Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

**UNIVERSIDAD DE JAÉN**
*Nombre del Centro*

Trabajo Fin de Grado

# ESTUDIO Y DESARROLLO DE UN PROTOTIPO DE APLICACIÓN WEB PARA LE GESTIÓN DE UN POLIDEPORTIVO

**Alumno: José Manuel García Ronda**

Tutor: José Ramón Balsas Almagro
Dpto: Informática

**Septiembre, 2022**

AGRADECIMIENTOS

En primer lugar, me gustaría mostrar mi agradecimiento a Jose Ramón Balsas Almagro por la oportunidad que me ha dado para realizar el trabajo con él y por la paciencia y dedicación que ha tenido en este tiempo.

A mis amigos de la universidad con los que he vivido muchas experiencias y vivencias tanto fuera como dentro de la universidad.

A mis compañeros de clase por generarme la ambición de superarme día a día.

A mis profesores de la universidad por formarme y ayudarme a crecer intelectualmente en el mundo de la informática.

A mis padres, por apoyarme en los momentos difíciles y ser el pilar fundamental para poder cumplir este objetivo.

A mi hermana María por hacerme ver en las adversidades que siempre hay luz al final del túnel.

A demás compañeros, amigos y familiares que me han apoyado hasta el final.

Gracias a todos.

# Tabla de contenidos

# Resumen

En este Trabajo de Fin de Grado se procede a realizar el diseño y la implementación de una aplicación web cuyo principal propósito reside en la gestión y realización de reservas de las instalaciones deportivas de un polideportivo de forma online.

Para conseguirlo se ha optado por dividir la aplicación principal en dos subproyectos, un proyecto para la parte que corresponde al Front-End de la aplicación y otro para la parte de Back-End de la misma. De esta manera se intenta que ambas partes del proyecto sean independientes entre sí y encapsular todo lo máximo posible para facilitar la implementación y desarrollar ambas partes consiguiendo disminuir la presencia de errores de forma considerada.

La parte de Back-End se ha desarrollado con SpringBoot haciendo uso de apis del propio Spring como JPA para la persistencia de datos, Spring Security para gestionar la seguridad de la aplicación y varias más para completar las funcionalidades de la misma.
Mientras que para la parte de Front-End se ha implementado mediante Ángular, ya que es un framework muy potente porque combina HTML,CSS y TypeScript.

Para realizar la persistencia de los datos del proyecto, se ha optado por usar una H2, ya que se complementa muy bien con SpringBoot.

# 1.- Introducción

## 1.1 Motivación

Actualmente, cada vez se hace más énfasis en la importancia de cuidar de la salud de la sociedad, es por ello, que desde los más pequeños hasta los mayores están introduciéndose en el mundo del deporte. Sin embargo, por falta de tiempo se suele dar de lado, ya que para acceder a las instalaciones deportivas de la propia ciudad, es necesario ir físicamente para realizar las reservas y esto precisa de tiempo el cual a veces no es fácil disponer. Esta razón ha sido el argumento principal que me impulsó a realizar este TFG, poder realizar reservas de las instalaciones deportivas mediante esta aplicación web garantizando así que se podrá llevar a cabo, sin esperas, la actividad deseada.

Con este proyecto, además de los objetivos funcionales que se desean llevar a cabo también existen una serie de objetivos personales que se quieren cumplir. En primer lugar formarme y enriquecerme en todas las facetas de la programación intentando aprender frameworks sobre desarrollo de aplicaciones en parte del servidor y en parte del cliente.

Pretendo enfrentarme a nuevos retos escogiendo tecnologías con las que no he trabajado antes para seguir formándome y adquiriendo nuevos conocimientos sobre programación y tecnologías de desarrollo.

Por último, gracias a la realización de este TFG me gustaría sentirme realizado ya que proviene de una necesidad existente en mi ciudad y resolverla mediante la implementación de esta aplicación web.

## 1.2 Objetivos

Los objetivos principales de este proyecto son:

- Realizar un estudio de necesidades y soluciones para la gestión de un polideportivo.
- Diseñar un sistema informático especialmente orientado a la utilización de arquitecturas y metodologías de desarrollo web.
- Implementar un prototipo de aplicación web usando tecnologías JEE en el lado del servidor y algún framework de desarrollo JS centrado en el cliente.

## 1.3 Metodología

El proceso para la realización de este proyecto consta de varias fases. En primer lugar, se realizará un trabajo de investigación sobre el tema elegido y las tecnologías que serán utilizadas en este.

Posteriormente, se realizará un estudio sobre las necesidades y requisitos que debe tener la aplicación que vamos a implementar. Además se usará como metodología ágil usando algunos de los elementos principales de *SCRUM de forma individual* para determinar la planificación del proyecto y la duración del mismo.

El proyecto se dividirá en varios sprints en los que tendremos definidas varias historias de usuario con sus respectivas tareas que tendremos que finalizar. A su vez cada historia de usuario tendrá una estimación basada de puntos de historia para reflejar la importancia de cada una a la hora de elegirlas y asignarlas para cada sprint.

Por último, se realizará el proceso de diseño del propio sistema mediante patrones de diseño y arquitecturas finalizando con la implementación del prototipo del sistema.

## 1.4 Estructura del documento

En este apartado se expondrán de forma sintetizada los diferentes puntos que componen este proyecto.

Inicialmente expondremos de forma breve el tema de este Trabajo de Fin de Grado junto a la forma en la que se realizará.

En el punto 1, se realiza una breve introducción del proyecto indicando el motivo por el que se realiza, una breve descripción de los objetivos principales del mismo y la metodología de trabajo.

