Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Senderos de Tenerife

Tenerife Hiking

Alberto Dorta Darias - alu0100785360@ull.edu.es

La Laguna, 10 de marzo de 2022

D. **Alejandro Pérez Nava**, con N.I.F. 43821179-S profesor asociado de Universidad adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna, como tutor.

D. **Francisco Javier Rodríguez González**, con N.I.F. 43618712-V profesor asociado de Universidad adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna, como cotutor.

C E R T I F I C A (N)

Que la presente memoria titulada:

“Senderos de Tenerife”

ha sido realizada bajo su dirección por D. **Alberto Dorta Darias**, con N.I.F. 78642214-Q.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de marzo de 2022.

Agradecimientos

Me gustaría agradecer a todo el equipo docente y administrativo de la Escuela Superior de Ingeniería y Tecnología de la Universidad de La Laguna. Ha sido un largo camino y es un privilegio ver reflejados en este Trabajo de Fin de Grado todos los conocimientos adquiridos durante la carrera.

Agradecer a mi tutor Alejandro Pérez Nava por su disponibilidad, flexibilidad, sugerencias, conocimientos y su comprensión tanto de las necesidades de este proyecto, como de las mías propias.

Por último, agradecer a mi familia, que siempre ha apostado e invertido en mi educación. Sin duda ellos han jugado el rol más importante, fomentando mis estudios, apoyando mis objetivos y formándome no solo profesionalmente sino también personalmente. De ellos también nace la idea detrás de este Trabajo; Senderos de Tenerife. Desde muy temprana edad, hemos estado en contacto con la naturaleza y gracias a ello he visto la oportunidad de trasladar un fragmento de esas experiencias a un entorno informático.

Licencia

Resumen

Todos sabemos que Tenerife es una isla de contrastes donde el turismo y los residentes pueden disfrutar de paisajes muy variados. El senderismo y montañismo son ejemplos de las actividades que permiten apreciar esta diversidad, donde se pueden encontrar desde bosques húmedos hasta zonas completamente áridas / volcánicas. Sin embargo, no existe un espacio exclusivo que permita promocionar, buscar o consultar acerca de ellos.

El objetivo de este proyecto es crear una plataforma que de visibilidad e información sobre los muchos senderos que hay en Tenerife, donde cada sendero incluirá un mapa indicando su ruta, la dificultad, ubicación, duración.. etc. La plataforma también tendrá funcionalidades de tipo “red social” donde los usuarios podrán registrarse y valorar los senderos. Además, de la opción de reservar rutas guiadas (*funcionalidad no implementada completamente*) para un posible uso comercial.

Esta plataforma será un proyecto web Full Stack. Hemos elegido crear una API ya que nos permitirá compartir los datos de la aplicación con los clientes y otros agentes externos. La podremos consumir independientemente del lenguaje que utilicemos para desarrollar el proyecto y además, hace que la interfaz del usuario y el almacenamiento de datos funcionen de manera independiente.

Palabras clave: Sendero, API, NodeJS, Mapa, MongoDB, JWT, Cookie, Postman, Express.

Abstract

We all know that Tenerife is an island of contrasts where tourists and residents can enjoy very varied landscapes. Hiking and mountaineering are examples of activities that allow us to appreciate this diversity, where we can find everything from wild forests to completely arid / volcanic areas. However, there is not an exclusive space that allows you to promote, search or consult about them.

The purpose of this project is to create a platform that provides visibility and information about the many hiking routes in Tenerife, where each of them will have a map indicating its path, its difficulty, location, duration, etc. The platform will also have "social network" features where users can register and rate the trails. In addition, there will be an option to book guided routes (*functionality not fully implemented*) for a potential commercial use.

This platform will be a Full Stack web project. We have chosen to create an API as it will allow us to share the application data with clients and other external agents. We can consume it regardless of the language we use to develop the project and also make the user interface and data storage to work independently.

Keywords: Hike, API, Node JS, Map, MongoDB, JWT, Cookie, Postman, Express.

Índice general

Índice de figuras

1. Introducción

1.1. Motivación

Actualmente, el sector turístico en Tenerife se encuentra en niveles históricos de baja actividad debido a la pandemia COVID-19. La primera industria económica de Canarias está sufriendo un duro golpe, y esto supone un revulsivo para avanzar en la sostenibilidad y la digitalización.

La principal finalidad de la plataforma *Senderos de Tenerife* es dar visibilidad a experiencias de montañismo y caminatas que tienen lugar en la isla, ya que no se ven afectadas por ningún tipo de restricción sanitaria. Este tipo de actividades pueden posicionarse como las más atractivas para los recientes visitantes, que están siendo claramente limitados.

Además, no existe un espacio virtual que funcione como un archivo y registre rutas y datos sustanciales sobre los senderos de Tenerife, lo que puede convertirse en una herramienta muy útil y un gran recurso para obtener información.

1.2. Objetivos

El principal objetivo de este proyecto es desarrollar una API y construir una aplicación web a partir de los datos que almacenemos / generemos en ella. Como ya he citado, almacenaremos caminatas de Tenerife donde se implementará un mapa indicando su ruta, su dificultad, ubicación, duración… etc. También, podremos hacer búsquedas complejas de senderos por nombre o ubicación.

Adicionalmente, habrán funciones clásicas de usuario, como registrar una cuenta e iniciar sesión, y existirá la posibilidad de que un usuario registrado realice reviews y valoraciones en los distintos senderos.

Lo más interesante de este trabajo, es que todas y cada una de las prestaciones de la aplicación funcionarán con distintos tipos de consultas a la API.

1.3. Mockup y modelo de base de datos

Primer boceto del diseño:

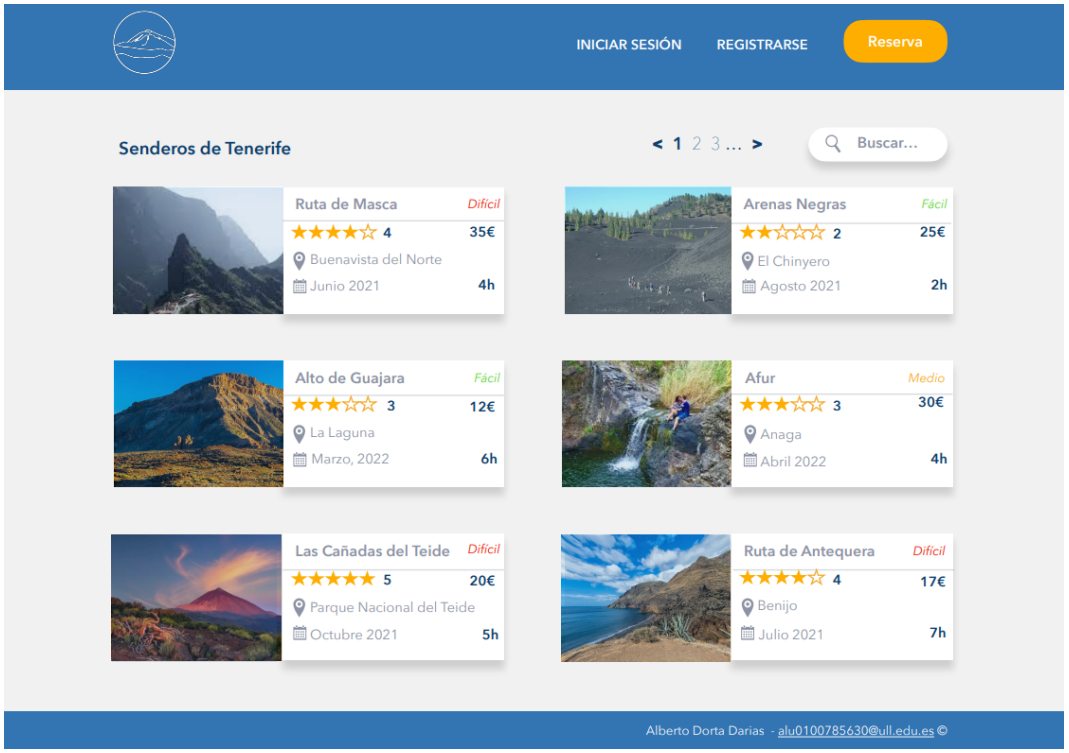

Figura 1.3.1: Overview de los senderos

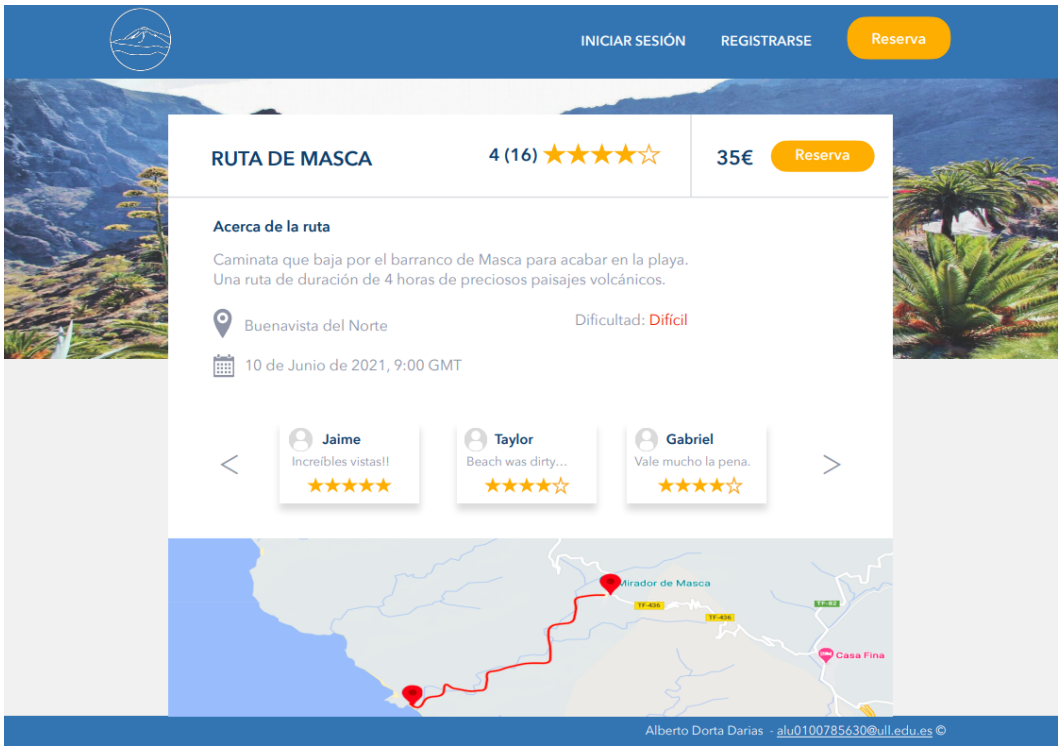

Figura 1.3.2: Sendero Individual

Modelo de la base de datos:

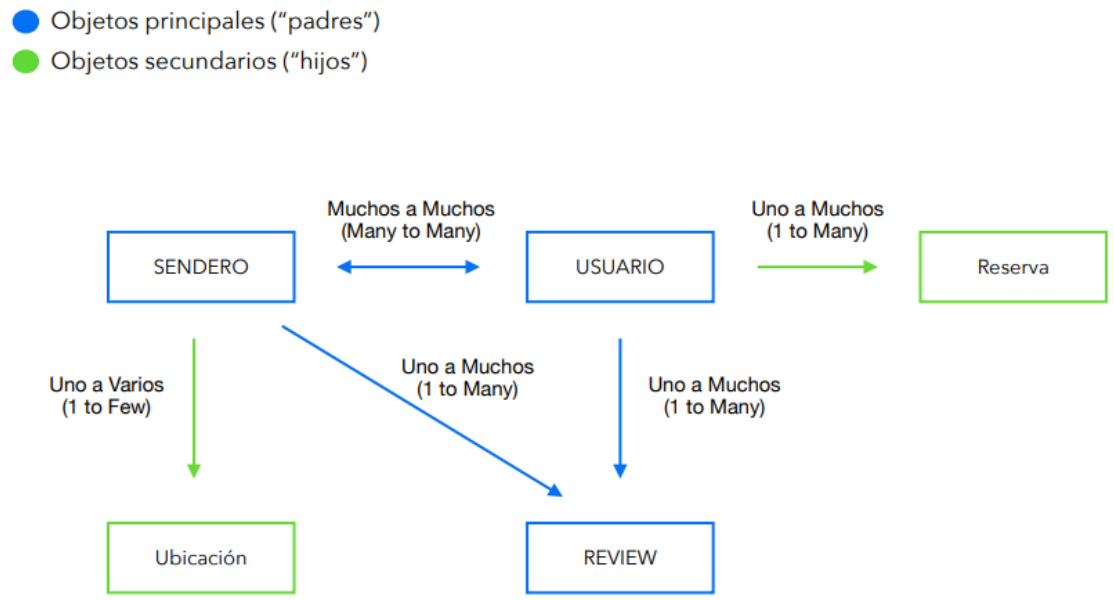

Figura 1.3.3: Modelo base de datos

1.4. Fases del proyecto

Fase 1 : Configuración del entorno de desarrollo

- Creación del entorno de desarrollo
- Instalación de herramientas software (Postman, Compass… etc)
- Creación una base de datos con el host en Atlas (se almacena en Cloud)
- Script de Webpack para compilar ficheros y desarrollar el front-end

Fase 2 : Creación e Interacción con la API

- Definir el schema de Senderos, Usuarios y Reviews en MongoDB - Mongoose.
- Crear rutas de la API
- Administrar diferentes peticiones (GET, POST, PATCH, UPDATE, DELETE)
- Filtros
- Administración de errores
- Autenticación, Autorización y Seguridad
    - Roles y permisos
    - JWT
    - Proteger rutas para usuarios no identificados

Fase 3 : Implementaciones avanzadas y renderizar resultados

- Modelado de la base de datos para crear reviews
- Mejorar el rendimiento
- Renderizar el resultado (front-end)
- Añadir mapas que señalen la ruta (Mapbox)

1.5. Estructura de la memoria

A lo largo de este documento encontraremos una descripción completa de las tareas que se han llevado a cabo en este proyecto. Se muestra la perspectiva de los desafíos lógicos y la exposición de las soluciones desde un punto de vista técnico. Podemos dividir la memoria en los siguientes apartados, que serán desarrollados posteriormente:

- Herramientas de software y configuración del proyecto: Comenzaremos detallando las tecnologías necesarias para la elaboración de esta idea, así como el modelo y el entorno de desarrollo.

