



2022

Balloon Simulator

TFG DESARROLLO DE APLICACIONES WEB
ÁLVARO CARBAJO ALCALDE

ÍNDICE

Portada.....	1
Índice	2
Introducción.....	4
Objetivos	5
Descripción general	6
Tecnologías utilizadas	7
Base de datos	7
Back-end.....	7
Front-end	8
Simulador	8
Servicios de terceros	9
Otras tecnologías.....	10
Despliegue	11
Docker	12
Fases del proyecto	13
Sprint 1.....	16
Mapeado del simulador	16
Diseño.....	16
Implementación.....	17
Modelo del globo	22
Resultado final de la escena	23
Físicas	25
Físicas del Viento	26
Interfaz de usuario	28
Diseño.....	28
Sprint 2.....	31
Base de datos	31
Back-end.....	33
Diseño de las vistas	37
Api	40
Pruebas.....	42

Inserción de datos	43
Sprint 3.....	44
Angular	44
Diseño.....	45
Implementación.....	46
Pruebas.....	47
Sprint 4.....	49
Sprint 5.....	56
Historial	56
Flightview	57
Sprint 6.....	63
Pruebas.....	64
Comentarios.....	65
Ampliación y mejoras.....	66
Conclusión.....	67
Bibliografía	68

INTRODUCCIÓN

Globos Arcoiris es una empresa riojana del sector turístico dedicada a actividades con **globos aerostáticos**. Está localizada en Cuzcurrita de Río Tirón. Este proyecto busca realizar una aplicación web para dicha empresa, en la cual se puedan simular los vuelos que se desarrollan por La Rioja Alta, Burgos y Álava.



El **objetivo** del proyecto será diseñar e implementar una **aplicación web** en la que se puedan simular de la forma más realista posible vuelos en globo aerostático. La aplicación contará con varias secciones para la visualización de los datos que se recojan durante la simulación, rutas del trayecto recorrido...

La web tendrá también un apartado específico que permitirá realizar **consultas meteorológicas** que ayuden a los pilotos con la planificación de los vuelos y les permita seleccionar el punto de despegue idóneo para un trayecto agradable.

La idea para este proyecto me la dio **Valentín Carbajo**, piloto de Globos Arcoiris que me motivó durante el desarrollo de la web y me aportó ideas e información sobre los vuelos y su funcionamiento, así como sobre los puntos de despegue más comúnmente usados por la empresa y la localización de los mismos.

OBJETIVOS

Estos son los **objetivos** que me he propuesto para la realización de este proyecto:

- Crear un **simulador** con un **mapa de la Rioja Alta**, con unas **físicas realistas** y controles basados en el manejo real de un globo aerostático.
- Desarrollar un **front-end** que aloje el simulador y varias páginas con utilidades.
- Diseñar e implementar una **base de datos** que permita guardar datos de vuelos, puntos de despegue...
- Servirse de alguna API de terceros para poder **visualizar en tiempo real la ruta seguida por el aerostato** y usarla, además, para **elegir punto de despegue**.
- Insertar **puntos de despegue reales** usados por la empresa.
- Listar los vuelos efectuados en el simulador y extraer datos de los mismos como su duración, velocidades medias...
- Mostrar con **gráficas** los **datos** de altura, velocidad... del vuelo.
- Mostrar las **rutas realizadas en simulaciones anteriores** en un mapa interactivo.
- Desarrollar un **back-end** que permita gestionar **CRUDs** de las tablas de la base de datos para insertar, visualizar o modificar la información.
- Implementar una **API** que permita al front-end comunicarse con el back-end mediante peticiones.
- Desplegar la aplicación **con Docker**, creando las imágenes, contenedores y volúmenes necesarios.
- Visualizar en el simulador a modo de **repetición** los **vuelos** una vez terminados, permitiendo cambiar la velocidad de reproducción.
- Generar un **código legible**, claro, en **inglés** y con **comentarios**.

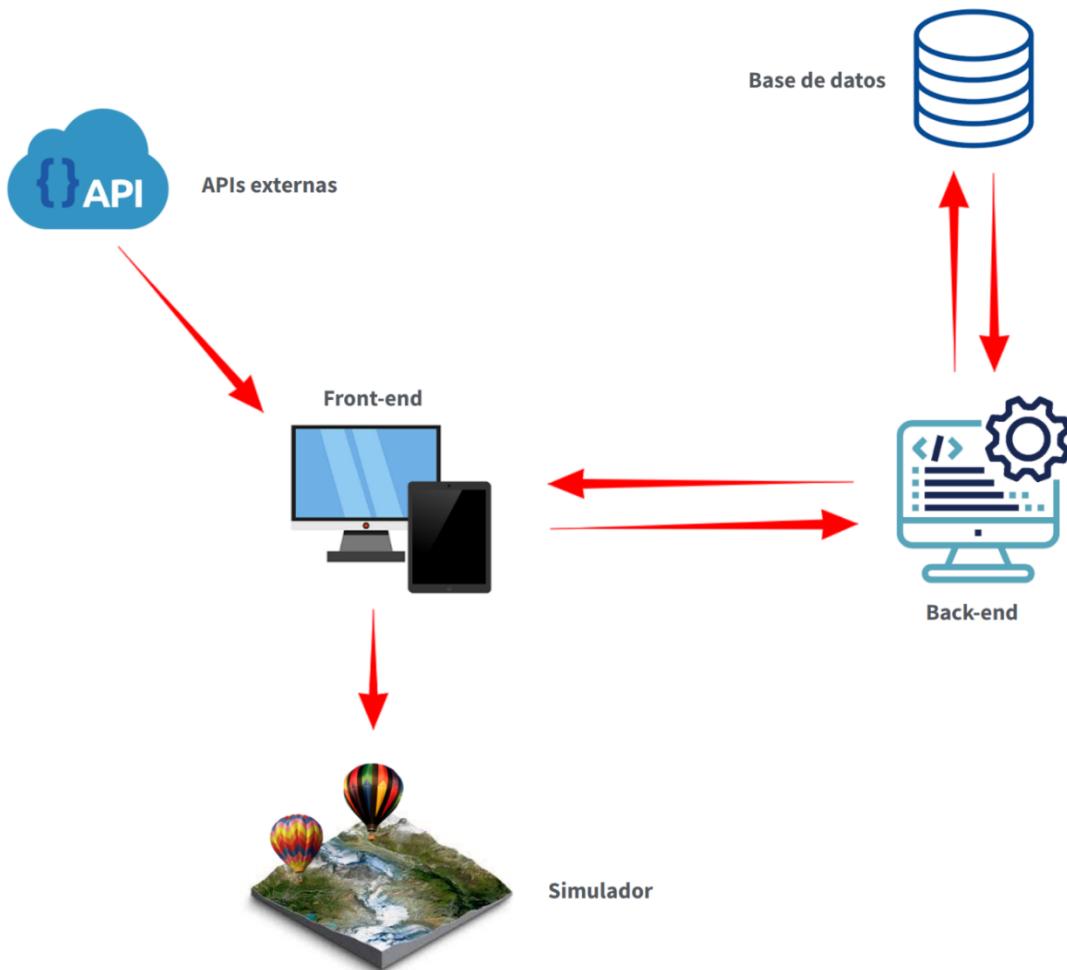
DESCRIPCIÓN GENERAL

La web estará formada por **cuatro partes** que funcionarán en conjunto para cumplir con los objetivos especificados. Estas partes son: el simulador, un front-end, un back-end y un sistema **gestor de base de datos**.

El **back-end** contendrá dos partes principales: una **API** para comunicarse con el front-end y una serie de **vistas** que permitirán a un usuario administrador gestionar la base de datos. Será el encargado de intercambiar información con la base de datos.

El **front-end** se servirá del back-end y de APIs externas para obtener los datos necesarios para el funcionamiento de la web. Esta parte estará dirigida al usuario final y será la encargada de lanzar el **simulador**.

Diagrama de diseño: Las flechas rojas representan el flujo de datos.



TECNOLOGÍAS UTILIZADAS

BASE DE DATOS

Para la gestión de datos utilizaré **MariaDB**. Una base datos relacional de código abierto alternativa a MySQL que nace tras la compra de éste por parte de Oracle. He escogido MariaDB sobre **otras opciones** como MySQL o SQLServer porque es software libre y tiene un mayor rendimiento que MySQL.



BACK-END

El back-end estará desarrollado en **Laravel**, un framework de **PHP** muy potente y muy orientado a desarrollo servidor. Utiliza como lenguajes HTML, PHP y CSS. También usa **Blade**, un motor de plantillas simple que otorga muchas facilidades para desarrollar vistas.



Desarrollaré una **API**, que permita al front-end comunicarse con los datos, y una serie de vistas orientadas a un administrador que realice la gestión de los datos. Me serviré de la estructura **MVC** que aporta Laravel para organizar el código de forma correcta y escalable. Para las vistas usaré Blade, HTML y **Bootstrap**. Bootstrap es un framework de CSS que ofrece un conjunto de herramientas de código abierto para el diseño de sitios web. He elegido Bootstrap porque puedes instalarlo por defecto en Laravel y ahorra mucho tiempo de maquetación de estilos.



He rechazado **otras opciones** como Symfony porque Laravel tiene una curva de aprendizaje más sencilla y tiene un mejor rendimiento. Laravel cuenta además con una documentación más concisa y permite separar con facilidad la parte de API de la parte web.



Bootstrap

FRONT-END

En el front-end, **Angular** era la opción más clara. Es un framework de JavaScript desarrollado por Google en 2010. Es uno de los frameworks front-end más populares junto con React. Sus grandes ventajas son la flexibilidad, la integración de gran cantidad de librerías de terceros y el uso de TypeScript.



TypeScript es un lenguaje de programación libre y de código abierto desarrollado por Microsoft. Es un superconjunto de JavaScript, que esencialmente añade un tipado de clases. Su lenguaje, al contrario que JavaScript, está muy orientado a objetos, pero su gran desventaja es que requiere una compilación a JS previa a su ejecución.



Para las vistas no utilizaré ningún framework de CSS/HTML porque quiero diseñar las páginas de forma muy concreta y personalizada para el usuario final de la web.

SIMULADOR

Para el simulador he escogido un framework llamado **Babylon.js** que es un motor 3D desarrollado por Microsoft escrito en **JavaScript**. Se basa en una escena que se renderiza embebida en un elemento canvas del HTML. Permite cargar modelos en 2 y 3 dimensiones, crear puntos de luz, cámaras, físicas...



He escogido a Babylon.js por encima de **otras opciones** como Three.js o Pixi.js porque cuenta con una documentación mucho más extensa con ejemplos ejecutables y soporte de Microsoft.

SERVICIOS DE TERCEROS

El front-end se nutrirá de una serie de **APIs** de terceros para extraer distintos datos. Para la creación del mapeado del simulador obtendré una serie de imágenes extraídas de **MapQuest**. MapQuest es un servicio de cartografía web del que se pueden extraer imágenes satelitales.



Para obtener datos de altura de distintos puntos del mapeado realizaré consultas a **OpenTopography**, una API que devuelve información geográfica como la altitud de unas coordenadas dadas.



Para los mapas que mostrarán tanto la posición en tiempo real del globo como los puntos de despegue y las trayectorias, usaré **Leaflet**, una librería de JavaScript de código abierto provista de herramientas útiles como creación de marcadores, rutas, etc. que nos permitirán crear mapas interactivos.



Leaflet a su vez utiliza una API llamada **Open Street Map**. Es un proyecto colaborativo para crear **mapas editables y libres**. Tiene una base de datos muy amplia de carreteras, información geográfica, relieve... Proporciona imágenes de mapas.



Windy es un servicio web que proporciona datos meteorológicos. La web cuenta con una API que devuelve **datos del viento** a distintas alturas, esto me permitirá realizar los cálculos de movimiento del globo aerostático. La versión gratuita de la API devuelve los datos mezclados y ligeramente modificados y sólo permite hacer 500 llamadas diarias.



OTRAS TECNOLOGÍAS

Otras tecnologías con las que trabajaré durante el desarrollo de la aplicación:

Para poder realizar el control de versiones de la web usaré **GitHub**, una plataforma para alojar proyectos que usa el sistema de **control de versiones Git**. El uso de GitHub me permite almacenar el código online y la posibilidad de realizar backups a estados anteriores de la aplicación si se produce algún inconveniente. Para facilitar el uso de Git trabajaré con la aplicación **GitHub-Desktop**.



Para realizar **pruebas** de los servicios de terceros, así como de la API que voy a desarrollar, me serviré de **Postman**. Una aplicación que permite realizar de forma cómoda **peticiones HTTP** a través de una interfaz de usuario amigable.



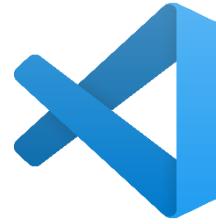
Gimp 2 es un **editor de recursos gráficos** similar a Photoshop, gratuito y de código abierto. Será necesario para modificar los recursos visuales que contendrá la web y los diversos assets. Aunque Gimp 2 contiene muchas funcionalidades también usaré la prueba gratuita de **Photoshop** para una tarea concreta que no podía ser realizada con este editor.



Para poder incluir gráficos de los datos de vuelos, utilizaré **Chart.js**. Es una librería de JavaScript de código abierto que, utilizando un elemento canvas HTML, permite la creación y visualización de **gráficos de datos**. Estos datos pueden ser representados mediante barras, gráficos circulares o líneas de puntos.



El entorno de desarrollo que he elegido es **Visual Studio Code**, un **editor de código** desarrollado por Microsoft. Es uno de los más utilizados para el desarrollo web debido a sus **extensiones** y posibilidades en cuanto a personalización. Me he decantado por este editor ya que es mi favorito por su comodidad y diseño.



Fontawesome es un conjunto de herramientas de fuentes e iconos basado en CSS y Less. Permite incluir en el HTML de la web una gran variedad de **iconos** que me permitirán mejorar el estilo y diseño de la misma. En su página web se encuentra un listado de sus más de 16.000 iconos en su versión premium.



Para la **comunicación de datos** entre API y Front-end utilizaré **JSON**. Lenguaje de marcado alternativo a XML que ha ganado mucha ventaja sobre éste debido a su sintaxis más simple y su menor peso. Integrado a la perfección con los lenguajes PHP y JavaScript, es mejor opción que XML.



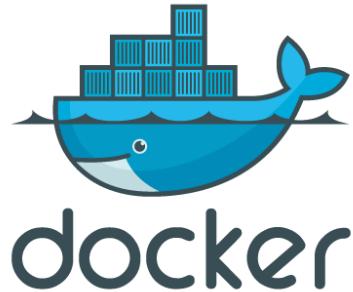
DESPLIEGUE

Para **desplegar** la aplicación necesitaré:

- **XAMPP**, una distribución de **Apache** fácil de instalar que contiene **MariaDB**, **PHP** y **phpMyAdmin**.
- **Node.js**, un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa servidor basado en JavaScript. Se requiere su instalación para utilizar Angular.
- **Administradores de paquetes**: Angular requiere de **npm**, el administrador de paquetes de Node. Laravel a su vez precisa de **Composer**, un administrador de paquetes para PHP que proporciona un estándar para administrar, descargar e instalar dependencias y librerías.

DOCKER

Para desplegar la aplicación independientemente del sistema operativo en el que se ejecute utilizaré **Docker**. Docker empaqueta software en unidades estandarizadas llamadas **contenedores** que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema, código...



Docker utiliza un **Sistema Operativo interno** basado en Linux. Esto permite a cada contenedor ejecutarse en una máquina virtual independiente, con todos los recursos del sistema a su disposición. Con la aplicación **Docker-Desktop** se pueden gestionar de forma cómoda los contenedores que hay en ejecución.

Para desplegar una aplicación con Docker se necesita un archivo de texto plano llamado **Dokerfile**. En este archivo se definen los comandos que se deben ejecutar en el S.O. interno del contenedor. En el ejemplo, el Dokerfile del front-end, que parte de una imagen de Node.js y ejecuta un “*npm install*” después de copiar el archivo “*package.json*”.

```
FROM node:14
RUN apt-get update && apt-get install
RUN npm install
WORKDIR /var/www
COPY ["package.json", "package-lock.json", "./"]
RUN npm install
```

```
services:
  app:
    container_name: laravel
    build:
      context: .
      dockerfile: .docker/Dockerfile
    image: "laravelapp"
    ports:
      - "8000:80"
    volumes:
      - ./:/var/www/html

  mariadb:
    image: mariadb
    container_name: mariadb
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: "root"
      MARIADB_DATABASE: "balloonsim"
      MARIADB_USER: "root"
      MARIADB_PASSWORD: "root"
    volumes:
      - ./data:/var/lib/mysql
    ports:
      - "3306:3306"
```

Docker Compose es una herramienta que permite simplificar el uso de Docker. A partir de un archivo con extensión **YAML** llamado `docker-compose`.

Como se muestra en la imagen este archivo permite **levantar de forma automática** contenedores de Docker, en este caso dos servicios: Laravel y MariaDB. Docker Compose permite también asignar los puertos externos que reflejan los puertos internos del contenedor y que permiten acceder a la aplicación desde la máquina cliente. Mediante el uso del comando “*docker-compose up*” la aplicación se desplegará con todo lo necesario para su funcionamiento.

FASES DEL PROYECTO

Sprint 1 (24-12-2021 >> 14-01-2022)

- Investigación y documentación sobre Babylon.js.
- Mapeado del simulador.
 - Acotar el mapa.
 - Obtener imágenes satelitales.
 - Obtener imágenes topográficas.
 - Implementar la altitud para un mapa 3D.
- Modelo del globo.
 - Cargar un modelo en 3D para el globo.
- Físicas del globo.
 - Calcular coordenadas en tiempo real.
 - Calcular velocidades de ascenso, descenso y traslación.
 - Establecer velocidad y dirección a partir de un viento.
 - Establecer un cálculo de temperaturas.
 - Calcular altura del aerostato en tiempo real.
- Interfaz de usuario.
 - Diseño de una interfaz que incluya todos los elementos básicos.
 - Desarrollo de animaciones CSS para los controles.

Las horas estimadas para la realización de estas tareas fueron 40, sin embargo, las dificultades surgidas durante el desarrollo y el proceso de investigación sobre Babylon.js provocaron que me llevara unas 60 horas.