En el punto 2, se realizará una explicación de las diferentes fases de desarrollo del proyecto junto a una descripción de las tecnologías utilizadas y análisis de la gestión de polideportivos.

En los puntos 3,4 y 5 se va a realizar un diseño en base al análisis realizado anteriormente junto al diseño del proyecto,finalizando con un plan de pruebas para la parte del servidor.

En el punto 6, se explica las conclusiones conseguidas realizando una retrospectiva para analizar los requisitos alcanzados. Tras ello, se indicarán posibles mejoras para conseguir una mejor aplicación.

Finalmente, en el punto 7 se dispondrá de la bibliografía consultada para realizar este Trabajo de Fin de Grado.

Además, junto a los puntos anteriores, se añadirá un pequeño anexo que consta con un manual de usuario en el que se indicará el correcto uso de la aplicación web junto a un pequeño manual de gestión para tareas de administración.

# 2.- Análisis

## 2.1 Análisis preliminar

Actualmente, vivimos en una sociedad en la que el campo de la tecnología sigue una tendencia ascendente y más concretamente el uso de Internet. Conforme pasa el tiempo, mayor es el número de usuarios que comienzan a realizar todo tipo de tareas mediante Internet, y a su vez, las empresas ofertan sus servicios mediante esta vía, lo que está haciendo que cada vez los negocios actuales tengan que adaptarse siguiendo esta tendencia.

Otro factor a tener en cuenta, es el aumento de uso de aplicaciones web por parte de los usuarios, estas proporcionan una gran cantidad de beneficios para ellos entre la que destacan el libre acceso a la aplicación desde cualquier dispositivo o sistema operativo y no tienen porqué disponer de una gran cantidad de almacenamiento puesto que se puede acceder a ella mediante navegadores webs.[1]

El sector de gestión de reserva de pistas de polideportivos es muy variable ya que depende principalmente del municipio en el que te encuentres y el poder adquisitivo del mismo para poder realizar una aplicación. Este sistema puede ser valioso si el municipio

tiene constancia de la presencia de un elevado número de ciudadanos que realicen deporte y hagan uso de las instalaciones deportivas para ello.

Sin embargo, hay muchos municipios que optan por tener la gestión de reservas de forma manual y no tienen una plataforma web para llevar todo esto a cabo. Teniendo en cuenta la limitación de tiempo de los usuarios ya que si quieren hacer uso de las instalaciones deportivas deben de acudir al edificio para llevar a cabo la reserva.

### 2.1.1 Estudio sobre la gestión de reservas de polideportivos

En el entorno de este Trabajo de Fin de Grado, existen diversas soluciones que ya están implementadas, aunque tienen alguna desventaja.

Las aplicaciones de gestión de reservas normalmente están enfocadas para un solo tipo de pistas; la mayoría de pádel puesto que es un deporte muy popular que en los últimos años está en una tendencia ascendente.[2]

Por otra parte, las aplicaciones webs que hay para la gestión de reservas son de un carácter muy general y no permite realizar una especificación de tus propias instalaciones.

Por último, es importante destacar la escasez de este tipo de aplicaciones y la mayoría de ellas se corresponden con proyectos como el que se está desarrollando ya que generalmente dependen del municipio en el que te encuentres, la actividad deportiva que se realice y los medios suficientes para poder realizarla.

1) Reservas24h


*Ilustración 1: Interfaz de Reservas24h*

Como la imagen adjunta muestra, es una aplicación ya existente y totalmente personalizada de forma que puede incluir funcionalidades o reserva de instalaciones que no estén presentes en las instalaciones de la ciudad.[3]

Un punto débil que encontramos en la aplicación es lo poco intuitiva que es para el usuario de forma que dificulta el uso para aquellos que no estén familiarizados con las nuevas tecnologías. Por lo tanto, este es un punto a tener en cuenta en nuestra aplicación de cara al desarrollo.

2) Clicac


*Ilustración 2: Clicac*

Clicac es otra aplicación web para la gestión de reservas de instalaciones deportivas pero, como se observa, solo permite realizar reservas dentro de los centros dados de alta, ya que las ciudades adquieren los servicios de esta aplicación para su municipio.[4]

3) MyTurn

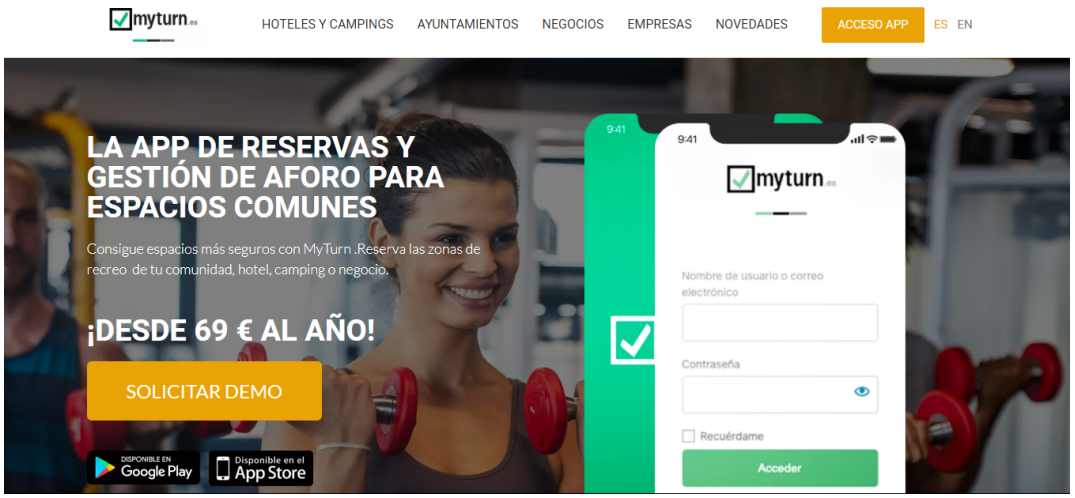



*Ilustración 3: MyTurn*

En este caso, esta aplicación web ofrece un servicio de gestión de aforo para espacios comunes, no es específica para polideportivos de forma que puede no adaptarse correctamente a los requisitos que tenga el propio polideportivo, o ser más efectiva en otro tipo de actividades. [5]