- Estudio de las API y el tratamiento de datos: Analizamos el funcionamiento y los beneficios de implementar una API en nuestro proyecto, así como la estructura de datos y las relaciones entre ellos. Describimos la arquitectura cliente-servidor y el protocolo HTTP.

- Funcionalidades específicas del proyecto: Se presenta el funcionamiento detallado de las características técnicas de la aplicación web, como por ejemplo: operaciones CRUD con la API, middlewares de Express, filtrado de datos, registro e inicio de sesión, modelado de datos avanzado… etc.

- Conclusiones: Se expondrán las conclusiones que se han obtenido durante el proceso de realización del proyecto, donde se destacan potenciales puntos de mejora, funcionalidades no implementadas y una valoración general en la que analizamos las líneas futuras.

2. Software y herramientas utilizadas

- **Visual Studio Code**

Figura 2.1: Logo Visual Studio Code

El ampliamente conocido editor de Microsoft ha sido el elegido para desarrollar el código fuente de nuestra aplicación. A parte de una muy buena UI/UX, este ofrece un gran soporte para tareas de depuración, control de versiones Git integrado y un market de extensiones para completar la experiencia de desarrollo.

- **NodeJS**

Figura 2.2: Logo NodeJS

NodeJS debido a su robustez y la posibilidad de ejecutar Javascript fuera del navegador y scripts y código shell, para importar ciertos archivos u otras tareas específicas. Además, eligiendo NodeJS como tecnología para el servidor, usaremos únicamente un lenguaje de programación para todo el proyecto (*JavaScript*).

  - **Express**

Nos ayudaremos del framework Express ya que facilita la escritura del código y nos permite gestionar de forma separada las peticiones por medio de diferentes direcciones URL. Adicionalmente, podremos analizar el cuerpo de las peticiones HTTP y las cookies.

- **MongoDB**

Figura 2.3: Logo MongoDB

Para almacenar los datos de la aplicación usaremos MongoDB, que es un sistema de base de datos no relacional que permite operaciones complejas y otras funciones como “populating” y “virtual populating” que son convenientes para el proyecto.

- **Atlas**

Atlas es un servicio Cloud de MongoDB que nos permite trabajar con nuestra base de datos desde cualquier parte del mundo, siempre estará en Cloud y además, no habrá problemas al exportar en caso de mover nuestro proyecto a otro servidor.

- **Compass**

Permite visualizar e interactuar el contenido de la base de datos mediante una interfaz de usuario.

- **Mongoose**

Podríamos definir Mongoose como un driver para que NodeJS pueda controlar la base de datos. Mongoose nos permite escribir consultas a la base de datos de MongoDB con funcionalidades añadidas como validaciones y middlewares. Además, podremos fijar un modelo de datos para cada ítem de nuestra aplicación (*sendero, usuario, review*).

- **Postman**

Figura 2.4: Logo Postman

Sin duda una herramienta indispensable para el desarrollo de este proyecto. Postman permite realizar cualquier tipo de petición HTTP a una API, donde también podemos configurar headers, cuerpos en formato JSON para peticiones POST, guardas rutas, etc.

- **Paquetes NPM**

NPM es el gestor de paquetes de JavaScript de NodeJS. La aplicación hace uso de las siguientes dependencias:

- cookie-parser: Nos permite interactuar con las cookies
- cors: Para realizar peticiones a nuestra API desde otro dominio.
- dotenv: Para poder usar nuestras variables de entorno.
- formidable: Permite acceder a los ficheros enviados desde un formulario.
- jsonwebtoken: Software que usaremos para las funciones de autenticación.
- morgan: Muestra detalles de cada petición en la terminal.
- nodemon: Reinicia el servidor cada vez que se produce un cambio en el código.
- slugify: Convierte strings en “formato URL”.

- **Git & GitHub**

Figura 2.5: Logo Git

Como no podía ser de otra manera, hemos utilizado Git como sistema de control de versiones y así poder registrar los cambios a medida que se ha progresado en el proyecto, y GitHub como plataforma online para trabajar con los recursos del mismo.

Aquí se encuentran los enlaces a los repositorios:

- https://github.com/alu0100785630/TFG-Senderos-Tenerife
- https://github.com/alu0100785630/TFG-Senderdos-Tenerife-FrontEnd

- **Webpack**

Figura 2.5: Logo Webpack

Para trabajar con el código JavaScript, las plantillas Pug y los estilos CSS de forma óptima, se usará Webpack. Este permite agrupar el código en diferentes módulos lo que facilita la depuración, el mantenimiento y la escalabilidad.

3. Configuración del entorno de trabajo

En este trabajo se va a diseñar e implementar una web *full stack*, lo que requiere producir tanto la parte visual (front-end) como la parte lógica (back-end).

Como hemos mencionado anteriormente, se ha elegido la pila Node - Express - Mongo por su capacidad de separar cliente y servidor, permitiendo que el desarrollo de la estructura del back-end sea totalmente independiente al desarrollo del front-end.

3.1. Front - End

Con el fin de crear la estructura del front-end, se han configurado diferentes scripts con la herramienta de compilación Webpack. Ésta permite agrupar y empaquetar diferentes archivos JavaScript para su uso en un navegador. También lo usaremos como automatizador de tareas, compilador de los ficheros CSS usando el preprocesador Sass y como compresor de imágenes.

En la configuración del proyecto front-end, existen dos modos de desarrollo: **Development y Production**. Después de importar los scripts deseados y ejecutar el comando `npm` adecuado, se generará una carpeta de distribución con los archivos compilados en el directorio raíz:

- Development: `$ npm start` ⇒ `./dist`
- Production: `$ npm run build` ⇒ `./prod`

Ambos modos inyectan los scripts y las hojas de estilos en el HTML, cargan JQuery de forma automática y eliminan los elementos que bloquean el renderizado. Estas son sus principales diferencias:

Development

- Observa los cambios en el ćodigo y compila automáticamente.
- Copia las imágenes.
- El CSS y JavaScript se incluyen juntos en un fichero JavaScript. El CSS se carga con una etiqueta `<style>` en el DOM.
- Muestra el HTML indentado.
- Compila todo el código de CSS de frameworks externos.

Production

- Solo compila cuando se ejecuta el comando `npm`.
- Comprime las imágenes.
- Compila y minifica el CSS y JavaScript en diferentes ficheros.
- Minifica el HTML.
- Elimina el CSS no usado. Por ejemplo, solo usamos un alrededor de un 10% del código de Bootstrap, el resto será eliminado.

3.2. Back - End

El back-end se basa en la arquitectura **MVC, Model - Controller - View**. Esto significa que el código estará estructurado de manera que seremos capaces de separar la lógica de la aplicación (*app logic*) de la lógica del negocio (*business logic*).

Los elementos de control interactúan con el modelo, por ejemplo, creando un nuevo documento en la base de datos. Después de haber interactuado con el modelo, podríamos estar preparados para mandar una respuesta al cliente, pero necesitamos una interfaz que renderizar, así que interactuamos con la parte visual (*View*), y luego enviamos la respuesta al cliente.

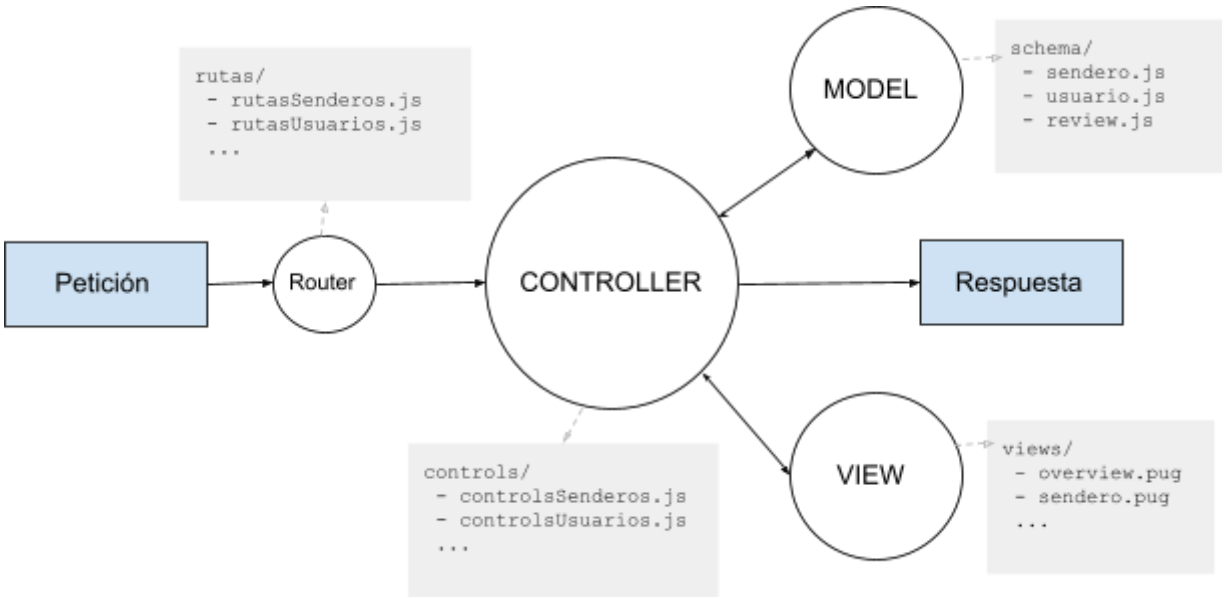

Figura 3.2: Arquitectura MVC

4. Application Programming Interface. API

Es un fragmento de software que puede ser usado por otro fragmento de software para permitir que distintas aplicaciones puedan comunicarse entre sí. Una API facilita la programación ya que se implementan procedimientos y funciones que solo necesitarán consumirse y no desarrollarse desde cero.

El funcionamiento web se basa en que un usuario realice una petición a un servidor, este servidor web accede a una base de datos y devuelve los datos solicitados al cliente como una respuesta. Este mismo ciclo de petición/respuesta se utiliza cuando se accede a una API. La principal diferencia entre una "petición a una API" y una "petición a una página web" es el tipo de datos se proporcionan en la respuesta.

Las API responden con datos en un formato sin formato, es decir, que no está diseñado para que un navegador los convierta en una experiencia de usuario. JSON y XML son los formatos más comunes utilizados para estos datos sin procesar, y ambos son formatos de texto flexibles para almacenar datos. La gran mayoría de los lenguajes de programación tienen bibliotecas que pueden analizar JSON y XML, lo que los convierte en las opciones óptimas para representar los datos de la API ya que serán fácilmente manipulables para otros desarrolladores.

4.1. Beneficios de una API

- Mejora la portabilidad a otras plataformas, Android, iOS, software de escritorio, etc.
- Facilitar y agiliza la programación a otros desarrolladores. Los datos se podrán consumir independientemente del lenguaje de programación y no necesitarán instalar ningún software adicional o librería externa, ya que funciona con protocolos HTTP.
- Automatización. Esto puede ser tan simple como hacer que una actualización de contenido se propague a varias secciones de una plataforma (o varias plataformas) a la vez.

Para construir una API de valor debemos seguir ciertas reglas. Seguiremos los principios de la **arquitectura REST**.

5. La arquitectura REST

La arquitectura REST o Representational State Transfer es básicamente un método para construir API 's de una forma lógica y hacer su uso más fácil de consumir.

Sigue los siguientes principios:

1. Todos los datos que queramos compartir en la API deberán estar divididos en recursos lógicos, siendo estos objetos o representaciones de elementos que tienen datos asociados. Cualquier información que se pueda nombrar puede ser un recurso. Normalmente se nombran en plural. En el caso de nuestra aplicación, serán:

   `senderos` `usuarios` `reviews`

2. Todos los datos tienen que estar disponibles usando URLs estructuradas donde el cliente pueda realizar las peticiones.

   `https://senderos-tfg.com/api/`**`crearSendero`**

3. Los endpoints solo deben contener los recursos y no las acciones que se van a realizar a estos. El ejemplo anterior (*punto nº 2*) es un caso de uso incorrecto. Con Express podemos definir el método de acceso y el resultado que queremos mostrar.

   `GET` `https://senderos-tfg.com/api/`**`senderos`** ⇒ Lista todos los senderos

   `POST` `https://senderos-tfg.com/api/`**`senderos`** ⇒ Crea un nuevo sendero

4. Los datos se deben enviar en formato JSON.

5. Una RESTful API no debe tener estado. El estado se refiere a un fragmento de los datos de la aplicación que puede cambiar a lo largo del tiempo. Todas las peticiones del cliente deben contener la información necesaria para que puedan ser procesadas en el servidor. El servidor no debe recordar la petición anterior para que pueda ejecutar la siguiente petición.

   `GET` `https://senderos-tfg.com/api/`**`senderos/nextPage`** ❌

   `GET` `https://senderos-tfg.com/api/`**`senderos/page/4`** ✔

6. Modelo de almacenamiento de datos

Anteriormente citado, usaremos MongoDB como sistema de base de datos. Es una base de datos llamada NoSQL, ésta contiene **colecciones**, las cuales se podrían identificar como tablas en las bases de datos relacionales, donde cada colección contiene diferentes estructuras de datos, llamadas **documentos** (identificables como filas en las bases de datos relacionales).

MongoDB almacena los datos en un fichero que sigue una estructura de pares, campo-valor. Se denomina formato **BSON**, ya que tiene una especificación muy similar a JSON pero los datos estarán tipados, es decir, serán valores `int`, `char`, `Array`… etc. Cada documento BSON tiene un máximo de 16MB y un identificador de documento único.