Sprint 2 (15-01-2022 >> 21-02-2022)

- Base de datos.
 - Diseño de las tablas.
 - Creación de las tablas.
- Desarrollo del Back-end.
 - Investigar sobre Laravel
 - Crear un CRUD en la parte web.
 - Crear una API.
 - Implementar un sistema de login.
 - Diseñar el estilo del back-end con Bootstrap.
- Insertar datos de los puntos de despegue.

Para este sprint estimé una duración de 25 horas, pero las facilidades que aporta Laravel a la hora de programar hicieron que sólo tardara unas 15.

Sprint 3 (22-02-2022 >> 12-03-2022)

- Preparación del front-end.
 - Componentes de Angular para las páginas.
 - Integrar el simulador.
 - Pantalla de carga para el simulador.
 - Pantalla principal (home).
- Despliegue con Docker.
 - Desplegar front-end.
 - Desplegar back-end.
 - Desplegar base de datos y phpMyAdmin.
- Implementar una clase para realizar peticiones a la API.
 - Crear los modelos de datos en el front-end.

Para este sprint estimé una duración de 15 horas y más o menos éas fueron las realmente invertidas.

Sprint 4 (13-03-2022 >> 31-03-2022)

- Hoja de ajustes en el front-end.
 - Estructurar la página ajustes.
 - Implementar mapas interactivos.
 - Mostrar los puntos de despegue en un mapa con marcadores.
 - Realizar peticiones a Windy para obtener datos de los vientos.
 - Mostrar los vientos de forma gráfica sobre un mapa.
- Implementar métodos para exportar / importar los ajustes.
 - Crear un JSON con los ajustes.
 - Descargar el JSON al exportar.
 - Implementar una función para importar el JSON y editar los ajustes automáticamente.

Para el cuarto sprint estimé una duración de unas 40 horas e invertí aproximadamente unas 35.

Sprint 5 (01-04-2022 >> 15-04-2022)

- Crear el mapa de la tablet del simulador.
 - Cargar el mapa interactivo con Leaflet.
 - Mostrar la ruta seguida por el aerostato en tiempo real.
- Desarrollar las pestañas del historial de vuelos.
 - Pestaña con el listado de vuelos pasados.
 - Pestaña con los datos específicos de un vuelo.

- Pintar gráficos con los datos del vuelo (altitud, combustible restante...) utilizando para ello Charts.js.
- Mostrar en una tabla los cálculos de altitud máxima, media, velocidades máximas, medias, distancia recorrida...

Para el quinto sprint estimé una duración de unas 35 horas e invertí aproximadamente unas 40.

Sprint 6 (16-04-2022 >> 25-04-2022)

- Crear el componente flightreview.
 - Cargar el simulador para ver la repetición de un vuelo.
 - Permitir al usuario seleccionar el segundo que ver.
 - Permitir poner en play / pause la simulación.
 - Permitir cambiar la velocidad de reproducción.
- Realizar vuelos y pruebas de la aplicación.
 - Corregir posibles bugs.
 - Pedir a alguien que utilice la aplicación para analizar cómo se desenvuelve un usuario novato.
- Comentar todo el código de la aplicación.
 - Comentarios front-end y simulador.
 - Comentarios back-end.

Para el sexto y último sprint estimé una duración de unas 10 horas, pero invertí aproximadamente unas 20 porque surgieron bastantes bugs, aunque pequeños, la mayoría de ellos visuales y de CSS.

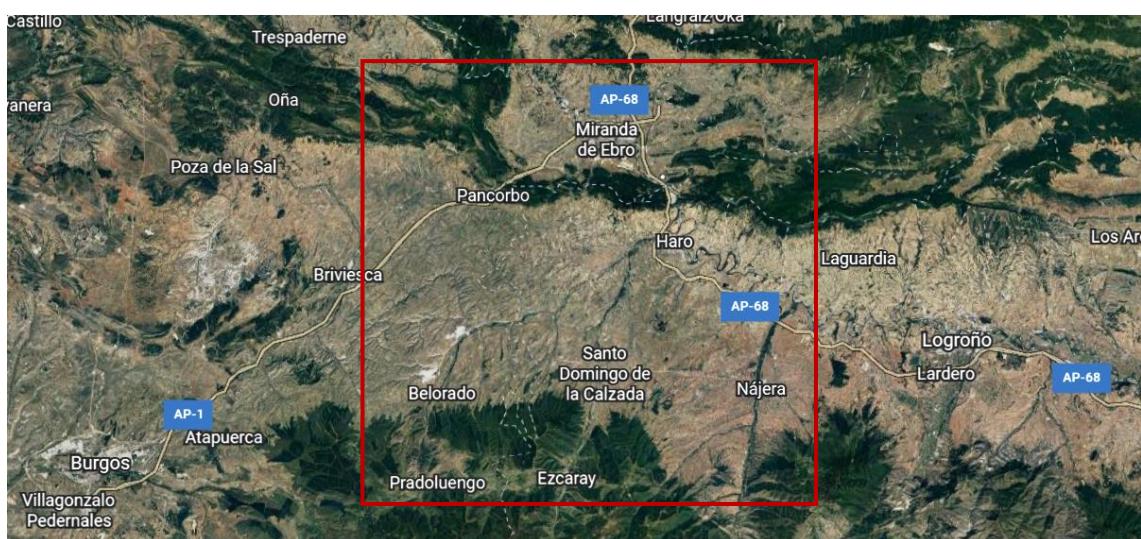
SPRINT 1

El simulador es la parte más crítica del proyecto, la que entraña más dificultad y presenta una mayor cantidad de retos, por lo que decidí empezar a desarrollar este apartado en primer lugar. Tiene **tres partes** diferenciadas: el **mapeado**, las **físicas del globo** aerostático y la **interfaz de usuario**.

MAPEADO DEL SIMULADOR

DISEÑO

El **mapeado** es la parte más problemática de toda la aplicación. En primer lugar, acoté junto a uno de los pilotos de la empresa, la **zona de vuelo habitual** para representarla en la aplicación. El mapeado debía limitar al norte con Miranda de Ebro, al sur con Ezcaray, al este con Nájera y al oeste con Briviesca. La **zona representada en rojo** ocupa un **área total de 2704 Km²**, un cuadrado con un lado de 52Km. Será la zona que deberá ser cargada en el simulador.



La idea del simulador era crear un **mapa en 3D** que lo dotara de realismo, por lo que era necesario encontrar una forma de hacerlo con Babylon.js. También había que incluir un cielo para una mejor sensación en el aire.

IMPLEMENTACIÓN

Para representar el mapa dentro del Babylon.js decidí cortar el mapa en forma de cuadrícula. Mediante un **script PHP** realicé peticiones a **MapQuest** para extraer **255 imágenes** correspondientes a una **cuadrícula de 15 x 15**. Cada imagen es de 1920 x 1920 píxeles. Por lo que en su conjunto ocupan 740MB, que deben ser cargados en su totalidad durante la simulación. Esto conlleva un **tiempo de carga aproximado de unos 30 segundos** y un **consumo de RAM de unos 3GB**. Una opción que barajé fue cargar dinámicamente los trozos del mapa; sin embargo, la distancia de visión en el aire era escasa y cada imagen tardaba unos 20 segundos en ser cargada, por lo que descarté esta posibilidad.

Este es el **resultado de las peticiones**, 255 imágenes, en una cuadrícula de 15 x 15 alojado en los assets del simulador. El tiempo de ejecución del script de descarga fue superior a la media hora ya que la API de MapQuest es bastante lenta y las imágenes obtenidas eran de gran tamaño (unos 3MB cada una).



```
function createScene() {
    canvas = document.createElement("canvas");
    game.innerHTML = "";
    engine = new BABYLON.Engine(canvas, true);
    scene = new BABYLON.Scene(engine);

    setLight();
    setCamera();
    setPointer();
    setSkybox();
    setGround();
    setBalloon();
}
```

Con las imágenes ya obtenidas, pude empezar con la creación de la **escena** en Babylon.js. Una escena contiene un **motor gráfico**, **varios objetos para renderizar**, una **cámara** y una o varias **luces**. Con la escena creada ya pude cargar el mapeado.

Para cargar la malla de 15 x 15 imágenes realicé un script (debajo) que calculaba la posición en la que debía encontrarse cada una y cargaba en el objeto “ground” la textura adecuada.

```
/**  
 * Creates the ground map  
 */  
function setGround() {  
    let k = 1;  
  
    for (let i = 0; i < 15; i++) {  
        for (let j = 0; j < 15; j++) {  
            const positionZ = mapSize * -i - mapSize / 2 + mapTotalSize;  
            const positionX = mapSize * j + mapSize / 2;  
  
            const materialGMap = new BABYLON.StandardMaterial("material", scene);  
            materialGMap.diffuseTexture = new BABYLON.Texture(`../assets/maps/sat/m_${i}_${j}.jpg`, scene);  
            materialGMap.diffuseTexture.wrapU = BABYLON.Texture.CLAMP_ADDRESSMODE;  
            materialGMap.diffuseTexture.wrapV = BABYLON.Texture.CLAMP_ADDRESSMODE;  
            materialGMap.specularColor = new BABYLON.Color4(0, 0, 0, 0);  
            materialGMap.alpha = 1.0;  
  
            const ground = BABYLON.MeshBuilder.CreateGround(`ground_${k}`, {width: mapSize, height: mapSize});  
            ground.material = materialGMap;  
            ground.position.x = positionX;  
            ground.position.z = positionZ;  
  
            k++;  
        }  
    }  
}
```

] Posicionamiento del trozo de mapa.

Con las imágenes cargadas el mapa se mostraba correctamente, pero, al estar completamente plano tenía un aspecto muy pobre. Para dar una sensación derealismo extra era necesario dotar al mapa de **elevación**. Sin duda, esto llevó un **gran tiempo de investigación** para entender el funcionamiento de Babylon.js en este aspecto. El framework permite aplicar una **imagen topográfica en blanco y negro** a los planos creados para la cuadrícula. Como vemos en el mapa de abajo, las zonas más blancas representan una altitud mayor (como la cordillera del Himalaya o los Andes) y las más oscuras, como los océanos, una menor. Cuanto más negro, más cercano a la altitud 0.



El **principal problema** con el que me topé para implementar la elevación del terreno fue la **inexistencia de una API** de la que pudiera extraer el mapeado topográfico en blanco y negro. Por lo que tuve que descargar una **imagen completa en 4K de toda España y recortarla con Photoshop** de la forma más precisa posible para que se ajustara al mapeado representado. Tras dividir la imagen resultante en 255 trozos iguales el mapeado del simulador estaba completo.

Algunos **ejemplos**: a la izquierda, el mapa de elevación en blanco y negro y a la derecha la vista satélite. Se aprecia la depresión en la que está situado Ezcaray, entre los montes y la curiosa forma del Barranco de Peña Redonda junto al municipio de Ocio.



Ahora podía cambiar el **código** de generación del mapa para poder ponerlo en **tres dimensiones**. Para ello en la carga de cada imagen se debía especificar cuál era la imagen topográfica que correspondía a dicha sección, así como **tres nuevas variables** muy importantes:

- “subdivisions”: especifica en cuantas subdivisiones debe calcular la altura, es decir, el número de polígonos del objeto. **Cuanto más alta**, daba una sensación de **mejor calidad**, pero **aumentaba** drásticamente el **tiempo de carga** y el **consumo de RAM**.
- “maxHeight”: **Altura máxima** que puede alcanzar el mapeado, es decir, la altura asignada a los píxeles más blancos de la imagen.
- “minHeight”: **Altura mínima** que puede alcanzar el mapeado, la asignada a los píxeles más oscuros de la imagen.

Cuanto mayor sea la diferencia entre “maxHeight” y “minHeight” más altas serán las montañas. Requirió de una **gran cantidad de pruebas** cambiando estos valores hasta producir un resultado con el que me sintiera satisfecho sin implicar unos tiempos de carga o consumos de RAM excesivos. Finalmente obtuve un gran resultado que requería de unos **40 segundos de carga y 4GB de RAM**.

En estas dos imágenes se observa el código de la nueva carga de los trozos con la elevación. El único problema que generaría el mapeado en tres dimensiones es que **no** me permitía **detectar** las **colisiones** del globo aerostático **con el suelo**. Sin embargo, quedaba tan bien que el suelo fuera en tres dimensiones que no me importó **perder la posibilidad de detectar aterrizajes**.

```
/** * Creates the ground map */
function setGround() {
    let k = 1;

    for (let i = 0; i < 15; i++) {
        for (let j = 0; j < 15; j++) {
            buildGround(i, j, k);
            k++;
        }
    }
}
```

```
/** * Builds the ground map at the given position *
* @param {number} i x position
* @param {number} j y position
* @param {number} k order of the ground */
async function buildGround(i, j, k) {
    const positionZ = mapSize * -i - mapSize / 2 + mapTotalSize;
    const positionX = mapSize * j + mapSize / 2;

    //Material
    const materialGMap = new BABYLON.StandardMaterial("material", scene);
    materialGMap.diffuseTexture = new BABYLON.Texture(`../assets/maps/sat/m_${i}_${j}.jpg`, scene);
    materialGMap.diffuseTexture.wrapU = BABYLON.Texture.CLAMP_ADDRESSMODE;
    materialGMap.diffuseTexture.wrapV = BABYLON.Texture.CLAMP_ADDRESSMODE;
    materialGMap.specularColor = new BABYLON.Color4(0, 0, 0, 0);
    materialGMap.alpha = 1.0;

    //Elevation
    const ground = BABYLON.MeshBuilder.CreateGroundFromHeightMap(
        `ground_${k}`,
        `../assets/maps/topo/topo_${k}.gif`, {
            width: mapSize,
            height: mapSize,
            subdivisions: 100,
            maxHeight: 420,
            minHeight: 0,
        }
    );
    ground.material = materialGMap;
    ground.position.x = positionX;
    ground.position.z = positionZ;

    //Shows the collision box of the ground
    if (showCollisions) ground.showBoundingBox = true;
}
```

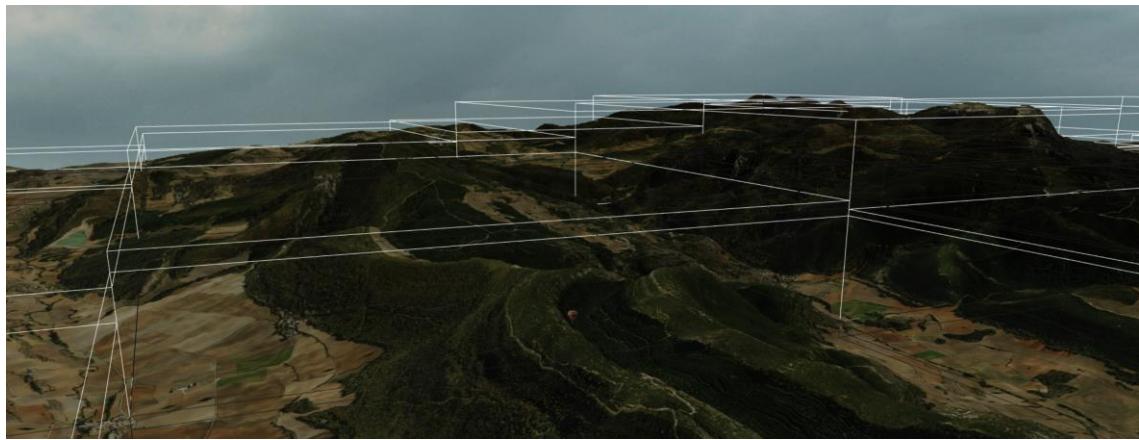
Carga de la imagen satelital

Carga de la imagen topográfica

Variables

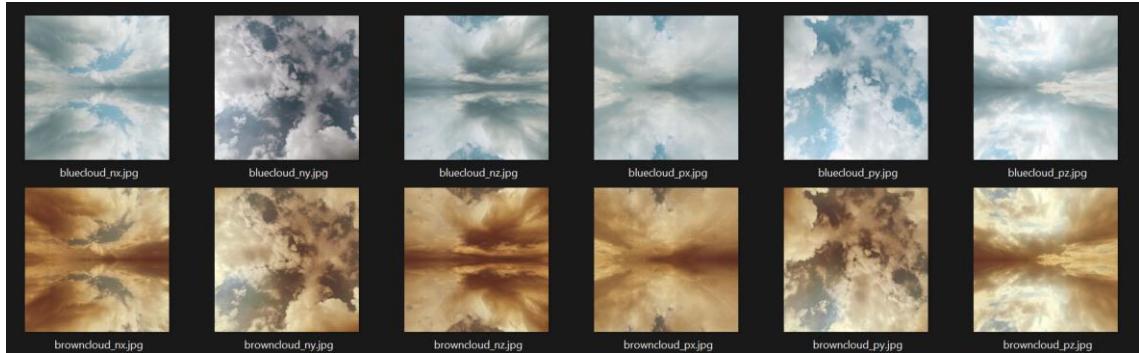
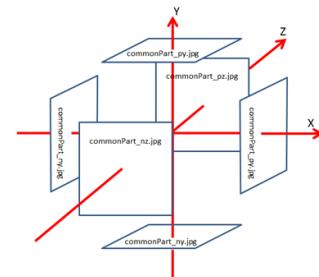
Para mostrar las colisiones

Como se muestra en la imagen de abajo, las **cajas de colisión** (en blanco) que permiten detectar colisiones **no se adaptan a la geografía del terreno**, sino que se colocan a la altura del punto más alto. Esto impide determinar en qué momento el globo aerostático desciende por debajo del nivel del suelo.



Un buen simulador de vuelo debe tener no sólo un mapeado sino un **cielo** visible. Para esto usé un **skybox**, clase de Babylon.js que permite colocar un cubo con sus caras orientadas hacia el interior manteniendo la cámara dentro. Esto permite crear el efecto de un cielo dentro del simulador.