### 2.1.2 Selección de tecnologías

Tras ver el objetivo de la aplicación, se va a realizar un estudio sobre diferentes herramientas para la implementación de la parte de Back-end por parte del servidor y para la parte de Front-end por parte del cliente.

#### 2.1.2.1 Back-end

Como explica Rafa Arjonilla en [6]: *"El backend es la parte del desarrollo web que se encarga de que toda la lógica de una página web funcione. Se trata del conjunto de acciones que pasan en una web pero que no vemos como, por ejemplo, la comunicación con el servidor."*

Cada vez, se le da mayor importancia al back-end de una aplicación y ha habido un aumento en el número de programadores interesados en esta parte de los proyectos. A continuación en la Ilustración 4 se puede observar en qué área se identifican los propios programadores.[7]

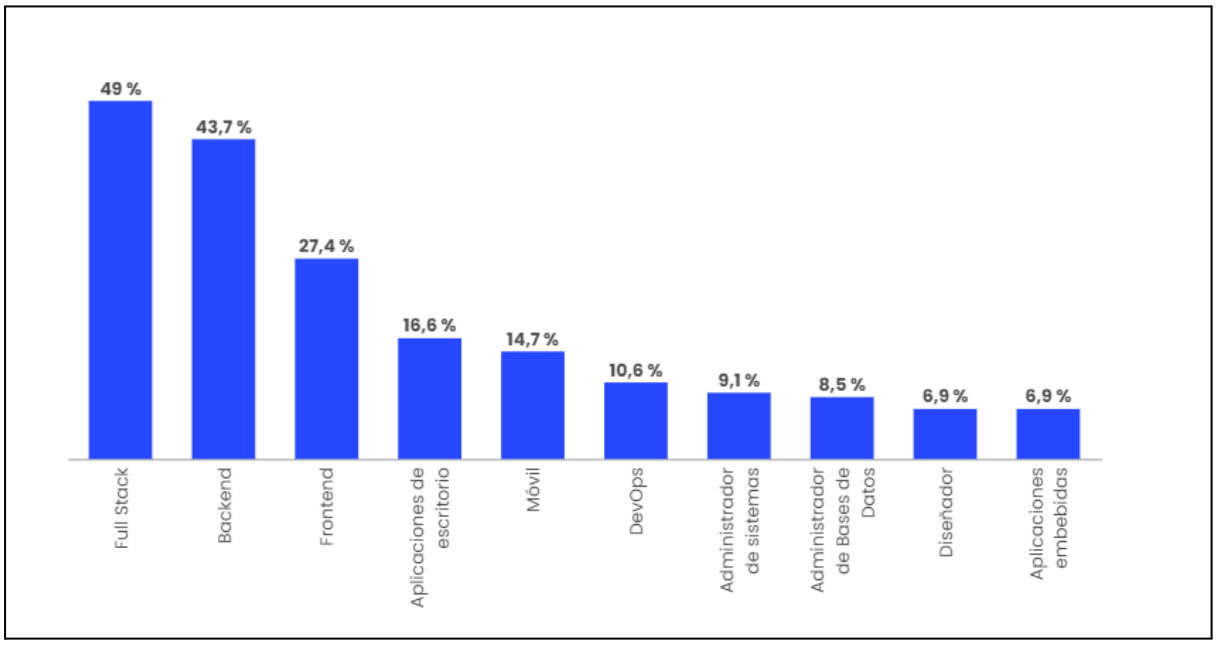



***Ilustración 4: Identificación de los desarrolladores de sus propios roles en 2021 (% de programadores que se identifican a sí mismos en un área).***

Se puede deducir tras ver la ilustración anterior las principales áreas en la que se engloban los programadores, son FullStack (desarrollador tanto de front-end como de back-end) y back-end, lo que nos hace ver la relevancia e importancia que tiene esta parte de los proyectos.

Además, estamos viviendo un momento de expansión en la que están surgiendo nuevos servicios web y empresas que necesitan realizar una buena lógica de su aplicación para garantizar un correcto funcionamiento de la misma.

Hoy en día, son numerosos los frameworks que surgen para el desarrollo en la aplicaciones del servidor (*backend*).

A continuación, enumeramos los más utilizados por la mayoría de empresas y analizaremos sus ventajas y desventajas.