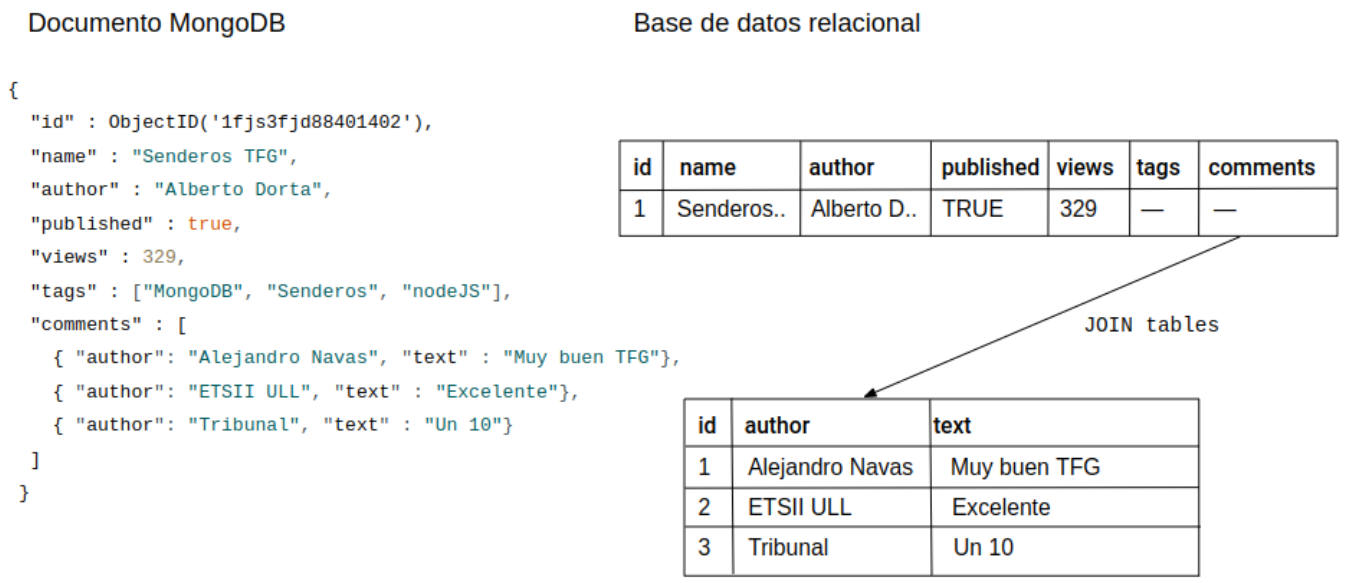

Figura 6.1: MongoDB vs Base de datos relacional

Para cada uno de los recursos de este proyecto hemos diseñado un modelo de datos definiendo un **schema** de Mongoose, donde describimos su estructura, tipo, valores por defecto, validaciones… etc.

6.1. Senderos

- ID: identificador del sendero.
- Nombre: nombre del sendero.
- Imagen: imagen que se usará para el "banner" del sendero.
- Descripción: descripción del sendero.
- Duración: duración del sendero.
- Dificultad: Fácil - Medio - Difícil.
- Precio: precio de reserva individual.
- Media de valoraciones: la media del total de las valoraciones de los usuarios.
- Número de valoraciones: Es necesario almacenar el número de valoraciones del sendero en el schema.
- Fecha: día y hora programada para la ruta.
- Ubicación principal: Municipio al que pertenece el sendero
- Ruta del sendero: Coordenadas - Descripción

```
{
   "name": "Chinamada",
   "duration": 3,
   "difficulty": "medio",
   "price": 10,
   "description": "<p>Chinamada es un antiguo caserío …",
   "image": "chinamada.png",
   "startDate": "2022-05-03,09:00",
   "mainLocation" : "Anaga",
   "routeLocations": [
     {
       "description": "Mirador Aguaide",
       "type": "Point",
       "coordenadas": [28.565309805276424, -16.294732860821522]
     },
     {
       "description": "Mirador hacia el Roque de los Pinos",
       "type": "Point",
       "coordenadas": [28.562840991450457, -16.293381027436315]
     },
     ...
   ]
 }
```

6.2. Usuarios

- ID: identificador de usuario.
- Nombre: nombre de usuario.
- Email: email para notificaciones y funcionalidad sign in.
- Rol: rol del usuario (administrador, guía o usuario regular).
- Contraseña: contraseña de usuario con cifrado Hash.

```
{
   "_id":{
      "$oid":"61edb40eff2751dd7052a606"
   },
   "role":"usuario",
   "name":"Alberto Dorta",
   "email":"aldo@gmail.com",
   "password":"1234",
   "__v":0
}
```


6.3. Reviews

- ID: identificador de review.
- Review: comentario acerca del sendero.
- Valoración: valoración del usuario de 1 a 5.
- Usuario: usuario que realiza la valoración.
- Sendero: sendero al que el usuario realiza la valoración.

```
{
   "_id":{ "$oid":"61edb4ccff2751dd7052a608" },
   "publishedTime":{ "$date":"2022-01-23T19:59:16.686Z" },
   "usuario":{ "$oid":"61edb40eff2751dd7052a606" },
   "sendero":{ "$oid":"61ea9bdfcc059e291dd472c6" },
   "review":"Sorprendente las formaciones rocosas …",
   "rating":4,
   "__v":0
}
```

7. Refactorización del código

Puesto que seguimos una estructura MVC para nuestro código, hemos separado los controladores de cada petición y las rutas es distintos ficheros y directorios:

```
/rutas
    ├── rutasReviews.js
    ├── rutasSenderos.js
    ├── rutasUsuarios.js
    └── rutasVisual.js
```

```
/controls
    ├── controlsReviews.js
    ├── controlsSenderos.js
    ├── controlsUsuarios.js
    └── controlsViews.js
```

Cada controlador incluirá el schema del recurso a manipular, situados en el directorio `/schema`, y cada ruta incluirá las funciones de cada controlador para administrar las respuestas según el tipo de petición:

Controlador

controlsSenderos.js

```
const Sendero = require('./../schema/sendero');
...
exports.allSenderos = async(req, res) => {
      ...
};
```

Ruta

rutasSenderos.js

```
const controlsSenderos = require('./../controls/controlsSenderos');
...
router
  .route('/')
  .get(controlsSenderos.allSenderos)
  .post(controlsSenderos.createSendero)
```

A su vez, incluiremos todas las rutas en el fichero `app.js`, que administra las llamadas a las funciones (*routers*) de Express:

app.js

```
const rutasSenderos = require('./rutas/rutasSenderos');
...
app.use('/api/senderos', rutasSenderos);
```

8. Middleware y el ciclo de respuesta de una petición

Cuando un cliente se conecta a un servidor, se crea un objeto *request* y un objeto *response*, cuyos datos se usarán para enviar la respuesta adecuada. Para procesar estos datos, en Express se usa **Middlewares**.

Se les llama middlewares porque son funciones que se ejecutan entre la recepción de la petición y la ejecución de la respuesta. Los middlewares pueden manipular los objetos *request* y *response,* y/o ejecutar cualquier otro tipo de código que necesite la aplicación.

Existe una pila de ejecución de middlewares que funciona de forma secuencial, es decir, lo que se declara primero en el código también será ejecutado primero. Luego, para acceder al siguiente middleware de la pila de ejecución, existirán llamadas a la función `next()`. Esta serie de pasos se lleva a cabo continuamente hasta llegar a la respuesta final. Ejemplo:

1. Petición ⇒ Objeto request y Objeto response
2. Middleware : analizar el body y `next()`
3. Middleware : hacer una función de login y `next()`
4. Middleware : configurar los headers y `next()`
5. Middleware : enviar la respuesta (`res.send() o res.status().json()`)
6. Respuesta

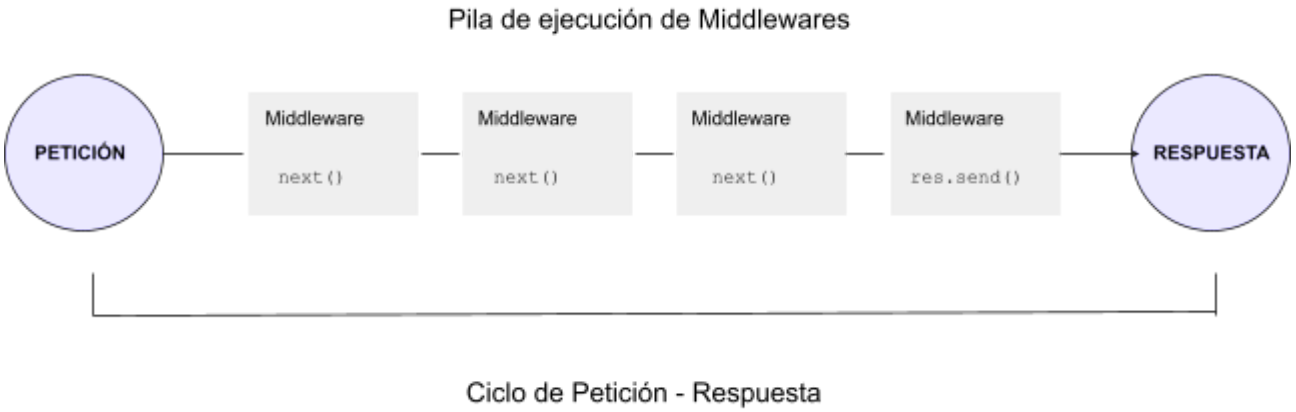

Figura 8.1: Middleware y el ciclo de respuesta de una petición

Para hacer uso de los middleware, necesitamos el método `use` de Express:

```
app.use(express.json());
```

9. Operaciones CRUD en el proyecto

CRUD es un acrónimo que hace referencia a las cuatro funciones que se consideran necesarias para implementar una aplicación que almacena datos de forma consistente: Create (crear), Read (leer), Update (actualizar) y Delete (eliminar).

Al realizar una consulta a la base de datos se genera una promesa de JavaScript. Es por ello que las funciones de nuestros controladores trabajan de forma asíncrona usando la sintaxis `async-await` para consumir estas promesas. Además, usaremos bloques `try-catch` para atrapar los errores cuando la promesa es rechazada y así evitar pequeños *delays* en la respuesta.

9.1. READ. Peticiones GET

Las operaciones de lectura se realizan mediante peticiones GET. Para ello usamos consultas a la base de datos con las funciones de mongoose **`find()`**, que devuelve todas las instancias de una colección, y **`findById()`**, que devuelve un único elemento de la base de datos que coincida con el ID que se pasa como parámetro. Ejemplo:

```
exports.allSenderos = async(req, res, next) => {
  try {
    const senderos = await Sendero.find(); //Obtener todos los senderos

    res.status(200).json({ //Enviar la respuesta como un JSON
      status: 'success',
      data: { senderos }
    });
  } catch (err) {
    res.status(404).json({ //Enviar el error como un JSON
      status: 'fail',
      message: err
    });
    return next(err);
  }
};
```

9.1.1. Respondiendo a parámetros en la URL

En Express podemos crear variables en la URL de la siguiente manera:

```
app.get('/api/senderos/:id', (req, res) => {
   console.log(req.params);
});
```

Esto crea una variable "*id*" que se almacena en el objeto "*params*" de la petición. De esta manera, podemos enviar el ID de un sendero en la URL y hacer una consulta a la base de datos con la función **`findById()`**, que nos devolverá el elemento que coincida.

```
router
  .route('/:id')
  .get(controlsSenderos.singleSendero)
  ...
```

Como ya hemos citado, en Express podemos manipular todas las solicitudes del cliente accediendo a los objetos *request* y *response*, definidos en las variables `req` y `res` de cada una de nuestras funciones asíncronas:

```
exports.singleSendero = async(req, res, next) => {
   try {
   const sendero = await Sendero.findById(req.params.id);

    res.status(200).json({
      status: 'success',
      data: {
        sendero
      }
    });
   } catch(err) {
      return next(err);
   }
});
```

9.2. CREATE. Peticiones POST

Para crear cualquier recurso de nuestra aplicación, usaremos peticiones POST a la vez que enviamos los datos necesarios en el cuerpo de la petición. Mongoose añade nuevos documentos en la base de datos con el método **`create()`**.

```
exports.createSendero = async(req, res, next) => {
  try {
    const newSendero = await Sendero.create(req.body);

    res.status(201).json({
      status: 'success',
      data: { sendero: newSendero }
    });
  } catch (err) {
      ...
    return next(err);
  }
};
```

Nótese que ahora enviamos el estado de éxito 201, que es el código HTTP que indica que el recurso ha sido creado satisfactoriamente.

En Postman no solo podemos hacer todo tipo de peticiones HTTP, sino que también podemos enviar datos en formato JSON en el cuerpo de la petición. Esto nos ha permitido realizar las comprobaciones pertinentes para asegurarnos de que todo funciona como se espera.

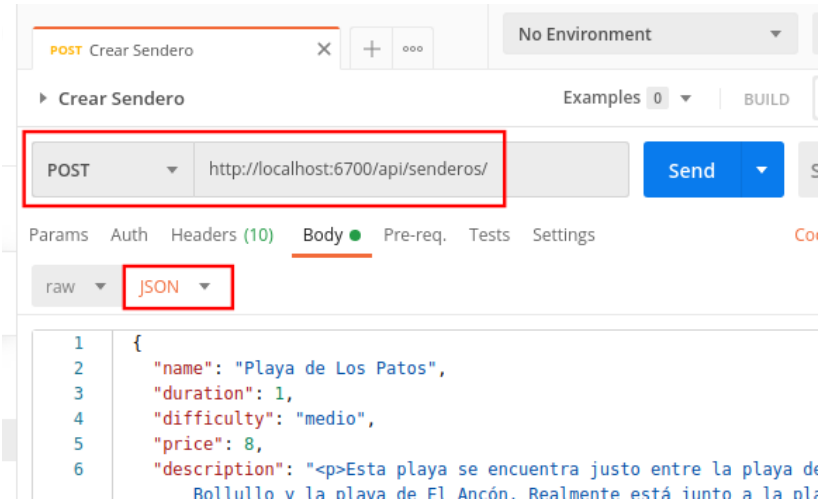

Figura 9.1: Petición POST en Postman

9.3. UPDATE. Peticiones PATCH

Para actualizar (modificar) ciertos valores de los campos de nuestros documentos realizaremos peticiones tipo PATCH. Usaremos el método **`findByIdAndUpdate()`** de Mongoose.