En la imagen (abajo) se muestran las texturas de este skybox, 6 caras de un cubo. Para dar un efecto extra de realismo, hay **cuatro tonalidades diferentes** para el cielo en colores distintos para que el usuario decida qué aspecto debe tener el cielo que se va a mostrar.



```
/**
 * Sets the skybox
 */
function setSkybox() {
    skybox = BABYLON.Mesh.CreateBox("skyBox", skyboxSize, scene);
    const skyboxMaterial = new BABYLON.StandardMaterial("skyBox", scene);
    skyboxMaterial.backFaceCulling = false;
    skyboxMaterial.reflectionTexture = new BABYLON.CubeTexture(`../assets/textures/skybox/${selectedSkybox}`, scene);
    skyboxMaterial.reflectionTexture.coordinatesMode = BABYLON.Texture.SKYBOX_MODE;
    skyboxMaterial.disableLighting = true;
    skybox.material = skyboxMaterial;
}
```

MODELO DEL GLOBO

Con el mapa terminado solamente quedaba seleccionar un **modelo en tres dimensiones** para el **globo aerostático** e incluirlo. Usando la biblioteca de imágenes 3d que ofrece Microsoft en la aplicación **Visor 3D**, escogí el modelo del aerostato (abajo) y lo agregué en la simulación.



Para que la **cámara** de la escena siguiera de forma constante al modelo del globo aerostático, a la hora de cargar el objeto se lo aplicamos a la cámara como **target**. Esto hace que la visión del usuario se **centre automáticamente** sobre el modelo.

```
/**  
 * Sets the balloon  
 */  
function setBalloon() {  
    BABYLON.SceneLoader.ImportMesh(null, "../assets/models/", "balloon.glb", scene, () => {  
        balloonModel = scene.getMeshByName("hot_air_balloon.1");  
  
        //Scaling  
        balloonModel.scaling = new BABYLON.Vector3(balloonScaling, balloonScaling, balloonScaling);  
  
        //Camera target  
        camera.setTarget(balloonModel); } } Target de la cámara.  
        return true;  
});  
}
```

RESULTADO FINAL DE LA ESCENA

Resultado final del mapeado y la escena de Babylon.js:

Haro y los meandros del río Ebro:



Ezcaray:



Miranda de Ebro:



Los montes Obarenes:



FÍSICAS

Lo siguiente son las **físicas**. Un globo aerostático vuela gracias a la **diferencia de presión entre el interior y el exterior** de la vela. Esto se consigue calentando el aire que se encuentra dentro de la lona, a una temperatura de unos 60-80 grados superior a la temperatura ambiente. Cuanta más temperatura tiene el aerostato más rápido se eleva. Un globo de aire caliente **no lleva timón** ni forma de dirigirse, los pilotos usan la altura para buscar capas de viento que lo empujen hacia la dirección adecuada.

Para mover el globo por el mapa es necesario realizar un **cálculo** entre las coordenadas que usa Babylon.js para sus escenas y la latitud-longitud que representan. Estos cálculos, como se muestran abajo, son **reglas de tres** y son necesarios para calcular la posición, velocidad, altura...



Los 52 km que mide cada lado del mapeado **representan en grados 0,617** ya que **1 grado** de latitud equivale aproximadamente a **111,12 km**. Cada imagen del mapa ocupa 500 coordenadas de Babylon.js por lo que las 15 que hay en cada fila y columna ocupan 7500. Cada **1 de coordenada de Babylon.js representa** entonces **6,933 metros y 0,00008227 grados**. Con estas medidas puedo conocer el punto exacto en el que se encuentra el aerostato tanto en metros como en coordenadas de Babylon.js y coordenadas reales.

Para calcular si el globo asciende o desciende he implementado **un sistema de temperaturas** que permite ascender o descender dependiendo de la diferencia con el exterior, modificando velocidades de ascenso o descenso. La temperatura inicial se establece en tierra y **cada 154 metros ascendidos baja 1 grado centígrado**.

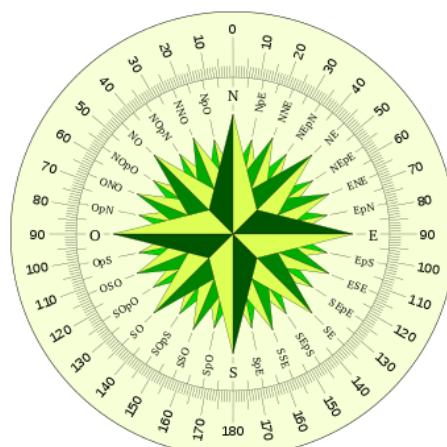
Ejemplo de cálculos en código:

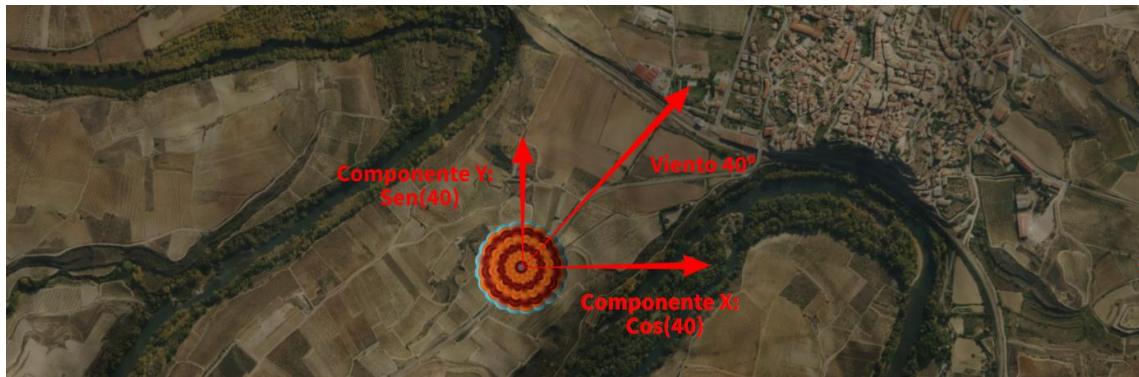
```
/**  
 * Method to calculate latitude degrees  
 *  
 * @return {number} latitude degrees of the balloon  
 * @memberof Balloon  
 */  
calcDegreesLat() {  
    const latDeg = upDeg + ((this.pointer.position.z - mapTotalSize) / mapTotalSize) * diffDegY;  
    return latDeg.toFixed(5);  
}  
  
/**  
 * Method to calculate longitude degrees  
 *  
 * @return {number} longitude degrees of the balloon  
 * @memberof Balloon  
 */  
calcDegreesLon() {  
    const lonDeg = leftDeg - (this.pointer.position.x / mapTotalSize) * diffDegX;  
    return lonDeg.toFixed(5);  
}  
  
/**  
 * Sets the meters from position transforming x, y and z to real meters  
 * @memberof Balloon  
 */  
setMetersFromPosition() {  
    this.mX = (this.pointer.position.x / mapTotalSize) * mapSizeMeters;  
    this.mZ = (this.pointer.position.z / mapTotalSize) * mapSizeMeters;  
    this.altitude = pointer.position.y * convAltitude + altitudeAddition;  
    this.moveBalloonToPointer();  
}
```

FÍSICAS DEL VIENTO

La dirección del viento se mide en **grados**, estos grados, como se muestra en la imagen, están divididos en **360**. Estas direcciones deben ser **divididas en sus componentes** este/oeste y norte/sur de forma que mediante Babylon.js podamos mover el globo en sus coordenadas X y Z (la Y es la altura).

Para separar el viento en sus diferentes componentes empleé los siguientes **cálculos**, usando el **seno** y **coseno** del ángulo. Supongamos un viento de dirección nordeste.





Mediante el siguiente **código** (abajo) se establecen las velocidades en los ejes de coordenadas X y Z a partir de un **ángulo** y una **velocidad**.

```
/**
 * Method to set the speeds in m/s to move the balloon
 * Depending on the degrees given.
 * X = Sin(angle * PI / 180)
 * Y = Cos(angle * PI / 180)
 *
 * @param {number} deg wind direction in degrees
 * @param {number} speed wind speed in m/s
 * @memberof Balloon
 */
setSpeeds(deg, speed) {
    this.actSpeedX = Math.sin((deg * Math.PI) / 180) * speed;
    this.actSpeedZ = Math.cos((deg * Math.PI) / 180) * speed;
}
```

Para poder realizar **pruebas** en el simulador sin que las físicas afectasen, utilicé una variable de tipo booleano que desactiva las físicas, permitiéndome mover el aerostato con las flechas del teclado. Esto permite realizar capturas de pantalla, moverse hacia posiciones concretas para guardar coordenadas, etcétera.

Las físicas iniciales, a medida que se desarrollaba el proyecto y a lo largo de todos los sprints, recibieron **numerosos cambios** debido a **bugs**, pequeños **ajustes de cálculo** de algunas variables... de forma que fueron siendo cada vez cálculos más precisos.

INTERFAZ DE USUARIO

Con las físicas completas sólo queda la **interfaz de usuario**. Babylon.js dispone de métodos y clases para incluir una interfaz dentro del propio motor; sin embargo, quiero incluir el uso de mapas dinámicos que deben cargarse dentro del **DOM**, por lo que me he decantado por crear una interfaz de usuario desarrollada en **HTML y CSS** para poder embeber código de terceros, canvas, mapas...

DISEÑO

La **interfaz** se compone de varios elementos que **simulan** los **aparatos reales** de navegación que usan los pilotos.

El **quemador** utiliza gas propano para producir una intensa llama que calienta el interior de la vela. Para accionarlo el usuario empuja unos gatillos hacia abajo.



El **altímetro** es un aparato que muestra las velocidades de ascenso y descenso, así como la altura en pies y m s.n.m (metros sobre el nivel del mar). También se muestran otros datos como la presión, la temperatura, la dirección y velocidad del viento. Tiene un medidor en su parte izquierda que muestra de manera más visual la velocidad de ascenso o descenso que deberá **animarse** por CSS.

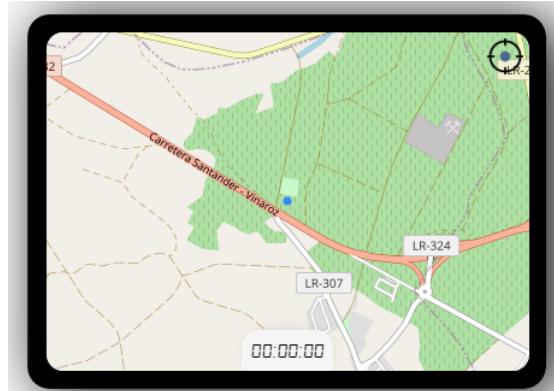


Las **medidas** que se muestran en el altímetro se corresponden con la posición del globo en el simulador y serán todas calculadas en **tiempo real**. El **diseño** del aparato está basado en altímetros reales utilizados por la empresa.

Una **cuerda** que, anclada al paracaídas superior del globo aerostático, permite abrirlo ligeramente escapándose así el aire cálido y permitiendo el descenso. Al ser accionada por el usuario deberá **animarse** mediante CSS, desplazando la cuerda hacia abajo simulando que el se está tirando de ella.



Una **tablet** que es usada por los pilotos para ver la posición en la que se encuentra el globo en cada momento. Deberá contener un mapa interactivo para mostrar la ruta y posición. También incorporará un botón que centrará la ubicación sobre la posición del aerostato y un cronómetro con la duración del vuelo.



Un **indicador de combustible** que muestre el combustible restante y tenga una flecha que se mueva indicándolo visualmente.



Una **barra de menú** con iconos que permitan **salir** del simulador y **ocultar** la interfaz de forma que podamos ver todo el mapeado sin que la interfaz nos lo dificulte.

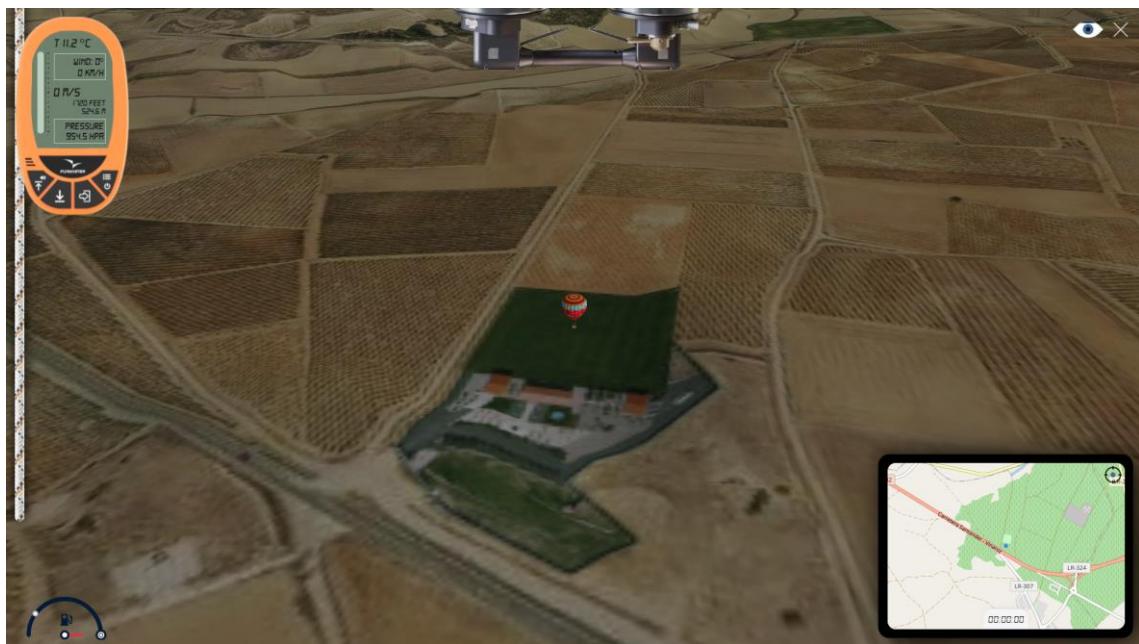
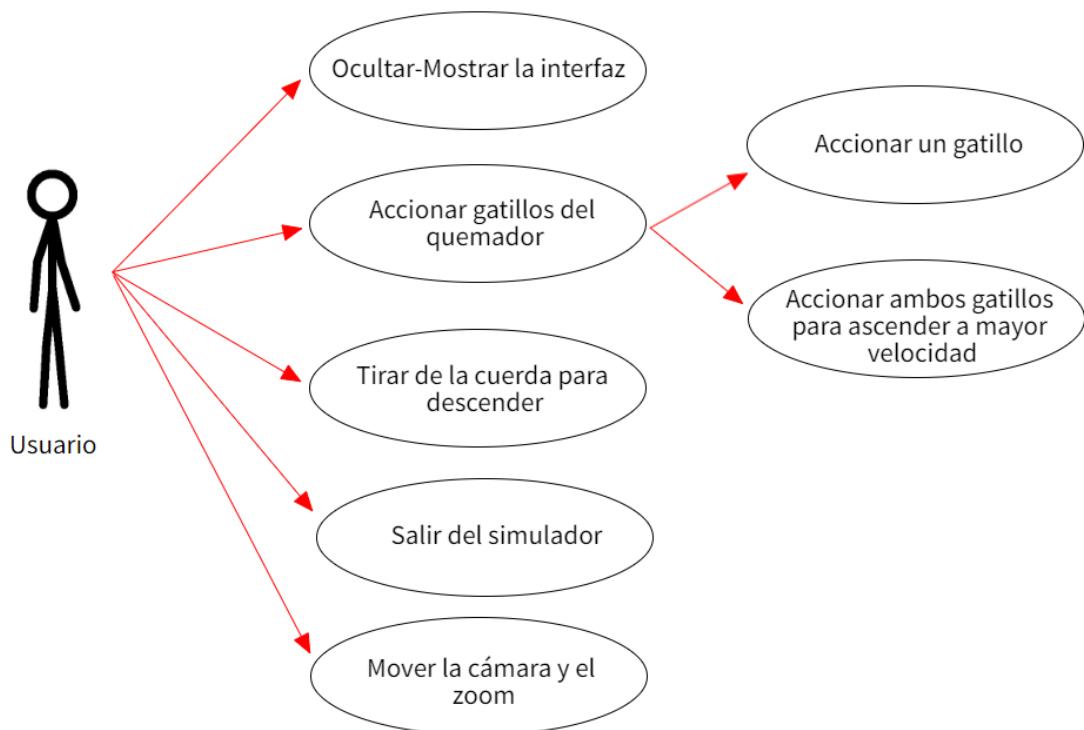


Diagrama de **casos de uso** de la interfaz del simulador. El usuario debe poder interactuar con la cámara de la escena de Babylon.js de forma que pueda cambiar la vista y el zoom de la misma. Podrá accionar uno o ambos gatillos del quemador para hacer ascender a la aeronave y tirar de la cuerda para hacerla descender. Mediante la barra de opciones el usuario del simulador podrá ocultar/mostrar la interfaz de usuario y salir del mismo al finalizar el vuelo.