1) Laravel


*Ilustración 5: Logo de Laravel*

Laravel es un framework implementado en PHP para desarrollar aplicaciones webs complejas que normalmente usan el modelo-vista-controlador. Las principales ventajas de este es su código ya que es muy simple e intuitivo para todo desarrollador. Además, cuenta con una gran cantidad de documentación lo que puede ayudar a dar soporte en los problemas que aparezcan durante el desarrollo. La principal desventaja es que posee librerías que dependen de Symphony, otro framework similar pero algo más complejo.[8]

2) Django


*Ilustración 6: Logo de Django*

Django es un framework de Python que se usa para el desarrollo de aplicaciones de alto nivel. Al igual que Laravel se basa en el modelo-vista-controlador. Las principales ventajas de Django es su fácil adaptación a tu aplicación, la escalabilidad del mismo y seguridad y velocidad que este proporciona. Las principales desventajas que posee es la diversidad de documentación ya que puede dar lugar a confusión y su complicada adaptación a las implementaciones de Api Rest. [10]

3) Express Js


*Ilustración 7: Logo de Express Js*

ExpressJS es un framework de Node.js que se usa para el desarrollo de la parte de servidor de aplicaciones web e implementación de APIS. Las principales ventajas que posee este framework son su curva de aprendizaje que es pequeña, ya que JavaScript actualmente es uno de los lenguajes más usados para el desarrollo de aplicaciones y ofrece un buen rendimiento. Además, ofrece un sistema de depuración para facilitar la detección de errores a los usuarios junto a una programación rápida al lado del servidor. La principal desventaja que posee Express es la falta de compatibilidad entre versiones, lo que conlleva realizar cambios en tu aplicación para mantener su correcto funcionamiento. [11]

4) Spring Framework

*Ilustración 8: Logo de Spring Framework*

Spring es un framework para el desarrollo de aplicaciones web fundamentado en Java, además de ser Open Source [12]. También ofrece una configuración para el desarrollo de aplicaciones empresariales apoyándose en Java junto a un patrón completo de programación. Esto es muy importante ya que ayuda a centrar todo el desarrollo en la parte relacionada con la lógica de negocio de la aplicación [13].

Spring es un framework muy utilizado por muchas empresas actualmente y sus principales características son las siguientes [14]:

- Es un framework muy flexible ya que ofrece una gran compatibilidad con una gran cantidad de aplicaciones.

- Mantiene la compatibilidad con versiones anteriores lo que hace que no se tenga que realizar muchos cambios entre una versión y otra.

- Posee de una cantidad bastante elevada de estándares para garantizar la calidad del código
- Ofrece un gran soporte para la integración de datos en una aplicación con JDBC, ORM o JMS entre otros.

- Posee una gran para servicios RestFul y aplicaciones web con modelo MVC.

- Otra característica muy interesante de Spring Boot es la inyección de dependencias lo que facilita mucho la comunicación en la aplicación y el desarrollo.

#### 2.1.2.2 Front-end

La parte de Front-end de una aplicación que permite realizar la comunicación entre el cliente y el servidor para que este pueda acceder a los recursos del servidor. Normalmente una aplicación de frontend está compuesta por una interfaz gráfica para el usuario o un desarrollo para el cliente que establece una conexión con la parte del servidor.[15]

Actualmente, hay una gran presencia de frameworks para el desarrollo de aplicaciones web por parte del cliente. A continuación, se explicará brevemente las características de los más populares y utilizados.

1) React


*Ilustración 9: Logo de React*

[16] React es una librería implementada en JavaScript para el desarrollo de interfaces gráficas para el usuario. Las principales características es que es una programación basada en componentes, es decir, que cada uno trabaja de forma independiente al resto y posee su propio estado. Además, al estar implementado en JavaScript puedes pasar datos entre los componentes de tu aplicación de forma sencilla

2) VueJs


*Ilustración 10: Logo de Vue.Js*

[17] Al igual que React, Vue es un framework para la implementación de interfaces para el usuario. Está fundamentado en JavaScript con la diferencia de que a la vez trabaja

con herramientas como HTML y CSS lo que le permite implementar aplicaciones de una forma rápida debido a que las herramientas en las que se apoya son muy conocidas por la mayoría de desarrolladores. Al igual que React está basado en la programación mediante componentes lo que permite realizar una comunicación en tu aplicación efectiva y eficaz.

3) Angular


*Ilustración 11: Logo de Ángular*

[18] Angular es una herramienta de desarrollo para aplicaciones web para la parte del cliente. Su programación está basada en componentes y las herramientas que usa son HTML,CSS y TypeScript. Angular es uno de los frameworks más usados actualmente por la mayoría de empresas ya que posee una serie de características que lo hacen muy atractivo para las mismas.

[19] La principal característica de Angular es el mantenimiento ofrecido por Google. Esto le hace tener una potencia muy grande ya que Google es una de las empresas más potentes que existe actualmente del mundo. Además, esto le proporciona una aceptación muy buena por parte de la comunidad, lo que hace que los desarrolladores quieran usarlo y a su vez, existe un buen soporte por la cantidad de desarrolladores que lo acaban utilizando.

También destacar la gran cantidad de documentación que existe para Angular y a su vez la cantidad de librerías existentes, lo que hace que el aprendizaje del mismo sea más sencillo y el uso de este framework posea una menor dificultad para el programador durante su inicio.

## 2.2 Propuesta de solución

En este punto se va a explicar la idea principal de la aplicación que se quiere implementar junto a las tecnologías y metodologías que se van a utilizar durante el desarrollo del proyecto.

Se quiere crear una aplicación que permita a los usuarios ver información sobre las diferentes pistas que disponga el polideportivo además de ver un calendario con la disponibilidad de estas para poder reservarlas.

Además, también se desea que los usuarios puedan leer noticias de interés sobre actividades que haya en el polideportivo junto a la posibilidad de poder apuntarse a eventos, que serán exclusivos.

Para realizar todo esto, se hará uso de la metodología y de las tecnologías que se han visto en el anterior apartado con la que implementaremos una aplicación dividida en dos, una aplicación web para el servidor y otra para el cliente que se comunicarán usando una API- REST.