Una vez más, necesitamos identificar el recurso que se va actualizar, por lo tanto también enviaremos el ID del sendero como un parámetro en la URL, que enviaremos junto con el cuerpo de la petición al método `findByIdAndUpdate()`.

```
router
  .route('/:id')
  .patch(controlsSenderos.updateSendero)
  ...
```

```
exports.updateSendero = async(req, res, next) => {
  try {
    const sendero = await Sendero.findByIdAndUpdate(req.params.id, req.body, {
      runValidators: true    //Para que se hagan las validaciones del schema
    });
    res.status(200).json({
      status: 'success',
      data: { sendero }
    });
  } catch (err) {
    return next(err);
  }
};
```

En el cuerpo de la petición enviaremos (en formato JSON)los datos que nos interese actualizar:

```
{
    "name" : "Nuevo Sendero",
    "price" : "14"
}
```

9.4. DELETE. Peticiones DELETE

Seguiremos el mismo procedimiento para eliminar un recurso de nuestra aplicación. Utilizaremos una petición HTTP tipo DELETE, identificamos el recurso enviando el ID como parámetro en la URL, y usaremos el método **`findByIdAndDelete()`** de Mongoose para llevar a cabo la función de borrado en la base de datos.

```
router
  .route('/:id')
  .delete(controlsSenderos.deleteSendero)
  ...
```

```
exports.deleteSendero = async(req, res, next) => {
  try {
    await Sendero.findByIdAndDelete(req.params.id);

    res.status(204).json({
      status: 'success',
      data: 'El sendero ha sido eliminado correctamente'
    });
  } catch (err) {
    err.message = `No se encuentra un sendero con ese ID.`;
    err.status = 'fail';
    err.statusCode = 404;
    return next(err);
  }
};
```

En esta función no almacenamos la respuesta del *await* en una variable, ya que el recurso no existirá si la petición es exitosa. Simplemente enviamos un mensaje.

Además, enviamos el código HTTP 204, que significa que la respuesta es exitosa pero no tiene contenido.

10. Operaciones complejas con la API

Una característica común en las API es tener consultas que permitan a los usuarios filtrar datos. Estas consultas se usarán mediante palabras clave en la URL de la petición. Por ejemplo:

```
GET https://senderos-de-tenerife.com/api/senderos?sort=price
```

Solo por la sintaxis utilizada, entendemos que una solicitud a esta URL devolverá todos los senderos ordenados por precio.

Para incorporar esta funcionalidad a nuestra API, hemos creado una clase llamada *APIOperations*. Su constructor recibirá dos parámetros, la consulta a la base de datos (`query`) y la cadena de la URL (`cadenaQuery`), a la que podemos acceder mediante el objeto *request*, más concretamente en `req.query`.

APIOperations tendrá distintos métodos que realizarán diferentes consultas a la base de datos según las palabras clave en la URL:

- **`ordenar()`** : Si existe la palabra "*sort*" en la URL, selecciona los campos manipulando el string de entrada y ejecuta la consulta *sort* en la base de datos:

```
const orderBy = this.cadenaQuery.sort.split(',').join(' ');
this.query.sort(orderBy);
```

```
/api/senderos?sort=price,gradeAverage
```

Si incluyo "-" antes del campo, los clasifica en orden inverso:

```
/api/senderos?sort=-price
```

- **`seleccionarCampos()`** : Devolverá los resultados sólo los campos que se indiquen. Si existe la palabra clave "*fields*" en la URL, selecciona los campos manipulando el string de entrada y ejecuta la consulta *select* en la base de datos:

```
const fields = this.cadenaQuery.sort.split(',').join(' ');
this.query.select(fields)
```

```
/api/senderos?fields=name,price
```

- **`filtrar()`** : Es el método más complejo de la clase ya que permitirá dos tipos distintos de filtrado.

    - El primero lo usaremos como motor de búsqueda de nuestra aplicación, es decir, el término o palabra introducida filtrará senderos por nombre o por ubicación. Para ello se comprobará que exista la palabra clave "*regex*" en la URL y creará una expresión regular para evitar excluir resultados por uso de mayúsculas. Seguidamente, realizará una consulta a la base de datos con una condición "*or*", que encontrará las coincidencias según nombre o ubicación:

```
let queryRegEx = new RegExp(this.cadenaQuery.regex, 'i' );
this.query = this.query.find({
  $or: [
     {'name' : queryRegEx},
     {'mainLocation': queryRegEx}
   ]
});
```

`/api/senderos?regex=anaga`

    - El segundo lo usaremos para filtrar por valores, con la opción de comparar resultados. Si no existe la palabra clave "regex" en la URL, entonces reemplaza los operadores relacionales (*gte, gt, lte, le*) en una expresión regular para que coincidan con el formato de MongoDB:

```
jsStr = jsStr.replace(/\b(gte|gt|lte|lt)\b/g, match => `$${match}`);
this.query = this.query.find(JSON.parse(jsonStr));
```

`/api/senderos?price[lt]=20`

Una vez construída la clase, podemos ejecutar las operaciones mencionadas creando un objeto y encadenando las funciones. Esto es posible porque todos los métodos de la clase retornan *this* (`return this;`), de esta manera siempre se puede acceder al objeto actual.

```
const operations =
new APIOperations(Sendero.find(), req.query).filtrar().ordenar().selCampos();
const senderos = await operations.query;
```

11. Controlador de errores global

Cuando un usuario intenta acceder a una ruta que no existe, Express envía como respuesta un código HTML por defecto:

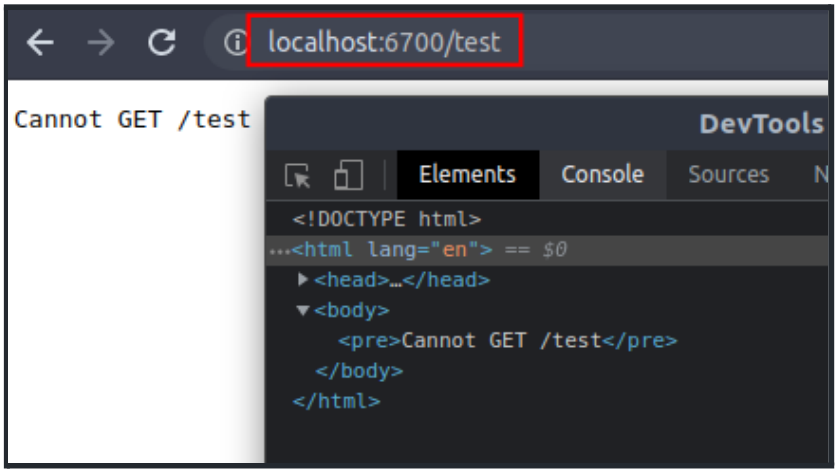

Figura 11.1: Error de Express

Sin embargo, si probamos con una ruta como: `/api/senderos/error-test`, Express no enviará el código HTML porque hemos definido que puede aceptar un parámetro (`:id`) en ese elemento. Como los middleware se ejecutan de forma secuencial, podemos crear un nuevo middleware para administrar los errores que se produzcan en los controladores previos:

```
app.use('/api/senderos', rutasSenderos);
app.use('/api/usuarios', rutasUsuarios);
app.use('/api/reviews', rutasReviews);

app.all('*', (req, res, next) => {
  //req.originalUrl muestra la URL de la petición
  const err = new Error(`No se encuentra la ruta: ${req.originalUrl}`);
  err.status = 'fail';
  err.statusCode = 404;
  next(err);
});
```

Al pasar el error a *next*, el siguiente middleware será de error sin importar el orden en la pila de middlewares. Para ello, creamos un **controlador de errores global**.

Declararemos la función en un fichero independiente que será posteriormente importado en *app.js*. Si especificamos cuatro parámetros en la función, Express identifica un middleware para manejar errores. Desde ahí leemos el estado y enviamos la respuesta en formato JSON:

```
module.exports = (err, req, res, next) => {
  //El status code será igual al status code (si existe) o 500
  err.statusCode = err.statusCode || 500;
  err.status = err.status || 'error';

  res.status(err.statusCode).json({
    status: err.status,
    message: err.message
  });
};
```

Una vez implementado el controlador de errores global, no necesitamos enviar los errores dentro del bloque catch de cada función asíncrona. Simplemente retornaremos una llamada al siguiente middleware con el error encontrado:

```
try {
   ...
}
catch (err) {
    res.status(404).json({
      status: 'fail',          ❌
      message: err
});
```

```
try {
   ...
}
catch (err) {
    return next(err);          ✔
});
```

12. Autenticación y Autorización

Durante el proceso de autenticación, verificamos quién es el usuario, es decir, si el usuario se ha registrado en nuestra plataforma, con el fin de que puedan interactuar con los recursos protegidos. Con las funcionalidades de autorización, verificamos si el usuario tiene acceso a diferentes ficheros, recursos o datos específicos de nuestra aplicación.

Crearemos un controlador específico que separa las funciones de autenticación de las operaciones CRUD del recurso “Usuario”.

```
/controls
    ├── controlsAutenticacion.js
    └── controlsUsuarios.js
```

12.1. JWT - JSON Web Tokens

JWT es una solución estable para la autenticación. No se necesita guardar ninguna sesión en el servidor, lo que es perfecto para nuestra API.

¿Cómo inicia sesión un usuario en nuestro sistema?

1. El usuario realiza una petición POST con el email y la contraseña.
2. La aplicación comprueba si el usuario existe y la contraseña es correcta. Si es así, se crea un token (*JWT*) único para ese usuario usando un string secreto en el servidor.
3. El servidor manda de vuelta el token (*JWT*), que se almacenará en una cookie o en el *localStorage* del navegador del cliente.
4. El servidor realmente no sabe qué usuarios han hecho Log in, pero el usuario sí, ya que tiene acceso a los elementos protegidos de la web.

JSON Web Tokens (JWT)

Un JWT es un string codificado que se compone de tres partes:

1. **Header**: Datos acerca del propio token
2. **Payload**: Datos que podemos codificar en el token. Puede ser cualquier dato que queramos, en nuestro caso, usaremos el ID del usuario.
3. **Signature**: Se crea usando el Header, el Payload y una clave secreta en el servidor. Este proceso se llama “Sign the JWT”.

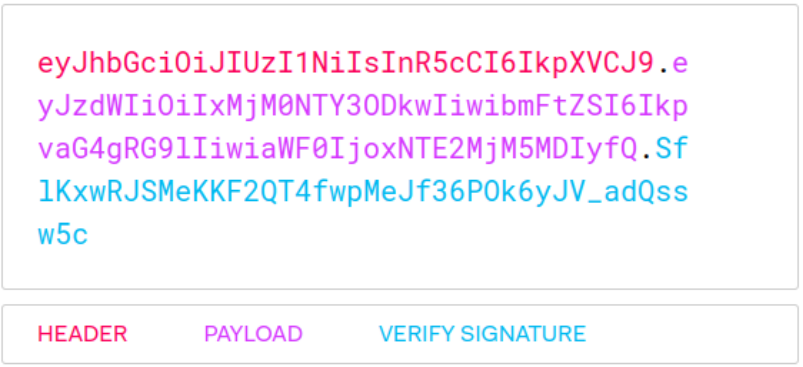

Figura 12.1: Estructura de un JWT

Verificar el JWT

1. El algoritmo de JWT selecciona el Header, el Payload y la clave secreta del servidor para crear una firma (signature) única.
2. Luego, esta firma, junto con el Header y el Payload, crean el JWT que será enviado al cliente.

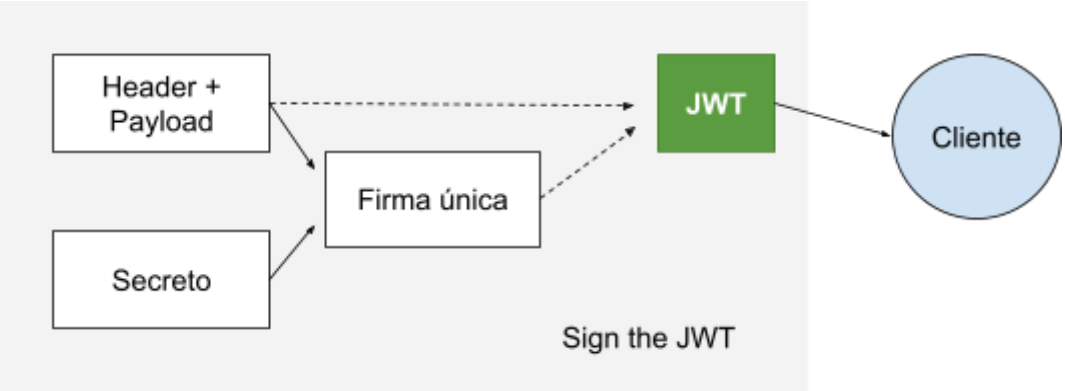

Figura 12.2: Sign the JWT

3. Ahora el cliente envía la petición con el JWT que se ha creado.
4. Una vez el servidor recibe el JWT del cliente, verifica si se han modificado el Header o el Payload.
5. Selecciona el Header y el Payload junto con la clave secreta que sigue almacenada en el servidor y crea una firma de prueba.
6. La firma original, la que se creó para enviar el JWT, también sigue almacenada en el servidor.
7. Finalmente, se compara la firma original con la firma de prueba. Si son iguales, significa que el Header y el Payload no han sido modificados, por tanto podemos dar acceso al usuario.