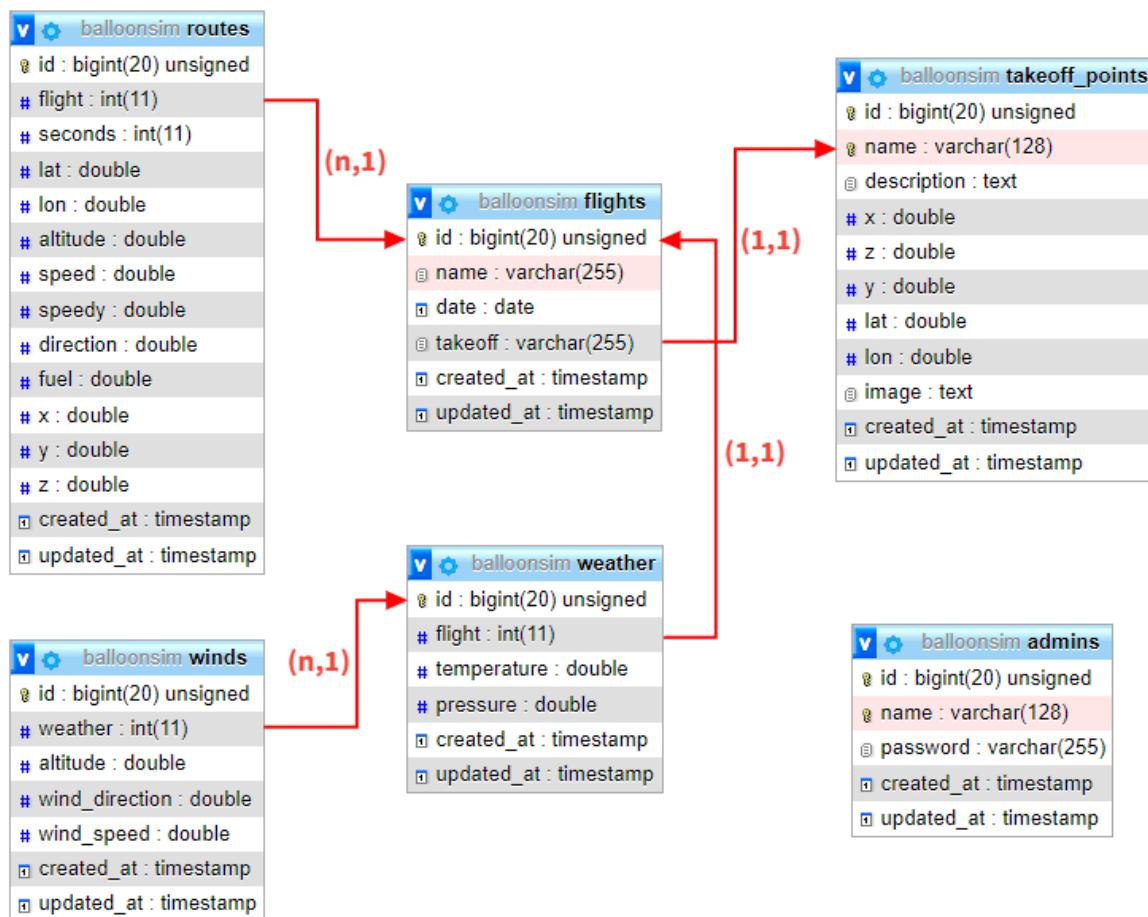
SPRINT 2

Para el segundo de los sprints me propuse realizar toda la parte **back-end** de la aplicación y la **base de datos**.

BASE DE DATOS

La base de datos está realizada con **MariaDB** por lo que sigue un esquema relacional. Las tablas contienen todos los datos necesarios para el funcionamiento de la aplicación, debe cumplir con los siguientes **requisitos**: almacenar los datos de login del administrador del back-end, almacenar los puntos de despegue habituales de la empresa Globos Arcoiris y almacenar los vuelos realizados con todos sus datos. Para ello **diseñé** las siguientes tablas:

Esquema de la base de datos:



Tablas de la base de datos:

Todas las tablas tienen un **Id** como clave primaria autogenerada y dos timestamp: **created_at** y **updated_at**, usados de forma automática por Laravel para guardar el momento en el que se creó el dato y la fecha de su última modificación. Estos datos son muy útiles a la hora de establecer migraciones de la base de datos.

- **Admins:**

Contiene los datos necesarios de **inicio de sesión** para los usuarios administradores del back-end.

- **Name:** nombre del usuario.
- **Password:** contraseña **encriptada** mediante **SHA1**.

- **Takeoff_points:**

Contiene los datos de los **puntos de despegue** que son utilizados habitualmente por la empresa.

- **Name:** nombre del punto de despegue.
- **Description:** breve descripción del lugar.
- **X:** coordenada x de Babylon.js.
- **Z:** coordenada z de Babylon.js.
- **Y:** coordenada y de Babylon.js, altura.
- **Lat:** latitud en la vida real.
- **Lon:** longitud en la vida real.
- **Image:** imagen del lugar para ser mostrada en el front-end.

- **Flights:**

Contiene los datos básicos de los **vuelos** realizados.

- **Name:** nombre del vuelo para poder buscarlo.
- **Date:** fecha de realización.
- **Takeoff:** punto de despegue.

- **Weather:**

Contiene los datos del **clima** del vuelo.

Cada vuelo tiene un clima asignado.

- **Flight:** vuelo al que pertenece.
- **Temperature:** temperatura en tierra (grados centígrados).
- **Pressure:** presión a nivel del mar (hectopascales).

- **Winds:**

Contiene los datos de todos los **vientos** de un clima.

Un clima tiene n vientos.

- **Weather:** clima al que pertenece.
- **Altitude:** altitud a la que se encuentra la capa de viento.
- **Wind_direction:** dirección de la capa (grados).
- **Wind_speed:** velocidad de la capa de viento (km/h).

- **Routes:**

Contiene todos los **datos de cada segundo del vuelo**, esto permitirá la realización de gráficos para mostrar todos estos datos de forma visual.

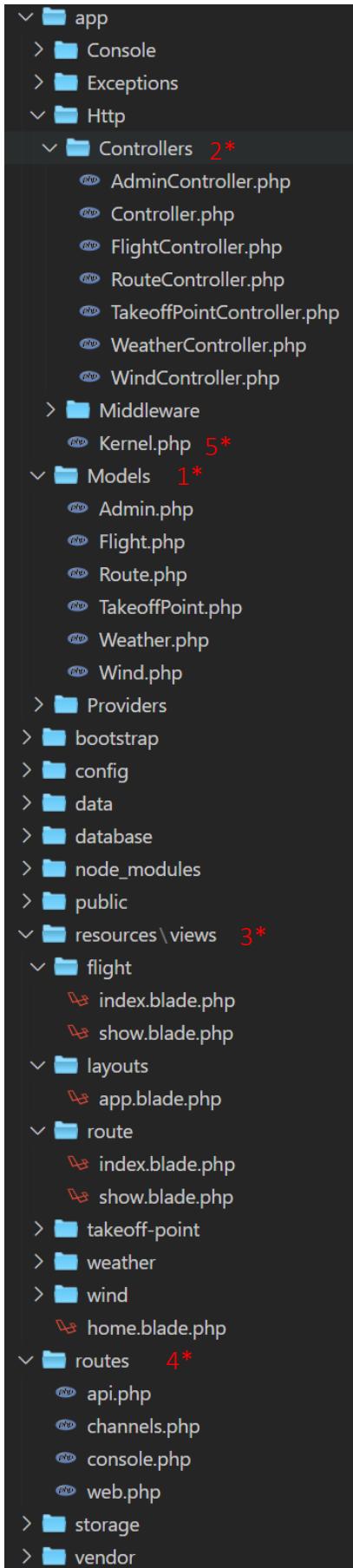
Un vuelo tiene n **puntos de ruta**.

- **Flight:** vuelo al que pertenece.
- **Seconds:** segundos transcurridos desde el despegue.
- **Lat:** latitud a la que se encuentra el globo.
- **Lon:** longitud a la que se encuentra el globo.
- **Altitude:** altitud en metros del aerostato.
- **Speed:** velocidad en km/h de la aeronave.
- **SpeedY:** velocidad de ascenso/descenso (m/s).
- **Direction:** dirección a la que se dirige (grados).
- **Fuel:** combustible restante (%).
- **X:** coordenada x en el simulador.
- **Y:** coordenada y del simulador.
- **Z:** coordenada z en el simulador.

BACK-END

El back-end está desarrollado utilizando **Laravel**, un framework que **nunca había usado** por lo que tuve que **investigar** cómo funcionaba. Laravel es un framework de código abierto para desarrollar aplicaciones y servicios web con PHP. Su filosofía es desarrollar código de forma elegante y simple. Está orientado a un modelo **MVC** (Modelo, Vista, Controlador). El MVC es un patrón de arquitectura de software, que separa los datos de su representación y el módulo encargado de gestionar los eventos y las comunicaciones: el controlador.

Para este proyecto quería un back-end sencillo que tuviera dos funcionalidades: un apartado **web** y una **API**. La parte web permitirá gestionar un **CRUD** (Create, Read, Update y Delete) de las tablas de la base de datos para que uno o varios usuarios administradores modifiquen los datos. La **API** debe permitir al front-end la **comunicación** con la base de datos a través del back-end mediante una serie de **endpoints**.



La **estructura de carpetas** de Laravel permite modularizar el código de forma muy clara y ordenada. En la carpeta **Models** (1) se almacena las clases que representarán nuestros objetos de la base de datos.

En la carpeta **Controllers** (2) se encuentra el código que contiene los métodos a realizar sobre el modelo, es decir, métodos para visualizar los datos, eliminarlos, listarlos, modificarlos, etcétera. Todos los controladores heredan de Controller, que es el controlador base. En los controladores se encuentra toda la lógica de la aplicación.

La carpeta **resources** (3) contiene todos los recursos visuales que tendrá la web: vistas y assets (imágenes, videos, sonidos, fuentes, etc.). Esta carpeta contiene a su vez la carpeta **views** donde se encuentran las vistas. Laravel incluye un sistema de procesamiento de plantillas llamado **Blade**. Este sistema favorece un código mucho más limpio en las vistas e incluye un sistema de Caché que lo hace mucho más rápido. Además, permite una sintaxis mucho más reducida en su escritura. Estos archivos tienen una extensión blade.php. Gracias a Blade las vistas pueden heredar aspectos entre ellas de forma cómoda y legible evitando duplicidad de código.

En la carpeta **routes** (4) se encuentra el código en el que especificaremos las rutas. El sistema de rutas de Laravel es bastante intuitivo y fácil de manejar, pero a la vez muy potente. Las rutas están divididas para las distintas funcionalidades, en mi caso, sólo me centraré en la parte web y en la API.

El archivo **Kernel.php** (5) es también muy importante para especificar el **número máximo de peticiones** por minuto que permite la API.

```

/**
 * Class Weather
 *
 * @property $id
 * @property $flight
 * @property $temperature
 * @property $pressure
 * @property $created_at
 * @property $updated_at
 */
class Weather extends Model
{
    static $rules = [
        'flight' => 'required',
        'temperature' => 'required|numeric',
        'pressure' => 'required|numeric',
    ];

    protected $perPage = 20;
}

/**
 * Attributes that should be mass-assignable.
 *
 * @var array
 */
protected $fillable = ['flight', 'temperature', 'pressure'];

```

Modelo: en este ejemplo se muestra el modelo de la clase Weather (la que contiene los datos del clima).

En el comentario de clase, Laravel nos permite especificar las **propiedades** que tiene la clase (referencian a las existentes en la base de datos). Laravel permite en sus modelos establecer una serie de **reglas de validación** que se comprobarán automáticamente a la hora de modificar o crear un nuevo objeto de la clase, mediante la variable **\$rules**. La variable **\$perPage** permite establecer una paginación automática.

En el ejemplo de **controlador** (abajo), se muestra el TakeoffPointController asociado con el modelo de los puntos de despegue. En la captura aparecen los métodos para listar y crear los puntos de despegue en la web que devuelven las vistas necesarias, comunicando el modelo con las vistas.

```

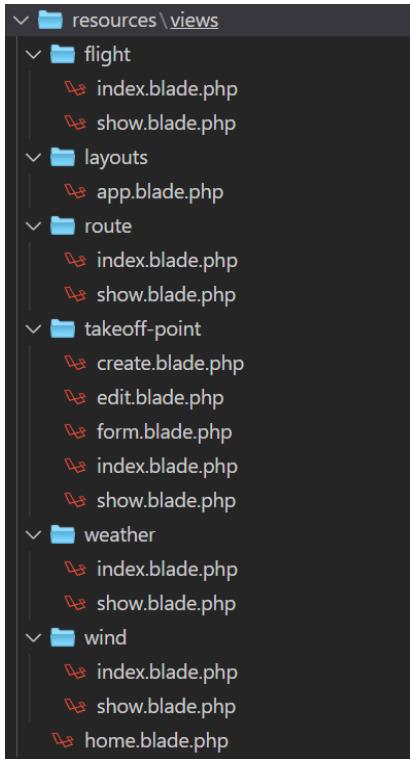
/**
 * Class TakeoffPointController
 * @package App\Http\Controllers
 */
class TakeoffPointController extends Controller
{

    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $takeoffPoints = TakeoffPoint::paginate();

        return view('takeoff-point.index', compact('takeoffPoints'))
            ->with('i', (request()->input('page', 1) - 1) * $takeoffPoints->perPage());
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        $takeoffPoint = new TakeoffPoint();
        return view('takeoff-point.create', compact('takeoffPoint'));
    }
}

```



Las **vistas** de la web heredan de un **Layout**, gracias a Blade. Por lo que será en el archivo **app.blade.php** donde se encuentre el menú, el head y body del HMTL y las distintas secciones. El resto de vistas contienen únicamente tablas y formularios. Para cada clase del modelo hay una carpeta con las vistas: **index** que contiene el listado, **show** que tiene un listado de las propiedades del modelo, **form** que es el **formulario** de inserción o modificación y **edit** y **create** que contienen este formulario. Dependiendo de si me interesa que el administrador pueda insertar o modificar datos habrá unas vistas y faltarán otras.

La vista **home** es la principal y contiene el **login** si la sesión no está iniciada y un **listado con todas las tablas** en caso contrario.

Las **rutas** se especifican de forma muy simple. Se basan en dos parámetros: la ruta que las ejecuta y el controlador encargado de responder o un script (función) que se lanzará al resolver la ruta. En el ejemplo (abajo) algunas **rutas** de la parte **web**.

```
/*
|-----[{"x": 550, "y": 650, "x2": 550, "y2": 750}]-|
| Web Routes
|-----[{"x": 550, "y": 750, "x2": 600, "y2": 750}]-|
*/
Route::get('/', function () {
    return view('home');
})->name('home');

Route::get('/login-failed', function () {
    return view('home', ['message' => 'Invalid username or password']);
});

Route::get('logout', 'App\Http\Controllers\AdminController@logout');
Route::get('login', 'App\Http\Controllers\AdminController@login');

Route::resource('takeoff-points', TakeoffPointController::class);
Route::resource('users', UserController::class);
Route::resource('winds', WindController::class);
Route::resource('weather', WeatherController::class);
Route::resource('flights', FlightController::class);
Route::resource('routes', RouteController::class);

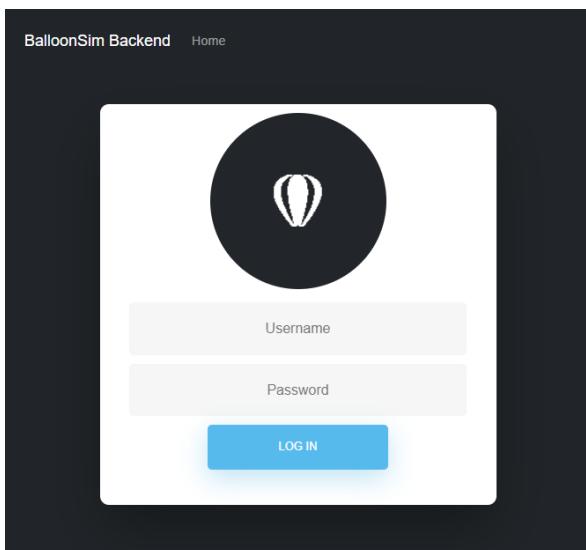
Route::get('/deleteAllFlights', function () {
    foreach (\App\Models\Weather::all() as $weather) $weather->delete();
    foreach (\App\Models\Wind::all() as $wind) $wind->delete();
    foreach (\App\Models\Route::all() as $route) $route->delete();
    foreach (\App\Models\Flight::all() as $flight) $flight->delete();
    return redirect()->route('flights.index')->with('success', 'All flights deleted successfully');
});
```

Rutas que llaman a controladores

Rutas que lanzan una función.

DISEÑO DE LAS VISTAS

Para las **vistas** de mi back-end tenía en mente basarme en phpMyAdmin, es decir, establecer un CRUD en el que el administrador vea mediante tablas HTML los datos que contienen las distintas tablas de la base de datos. Para simplificar el proceso de estilos, he aprovechado la integración de **Bootstrap** en Laravel, que permite establecer el CSS de la página de forma muy rápida, y **responsive**, aunque el resultado final no sea muy personalizado. Para modificar los datos o insertar nuevos crearé formularios al igual que para el login. El menú constará de una pantalla inicial en la que se mostrarán las distintas tablas.



La página tiene un **diseño** elegante y sobrio, como es habitual con Bootstrap. He elegido un color oscuro para los fondos y un blanco para los elementos que se muestran dando una buena sensación de contraste.

El **login** es una sencilla tarjeta que contiene el ícono de Globos Arcoiris y los inputs de nombre y contraseña. Tiene una **animación** por CSS cuando el componente se carga.

Una vez hemos iniciado sesión en la página se muestra la **pantalla principal** o home del back-end. Tiene un **menú** en la parte superior que permite volver al home cuando nos hallemos en otra pantalla, también permite **cerrar sesión** y contiene un enlace a phpMyAdmin.

En la captura (abajo) se muestra la **home** o página inicial, tiene un diseño sencillo maquetado mediante Bootstrap. Contiene un listado de todas las tablas de la base de datos. Cada campo incluye el **nombre** de la tabla, una breve **descripción** de la misma, las **filas** que contiene y el **espacio que ocupa** esa tabla en base de datos. Un botón a la derecha de cada campo permite navegar hasta la vista de cada clase del modelo.

Tiene además un botón (en verde) en la parte de abajo que **exporta la base de datos** y descarga un archivo .sql para poder realizar **backups** de los datos.

Tables of balloonsim					PhpMyAdmin
No	Name	Description	Rows	Size	
1	takeoff_points	List of takeoff places.	28	0.03 MB	Show
2	flights	List of flights.	4	0.02 MB	Show
3	routes	List of route points for each flight.	2382	0.33 MB	Show
4	weather	List of weathers for each flight.	4	0.02 MB	Show
5	winds	List of winds for each weather.	36	0.02 MB	Show
6	admins	List of admin users.	1	0.03 MB	

[Export database](#)

Cada una de las vistas de los modelos contienen los datos más importantes listados en la tabla y tres botones a la derecha que permiten **borrar**, **modificar** o **mostrar** los datos. En la parte superior derecha de la tarjeta está el botón para **agregar** un **nuevo** elemento. Gracias a Bootstrap el **diseño** se mantiene casi sin variaciones entre todas las vistas. Los datos que se muestran en las páginas son automáticamente **paginados** gracias a Laravel.