Tras ver las diferentes características de varios de los frameworks más usados para desarrollar aplicaciones en la parte del servidor se ha optado por Spring, ya que ofrece una enorme variedad de opciones a la hora de realizar tu propia implementación y es un framework con el que hemos trabajado en el grado y me siento muy cómodo con él.

Por último, hemos decidido realizar la parte del cliente de la aplicación con Angular, ya que nos ha parecido muy atractivo e interesante. Tiene una gran potencia además de una gran comunidad que lo respalda junto a una tecnología como TypeScript, que actualmente también está presente en muchos proyectos. Por todo ello he decidido usar Angular como framework para la parte de frontend de este proyecto.

## 2.3 Requisitos

En este punto se explica detalladamente los requisitos tanto funcionales como no funcionales de la aplicación.

La idea principal reside en crear una aplicación que permita al usuario realizar reservas de las instalaciones deportivas del polideportivo asociado, que pueda consultar y ver una breve descripción de las pistas disponibles y tener un historial con las reservas realizadas.

- Requisitos funcionales

  A continuación, se exponen las funcionalidades básicas y restricciones que debe cumplir la implementación de esta aplicación.

  - (1) El sistema debe poseer un conjunto de funcionalidades básicas generales como la mayoría de aplicaciones web como el registro en la web, realizar login en la misma, visualizar su perfil, realizar logout…

  - (2) Tras darle al usuario el acceso a la aplicación, se le debe de proveer de las funcionalidades básicas del proyecto, como en este caso: poder ver las diferentes pistas disponibles del polideportivo, poder buscar las pistas según el deporte que el usuario quiera practicar dentro de los disponibles, además de poder realizar una consulta de las horas disponibles de una pista y realizar la reserva de la misma.

  - (3) También se debe incluir un usuario con rol Administrador para que además de poseer los mismos permisos que cualquier usuario, sea el encargado de realizar el mantenimiento y la gestión de la aplicación. A este tipo de usuario le corresponderá tareas como añadir pistas del polideportivo, editarlas o eliminarlas en caso de que sea necesario. Poder ver un listado de los usuarios registrados y acceder a su historial de reservas, junto a la opción de poder editarlos. Por último acceder al historial completo de las reservas realizadas por todos los usuarios para poder gestionarlas de una manera adecuada

- Requisitos no funcionales

En este punto se tratarán aquellos aspectos relacionados con conseguir una satisfacción con el funcionamiento del sistema para el cliente.

  - (4) El sistema debe de tener una capacidad de respuesta rápida hacia cualquier tipo de petición.

  - (5) La interfaz de la aplicación debe poseer una interacción muy intuitiva para facilitar el uso.

  - (6) El sistema deberá de ser seguro y no permitir que un usuario no pueda hacer acciones que no tengan permisos.

## 2.4 Planificación inicial de tareas

En este punto se procede a explicar como se ha hecho la planificación de las tareas para la realización de este proyecto. Para ello se han utilizado Historias de Usuario, ya que se ha decidido usar una metodología ágil con principios de SCRUM para la gestión del mismo.

### 2.4.1 Historias de Usuario

En este apartado vamos a realizar una descripción de las historias de usuario que se intentarán conseguir durante la implementación de este proyecto.

Así lo describe Max Rehkopf en [20] : "*Una historia de usuario es la unidad de trabajo más pequeña en un marco ágil. Es un objetivo final, no una función, expresado desde la perspectiva del usuario del software*".

Al utilizar una metodología ágil con ideas SCRUM, se usan las Historias de Usuario en vez de casos de uso para definir los requisitos [21]. A continuación en la ilustración 12 se muestra la estructura que debe seguir una historia de usuario.

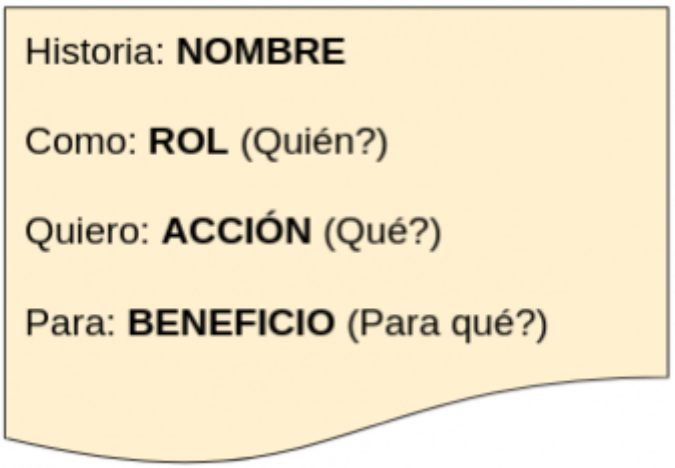



*Ilustración 12: Ejemplo de estructura de Historia de Usuario*

Para reflejar la importancia de las Historias de usuario se va a usar el método MosCow en el que las prioridades posibles serán las siguientes:

- ***Must Have (M)***
- ***Should Have (S)***
- ***Could Have(C)***
- ***Won't Have (W)***

Hay varios métodos para realizar la estimación de las historias de usuario, en mi caso he usado Puntos de Historia de la siguiente forma: 1,2,3,5,8. Hay muchos más pero para esta aplicación con este rango es más que suficiente.

