columna usa ese valor como clava de un mapa y luego crea un fichero con las mismas cabeceras del fichero origina y todos los valores que ha ido almacenando el valor del mapa.

```
import csv
import json
import logging
with open('NPD_data.csv') as f:
    reader = csv.reader(f, delimiter=";")
    header = next(reader) # Lee la primera fila del archivo CSV y la almacena
        en la variable "header"
    data_dict = {}
    for row in reader:
        key = row[0]
        if key not in data_dict:
            data_dict[key] = []
        data_dict[key].append(row)
    for key, value in data_dict.items():
        with open(f'{key}.csv', 'w', newline='') as f:
            writer = csv.writer(f, delimiter=";")
            writer.writerow(header)
            for row in value:
                writer.writerow(row)
```

El segundo *script* que se ha desarrollado se encarga de recorrer un fichero CSV también descargado de la página web de . Este *script* crea un fichero JSON que será usado en

la aplicación con distintos datos necesarios para el uso del servidor que no devuelven las API de navegación. Para este caso se extraen los siguientes datos:

- AcftId: modelo de forma abreviada.
- Description: modelo del avión.
- EngineType: hélice o jet.
- NpdId: fichero de datos a usar
- EngineMounted: montaje motor en alas o fuselaje.

```
import csv
import json

csv_filename = "Aircraft.csv"
json_filename = "aircrafts.json"
data = []
with open(csv_filename) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=";")
    next(csv_reader)
    for row in csv_reader:
        acft_id = row[0]
        description = row[1]
        engine_type = row[2]
        npd_id = row[11]
        engine_mounted = row[15]
        row_data = {
            "ACFT_ID": acft_id,
            "description": description,
            "engineType": engine_type,
            "NPD_ID": npd_id,
            "engineMounted": engine_mounted
        }
        data.append(row_data)
with open(json_filename, "w") as json_file:
    json.dump(data, json_file)
```

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**	X			
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**		X		
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**			X	
ODS 12. **Producción y consumo responsables.**	X			
ODS 13. **Acción por el clima.**	X			
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**	X			
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

El desarrollo de una web para la evaluación del nivel de ruido generado por los aviones en aeropuertos. es un proyecto que tiene implicaciones importantes en varios Objetivos de Desarrollo Sostenible (ODS). En esta sección se hará una reflexión sobre cómo estos objetivos se relacionan con el trabajo y cuáles son los más relevantes.

ODS 3 El primer objetivo de desarrollo sostenible que se relaciona con este proyecto es el número 3, "Salud y Bienestar". El ruido de los aviones puede ser perjudicial para la salud de las personas que viven cerca de los aeropuertos. Las personas expuestas a niveles altos de ruido pueden experimentar problemas de sueño, estrés, hipertensión y enfermedades cardiovasculares. La creación de un servicio web que calcule el ruido generado por los aviones en los alrededores de los aeropuertos puede ayudar a minimizar los impactos negativos en la salud de las personas y contribuir al objetivo de mejorar la salud y el bienestar de la población.

ODS 9 El segundo objetivo que se relaciona con este proyecto es el número 9, "Industria, Innovación e Infraestructura". Tiene como función el fomento de la innovación y desarrollo tecnológico así como la promoción de infraestructuras resilientes, sostenibles y de calidad. El presente trabajo contribuye a este objetivo al ofrecer una herramienta innovadora y tecnológica que puede mejorar la calidad de vida de las personas que viven cerca de los aeropuertos al poder tener una idea más clara de la posible contaminación acústica que se puede generar por la proximidad de las aeronaves para la construcción de viviendas con características especiales para resistir este tipo de contaminación y de esta forma conseguir la minimización de los impactos negativos en el medio ambiente.