Takeoff Point								Create New
Name	Description	X	Z	Y	Lat	Lon	Image	
Instalaciones de Globos Arcoíris	Finca de despegue de Globos Arcoíris, Km 459, N-232, Cuzcurrita de Río Tirón.	3830	3945	42.82	42.55654	-2.97265	instalaciones.png	Show Edit Delete
Carretera Anguciana	Finca de despegue situada entre las localidades de Haro y Anguciana, carretera LR-202.	5015	4190	34.57	42.57236	-2.87522	haro3.png	Show Edit Delete
Bugedo	Finca de despegue en Bugedo, Burgos. Despegue ideal para cruzar los Obarenes.	3240	5470	46.13	42.64976	-3.02136	bugedo.png	Show Edit Delete
Miranda de Ebro	Punto de despegue en Miranda de Ebro, provincia de Burgos	4065	6140	39.94	42.6905	-2.95349	miranda.png	Show Edit Delete

Tanto la **vista de agregar** como la de **modificar** contienen el mismo **formulario** de forma que se evita la duplicidad de código. Los datos introducidos en el formulario son **automáticamente comprobados** tanto con JavaScript como con PHP gracias a las **reglas de validación** establecidas en el modelo de datos.

The screenshot shows a web page titled "Update Takeoff Point". The page has a dark header with "BalloonSim Backend" and navigation links "Home" and "PhpMyAdmin" on the left, and "Log out" on the right. The main content area contains fields for updating a takeoff point:

- Name: Instalaciones de Globos Arcoíris
- Description: Finca de despegue de Globos Arcoíris, Km 459, N-232, Cuzcurrita de Río Tirón.
- X: 3830
- Z: 3945
- Y: 42.82
- Lat: 42.55654
- Lon: -2.97265
- Image: instalaciones.png

At the bottom left is a blue "Submit" button.

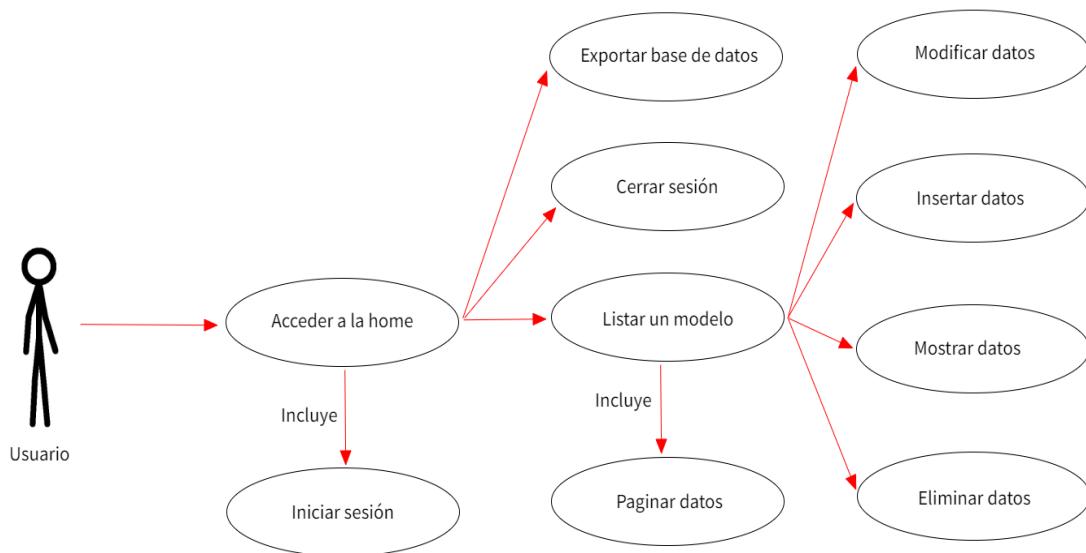
En la vista **show** se listan todos los datos del modelo de forma que se puedan visualizar cómodamente los datos almacenados.

The screenshot shows a web page titled "Show Takeoff Point". The page has a dark header with "BalloonSim Backend" and navigation links "Home" and "PhpMyAdmin" on the left, and "Log out" on the right. On the right side of the header is a blue "Back" button. The main content area displays the details of a takeoff point:

Name: Instalaciones de Globos Arcoíris
Description: Finca de despegue de Globos Arcoíris, Km 459, N-232, Cuzcurrita de Río Tirón.
X: 3830
Z: 3945
Y: 42.82
Lat: 42.55654
Lon: -2.97265
Image: instalaciones.png

Diagrama de **casos de uso** de la web del back-end. El usuario (administrador de la web) iniciará sesión para acceder a la pantalla home, puede realizar las acciones de cerrar sesión, exportar la base de datos y listar un modelo (paginando los datos).

Una vez en la vista de un determinado modelo, el usuario podrá insertar nuevos datos, mostrarlos, modificarlos y eliminarlos.



API

La **API** será la encargada de recibir las **peticiones** del front-end y devolver o insertar los datos requeridos mediante una serie de **endpoints** programados en el archivo api.php. Los endpoints son las URLs de un API o un back-end que responden a una petición.

En el ejemplo (abajo) el endpoint que crea un nuevo vuelo en la base de datos. Recibe como **parámetros** el nombre del vuelo (utilizado para poder realizar búsquedas en la tabla vuelos) y el punto de despegue. El script guarda un nuevo vuelo y **devuelve** el id insertado para que lo reciba el front-end y lo utilice para poder realizar nuevas peticiones.

```

Route::get('newflight/{name}/{takeoff}', function ($name, $takeoff) {
    $flight = new Flight();
    $flight->date = date('Y-m-d H:i:s');
    $flight->name = $name;
    $flight->takeoff = $takeoff;
    $flight->save();
    return DB::table('flights')->latest('updated_at')->first()->id;
});

```

Listado de los **endpoints** programados:

- “api/takeoffs”
 - Devuelve: el listado de puntos de despegue.

- “api/newflight/{name}/{takeoff}”
 - Guarda un nuevo vuelo.
 - Parámetros: nombre del vuelo y punto de despegue.
 - Devuelve: el id del vuelo.

- “api/newpoint/{flight}/{s}/{lat}/{lon}/{alt}/{speedy}/{direction}/{fuel}/{x}/{y}/{z}”
 - Guarda un nuevo punto de ruta.
 - Parámetros: id del vuelo, segundos desde el despegue, latitud, longitud, altitud, velocidad, velocidad de ascensión, dirección, combustible restante y posiciones X, Y, Z en el simulador.
 - Devuelve: el id del punto de ruta.

- “api/addwind/{weather}/{altitude}/{winddir}/{windspeed}”
 - Guarda un nuevo viento.
 - Parámetros: id del clima, altitud, dirección y velocidad del viento.

- “api/newweather/{id}/{t}/{pressure}”
 - Guarda un nuevo clima.
 - Parámetros: id del vuelo, temperatura y presión.

- “api/flights/{searchfor}”
 - Devuelve: el listado de vuelos que cumplan con la condición especificada en el parámetro searchfor.
 - Parámetros: condición de búsqueda.

- “api/flight/{id}”
 - Devuelve: el vuelo con el id especificado.
 - Parámetros: id del vuelo.

- “`api/routes/{flight}`”
 - Devuelve: los puntos de ruta del vuelo especificado.
 - Parámetros: id del vuelo.
- “`api/winds/{flight}`”
 - Devuelve: el listado de vientos del vuelo especificado.
 - Parámetros: id del vuelo.
- “`api/weather/{flight}`”
 - Devuelve: el clima del vuelo especificado.
 - Parámetros: id del vuelo.

Todos estos endpoints permiten la comunicación entre las partes de la web y todos los parámetros son validados conforme a las **reglas de validación** especificadas en cada uno de los modelos.

PRUEBAS

Para realizar las **pruebas** de mis endpoints he utilizado **Postman**. Postman es una herramienta que ayuda en todo el proceso de desarrollo de APIs. Se utiliza principalmente para testing web. Gracias a esta herramienta, además de testear y consumir APIs, podremos monitorear, documentar, simular y desarrollar pruebas automatizadas sobre las mismas.

En la captura (abajo) se muestra un ejemplo de **petición** a mi API mediante **GET**. La petición ataca al endpoint de obtención de los puntos de despegue. Postman permite visualizar los datos obtenidos en formato **JSON** de forma muy visual y tabulada. Las peticiones se pueden agrupar formando colecciones y nos permite **lanzarlas de forma automatizada** en un orden especificado.

Además de probar todos los endpoints que he desarrollado en la API del backend, Postman me ha servido para probar todos los servicios de terceros que he usado durante el desarrollo de la aplicación.

Endpoint

KEY	VALUE	DESCRIPTION	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

```

1
2 {
3     "id": 1,
4     "name": "Instalaciones de Globos Arcoiris",
5     "description": "Finca de despegue de Globos Arcoiris, Km 459, N-232, Cuzcurrita de Rio Tixón.",
6     "x": 3830,
7     "z": 3945,
8     "y": 42.82,
9     "lat": 42.55654,
10    "lon": -2.97265,
11    "image": "instalacines.png",
12    "created_at": "2022-02-28T09:41:47.000000Z",
13    "updated_at": "2022-04-12T12:23:09.000000Z"
14 },
15 {
16     "id": 3,
17     "name": "Carretera Anguciana",
18     "description": "Finca de despegue situada entre las localidades de Haro y Anguciana, carretera
19         LR-202.",
20     "x": 5015,
21     "z": 4190,
22     "y": 34.57,
23     "lat": 42.57236,
24 }

```

200 OK 290 ms 8.39 KB Save Response

Respuesta obtenida

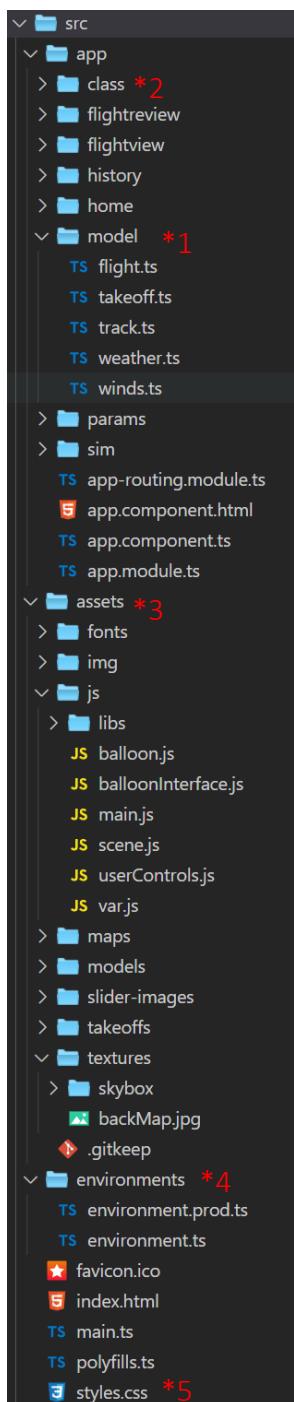
INSERCIÓN DE DATOS

Los únicos datos que no son autogenerados desde el front-end son el **listado de puntos de despegue**. Para **insertar** estos **datos** hice uso de la web creada en el back-end lo que me sirvió para realizar **pruebas** sobre este apartado. Para poder insertar puntos de despegue reales utilizados por la empresa, me puse en contacto con uno de los pilotos. Un punto de despegue es, básicamente, una finca que no esté siendo cultivada o cualquier otro espacio al aire libre con suficiente espacio para realizar el inflado y despegue de un globo aerostático que suele medir unos 30 metros, similar a un edificio de 8 plantas. **Para generar estos datos** hay que saber cuál es la finca que se desea incluir y cargar el simulador; a continuación, se sitúa el globo en la posición de dicha finca y se imprimen por consola los datos de latitud, longitud, X, Y, y Z. Al final la aplicación incluye **28 puntos de despegue** reales distintos que se pueden seleccionar para iniciar el vuelo desde el front-end.

SPRINT 3

Para el sprint 3 empecé a trabajar en el apartado **front-end** del proyecto y **Docker**.

ANGULAR



En el front-end he seguido la **estructura de carpetas** básica de Angular, esta se basa en una serie de componentes formados por un archivo **HTML**, un archivo **TypeScript** y, opcionalmente una hoja de estilos (**CSS**). Cada uno de estos componentes representa una página dentro de la web. Para mi proyecto creé **6 páginas distintas**:

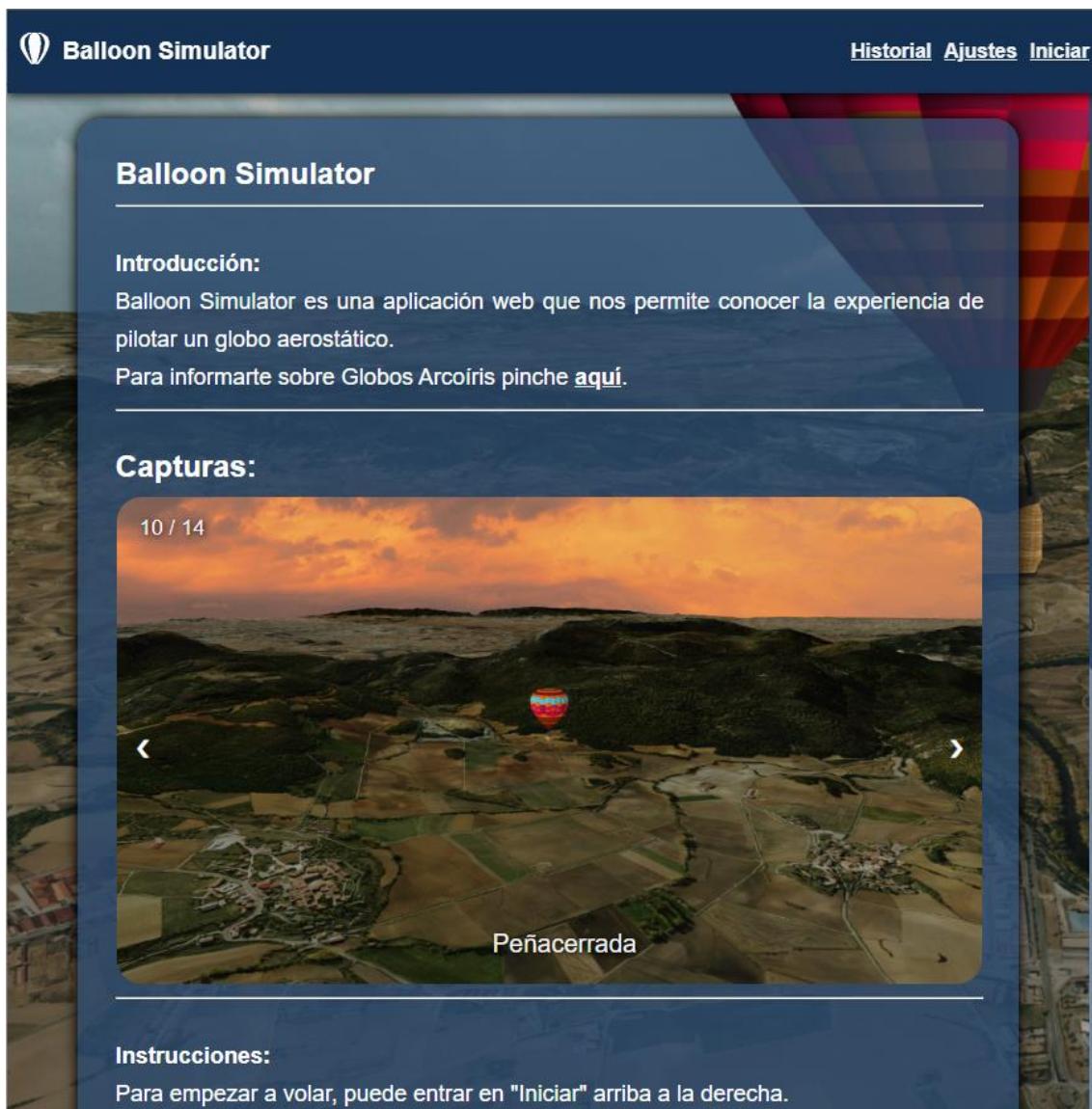
- **sim**: que es la página encargada de ejecutar el **simulador** (Babylon.js).
- **params**: un apartado en el que el usuario podrá seleccionar los **ajustes** del simulador.
- **home**: la **página principal** con las instrucciones y una galería de imágenes.
- **history**: un apartado que contendrá un **listado de los vuelos** realizados y un **buscador**.
- **flightview**: una página que contendrá los **datos** de un vuelo concreto.
- **flightreview**: un apartado que cargará de nuevo el simulador para **volver a ver un vuelo**.

Además de las páginas también está la carpeta **“model”** (1) en la que se encuentra el modelo de datos y la carpeta **“class”** (2) que contiene las clases necesarias para el funcionamiento de la web. La carpeta **“assets”** (3) contiene todos los recursos del front-end, imágenes, fuentes, texturas, modelos 3D, etc. Además, también contiene el código JavaScript del simulador y las librerías de Babylon.js. En la carpeta **“environments”** (4) se encuentran las variables de entorno, que serán modificadas si nos encontramos en un entorno de **desarrollo** o de **producción**. Por último, el archivo **“styles.css”** (5) contiene los estilos generales de la aplicación.