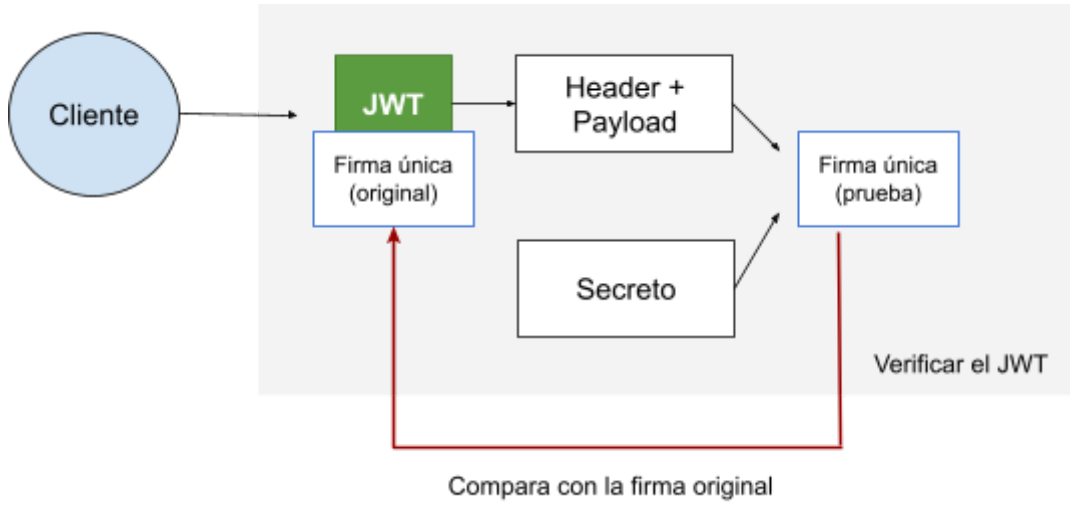

Figura 12.3: Verificar el JWT

Enviar el JWT como una Cookie

El algoritmo desarrollado previamente, es usado para las funciones de registro e inicio de sesión. Serán peticiones tipo POST donde se enviará los datos del usuario en el cuerpo de la petición. Una vez obtenido o creado el usuario mediante consultas a la base de datos (`Usuario.create()` o `Usuario.findOne()`), crearemos el JWT a través del ID del usuario y lo enviaremos como una cookie en el navegador usando las funciones del paquete *cookie-parser* de npm:

```
res.cookie('jwt', token, cookieOpt);
```

Una vez implementada la cookie, podemos crear un middleware *protect* que compruebe si esta existe. En caso afirmativo, consideramos que el usuario ha iniciado sesión y podrá acceder a los recursos protegidos de nuestra aplicación. La función *protect* guardará el usuario actual en la petición y siempre será ejecutada antes de la funcionalidad que queremos condicionar. Por ejemplo, modificar un sendero:

```
const auth = require('./../controls/controlsAutenticacion');

router
  .route('/:id')
  .get(controlsSenderos.singleSendero)
  .patch(
    auth.protect,
    controlsSenderos.updateSendero
  );
```

12.2. Autorización

Ahora verificamos si un usuario tiene los permisos adecuados para interactuar con ciertos recursos.

¿Cómo interactúa un usuario con los datos protegidos de nuestro sistema?

1. Envía su JWT con una petición al servidor.
2. El servidor verifica que el JWT es válido
3. Si es validado, se verifica la capacidad del rol de usuario y los datos requeridos serán enviados al cliente

En el ejemplo anterior, carece de sentido que un usuario regular pueda modificar un sendero en nuestra base de datos, no solo por motivos de seguridad, sino también por motivos de contenido, ya que podrían insertar datos inadecuados, irrelevantes o erróneos.

Puesto que hemos declarado distintos roles en el recurso *Usuario*, podemos limitar las funcionalidades que modifiquen, creen o eliminen otros recursos en nuestro sistema para que sean accesibles solo por cuentas de administración.

Crearemos un nuevo middleware *restrict* que se ejecutará siempre después del middleware *protect*, así podremos identificar el usuario y acceder a su rol. Si el tipo de usuario coincide con el rol que pasamos como parámetro, entonces tendrá acceso al componente restringido:

```
router
  .route('/:id')
  .patch(
    auth.protect,
    auth.restrict('admin'),
    controlsSenderos.updateSendero
  );
```

13. Modelado de datos y Mongoose Avanzado

El modelado de datos es el proceso de recopilación de datos desestructurados en un escenario real y ordenarlos de forma estructurada diferenciando los tipos de relaciones entre ellos siguiendo una configuración lógica.

Existen varios tipos de relaciones:

- 1 : 1 ⇒ Por ejemplo, una película solo puede tener un nombre.
- 1 : Muchos ⇒ Por ejemplo, una película puede tener muchos premios.
- Muchos : Muchos ⇒ Por ejemplo, una película tiene varios actores y esos actores trabajan en varias películas.

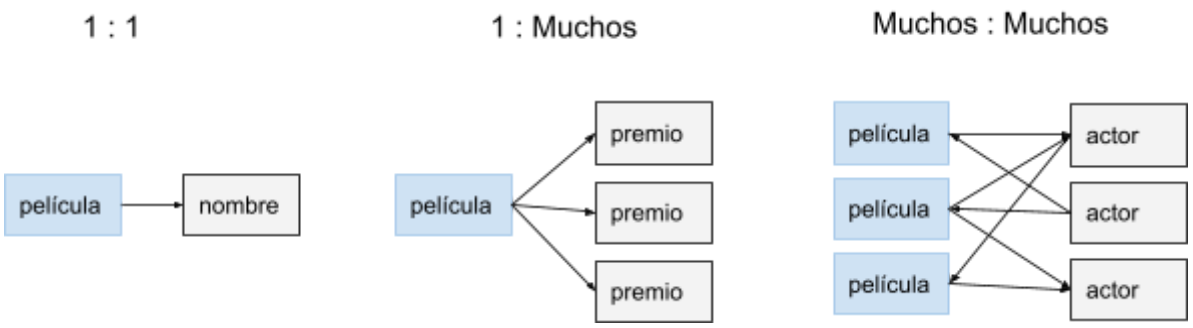

Figura 13.1: Tipos de relaciones entre datos

Siempre que tengamos una serie de datos relacionados, se pueden representar haciendo uso de referencias que definen dichas relaciones. Estas pueden ser mediante:

- **Referencing (referencia)**: se guardan los elementos en diferentes documentos que son referenciados a través de un valor común. En el ejemplo a continuación, se hace referencia a los elementos “hijo”.

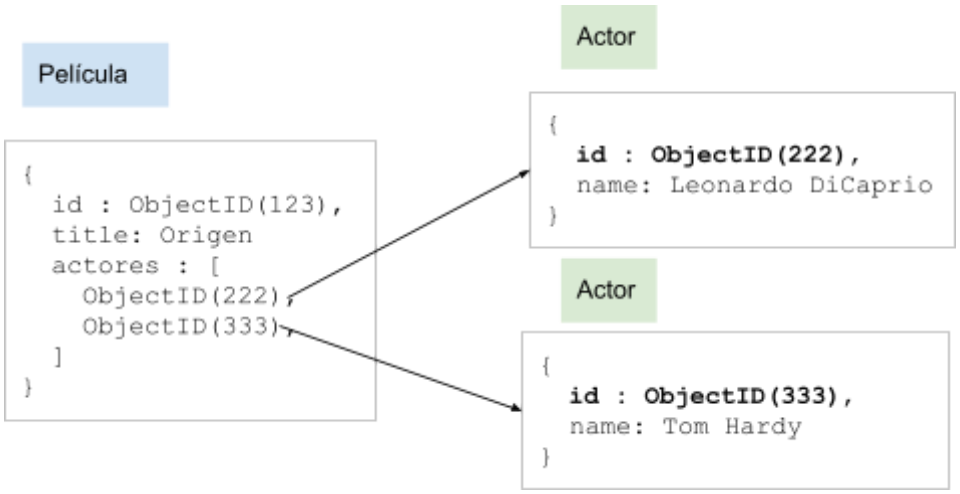

Figura 13.2: Relación de datos (Referencing)

- **Embedding (incrustar)**: se guardan todos los elementos en el mismo documento. Permite obtener toda la información a través de una sola consulta.

Figura 13.3: Relación de datos (Embedding)

13.1. Reviews

Para implementar las reviews en nuestra aplicación, debemos definir las relaciones y el tipo de referencia en nuestra base de datos. Como es de esperar, éstas serán elementos hijo del elemento padre Sendero, ya que un sendero puede tener múltiples reviews. Podríamos usar la técnica de “Embedding” y así tener toda la información disponible en el mismo lugar. No obstante, el documento podría crecer indefinidamente y MongoDB acepta un límite de consulta de 16 MB, lo que afectaría gravemente al rendimiento de nuestra web.

En el schema de las reviews, hemos definido propiedades que hacen referencia a otros recursos de la aplicación. Estos son el Usuario y el Sendero, ya que una review debe pertenecer a un usuario y a un sendero.

Para crear las valoraciones de los usuarios de forma óptima, definimos **propiedades virtuales** en el schema Sendero y el schema Review. Estas propiedades son campos que no se guardarán en la base de datos, por tanto no podremos usar consultas para visualizarlos.

```
const reviewSchema = new mongoose.Schema(
  {
    //definiciones
  },
  {
    //Cada vez que el output es un JSON o un objeto, muestra los virtuals.
    toJSON: { virtuals: true },
    toObject: { virtuals: true }
  }
);
```

Podremos crear una review enviando el ID del usuario y el ID del sendero en la petición, pero esto no creará una referencia. Para ello necesitamos implementar la técnica de **Populating**.

Populate es simplemente el proceso que nos permite hacer referencia a documentos en otras colecciones. Crearemos un middleware en el documento de las reviews para todas las consultas que incluyan una búsqueda, donde incluiremos las propiedades del schema que hacen referencia a otros elementos, es decir, sendero y usuario:

```
reviewSchema.pre(/^find/, function(next) {
   this.populate({
      path: 'sendero',
      select: 'name'
   }).populate({
      path: 'usuario',
      select: 'name'
   });
  next();
});
```

Ahora las reviews sabrán a qué sendero pertenecen, pero los senderos no serán capaces de identificar si la review pertenece a ellos. Con **Virtual populating** podemos hacer esa referencia permitiendo el acceso a todas las reviews sin tener que almacenar el array de IDs en el Sendero. Esto soluciona el problema de las referencias al hijo, donde los documentos pueden crecer indefinidamente.

```
senderoSchema.virtual('reviews', {
  ref: 'Review',
  foreignField: 'sendero',
  localField: '_id'
});
```

Sólo queremos mostrar las reviews para las peticiones a senderos individuales y no a todos los senderos, ya que afectaría al rendimiento. Así que haremos populate solo en el controlador de senderos individuales:

```
exports.singleSendero = async(req, res, next) => {
  try {
    const sendero = await Sendero.findById(req.params.id).populate('reviews');
  }
  ...
};
```

Uno de los problemas encontrados en la configuración actual, es que las reviews harán *populate* del propio sendero, es decir, dentro de los datos del sendero aparecerá la review, y dentro de la review aparecerá de nuevo el sendero. Para solucionarlo, hemos eliminado el *populate* de los senderos en el schema de las reviews, así conservamos la referencia al padre y eliminamos el exceso de datos para mejorar el rendimiento:

```
reviewSchema.pre(/^find/, function(next) {
   //Eliminamos populate a senderos.
   this.populate({
      path: 'usuario',
      select: 'name'
   });
  next();
});
```

Figura 13.4: Solución al problema de populate

13.2. Cálculo de la media de reviews

Almacenar un resumen de un conjunto de datos relacionados en el conjunto principal de datos es una técnica muy popular en el modelado de datos.

En nuestra aplicación, un gran ejemplo de esta técnica es almacenar la media de las reviews (*gradeAverage*) y el número de ratings (*gradeQuantity*) en cada sendero. De esta manera no tenemos que hacer una consulta a Reviews y calcular la media cada vez que se ejecute una petición a los senderos.

Calcularemos la media de ratings y el número de ratings de un sendero cada vez que una nueva review se añada a dicho sendero. También cuando una review se actualice o elimine.

Crearemos una función en el modelo de Reviews que tomará el ID del sendero y calculará la media y el número de ratings que existen en nuestra colección para ese sendero. Usaremos un middleware para llamar a esta función cada vez que se genera, actualiza o elimina una review.

Escribiremos un método *estático* en el Schema, ya que permite llamar a la función en el propio modelo.

13.2.1. MongoDB Aggregation Pipeline

La idea es definir un conducto donde los documentos de una colección y sean procesados según unos pasos o etapas que definimos en el código para que sean transformados en resultados que se agregan al documento:

1. En el primer paso filtramos todas las reviews que tengan el mismo sendero.

```
{
  $match : { sendero : senderoId}
},
```

2. En el segundo paso agrupamos los documentos usando acumuladores.

```
{
  $group : {
    _id: '$sendero',
    nRating : { $sum : 1 },
    avgRating : { $avg : '$rating' }
  }
}
```

3. Finalmente, con los datos obtenidos en los cálculos del *aggregation pipeline* y almacenados en una variable llamada *ratings*, podemos actualizar los datos del sendero que nos interesan mediante una consulta a la base de datos.

```
await Sendero.findByIdAndUpdate(senderoId, {
  gradeQuantity: ratings[0].nRating,
  gradeAverage: ratings[0].avgRating
});
```

Una vez definida la función para calcular el número de reviews y la media de las valoraciones, podremos ejecutarla cada vez que se crea, se modifica o se elimina una review. Para ello, utilizaremos middlewares en los documentos. No obstante, surgen dos problemas:

1. No podremos acceder al modelo “Reviews” ya que este se declara al final del schema.

Solución: Llamar al constructor, ya que apuntará al modelo actual.

```
reviewSchema.post('save', function() {
  this.constructor.calcAverageGrades(this.sendero, this.constructor);
});
```

2. No podremos ejecutar un middleware en los documentos para las operaciones de actualización o eliminado, ya que no existen. Seremos capaces ejecutar un middleware para esos tipos de consulta, pero surge otro problema; no podremos usar el método `post()` porque en ese punto el documento ya ha sido guardado y por tanto, la consulta a la base de datos se habrá ejecutado y no tendremos acceso a `findOne()`.