ODS 11 El tercer objetivo que se relaciona con este proyecto es el número 11, "Ciudades y Comunidades Sostenibles". Este objetivo tiene como objetivo hacer que las ciudades y los asentamientos humanos sean más inclusivos, seguros, resilientes y sostenibles. El ruido de los aviones es un problema importante para las comunidades cercanas a los aeropuertos, ya que puede afectar negativamente la calidad de vida y el bienestar de las personas que viven allí. La creación de este servicio puede contribuir a hacer que estas comunidades sean más seguras y sostenibles al proporcionar información precisa y oportuna sobre los niveles de ruido y su impacto en el medio ambiente y la salud humana.

ODS 12 El Objetivo 12 de Desarrollo Sostenible es "Producción y Consumo Responsables", que tiene como objetivo garantizar patrones sostenibles de producción y consumo en todo el mundo, reduciendo el impacto ambiental negativo de los procesos de producción y consumo. Este objetivo también se encuentra directamente relacionado con el desarrollo de este trabajo. Primero, el transporte aéreo es una importante fuente de emisiones de gases de efecto invernadero, lo que contribuye al cambio climático. El desarrollo de un servicio web que calcule el ruido generado por aviones en los alrededores de los aeropuertos puede contribuir a reducir la necesidad de vuelos innecesarios o mal programados, lo que a su vez puede ayudar a reducir las emisiones de gases de efecto invernadero. Además, la información proporcionada por el servicio

web puede ayudar a las autoridades de aviación a tomar decisiones más informadas sobre cómo programar los vuelos para minimizar su impacto ambiental negativo.
Por otra parte, el proyecto de desarrollo del servicio web también puede ayudar a promover prácticas de producción y consumo más responsables. Al proporcionar información precisa sobre el ruido generado por los aviones, el servicio web puede ayudar a las empresas de transporte aéreo a ajustar sus operaciones y reducir el impacto negativo en el medio ambiente. Además, el acceso a esta información puede ayudar a los consumidores a tomar decisiones más informadas sobre sus opciones de viaje, lo que puede fomentar un cambio hacia prácticas de transporte más sostenibles.

ODS 13 El quinto objetivo que se relaciona con este proyecto es el número 13, "Acción por el Clima". El transporte aéreo es una importante fuente de emisiones de gases de efecto invernadero y contribuye significativamente al cambio climático. La creación de un servicio web que calcule el ruido generado por los aviones puede ayudar a minimizar el impacto negativo del transporte aéreo en el medio ambiente al proporcionar información sobre los niveles de ruido y su relación con las emisiones de gases de efecto invernadero.

ODS 15 Y para terminar el último objetivo relacionado es el Objetivo 15, "Vida de Ecosistemas Terrestres", que tiene como objetivo proteger, restaurar y promover la utilización sostenible de los ecosistemas terrestres, gestionar los bosques de manera sostenible, combatir la desertificación, detener la degradación del suelo y revertir la pérdida de biodiversidad. Si bien a primera vista puede parecer que el proyecto de desarrollo del servicio web que calcula el ruido generado por los aviones no tiene una relación directa con este objetivo, hay ciertos aspectos en los que se puede establecer una relación. El ruido de los aviones puede tener efectos negativos en la vida silvestre que habita en los alrededores de los aeropuertos. El ruido excesivo puede causar estrés, interferir en la comunicación y reducir la calidad del hábitat de las especies, lo que puede tener efectos negativos en la diversidad y la salud de las poblaciones de vida silvestre. La creación de un servicio web que calcule el ruido generado por los aviones en los alrededores de los aeropuertos puede ayudar a identificar las áreas donde el ruido es más intenso y, por lo tanto, tomar medidas para minimizar su impacto en la vida silvestre. Por otra parte, el transporte aéreo es una importante fuente de emisiones de gases de efecto invernadero, lo que contribuye al cambio climático, que a su vez puede tener efectos negativos en los ecosistemas terrestres. La creación de un servicio web que calcule el ruido generado por las aeronaves puede contribuir a reducir la necesidad de vuelos innecesarios o mal programados, lo que a su vez puede ayudar a reducir las emisiones de gases de efecto invernadero y, por lo tanto, proteger mejor los ecosistemas terrestres.

ETS Enginyeria Informàtica
Camí de Vera, s/n, 46022, València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es