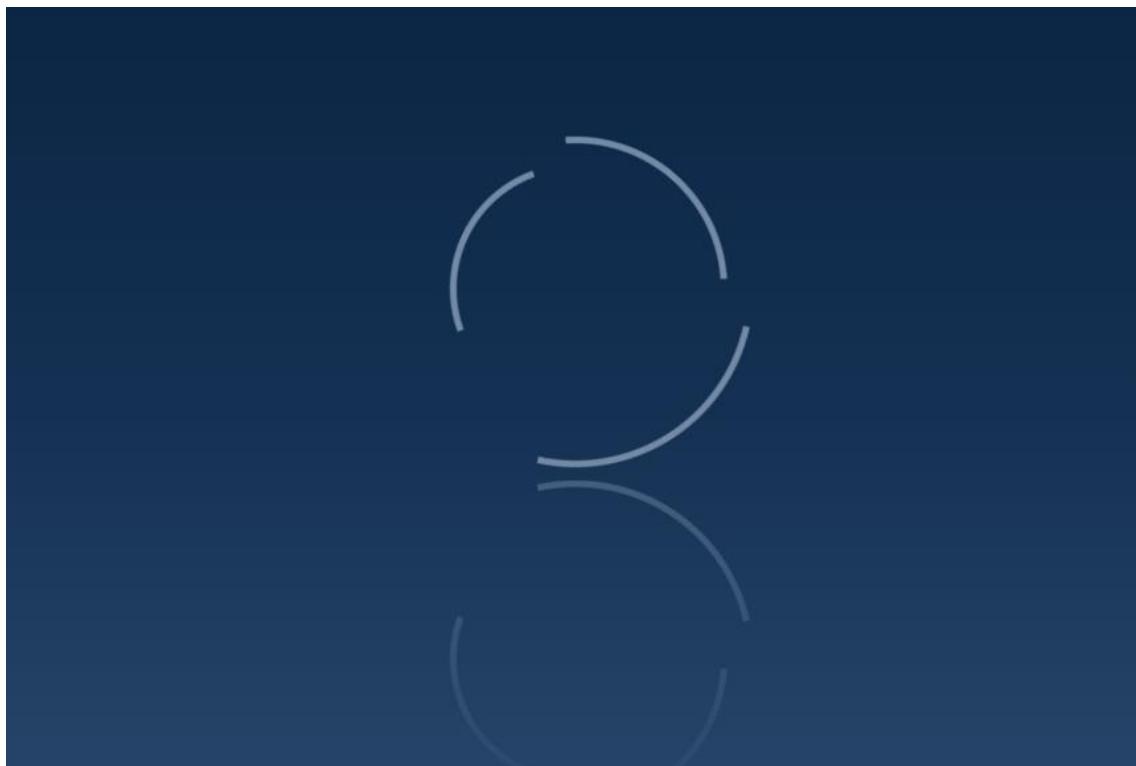
DISEÑO

Para el **diseño de la web** he escogido una barra de menú en la parte superior y un contenido en el centro en forma de tarjeta. El diseño se mantiene igual para todos los apartados. He establecido como fondo una imagen del simulador y he escogido una **paleta de colores** con tonos azulados extraída de **Paletton** (una web que proporciona paletas de colores para ayudar a los diseñadores web).

Para la pestaña **home** (abajo) incluí una descripción breve de la web, unas instrucciones de uso y una **galería de imágenes interactiva** con 14 imágenes extraídas del simulador. El **menú** se compone del logo de la web en la parte izquierda y la navegación con tres enlaces, historial, ajustes e iniciar. La **barra de desplazamiento lateral** ha sido modificada por CSS para que cumpla con el estilo general de la web.



Para la **pantalla de carga** del simulador he desarrollado una **animación** por CSS con unos semicírculos que rotan con un efecto de sombreado. Intenté poner además un indicador con el porcentaje de carga, pero Babylon.js no nos permite saber en qué estado se encuentra la carga.



IMPLEMENTACIÓN

Para el **modelo de datos** he creado un duplicado de los que se encuentran en Laravel, de forma que contengan las mismas propiedades. Esto ayuda a la hora de realizar las peticiones a la API del back-end.

En la imagen, de ejemplo el modelo de la clase flight, con todas sus propiedades y el constructor al que se le pasan todos los parámetros mediante una variable de tipo **JSON**.

```
export default class Flight {
    public id: number;
    public no!: number;
    public date: number;
    public name: string;
    public takeoff: string;
    public duration: string;
    public seconds: number;

    /**
     * Creates an instance of Flight.
     * @param params the parameters
     */
    constructor(params: any) {
        this.id = params.id;
        if (params.no) this.no = params.no;
        this.date = params.date;
        this.name = params.name;
        this.takeoff = params.takeoff;
        this.seconds = params.duration;
        this.duration = timeInSecondsToString(params.duration);
    }
}
```

Para realizar las **peticiones** tanto a la API del back-end como a las APIs de terceros he desarrollado un controlador llamado “**requestController**”. En el ejemplo (abajo) el método de la clase que realiza la petición para iniciar un **nuevo vuelo**. Pasaremos por parámetros el punto de despegue seleccionado y el nombre que asignaremos al vuelo. Mediante fetch, realizamos la petición y como **respuesta** de la API obtenemos el **ID** del nuevo vuelo. Como podemos apreciar en el código, utilizo las **variables de entorno** para poder seleccionar la ruta de mi API de forma que, mientras nos encontremos en un entorno de pruebas o desarrollo no afectemos a la misma base de datos que usamos en producción.

```
/**
 * Start a new flight and return the id
 *
 * @param {string} takeoff the takeoff name
 * @param {string} name name of the flight
 * @returns {Promise<number>} id of the flight
 */
public static async startFlight(takeoff: string, name: string): Promise<number> {
  let toReturn: number = 0; ← Variable de entorno
  await fetch(`${environment.apiRoute}newflight/${name}/${takeoff}`) } Petición
    .then((response) => response.json())
    .then((id) => {
      toReturn = id; } Respuesta
    });
  return toReturn;
}
```

PRUEBAS

Para probar estos métodos he desarrollado otra clase: “**consoleController**” que alberga métodos para introducirse mediante la consola del navegador web, que muestran por pantalla los resultados de las peticiones y otros parámetros.

```
/**
 * Shows the list of takeoffs
 */
private static async showTakeoffs() {
  const takeoffs = await RequestController.getTakeOffs();
  console.groupCollapsed('Takeoffs');
  console.table(takeoffs);
  console.groupEnd();
}
```

The screenshot shows the browser's developer tools console tab. The left pane displays a list of commands under the 'Help' section:

- help() --> Shows help
- showTakeoffs() --> Shows the list of Takeoffs
- showFlights() --> Shows the list of Flights
- showWinds() --> Shows the Winds response
- showGlobal() --> Shows the Global variables
- showInGamePosition() --> Shows the actual position variable

Below this, it says "Angular is running in development mode. Call enableProdMode() [webpack-dev-server] Live Reloading enabled."

A red bracket on the right side groups the first two items as "Ayuda de los comandos" (Command help).

The right pane shows the output of the 'showTakeoffs()' command, which is a ZoneAwarePromise object. A red bracket on the right side groups this as "Puntos de despegue" (Takeoff points). The output is a table:

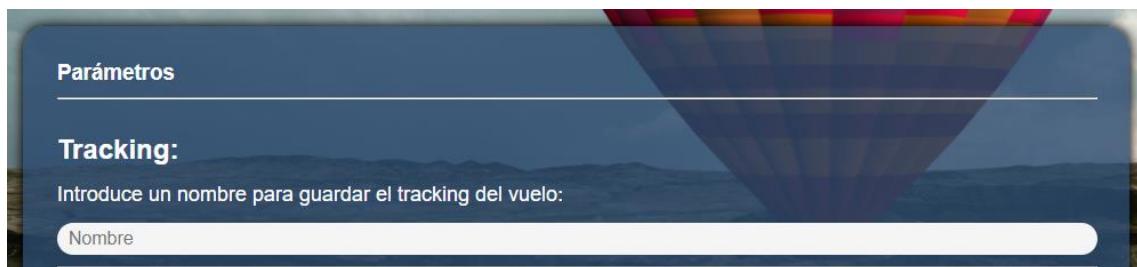
(índice)	id	name	descrip...	x	z	y	alt	lat	lon	img
0	1	'Insta...	'Finca ...	3830	3945	42.82	525	42.556...	-2.972...	'insta...
1	3	'Carre...	'Finca ...	5015	4190	34.57	487	42.572...	-2.875...	'haro3...
2	4	'Buded...	'Finca ...	3240	5470	46.13	539	42.649...	-3.021...	'buged...
3	5	'Miran...	'Punto ...	4065	6140	32.94	480	42.6905	-2.953...	'miran...
4	6	'Parki...	'Punto ...	4430	5625	34.6	488	42.659...	-2.923...	'miran...
5	7	'Ocio'	'Despeg...	5760	5540	46.12	539	42.654...	-2.814...	'ocio...
6	8	'Santo...	'Punto ...	3955	1945	72.48	658	42.435...	-2.962...	'sto.p...
7	9	'Santo...	'Finca ...	4220	1945	70.82	651	42.435...	-2.940...	'ste.p...
8	10	'Ezcar...	'Punto ...	3310	205	108.71	821	42.329...	-3.0156	'ezcar...
9	11	'Badar...	'Finca ...	5895	835	60.94	606	42.367...	-2.802...	'badar...
10	12	'Campo...	'Finca ...	4840	1555	88.94	732	42.411...	-2.889...	'golf...
11	13	'Cidam...	'Punto ...	4995	2915	59.29	599	42.4944	-2.876...	'cidam...
12	14	'Haro,...	'Despeg...	5340	4325	28	458	42.580...	-2.848...	'bodeg...
13	15	'Haro,...	'Finca ...	5400	4110	32.95	480	42.567...	-2.843...	'mazo...

Estos métodos han sido de gran ayuda para la realización de **pruebas**, así como para conocer la **ubicación concreta del globo** en el simulador para guardar los puntos de despegue. En el ejemplo (arriba) la consola del navegador con el listado de puntos de despegue.

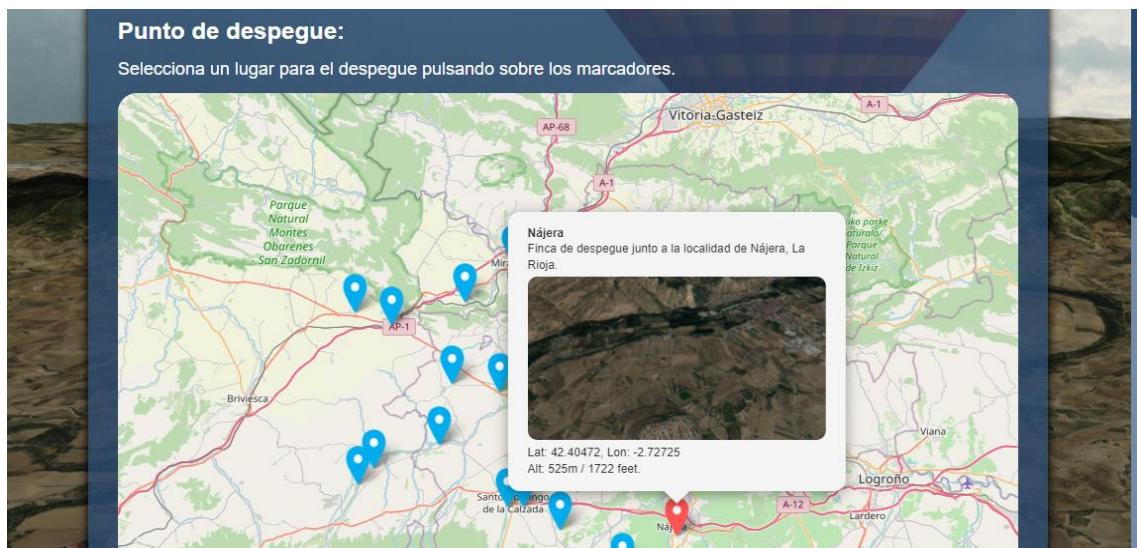
SPRINT 4

Para el **sprint** número **cuatro** me propuse realizar la **página de ajustes**, una de las más complejas de la web. Este apartado es básicamente un gran **formulario** que permite al usuario seleccionar los ajustes del simulador (nombre del vuelo, vientos, punto de despegue, color del cielo...).

El **primer campo** del formulario (abajo) es el **nombre del vuelo**. Es un input de tipo texto que permite seleccionar un nombre para el vuelo, esto nos permitirá realizar búsquedas por nombre a la hora de mostrar el historial.



El **segundo campo** utiliza **Leaflet** (una librería para crear **mapas interactivos**) para mostrar los **marcadores** de los **puntos de despegue**, el usuario elegirá de esta forma desde donde comenzar el vuelo. Al presionar un marcador, éste cambia su color a rojo indicando que es el elegido y se muestra una pestaña con los datos del punto de despegue extraídos de la API del front-end. En la pestaña se muestran la imagen, descripción, coordenadas y altitud del punto de despegue.



Código de creación del mapa con Leaflet y los marcadores, alojado en el archivo “params.component.ts”.

```
this.map = L.map('map', { center: GLOBAL.MAP_CENTER, zoom: 10 });
const tiles = L.tileLayer('https://s.tile.openstreetmap.org/{z}/{x}/{y}.png', {
  maxZoom: 15,
  minZoom: 10
});
tiles.addTo(this.map);    Recorremos los puntos
                          de despegue.
//#region Markers
this.markers = [];

GLOBAL.TakeoffPointsList.forEach((takeoff) => {
  const marker = L.marker([takeoff.lat, takeoff.lon], {
    title: takeoff.name,
    riseOnHover: true
  });

  //Image
  const img = document.createElement('img');
  img.src = `.../assets/takeoffs/${takeoff.img}`;
  img.style.width = '100%';
  img.alt = takeoff.name;
  img.style.borderRadius = '10px';
  img.style.marginTop = '3px';

  //Marker
  marker.addTo(this.map);
  marker.on('click', () => {
    GLOBAL.SelectedTakeoff = takeoff;
    this.markers.forEach((element: L.Marker) => {
      element.setIcon(this.defaultMapIcon);
      element.setZIndexOffset(0);
    });
    marker.setIcon(this.redMapIcon);
    marker.setZIndexOffset(100);
    this.windsMap.changeCenter([takeoff.lat, takeoff.lon]);
    this.windsMap.updateWindsMap();
  });
  marker.bindPopup(`<b>${takeoff.name}</b><br>
${takeoff.description}<br>
${img.outerHTML}<br>
Lat: ${takeoff.lat}, Lon: ${takeoff.lon}<br>
Alt: ${takeoff.alt}m / ${Math.round(metersToFeet(takeoff.alt))} feet.`);
  marker.setZIndexOffset(0);
  if (GLOBAL.SelectedTakeoff == takeoff) {
    marker.setIcon(this.redMapIcon);
    marker.setZIndexOffset(100);
  }
  this.markers.push(marker);
});
//endregion
```

Petición a OpenStreetMap

Recorremos los puntos de despegue.

Marcador

Imagen del lugar de despegue

Agregamos el marcador al mapa

Método on-click del marcador

HTML de la pestaña del popup.

El **tercer campo** es el de los **vientos**. Se compone de una tabla que muestra los datos de las distintas capas de viento extraídas de **Windy**. La tabla se compone de la altitud a la que empieza la capa de viento, la dirección en grados y su velocidad en kilómetros por hora y en nudos. En la columna derecha el usuario dispone de una serie de **acciones** a realizar con cada viento, utilizando los iconos de **Fontawesome**. En la parte inferior de la tabla se encuentra de nuevo un **mapa interactivo** en el que se visualizan los vientos marcados. Las **líneas representan la dirección y velocidad** del viento de forma visual y el usuario puede **cambiar el color** de cada viento o **mostrar / ocultar** el mismo mediante las acciones de la tabla. Las líneas parten de un punto central localizado en el punto de despegue escogido y si trazamos mentalmente un cono entre dos de ellas obtenemos los lugares que podremos sobrevolar si nos mantenemos en ese rango de altura. El usuario puede **eliminar** los vientos deseados pulsando el icono de la papelera o **añadir nuevos** pulsando el botón verde de la parte superior derecha, mediante un **popup**. Los iconos con forma de ojo sirven para ocultar o mostrar el viento y cambian su color mediante una animación CSS. Los selectores de color utilizan un elemento **color-picker**.

Vientos:
+ Añadir un viento

Altitud (s. n. m.)	Dirección	Velocidad	Acciones
432 m / 1417 feet	207.05°	6.92 km/h / 3.7 nudos.	█ █ █
580 m / 1903 feet	55.82°	3.21 km/h / 1.7 nudos.	█ █ █
712 m / 2336 feet	167.74°	7.02 km/h / 3.8 nudos.	█ █ █
821 m / 2694 feet	243.48°	23.42 km/h / 12.6 nudos.	█ █ █
1189 m / 3901 feet	3.03°	3.94 km/h / 2.1 nudos.	█ █ █
1457 m / 4780 feet	164.62°	29.89 km/h / 16.1 nudos.	█ █ █
1949 m / 6394 feet	92.61°	14.97 km/h / 8.1 nudos.	█ █ █

The map displays a region with various towns and roads. Wind vectors are shown as colored lines originating from a central point. Red arrows indicate the direction and strength of the wind at different altitudes. A large red shaded area represents the potential landing zones between the orange and dark blue wind layers. A green line also highlights a specific path or boundary on the map.

El área dibujada en rojo representa los posibles lugares que tenemos para volar si nos movemos entre las altitudes del viento representado en naranja y el coloreado en azul oscuro.