Solución: Guardar el objeto Review en la consulta actual usando el método `pre()` en el propio documento, que se ejecutará antes de guardar la review en la base de datos. Una vez almacenado en el objeto actual, podremos ejecutar la función de cálculo de medias en el método `post()`.

```
//findOneAndUpdate
//findOneAndDelete

reviewSchema.pre(/^findOneAnd/, async function(next) {
  this.currentReview = await this.findOne();
});

reviewSchema.post(/^findOneAnd/, async function(next) {
  await
this.currentReview.constructor.calcAverageGrades(this.currentReview.sendero);
});
```

14. Renderizando el contenido de la API

En vista de que los datos de nuestra API y la manera de interactuar con ellos ya han sido implementados, ahora podremos enfocar el proyecto en los resultados visuales. Mencionado anteriormente, usaremos plantillas **Pug**, que es un compilador de código HTML con una sintaxis simplificada, donde además podremos usar condicionales, bucles y dividir el HTML en fragmentos reusables. Guardaremos los ficheros Pug en el directorio *views*:

```
/views
    ├── _footer.pug
    ├── _header.pug
    ├── ...
    └── overview.pug
```

En el directorio *assets*, guardaremos los ficheros relevantes para la composición del front-end. Estos son el código CSS, JavaScript para interactuar con la web, fuentes, imágenes… etc.

```
/assets
    ├── /css
          └── main.css

    ├── /fonts
    ├── /img
    └── /js
```

Ahora, debemos indicar a Express que usaremos la ingeniería de Pug como herramienta para renderizar el contenido y en qué directorio se encuentran las plantillas:

```
app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'views'));
```

Para escribir código Pug, simplemente necesitamos escribir el nombre de las etiquetas HTML, su contenido e indentar correctamente:

```
doctype html
html
  head
    block head
      meta(charset='UTF-8')
      meta(name='viewport' content='width=device-width, initial-scale=1.0')
      link(rel='stylesheet' href='/css/main.css')
  body
    include _header
    block content
    include _footer
    script(src='/js/index.js')
```

Los links a ficheros CSS y JavaScript funcionarán ya que hemos definido un middleware donde “servimos” los elementos estáticos desde el directorio *assets*:

```
app.use(express.static(path.join(__dirname, 'assets')));
```

Una vez configurada la estructura del front-end y distribuidos sus ficheros, podemos renderizar el contenido. Para ello, crearemos nuevas rutas y un controlador de las funciones que se ejecutarán cuando accedemos a ellas. Seguiremos el mismo proceso utilizado hasta ahora, la principal diferencia es que en vez de enviar el resultado en formato JSON, lo enviaremos en formato HTML usando la función `render()`:

```
router.get('/overview', controlsViews.allOverview);
exports.allOverview = async(req, res, next) => {
  try {

    const senderos = await Sendero.find();

    res.status(200).render('overview', {
      title: 'Senderos',
      senderos: senderos
    });

  } catch (err) {
      ...
  }
};
```

El método `render()` recibe el nombre del fichero que será “traducido” a HTML y envía los datos que posteriormente serán accedidos en la plantilla Pug.

Para implementar las funcionalidades de la API en la interfaz web, usaremos peticiones AJAX a las rutas de la API. AJAX (Asynchronous JavaScript And XML) permite modificar páginas web mediante intercambios de datos de forma asíncrona con un servidor. Por ejemplo, para la funcionalidad de *log-in*, enviaremos una petición AJAX a la ruta de inicio de sesión de nuestra API (servidor):

```
let request = new XMLHttpRequest();

request.open('POST', 'http://localhost:6700/api/usuarios/login');
request.setRequestHeader("Content-Type", "application/json" );

request.send(JSON.stringify(dataObj));
```

El resto de funcionalidades como la búsqueda, registro, log-out… etc, también funcionan con peticiones AJAX al servidor.

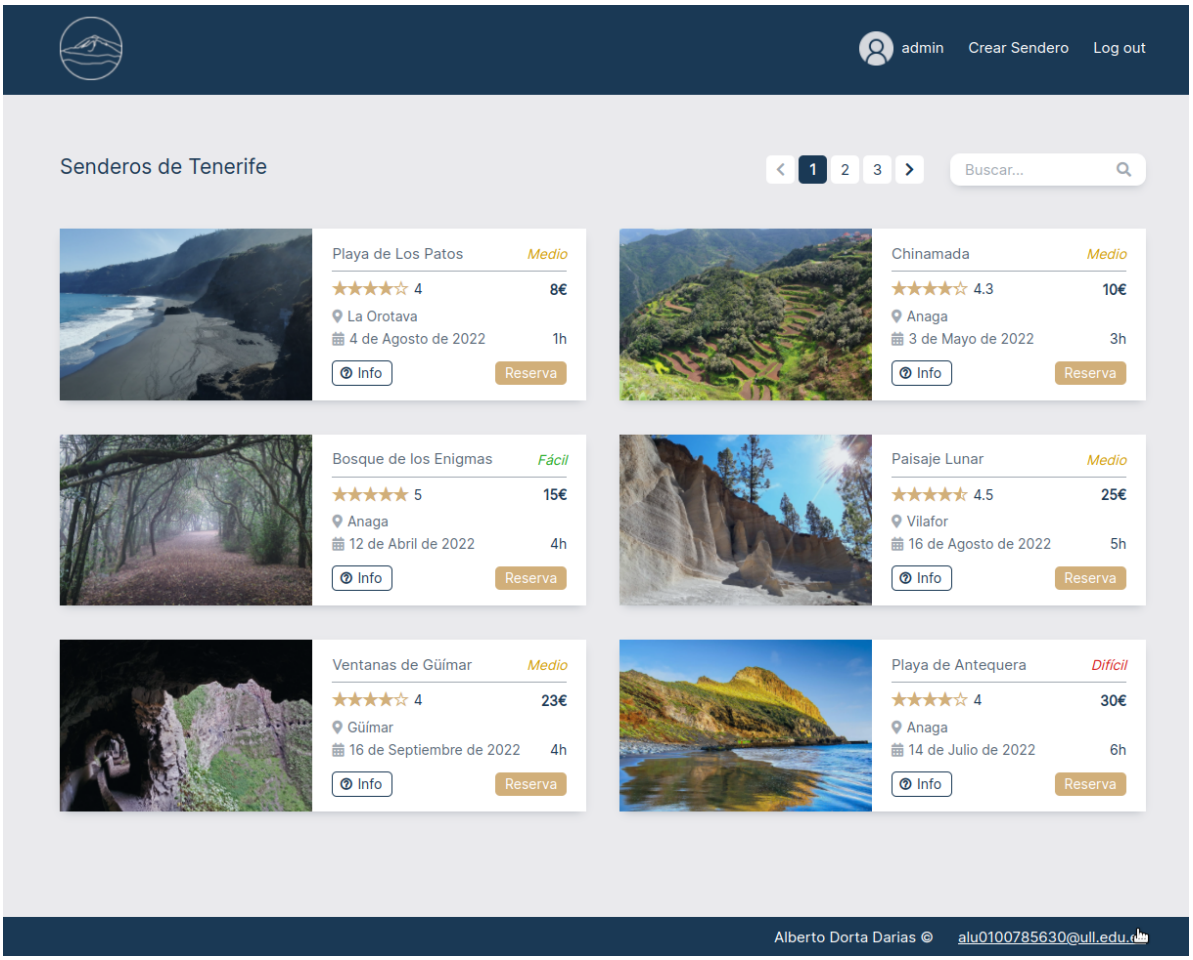

Figura 14.1: Overview de los senderos en el Front-End

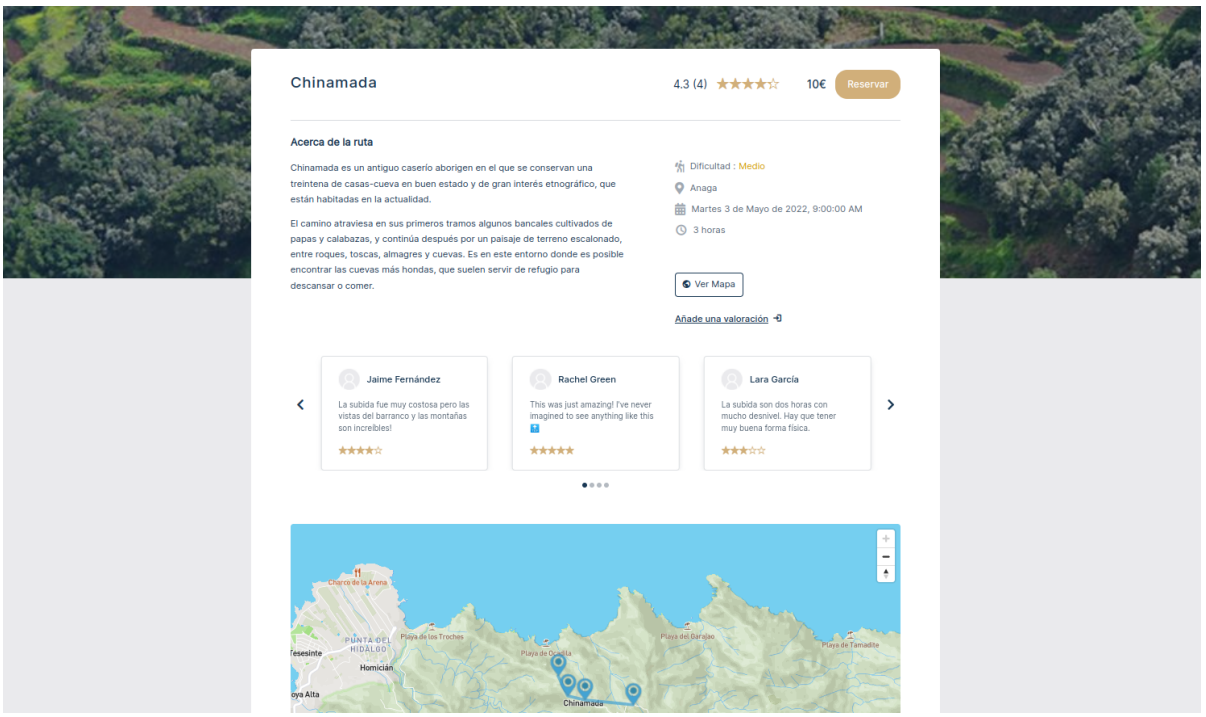

Figura 14.2: Sendero individual en el Front-End

15. Interfaz de usuario

A lo largo de esta sección haremos una breve descripción de las páginas y módulos disponibles en el sitio web una vez se ha renderizado correctamente el contenido de la API.

- **Header**

El header o encabezado se encuentra en la parte superior de cada página donde proporciona un espacio para el logotipo y diferentes links para facilitar la navegación.

Figura 15.1: Header

- **Footer**

El footer o pie de página se encuentra en la parte inferior de cada página y contiene un aviso de derechos de autor y la información de contacto.

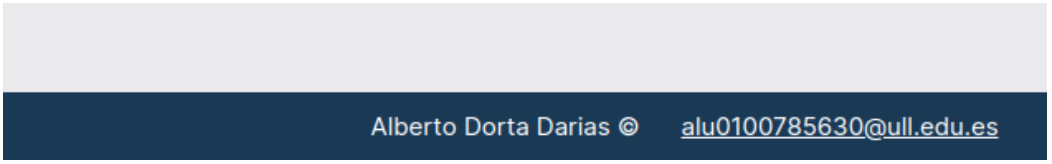

Figura 15.2: Footer

- **Home - Frontpage**

Como en toda web moderna, hemos implementado una landing page que muestra información básica sobre nuestra aplicación, así como distintos "*Call to Action*" para dirigir al usuario al contexto que nos interesa. Esta página no implementa ninguna funcionalidad de nuestra API, es simplemente una combinación de código HTML y CSS.

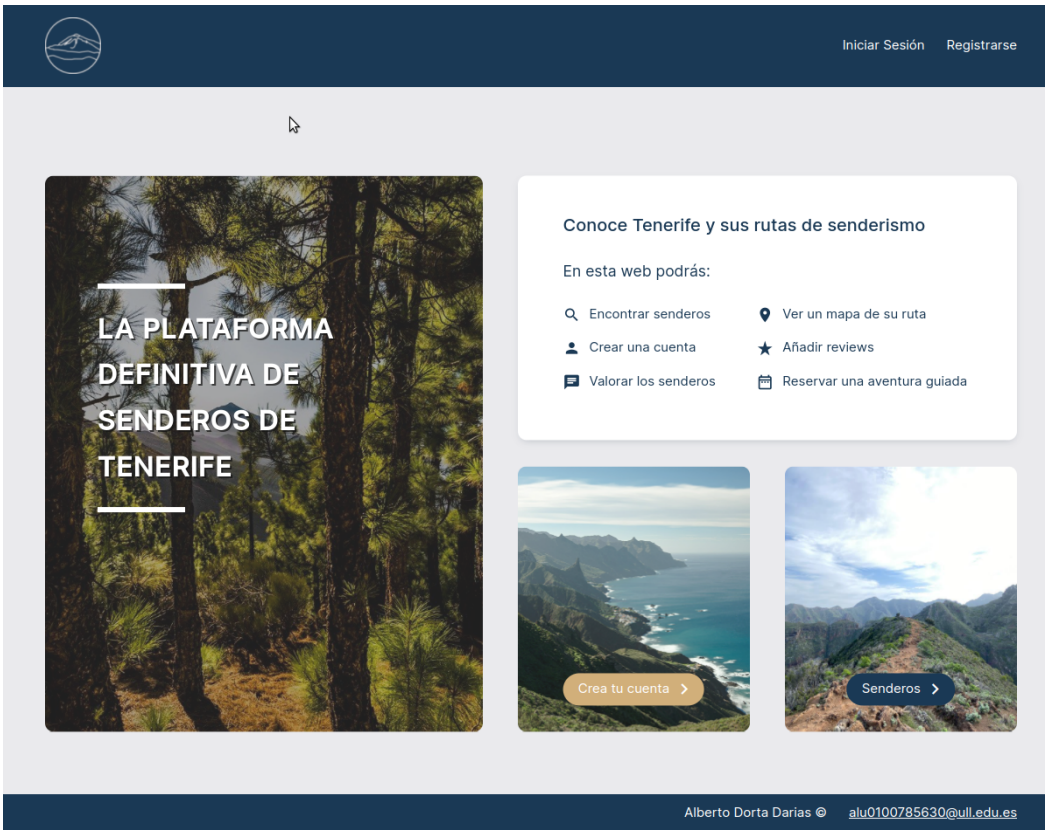

Figura 15.3: Home - Frontpage

- **Inicio de Sesión**

En la página de inicio de sesión, el usuario introduce sus credenciales donde se comprueba que existe y se valida el formato y el valor de los campos.