Las historias de usuario quedan reflejadas en la siguiente tabla:

| Número | Nombre | H.U | Requisitos | Estimación | Prioridad | Criterio de aceptación |
|---|---|---|---|---|---|---|
| **1** | Registro | Como usuario quiero poder realizar login en la aplicación para acceder a la web | 1,4 | 8 | M | El usuario registrado debe aparecer en la base de datos.<br><br>Los campos del usuario deben tener el formato adecuado, en caso contrario se mostrará un error. |
| **2** | Login | Como usuario quiero poder acceder a la página web mediante los datos indicados en el registro para iniciar sesión en la web | 1,6 | 8 | M | El usuario deberá poder navegar por la web de forma correcta en caso de tener acceso a las páginas que quiera acceder. |
| **3** | Editar usuario | Como usuario quiero poder modificar mis datos para cambiarlos cuando lo necesite. | 1,3,4 | 3 | S | En la base de datos debe aparecer el usuario con los campos actualizados. |

| | | | | | | |
|---|---|---|---|---|---|---|
| **4** | Listar reservas | Como usuario quiero poder ver todas las las reservas que he hecho para tener un historial de las mismas | 2,4 | 3 | M | Dado un usuario, debe aparecer la lista con las reservas que este ha realizado, en caso contrario se mostrará un error. |
| **5** | Listar usuarios | Cómo administrador quiero poder ver una lista de los usuarios existentes. | 2,4 | 3 | S | Dado un administrador, al pulsar sobre listar usuarios deben aparecer la lista de usuarios registrados. |
| **6** | Eliminar usuario | Como administrador quiero poder eliminar usuarios. | 2,3 | 2 | S | Dado un administrador, al pulsar sobre eliminar usuario debe desaparecer de la base de datos. |
| **7** | Listar pistas | Como usuario quiero ver las pistas disponibles para ver sus características. | 2,4 | 3 | M | Dado un usuario, al pulsar sobre reservar deben aparecer la lista de pistas existentes. |
| **8** | Filtrar pista por tipo | Como usuario quiero poder filtrar las instalaciones según un deporte determinado. | 2,4,5 | 3 | C | Al introducir el tipo de vista, se deben mostrar las diferentes pistas de ese tipo. |

| | | | | | | |
|---|---|---|---|---|---|---|
| **9** | Ver detalles de pista | Como usuario quiero ver los detalles de la pista para saber si me interesa o no. | 2,4,5 | 2 | S | Dada una pista, al poner el cursor sobre ella se deben ver los detalles de la misma. |
| **10** | Añadir pista | Como administrador quiero poder añadir una pista. | 2,3,6 | 2 | S | Dada una pista, al añadirla debe aparecer en la base de datos. |
| **11** | Editar pista | Como administrador quiero poder editar una pista para cambiar datos anómalos. | 3,5,6 | 2 | C | En la base de datos debe aparecer la pista con los campos actualizados. |
| **12** | Eliminar pista | Como administrador quiero poder eliminar una pista. | 3,5,6 | 2 | S | Dado un administrador, al pulsar sobre eliminar una pista debe desaparecer de la base de datos. |
| **13** | Realizar Reserva de una pista | Como usuario quiero realizar la reserva de una pista. | 1,2,4,5 | 8 | M | Debe aparecer en la base de datos la reserva realizada.<br>Dado un usuario, debe tener asociada la reserva realizada. |
| **14** | Recibir | Como usuario quiero | 1,4 | 5 | M | Dado un usuario, debe recibir un |

| | | | | | | |
|---|---|---|---|---|---|---|
| | email de confirmación | recibir un email de confirmación para obtener la confirmación de la misma. | | | | correo electrónico con los datos de la reserva realizada. |
| **15** | Listar reservas | Como administrador quiero poder ver una lista de las reservas realizadas por todos los usuarios. | 2,4 | 3 | S | Dado un administrador, al pulsar sobre ver reervas deben aparecer la lista de reservas realizadas por todos los usuarios. |
| **16** | Ver noticias | Como usuario quiero poder ver las noticias. | 2,4 | 2 | S | Dado un usuario, al pulsar sobre reservar deben aparecer la lista de pistas existentes. |
| **17** | Añadir noticia | Como administrador quiero poder añadir las noticias para que el usuario tenga noticias actuales. | 2,4,5,6 | 2 | S | Dada una noticia, al añadir una noticia debe aparecer en la base de datos. |
| **18** | Editar noticia | Como administrador quiero poder editar las noticias para corregir | 3,5,6 | 2 | C | Dado un administrador, al editar la noticia en la base de datos debe aparecer la noticia con los campos |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | errores | | | | actualizados. |
| **19** | Eliminar noticia | Como administrador quiero poder añadir las noticias para que el usuario no tenga noticias desfasadas. | 3,5,6 | 2 | C | Dado un administrador, al pulsar sobre eliminar una noticia debe desaparecer de la base de datos. |
| **20** | Apuntarse a evento | Como usuario quiero poder apuntarme a eventos para hacer actividades extraordinarias. | 1,4,5 | 2 | C | Dado un usuario, al apuntarse a un evento debe de aparecer en su lista de eventos.<br>Dado un evento, debe disminuir su capacidad. |
| **21** | Ver lista de eventos apuntados | Como usuario quiero ver una lista con todos los eventos disponibles para saber cuales me interesan | 2,4,6 | 2 | C | Dado un usuario, al pulsar sobre eventos deben aparecer la lista de pistas existentes. |
| **22** | Añadir evento | Como administrador quiero poder añadir eventos para que los usuarios se apunten. | 3,5,6 | 2 | C | Dado un evento, al añadir debe aparecer en la base de datos. |

| | | | | | | |
|---|---|---|---|---|---|---|
| **23** | Editar evento | Como administrador quiero poder editar eventos para que el usuario no tenga eventos anómalos | 3,5,6 | 2 | C | Dado un administrador, al editar el evento en la base de datos debe aparecer la noticia con los campos actualizados. |
| **24** | Eliminar evento | Como administrador quiero poder eliminar un evento. | 35,6 | 2 | C | Dado un administrador, al pulsar sobre eliminar un evento debe desaparecer de la base de datos. |
| **25** | Pago de reserva | El usuario deberá tener una plataforma de pago para garantizar el correcto pago de una reserva | 1,4,5 | 5 | C | La reserva deberá de ser realizada correctamente tras finalizar la simulación del pago. |