Como el mapa de vientos será reutilizado en la página del historial de vuelos, tiene una **clase propia** llamada “WindsMap” con los métodos necesarios para pintar cada viento, cambiar su color, cambiar el centro de las líneas, cambiar la visibilidad de cada viento... En los **ejemplos** (abajo) algunos métodos de la clase. Nótese que para poder pintar las líneas hay que realizar un **complejo cálculo** utilizando la dirección del viento, su velocidad y las coordenadas centrales.

```
/*
 * Shows a wind route representing wind speed and direction
 * @param {Wind} wind wind to show
 */
private seeWindOnMap(wind: Wind) {
    const kmx = wind.speedKMH * Math.cos(wind.direction * Math.PI / 180);
    const kmy = wind.speedKMH * Math.sin(wind.direction * Math.PI / 180);
    const lat = this.center[0] + kmx / 111.2;
    const lon = this.center[1] + kmy / (111.2 * Math.cos(this.center[0] * Math.PI / 180));

    const route = L.polyline([[this.center[0], this.center[1]], [lat, lon]], {
        color: wind.color,
        weight: 3
    });
    route.bindPopup(`<br>Altitud: ${wind.altitude.toFixed(0)} m / ${metersToFeet(wind.altitude).toFixed(0)} feet</b><br>
    Dirección: ${wind.direction.toFixed(2)}°<br>
    Velocidad: ${wind.speedKMH} km/h / ${kmPerHourToKnots(wind.speedKMH).toFixed(2)} nudos.<br>`);
    route.addTo(this.map);
}

/**
 * Changes wind color
 *
 * @param {Wind} wind the wind to change
 * @param {string} color color with which the wind will be changed
 */
public changeWindColor(wind: Wind, color: string) {
    wind.color = color;
    this.updateWindsMap();
}
```

Cálculo de la longitud de la línea, basado en 1h de vuelo

Pop up al pulsar la línea

Agregamos la línea al mapa

Creamos la línea

Cambiamos el color de un viento dado

```
/*
 * Updates the winds map
 */
public updateWindsMap() {
    if (!this.map) return;

    //Remove previous winds
    this.map.eachLayer((layer: any) => { if (layer.options.color) this.map.removeLayer(layer); });

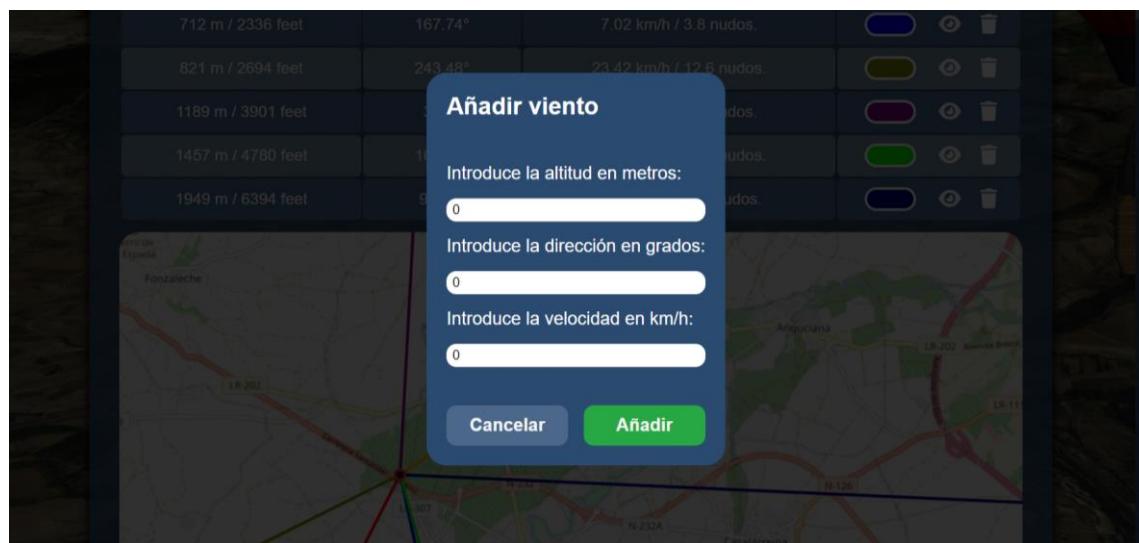
    //Add winds
    this.windsList.forEach((wind: Wind) => {
        if (!wind.color) {
            wind.color = this.colorsWindLines[this.actColor];
            this.actColor++;
            if (this.actColor >= this.colorsWindLines.length) this.actColor = 0;
        }
        if (wind.seeOnMap) this.seeWindOnMap(wind);
    });
    const marker = L.circleMarker([0, 0], { radius: 5, color: COLORS.red, fillColor: 'black', fillOpacity: 1 });
    marker.addTo(this.map);
    marker.setLatLng(this.center);
}
```

Borramos los vientos anteriores

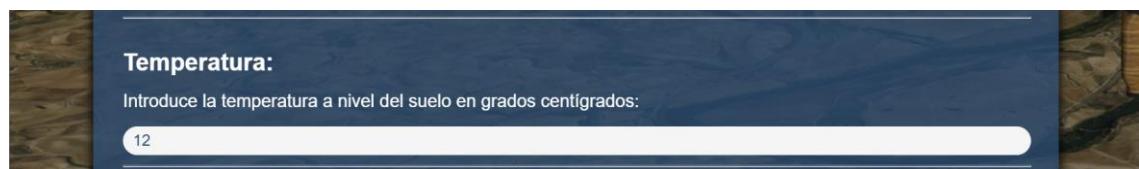
Añadimos todos los vientos

Agregamos el marcador central

Popup para introducir un **nuevo viento** (abajo), es un simple formulario que permite agregar nuevos vientos.



El **cuarto campo** del formulario (abajo) permite introducir la **temperatura** en tierra (ésta varía automáticamente con la altitud). Todos los campos numéricos y de texto cuentan con **validaciones** de forma que **no se puedan introducir datos erróneos** como velocidades negativas, texto en lugar de números, grados superiores a 360 o inferiores a 0, altitudes negativas, etc.



El **quinto campo** es un selector para el **color del cielo** que se verá en el simulador. El usuario puede elegir entre **cuatro tonalidades** diferentes, es un ajuste estético.



Por último, tenemos dos botones que nos permiten **importar y exportar** estos ajustes mediante un archivo **JSON**. Esto será muy útil si se desea **repetir vuelos** de sesiones anteriores sin tener que introducir todos los vientos a mano. La página del historial de cada vuelo contará también con el botón de exportar para poder extraer los ajustes.



Código con el método para **exportar** los ajustes (abajo). Gracias al atributo “download” del elemento “a” del HTML podemos decirle al navegador que **descargue el archivo**. La variable “settings” será el contenido de dicho archivo. Contiene el nombre del vuelo, la temperatura, el punto de despegue, el listado de las diferentes capas de viento y el color del cielo.

```
/**  
 * Export settings to file  
 */  
public exportSettings() {  
    const settings = {  
        "FlightName": GLOBAL.FlightName,  
        "Temperature": GLOBAL.Temperature,  
        "Takeoff": GLOBAL.SelectedTakeoff,  
        "Winds": GLOBAL.Winds.windsList,  
        "SkyboxColor": GLOBAL.SkyboxColor  
    }  
    const blob = new Blob([JSON.stringify(settings)], { type: "application/json" });  
    const url = window.URL.createObjectURL(blob);  
    const a = document.createElement("a");  
    a.href = url;  
    a.download = `ajustes_vuelo_${GLOBAL.FlightName.replace(' ', '-')}.json`;  
    a.click();  
}
```

Datos a exportar en formato JSON

Descargamos el archivo

A code snippet in JavaScript. It defines a method 'exportSettings' that creates a 'settings' object containing flight name, temperature, takeoff, winds, and skybox color. This object is then converted to a JSON string and used to create a blob. The blob is converted to a URL, which is then set as the href of a newly created anchor element. The download attribute of this anchor is set to 'ajustes_vuelo_{FlightName.replace(' ', '-')}.json'. Finally, the click event of the anchor is triggered. Red annotations with arrows point from the explanatory text to the 'settings' object and the 'download' attribute.

Al importar unos ajustes se **verificará** por código que los **datos** que contiene el JSON son **correctos**, además se modificará toda la página con el formulario de forma que represente los nuevos ajustes seleccionados. Esto implica modificar ambos mapas interactivos y los valores de los inputs.

En la imagen de **ejemplo** (abajo) se muestra el archivo “ajustes_vuelo_{nombre del vuelo}.json” descargado.

```
{  
    "FlightName": "Sin nombre",  
    "Temperature": 12,  
    "Takeoff": {  
        "id": 1,  
        "name": "Instalaciones de Globos Arcoíris",  
        "description": "Finca de despegue de Globos Arcoirirs, Km 459, N-232, Cuzcurrita de Río Tirón.",  
        "x": 3830,  
        "z": 3945,  
        "y": 42.82,  
        "alt": 525,  
        "lat": 42.55654,  
        "lon": -2.97265,  
        "img": "instalaciones.png"  
    },  
    "SkyboxColor": "browncloud",  
    "Winds": [  
        {  
            "altitude": 432,  
            "direction": 207.04571247053053,  
            "speed": 1.92107020740794,  
            "speedKMH": "6.92",  
            "altitudeFeet": 1417,  
            "seeOnMap": true,  
            "color": "#ff0000"  
        },  
        {  
            "altitude": 580,  
            "direction": 55.822119764857746,  
            "speed": 0.8911062672181163,  
            "speedKMH": "3.21",  
            "altitudeFeet": 1903,  
            "seeOnMap": true,  
            "color": "#ffa500"  
        },  
        {  
            "altitude": 712,  
            "direction": 167.74196954718744,  
            "speed": 1.949554153609391,  
            "speedKMH": "7.02",  
            "altitudeFeet": 2336,  
            "seeOnMap": true,  
            "color": "#0000ff"  
        },  
        {  
            "altitude": 821,  
            "direction": 243.47656563245172,  
            "speed": 6.50593085894491,  
            "speedKMH": "23.42",  
            "altitudeFeet": 2694,  
            "seeOnMap": true,  
            "color": "#808000"  
        },  
        {  
            "altitude": 1189,  
            "direction": 3.0250804671069376,  
            "speed": 1.0933048532293799,  
            "speedKMH": "3.94",  
            "altitudeFeet": 3901,  
            "seeOnMap": true,  
            "color": "#800080"  
        },  
        {  
            "altitude": 1457,  
            "direction": 164.6188202463725  
        }  
    ]  
}
```

SPRINT 5

Durante este sprint desarrollé el apartado del historial de los vuelos y alguna tarea más simple, como el mapa de la tablet del simulador.

El **mapa** que se muestra en la tablet del simulador es, de nuevo, un mapa cargado mediante **Leaflet**. Leaflet nos permite dibujar **rutas** mediante una serie de puntos, esto se puede usar para ir creando una ruta a medida que el globo aerostático se desplaza por el mapeado. En el **código** (abajo) se muestra el **intervalo** que dibuja los puntos de la ruta cada 300 milisegundos.



```
//Map update interval
this.mapUpdateInterval = setInterval(() => {
  if (started && balloon && this.flightID && this.seconds > environment.secondsBetweenRouteSaves) {
    const latLng = new L.LatLng(balloon.calcDegreesLat(), balloon.calcDegreesLon());
    marker.setLatLng(latLng);
    if (this.isMapCentered) this.map.panTo(latLng);
    route.addLatLng(latLng); ← Punto en el que se encuentra el globo
  } else {
    if (this.isMapCentered) this.map.panTo(marker.getLatLng()); ← Agregamos a la ruta
  }
  this.changeBalloonDirection();
}, 300); ← Centramos el mapa si está la opción activada
```

HISTORIAL

El **historial** es una pestaña que incluye una **tabla** en la que se muestran los **vuelos** que se han realizado anteriormente. Dicha tabla contiene los campos del **nombre** del vuelo, el lugar de **despegue**, la **duración** del vuelo, la **fecha** en la que se realizó y una columna extra con un botón (ícono de ojo) que permite dirigirnos a la pestaña con los datos en concreto de un vuelo. Un **buscador** en la parte superior derecha permite realizar una búsqueda por nombre en el evento “key-down” (mientras el usuario escribe).

En la imagen (abajo) la pestaña historial a la que se accede mediante la barra de navegación. Al realizar una búsqueda se aprovechan los métodos de Angular para **recargar dinámicamente** la tabla.

No	Nombre	Despegue	Fecha	Duración	Ver
1	Sin nombre	Instalaciones de Globos Arcoiris	2022-05-08	00:01:56	
2	Mi vuelo por la finca de Globos Arcoiris	Instalaciones de Globos Arcoiris	2022-04-20	00:26:04	
3	Vuelo desde Ezcaray	Ezcaray	2022-04-18	00:40:42	

FLIGHTVIEW

El componente **flightview** contiene los datos del vuelo. Tiene varios apartados en los que se muestran diferentes datos. El **primer apartado** (abajo) contiene el tracking del vuelo, es decir, la ruta que ha seguido. En esencia es igual al de la tablet. En la parte superior, el **nombre** del vuelo y un **botón** que nos redirige al componente **flightreview**.

Tracking:
Ruta seguida por el aerostato durante el vuelo.

El **segundo componente** es una **tabla** que contiene **datos** útiles sobre el vuelo, estos datos son calculados gracias a la tabla rutas de la base de datos que contiene todos los datos de cada segundo del vuelo. Los **datos a mostrar** son:

- Punto de despegue
- Fecha del vuelo
- Duración del vuelo
- Distancia recorrida (km)
 - Máxima
 - Media
 - Velocidad de ascensión máxima
 - Velocidad de descenso máxima
- Altitud (en metros y pies)
 - Máxima
 - Media
 - Mínima
 - Altitud del punto de despegue
- Desplazamiento (distancia en línea recta entre despegue y aterrizaje)
- Combustible restante (%)
- Temperatura en tierra (grados centígrados)
- Presión (sobre el nivel del mar)

Datos del vuelo:			
Despegue en:	Fecha del vuelo:	Duración del vuelo:	Distancia recorrida:
Instalaciones de Globos Arcoíris.	2022-04-20	00:26:04	4 km, 447 metros.
Velocidad máxima:	Velocidad media:	V. ascensión máxima:	V. descenso máxima:
41.80 km/h / 22.6 nudos.	10.24 km/h / 5.5 nudos.	4.83 m/s	-3.45 m/s
Altitud máxima:	Altitud mínima:	Altitud media:	Altitud despegue:
1232.00 m / 4042 feet	510.00 m / 1673 feet	814.99 m / 2674 feet	525.00 m / 1722 feet
Desplazamiento en línea recta:	Combustible restante:	Temperatura en tierra:	Presión s.n.m.
2 km, 370 metros.	79.36 %	12.00 °C	1013 hPa

En el **código** (abajo) se muestran los **cálculos** realizados para mostrar la ruta y los datos de la tabla. Para los cálculos de distancia recorrida, Leaflet tiene un método llamado “distanceTo” que devuelve la distancia en metros entre dos posiciones (conjunto de latitud y longitud) dadas. Además, guardo en unos arrays los datos necesarios para posteriormente representarlos en **gráficos**.

```

//ForEach route
this.maxSpeed = 0;
this.maxSpeedY = 0;
this.minSpeedY = 0;
this.maxAltitude = 0;
this.minAltitude = this.routes[0].altitude;
this.takeoffAltitude = this.routes[0].altitude;
this.distance = 0;
this.remainingFuel = this.routes[this.routes.length - 1].fuel;
const firstLatLng = new L.LatLng(this.routes[0].lat, this.routes[0].lon);
const lastLatLng = new L.LatLng(this.routes[this.routes.length - 1].lat, this.routes[this.routes.length - 1].lon);
this.straightLineDistance = firstLatLng.distanceTo(lastLatLng);
let previousLatLng = new L.LatLng(this.routes[0].lat, this.routes[0].lon);
let sumAltitude = 0;
const labelSeconds: string[] = [];
const dataAltitude: number[] = [];
const dataFuel: number[] = [];
const dataSpeed: number[] = [];
const dataDirection: number[] = [];
const dataSpeedY: number[] = [];

this.routes.forEach((point) => {
  if (point.altitude > this.maxAltitude) this.maxAltitude = point.altitude;
  if (point.altitude < this.minAltitude) this.minAltitude = point.altitude;
  if (point.speed > this.maxSpeed) this.maxSpeed = point.speed;
  if (point.speedy > this.maxSpeedY) this.maxSpeedY = point.speedy;
  if (point.speedy < this.minSpeedY) this.minSpeedY = point.speedy;
  labelSeconds.push(timeInSecondsToString(point.seconds, true));
  dataAltitude.push(point.altitude);
  dataFuel.push(point.fuel);
  dataSpeed.push(metersPerSecondToKmPerHour(point.speed));
  dataDirection.push(point.direction);
  dataSpeedY.push(point.speedy);
  const latlng = new L.LatLng(point.lat, point.lon);
  this.distance += latlng.distanceTo(previousLatLng);
  track.addLatLng(latlng); ← Cálculo de la distancia
  previousLatLng = latlng;
  sumAltitude += point.altitude;
});

this.avgAltitude = sumAltitude / this.routes.length;
this.distance = parseFloat(this.distance.toFixed(2));

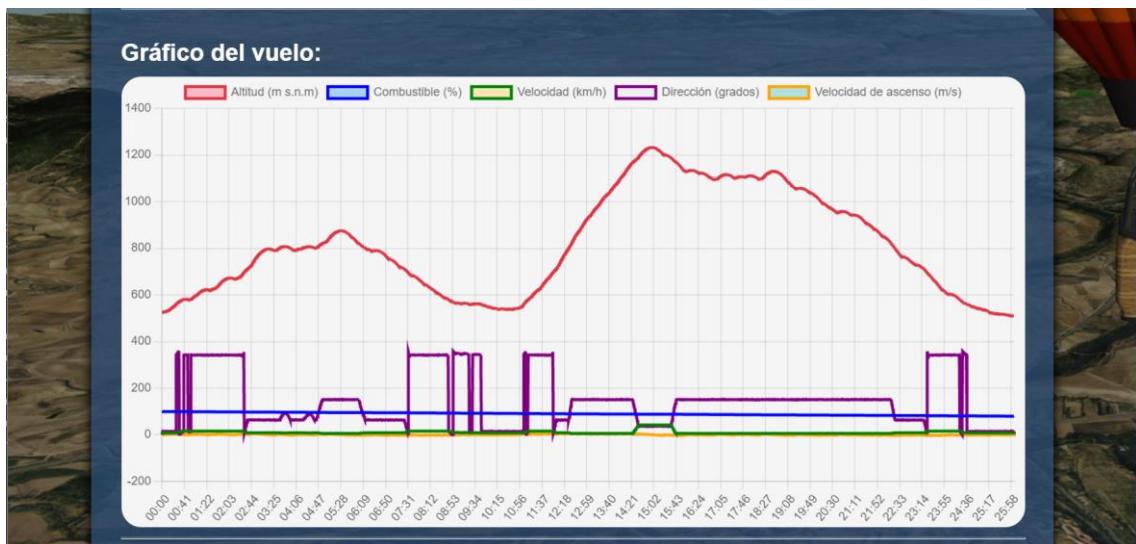
```

Recorremos los puntos de ruta

Cargamos los arrays para los gráficos

Añadimos el punto a la ruta del mapa

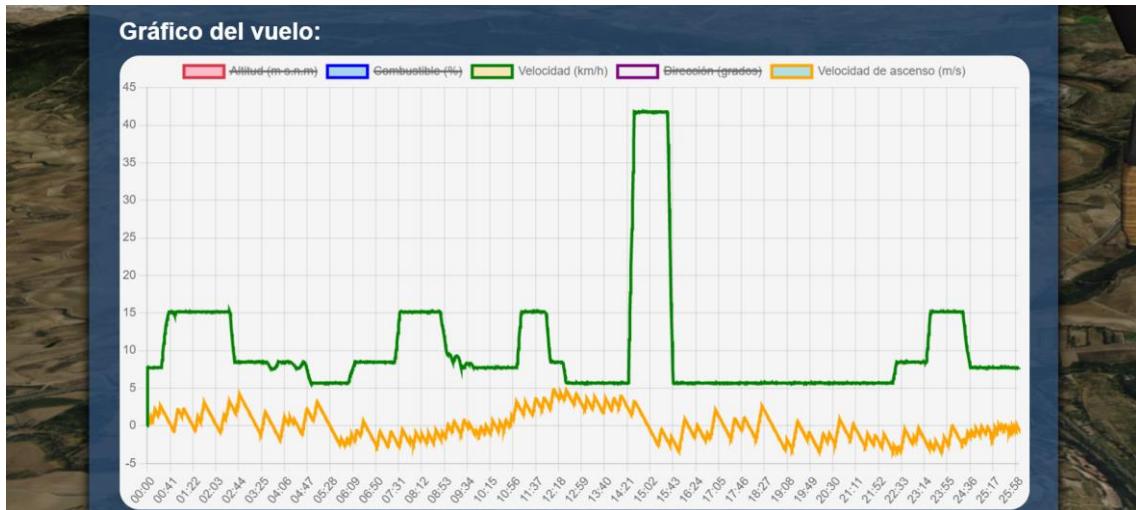
El tercer componente es un **gráfico interactivo** de los datos del vuelo. Utilizo para ello **Chart.js**, una librería de JavaScript. En la imagen (abajo) todos los gráficos juntos.