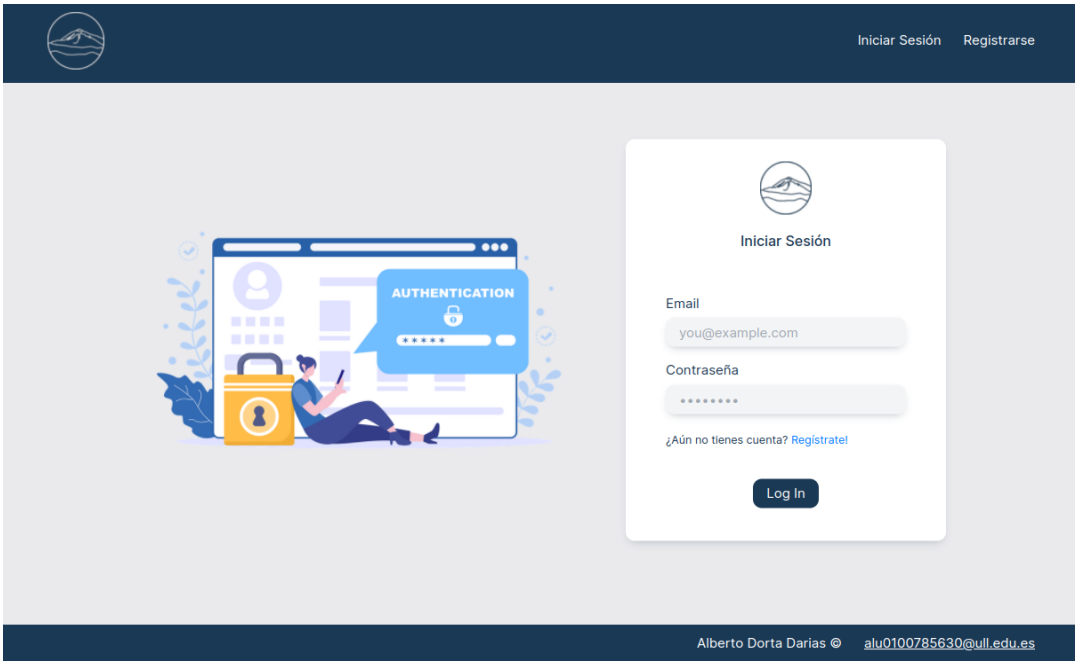

Figura 15.4: Inicio de Sesión

Una vez iniciada la sesión, el header cambiará sus links. Además, para las cuentas de administrador se muestra una ruta extra donde se puede crear un sendero.

Figura 15.5: Inicio de Sesión - Header

Figura 15.6: Inicio de Sesión - Header Administrador

- **Registro**

Aquí el usuario podrá darse de alta en la plataforma. Se comprueban y se validan varios campos que contienen la estructura de datos del recurso Usuario de nuestra API.

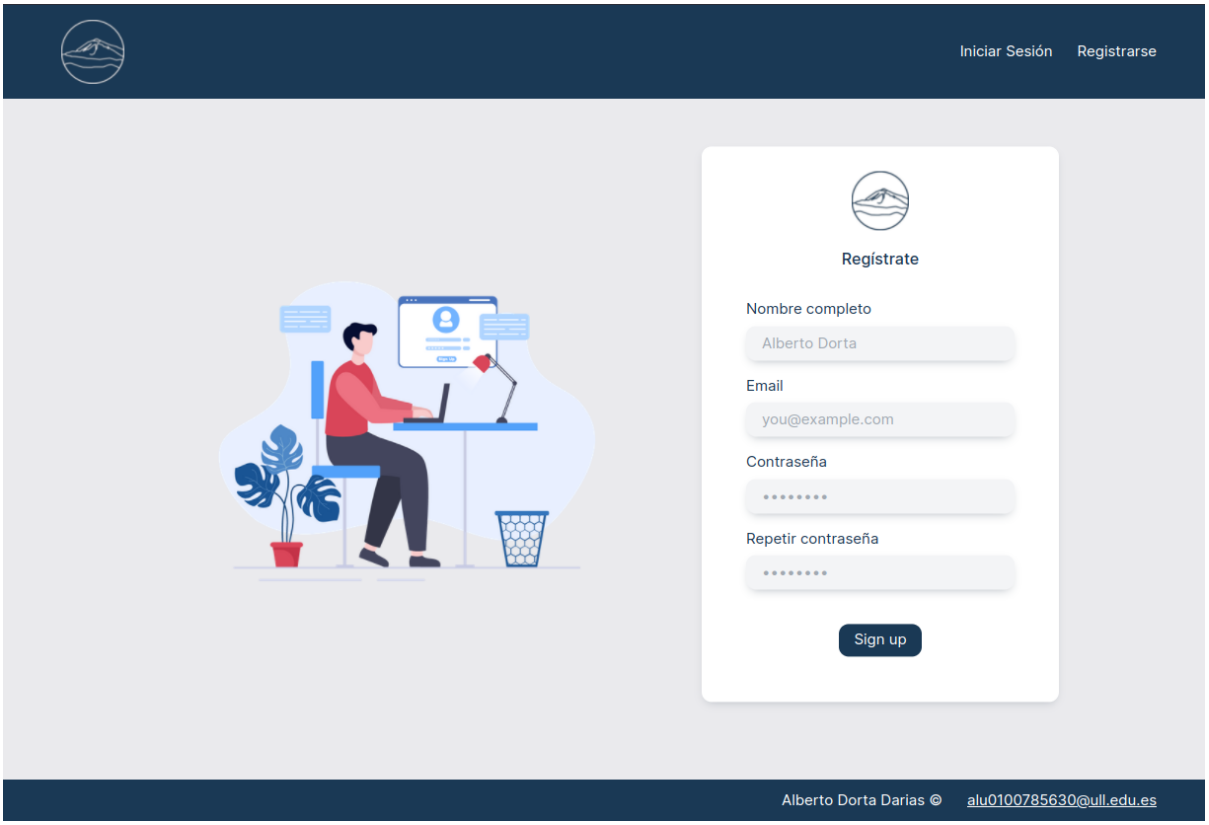

Figura 15.7: Registro

- **Overview**

En esta página encontramos el archivo de todos los senderos que han sido registrados en nuestra base de datos. Podremos consultarlos individualmente o usar las funciones de búsqueda y paginación implementadas al inicio de esta página.

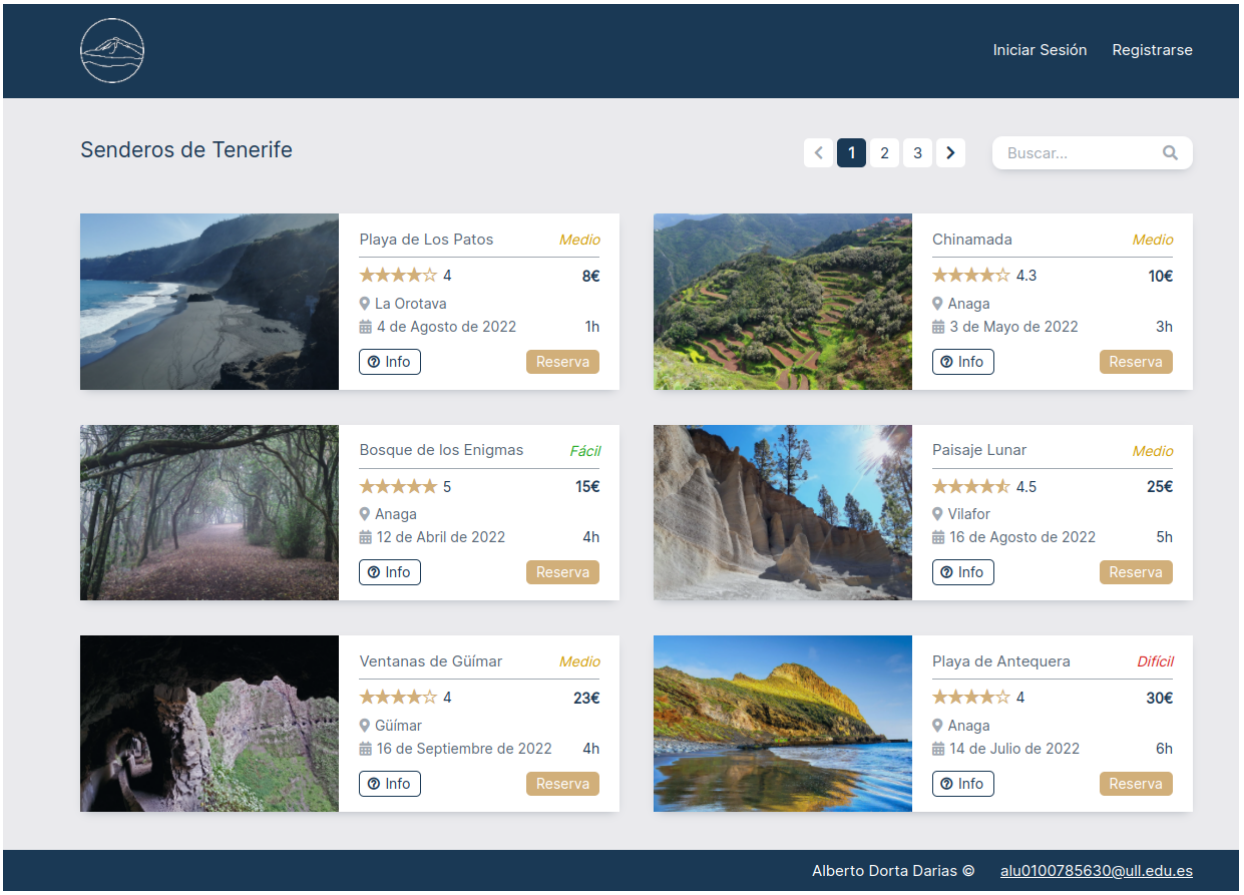

Figura 15.8: Overview

- **Páginas de Senderos**

Las URL de los senderos se crean automáticamente en el campo slug de cada documento, es ahí donde se renderiza el contenido. En cada URL asociada a los senderos encontramos toda su información: duración, ubicación, dificultad, media de las valoraciones… etc. También encontraremos un slider con cada una de las valoraciones de los usuarios y un botón “Ver Mapa” que nos lleva directamente a la ruta del sendero. En el mapa se pueden observar los distintos puntos de interés del sendero.

Si el usuario ha iniciado sesión, también encontraremos un input para añadir reviews, donde se indica la calificación y un breve comentario.