*Ilustración 13:Tabla de las Historias de Usuario*

### 2.4.2 Planificación inicial de las tareas

En este punto se va proseguir con la idea de aplicar técnicas de SCRUM para definir las tareas. Tras definir las historias de usuario, se va a realizar un desglose de tareas por cada una. Además se va a realizar un posible planteamiento del número de iteraciones (sprints) y la duración de los mismos.

Tras realizar la planificación de todos los sprints obtendremos una estimación sobre el tiempo de desarrollo del proyecto.

Inicialmente, hay un total de 79 puntos de historia para realizar y la fecha para finalizar el proyecto la estimaremos en 3 meses aproximadamente.

La duración de cada sprint va a ser de dos semanas, de forma que habrá 6 sprints en total, con un total de 50 horas de trabajo en cada sprint.

Para elegir las historias de usuario que se incluyen en cada sprint se tendrá en cuenta la prioridad asignada anteriormente, de mayor a menor, además de tener en cuenta las historias de usuario dependientes unas de otras. Por ejemplo, no se puede realizar el envío de confirmación de una reserva sin antes haber realizado la historia de usuario que se encarga de realizar la reserva.

Para elegir el ritmo de trabajo en cada sprint y finalizamos puntos de historia existen dos vertientes:

- La primera opción consiste en estimar la velocidad del sprint en base a la experiencia adquirida por la presencia en otros proyectos, de forma que así podríamos hacer una aproximación real del mismo.

- La segunda opción posible es realizar una estimación de la velocidad en base a la visión que se tenga del proyecto, tanto en duración como en dificultad.

Tras ver las dos posibles opciones para realizar la estimación de la velocidad de los sprints se ha escogido la segunda opción, puesto que no se tiene la suficiente experiencia en otros proyectos en los que se haya aplicado esta metodología de trabajo para poder

hacerlo de forma adecuada. Por ello, se va a llevar a cabo una estimación realista aunque con carácter positivo entendiendo que la velocidad serán los puntos de historia planeados en cada sprint. Por ello, sabiendo que hay 79 Puntos de Historia y 6 sprints, por cada sprint se deberán completar aproximadamente 13 puntos de historia.

Para indicar de las historias de usuario que componen este primer sprint, a continuación se va a mostrar una tabla con los índices de cada Historia de usuario junto al desglose de tareas que tiene cada una y las estimaciones iniciales realizadas anteriormente.

| **Registro (HU 1)** | |
|---|---|
| **Tareas** | **Descripción** |
| **1** | Iniciar proyecto del servidor (Configuración Inicial de Spring, Bases de datos…) |
| **2** | Iniciar proyecto cliente web (crear proyecto, dependencias Angular, enrutamiento, paquetes..) |
| **3** | Realizar conexión entre ambos proyectos |
| **4** | Realizar modelado del Usuario en el servidor y correspondiente mapeo en la base de datos |
| **5** | Servicios y controladores de usuario en la parte de frontend. |
| **6** | Crear vista para registrar usuarios |
| **7** | Implementar funcionalidad para realizar registro de un usuario |

*Ilustración 14: Tareas de Historia de Usuario Registro*

| Login (HU 2) | |
|---|---|
| **Tareas** | **Descripción** |
| **1** | Aprender funcionamiento de JWT como método de autenticación. |
| **2** | Realizar implementación de la autenticación en la parte de backend. |
| **3** | Implementar servicio/controlador/repositorio para hacer la comprobación de los usuarios mediante el token de autenticación. |
| **4** | Implementar autenticación en la parte de frontend. |
| **5** | Implementar funcionalidad de login en la parte del cliente |
| **6** | Crear vista de login |

*Ilustración 15: Tareas de Historia de Usuario Login*

Para este primer sprint, se ha optado por realizar más puntos de historia debido a que la parte del registro y login están muy relacionadas entre ellas para poder comenzar a realizar el logueo y registro en la aplicación de forma correcta, en los siguientes sprints se regulará esto. Finalmente el primer sprint queda de la siguiente manera:

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **1** | Registro | 8 | M |
| **2** | Login | 8 | M |

*Ilustración 16:Selección de HU para Sprint 1*

Una vez que se ha finalizado el primer sprint, se puede observar como todo lo relacionado con la configuración inicial del proyecto ahora los siguientes sprints se van a centrar en el desarrollo de nuevas funcionalidades.

Para los demás sprints se procederá de la misma forma. En primer lugar, implementaremos en el servidor la funcionalidad correspondiente en los respectivos servicios, repositorios y controladores y tras ello implementaremos el controlador del mismo en la parte del cliente y crearemos la vista para ello, por lo que el proceso es muy parecido en todas las entidades.

Como se observa a continuación, aunque la estimación media por sprint debe de ser de 13 Puntos de Historia, hay dos sprints con un valor de puntos de historia mayor. Esto es debido a que el primer sprint se corresponde con la inicialización del proyecto y son Historias de Usuario que están muy relacionadas de forma que aunque sea una mayor carga de trabajo estaríamos trabajando en aspectos muy similares.