El usuario puede **seleccionar** qué **gráfico visualizar** (pulsando sobre los cuadrados de la leyenda) de forma que puede comparar unos valores con otros. En la imagen (abajo) el gráfico de la **altitud** del aerostato.



Otro **ejemplo**: en amarillo la **velocidad de ascenso** y en verde la velocidad. Como podemos apreciar la **velocidad** se mantiene igual en su capa de viento, con pequeñas variaciones.



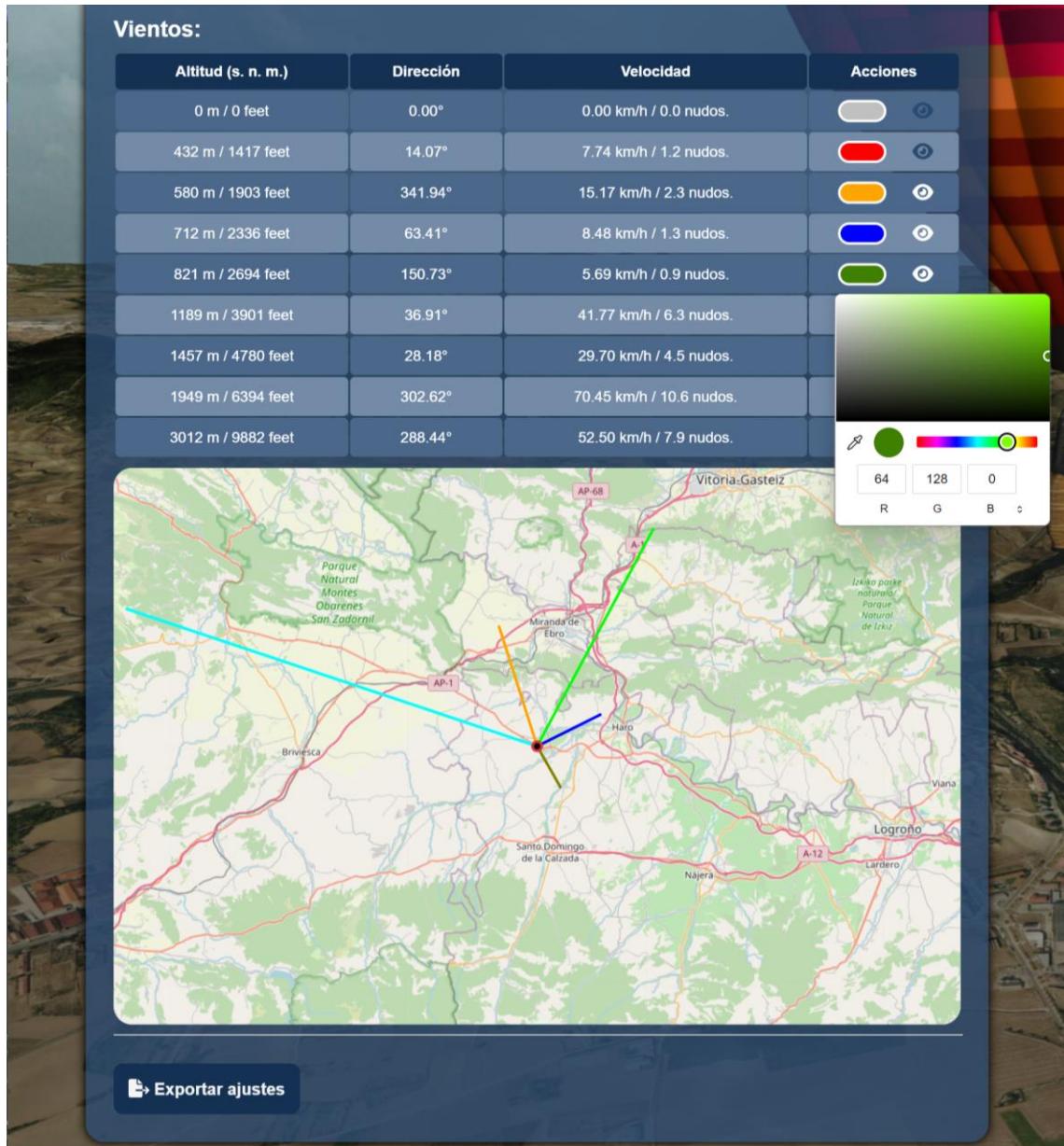
En el **código** (abajo) se muestra como son renderizados los gráficos. El eje inferior muestra el **tiempo** de vuelo mientras que el eje lateral muestra el dato numérico con la magnitud (varía dependiendo del dato: m/s, km/h, altitud, grados, etc.). Chart.js permite representar los gráficos como líneas, sectores, círculos, barras... Yo los he renderizado todos en forma de **línea** porque era la más adecuada para los datos representados.

```

/*
 * Creates a graphic with the altitude of the flight
 *
 * @param {string} labels labels for the chart, seconds
 * @param {number} dataAltitude altitude of the flight
 * @param {number} dataFuel fuel remaining
 * @param {number} dataSpeed speed of the balloon
 * @param {number} dataDirection direction of the balloon
 * @param {number} dataSpeedY speed in Y, ascension speed
 * @returns {void}
 */
public createGraphicData(labels: string[], dataAltitude: number[], dataFuel: number[], dataSpeed: number[],
    new Chart(document.getElementById('graphicAltitude') as HTMLCanvasElement, {
        type: 'line', ← Tipo de gráfico
        data: {
            labels: labels,
            datasets: [
                {
                    label: 'Altitud (m s.n.m)',
                    data: dataAltitude,
                    borderColor: COLORS.red,
                    pointBorderWidth: 0,
                    pointStyle: 'line',
                },
                {
                    label: 'Combustible (%)',
                    data: dataFuel,
                    borderColor: 'blue', ← Color de la línea
                    pointBorderWidth: 0,
                    pointStyle: 'line',
                },
                {
                    label: 'Velocidad (km/h)',
                    data: dataSpeed,
                    borderColor: 'green',
                    pointBorderWidth: 0,
                    pointStyle: 'line',
                },
                {
                    label: 'Dirección (grados)',
                    data: dataDirection, ← Etiqueta que se muestra
                    borderColor: 'purple',
                    pointBorderWidth: 0,
                    pointStyle: 'line',
                },
                {
                    label: 'Velocidad de ascenso (m/s)', ← Array a representar
                    data: dataSpeedY,
                    borderColor: 'orange',
                    pointBorderWidth: 0,
                    pointStyle: 'line',
                }
            ]
        },
        options: {
            scales: { y: { beginAtZero: true } }
        }
    });
}

```

Como **último apartado** del historial del vuelo se vuelven a representar los **vientos** de igual forma que se representaron en los ajustes, aunque sin permitir añadir nuevos o borrarlos. El botón de **exportar** en la parte inferior permitirá descargar el JSON con los ajustes del vuelo.



El mapa de vientos es el mismo que en los ajustes, gracias a la clase creada, esto permite la **reutilización** de código y **escalabilidad** a futuro. El **histórico** del vuelo permite ver todos los datos útiles extraídos del vuelo lo que permite realizar **análisis** de los mismos como comparativas entre velocidad de ascenso y altitud...

SPRINT 6

Gracias a los datos almacenados en la base de datos en cada vuelo y a las características del simulador programado era posible ejecutar el mismo en modo de **repetición**. En la repetición se cargará el simulador **sin la interfaz gráfica** (exceptuando el botón de salir), de modo que el usuario no tendrá el control del aerostato, sin embargo, dispondrá de un **slider** mediante el cual podrá avanzar o retroceder por la repetición. Hay cuatro modos de velocidad diferentes.



Como cada punto de la ruta está separado de su predecesor por un segundo, la reproducción daba la sensación de ir a trozos. Para conseguir un **movimiento fluido** desarrollé un código que dividía el segundo en 30 trozos, calculando la posición en la que debería encontrarse el globo aerostático en cada momento. En el **código** (abajo) el **intervalo** que mueve el globo dependiendo de la velocidad de la reproducción.

```
/**  
 * Starts the interval  
 */  
public startInterval() {  
    if (this.interval) clearInterval(this.interval);  
    this.interval = setInterval(() => {  
        if (balloon) {  
            this.timeValue++;  
            if (this.timeValue > this.routes.length - 1) return;  
            this.labelTime.innerHTML = timeInSecondsToString(this.routes[this.timeValue].seconds, true);  
            this.inputRange.value = this.timeValue.toString();  
  
            const previousPoint = this.routes[this.timeValue - 1].coords;  
            //Move balloon in 30 steps to simulate realistic movement  
            for (let i = 0; i < 30; i++) {  
                const nextPoint = this.routes[this.timeValue].coords;  
                const x = previousPoint.x + (nextPoint.x - previousPoint.x) / 30 * i;  
                const y = previousPoint.y + (nextPoint.y - previousPoint.y) / 30 * i;  
                const z = previousPoint.z + (nextPoint.z - previousPoint.z) / 30 * i;  
                setTimeout(() => {  
                    if (this.interval) this.moveBalloon({ x, y, z });  
                }, (this.secondsBetweenPoints * i * 100 / 3) / this.speed);  
            }  
            this.inputRange.style.backgroundSize = `${(this.timeValue) * 100 / (this.routes.length - 1)}% 100%`;  
        }  
    }, this.secondsBetweenPoints * 1000 / this.speed);  
}
```

PRUEBAS

En este sprint dediqué una gran cantidad de horas a la realización de **pruebas**. Estas pruebas se basaron en ejecución de las distintas partes del código comprobando la consola del navegador. La mayoría de **errores y arreglos** que realicé fueron ajustes de estilos con CSS. Otros **problemas comunes** fueron: errores con el número de decimales que tenían algunos valores, que se mostraban mucho más grandes de lo que debían y hubo que redondearlos.

Algún error más grave tuve con temas del simulador, al realizar vuelos algunos datos del altímetro no se correspondían con la realidad y hubo que realizar ajustes en dichos cálculos.

COMENTARIOS

Durante el sexto sprint realicé una revisión de todos los **comentarios** de la aplicación. Todos los comentarios y el código desarrollado están completamente en **inglés**. Los comentarios tienen siempre una explicación de lo que hace cada método con la descripción de sus parámetros de entrada y de salida. Abajo algunos **ejemplos** de funciones de mi web.

```
/**  
 * Initializes the map  
 * @param {string} mapId the id of the HTML element where the map will be shown  
 * @param {Wind[]} windList list of winds to show  
 * @param {L.LatLngTuple} center center of the map  
 */  
constructor(mapId: string, windList: Wind[], center: L.LatLngTuple) {  
    if(!document.getElementById(mapId)) return;  
    this.center = center;  
    this.windsList = windList;  
    this.map = L.map(mapId, { center: GLOBAL.MAP_CENTER, zoom: 10 });  
    const tiles = L.tileLayer('https://s.tile.openstreetmap.org/{z}/{x}/{y}.png', { maxZoom: 15, minZoom: 10 });  
    tiles.addTo(this.map);  
    this.actColor = 0;  
    this.map.panTo(this.center);  
}  
  
/**  
 * Display the specified resource.  
 *  
 * @param int $id  
 * @return \Illuminate\Http\Response  
 */  
public function show($id)  
{  
    $flight = Flight::find($id);  
  
    return view('flight.show', compact('flight'));  
}  
  
/**  
 * Saves a point of the route  
 *  
 * @param {number} flight id of the flight  
 * @param {number} s seconds since the start of the flight  
 * @param {number} lat latitude  
 * @param {number} lon longitude  
 * @param {number} alt altitude  
 * @param {number} sp speed in m/s  
 * @param {number} spy speed in y direction in m/s  
 * @param {number} dir direction in degrees  
 * @param {number} fuel fuel in %  
 * @param {number} x x position in simulator  
 * @param {number} y y position in simulator  
 * @param {number} z z position in simulator  
 */  
public static savePoint(flight: number, s: number, lat: number, lon: number, alt: number, sp: number, spy: number, dir: number, fuel: number)  
{  
    fetch(`${environment.apiRoute}newpoint/${flight}/${s}/${lat}/${lon}/${alt}/${sp}/${spy}/${dir}/${fuel}`)  
}
```

AMPLIACIÓN Y MEJORAS

Esta aplicación está basada desde un primer momento en las necesidades de la empresa Globos Arcoiris, sin embargo, se podría ampliar al uso de otras empresas agregando **nuevos mapeados** de zonas distintas a la Rioja Alta. Para agregar nuevas zonas el proceso sería sencillo, tan solo habría que elegir una nueva área y realizar las peticiones oportunas a MapQuest y volver a repetir el proceso de separación de las imágenes y el mapa de altitud.

Otra posible mejora sería añadir **modelos diferentes** para el globo aerostático de forma que se puedan escoger colores diferentes. Yo no sería capaz de modelar en 3D sin un estudio previo, pero podrían pedirse los nuevos modelos a terceros.

Siguiendo con el mapeado, se podrían **agregar modelos** de edificios emblemáticos como iglesias o castillos, así como aerogeneradores, líneas eléctricas, casas... Dependiendo de la cantidad de modelos añadida podría bajar mucho el rendimiento del simulador, pero sin duda sería una gran aportación.

Otra posible mejora sería **añadir páginas** al front-end que permitan realizar consultas meteorológicas para obtener datos climáticos (precipitaciones, nubes, temperaturas...). Este apartado podría tener un calendario con los días de la semana...

Realizar el **pago a Windy** también sería necesario para que los datos de las peticiones fuesen reales, permitiendo simular vuelos futuros o consultar estos datos para planear el punto de despegue.

Añadir al simulador **sonidos** podría ser muy interesante para lograr una sensación de inmersión mayor.

Aunque el simulador no está pensado para ejecutarse en un móvil, sí que se podría mejorar el diseño **responsive** del front-end (el back-end sí es completamente responsive).

Finalmente **contratar un servidor** para alojar la aplicación y que cualquier usuario en línea pueda utilizarla.

CONCLUSIÓN

A modo de **cierre de este trabajo** me gustaría incidir en la **dificultad** de las tareas planteadas. Tareas como implementar el mapeado 3D o las físicas del globo aerostático han llevado una gran cantidad de horas de **investigación y desarrollo**. Aunque la parte más visual del proyecto es el simulador, la solidez del back-end y el diseño y contenidos del front-end también han supuesto retos.

A fecha de inicio de este trabajo de fin de grado carecía de los conocimientos necesarios para su desarrollo. Nunca había programado con los frameworks utilizados (Laravel, Angular, Babylon.js...) ni con las librerías de terceros (Leaflet, Chart.js...) por lo que tras la conclusión de éste he ganado mucha **experiencia** en estos campos y crecido mucho como programador.

Los contenidos del proyecto abarcan todas las **asignaturas** del grado de Desarrollo de Aplicaciones Web (Despliegue de Aplicaciones Web, Desarrollo en Entorno Servidor, Acceso a Datos, Desarrollo de Interfaces, Diseño de Interfaces Web, Desarrollo en Entorno Cliente, etc.), tocando cada uno de los campos de estudio y yendo siempre más allá, **ampliando** los **contenidos** vistos en clase.

El **resultado final** del proyecto cumple con prácticamente todas las **tareas planteadas** de forma más que satisfactoria. Aunque debido a limitaciones del framework (Babylon.js) los aterrizajes, por ejemplo, no son detectables.

La **proyección de futuro** del proyecto es, o bien, desplegarlo de forma pública o comercializarlo para empresas de aerostación o particulares. El **código** resultante es **escalable** y utiliza **tecnologías novedosas** por lo que se puede continuar con su desarrollo, ampliando sus contenidos y funcionalidades.

BIBLIOGRAFÍA

Las **páginas y documentación** usada durante la realización del proyecto:

- GitHub personal: <https://github.com/AlvaroCarbajoAlcalde>
- Globos Arcoiris: <https://www.globosarcoiris.com>
- Angular: <https://angular.io>
- Laravel: <https://laravel.com>
- Bootstrap: <https://getbootstrap.com>
- Babylon.js: <https://www.babylonjs.com>
- Leaflet: <https://leafletjs.com>
- OpenStreetMap: <https://www.openstreetmap.org>
- MapQuest: <https://developer.mapquest.com>
- OpenTopography: <https://opentopography.org>
- Stack Overflow: <https://stackoverflow.com>
- Mockflow: <https://www.mockflow.com>
- GitHub: <https://github.com>
- GitHub Desktop: <https://desktop.github.com>
- Windy: <https://www.windy.com>
- Wikipedia: <https://es.wikipedia.org>
- Chart.js: <https://www.chartjs.org>
- CodePen: <https://codepen.io>
- XAMPP: <https://www.apachefriends.org>
- Postman: <https://www.postman.com>
- Paletton: <https://paletton.com>
- Gimp 2: <https://www.gimp.org>
- Playground <https://playground.babylonjs.com>