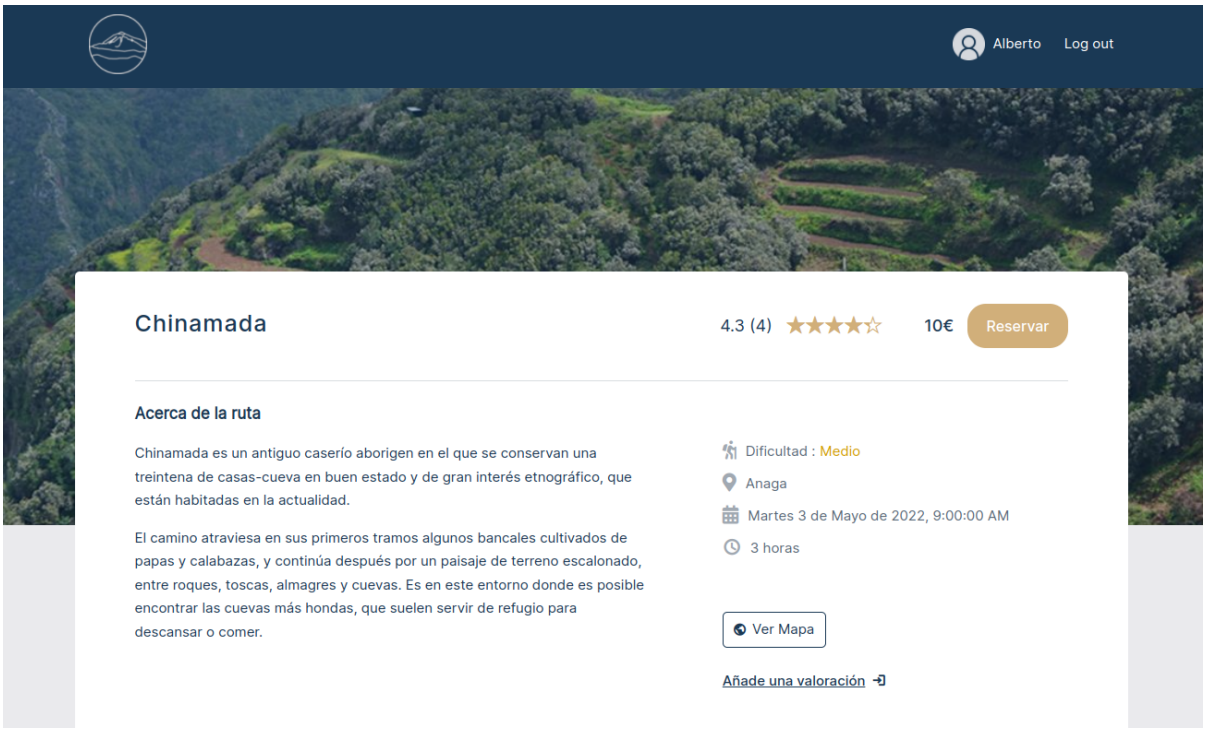

Figura 15.9: Parte superior del sendero

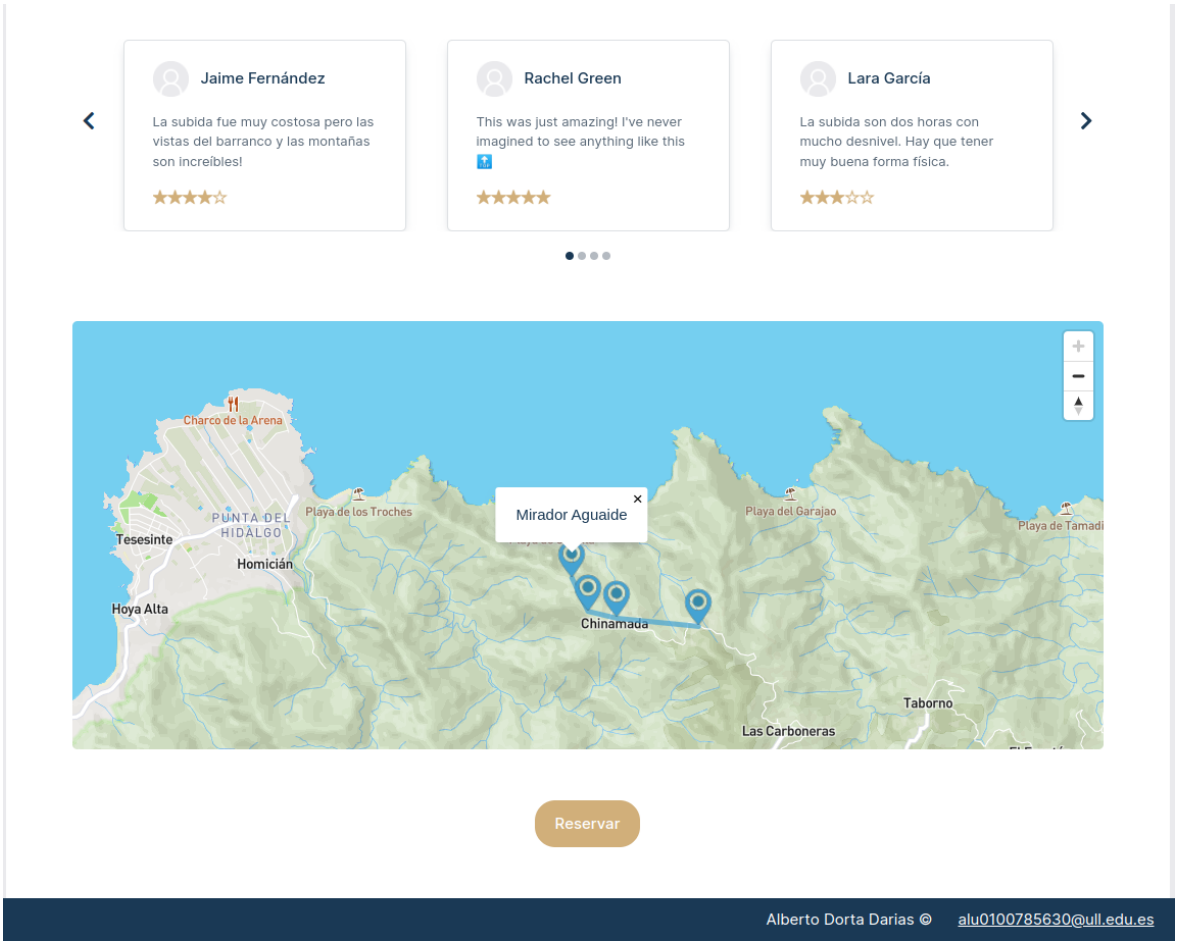

Figura 15.10: Parte inferior del sendero

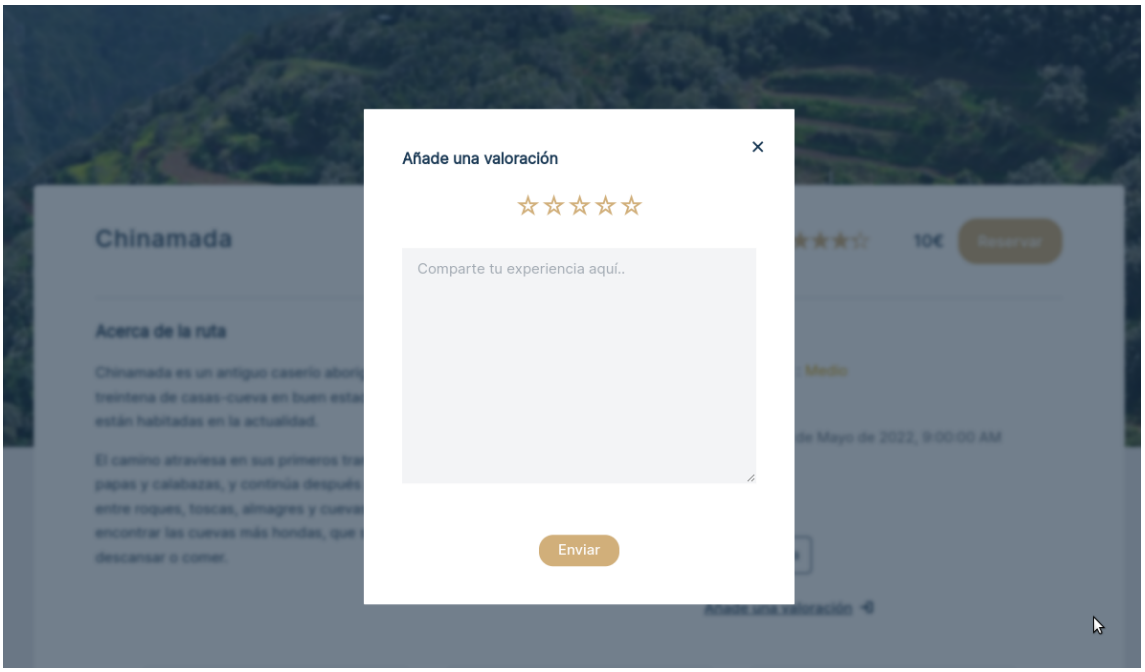

Figura 15.11: Pop-up para añadir reviews

- **Páginas de Senderos**

Por último encontramos la página para crear senderos, donde se añaden los datos que hemos requerido en el Back-End. Esta página está restringida solo para usuarios administradores.

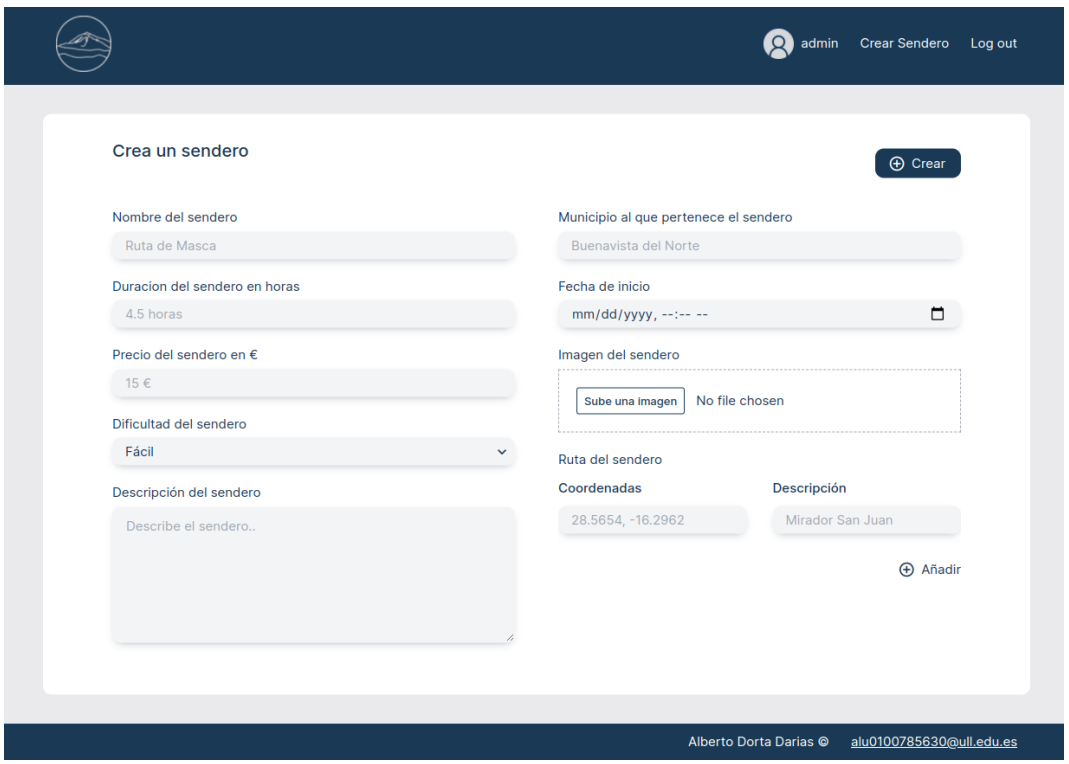

Figura 15.12: Crear senderos

16. Conclusiones y líneas futuras

Durante el desarrollo del proyecto, nos hemos dado cuenta de que el resultado más relevante de esta aplicación web es su potencial uso para empresas turísticas, o simplemente como reclamo y herramienta muy interesante para explorar los senderos de Tenerife. Gracias a la implementación de una API, la plataforma será fácilmente escalable puesto que puede adaptarse sin ningún tipo de problema al crecimiento de la base de datos de los senderos y los usuarios, sin que esto afecte al rendimiento. Asimismo, existe la posibilidad de generar nuevas formas de llegar a futuros clientes ya que podemos migrar la plataforma a cualquier otro sistema; aplicación de Android, aplicación de iOS, programa de escritorio… etc.

Pese a las numerosas virtudes y beneficios que encontramos en la aplicación, también encontramos posibles mejoras y funcionalidades incompletas:

- Implementación del sistema de pago y sistema de reserva
- Creación de un entorno de producción
- Encriptado de las contraseñas de los usuarios
- Funcionalidad de recuperación de contraseña
- Seguridad
    - Implementar Rate Limiting: limitar el número de peticiones HTTP desde una misma IP para así evitar ataques de fuerza bruta.
    - Configurar HTTPS para encriptar los datos de nuestra aplicación

- Construir funciones globales para las operaciones CRUD y así mejorar la mantenibilidad del código.

A modo de cierre de este trabajo, cabe destacar que se ha intentado trasladar los procedimientos de desarrollo a un entorno que fuera lo más profesional posible, donde el código se presenta de forma totalmente modular y estructurada según su función. Ha habido una gran curva de aprendizaje durante las fases de desarrollo, donde hemos corregido y optimizado cualquier tipo de error, con el fin y la esperanza de poder publicar un producto que presente una solución consistente y robusta.

17. Summary and conclusions

During the development of the project, we have realized that the most relevant result of this web application is its potential use for tourism companies, or simply as an attractive and very interesting tool to explore the hiking routes of Tenerife. Thanks to the implementation of an API, the platform will be easily scalable since it can adapt to the growth of the database of hiking routes and users, without affecting performance. In addition, there is the possibility of generating new ways of reaching future clients since we can migrate the platform to any other system; Android app, iOS app, desktop software… etc.

Despite the many virtues and benefits that we find in the application, we also find possible improvements and missing functionalities such as:

- Implementation of the payment system and reservation system
- Create a production environment
- Encryption of user passwords
- Password recovery functionality
- Security
    - Implement Rate Limiting: limit the number of HTTP requests from the same IP in order to avoid brute force attacks.
    - Configure HTTPS to encrypt the data of our application

- Build global functions for CRUD operations to improve code maintainability.

As a closure, we should mention that we have done quite an effort to transfer the development procedures to an environment that was as professional as possible, where the code is presented in a completely modular and structured way according to its function. There has been a great learning curve during the development phases, where we have corrected and optimized any kind of issues, with the aim and hope of being able to publish a product that presents a consistent and robust solution.

18. Bibliografía

[1] Universidad de La Laguna, Grado en Ingeniería Informática
https://campusingenieriaytecnologia2122.ull.es/

[2] Documentación oficial de NodeJS
https://nodejs.org/es/

[3] Paquete de NPM - JSON Web Tokens
https://www.npmjs.com/package/jsonwebtoken

[4] Técnica de populate en Mongoose
https://runebook.dev/es/docs/mongoose/populate

[5] Información sobre las API
https://es.wikipedia.org/wiki/Interfaz_de_programaci%C3%B3n_de_aplicaciones

[6] Arquitectura REST API
https://www.redhat.com/es/topics/api/what-is-a-rest-api

[7] Tutorial de configuración de Script de Webpack
https://www.youtube.com/playlist?list=PLblA84xge2_zwxh3XJqy6UVxS60YdusY8

[8] Documentación oficial de Mapbox
https://docs.mapbox.com/

[9] Documentación oficial de Pug
https://pugjs.org/api/getting-started.html

[10] Consultas para tareas específicas en Stack Overflow
https://stackoverflow.com/

[11] Aggregation Pipeline de MongoDB
https://docs.mongodb.com/manual/core/aggregation-pipeline/