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **3** | Editar Usuario | 3 | S |
| **5** | Listar Usuarios | 3 | S |
| **6** | Eliminar Usuario | 2 | S |

*Ilustración 17:Selección de HU para Sprint 2*

En el segundo sprint se van a abordar aquellas Historia de Usuario relacionadas con la gestión de usuarios ya que una vez hemos implementado el registro y login de la aplicación, podremos comprobar que funcionan correctamente. Además, al tener un sprint con un valor de 8 Puntos de Historia podremos realizar tareas análisis, documentación y pruebas además de tener una posibilidad de reajuste porque existe un pequeño margen.

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **7** | Listar Pistas | 3 | M |
| **8** | Filtrar tipo pista | 3 | C |
| **9** | Ver detalle pista | 2 | S |
| **10** | Añadir pista | 2 | S |
| **11** | Editar pista | 2 | C |
| **12** | Eliminar pista | 2 | S |

*Ilustración 18:Selección de HU para Sprint 3*

En el tercer sprint se ha escogido aquellas historias de usuario relacionadas con las pistas. En este caso los puntos de historia abordados son mayor que la media, pero en este caso es debido al número de Historias de Usuario elegidas más que por el coste de las mismas, ya que a priori tienen una estimación menor.

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **13** | Realizar reserva de una pista | 8 | M |
| **14** | Recibir email de confirmación | 5 | M |
| **25** | Pago con Stripe | 5 | S |

*Ilustración 19:Selección de HU para Sprint 4*

En el cuarto sprint, y su vez el que abarca un mayor número de Puntos de Historia, se han escogido las Historias de Usuario relacionadas con la realización de las reservas. En

este caso, el esfuerzo requerido será mayor que en otros sprints debido a que en él se concentra la funcionalidad más importante de la aplicación y el principal objetivo de la misma.

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **4** | Listar reservas | 3 | M |
| **15** | Listar reservas | 3 | S |
| **16** | Ver noticias | 2 | S |
| **17** | Añadir noticia | 2 | S |
| **18** | Editar noticia | 2 | C |

*Ilustración 20:Selección de HU para Sprint 5*

| Número HU | Nombre H.U | Estimación Asignada | Prioridad Asignada |
|---|---|---|---|
| **19** | Eliminar noticia | 2 | C |
| **20** | Apuntarse a evento | 2 | C |
| **21** | Ver lista de eventos apuntados | 2 | C |
| **22** | Añadir evento | 2 | C |
| **23** | Editar evento | 2 | C |
| **24** | Eliminar evento | 2 | |

*Ilustración 21:Selección de HU para Sprint 6*

Finalmente, en los dos últimos sprints se han elegido las Historias de Usuario restantes que están relacionadas principalmente con eventos y noticias. Son Historias de Usuario más sencillas y similares a las ya trabajadas en los anteriores sprints de forma que podremos aprovechar estos dos últimos sprints para reajustar e invertir más tiempo en aquellas tareas que no son de implementación.

En la siguiente tabla se van a mostrar a modo de resumen los diferentes sprints que se van a realizar junto a las Historias de usuario que van a acaparar cada uno. Viendo que tenemos 6 sprints la duración total va a ser de 300 horas.

| Sprint | Índices H.U | Puntos de Historia |
|---|---|---|
| **1** | 1,2 | 16 |
| **2** | 3,5,6 | 8 |
| **3** | 7,8,9,10,11,12 | 14 |
| **4** | 13,14, 25 | 18 |
| **5** | 4,15,16,17,18 | 12 |
| **6** | 19,20,21,22,23,24 | 12 |

*Ilustración 22:Sprints planificados para el desarrollo*

## 2.4.3.- Evolución de la planificación inicial

Una vez hemos comenzado con la implementación del proyecto hemos avanzado correctamente hasta que hemos llegado a la HU del pago con Stripe, debido a que han surgido problemas de implementación una vez encomendada y ver los problemas se decidió prescindir de ella ya que la prioridad de la misma es Should.

Por contratiempo se ha hecho la funcionalidad de administrador a la gestión de pistas en la parte del cliente aunque esté implementada en el servidor.

## 2.5 Estudio de viabilidad

En este punto se va a realizar un estudio sobre los costes a los que tendremos que hacer frente a la hora de desarrollar nuestra aplicación. Para ello tendremos en cuenta IDEs utilizados, nóminas de trabajadores, horas de trabajo y todo tipo de tecnología utilizada, tanto hardware como software.

## 2.5.1 Análisis de costes

Hay que tener en cuenta el equipo de trabajo del que se va a disponer para el desarrollo de esta aplicación. En este caso la plantilla de trabajo estará compuesta por un único integrante que será el autor del presente trabajo. Como herramienta de trabajo se ha utilizado el sistema operativo Windows 11. A su vez, como entorno de desarrollo se ha utilizado IntelliJ para la parte del servidor y Visual Studio Code para la parte del cliente. Como base de datos se ha utilizado H2 database.
A continuación, se explica cada uno detalladamente.

### 2.5.1.1 Costes de personal

Como se ha indicado anteriormente, la plantilla de desarrollo de este proyecto solo estará compuesta por una persona. Para estimar el coste que requerirá el trabajo hay que tener en cuenta las características del trabajador.
En este caso se corresponde con un estudiante en Ingeniería Informática, concretamente en la rama de Ingeniería del Software sin experiencia en el mundo laboral. Teniendo en cuenta esto, el salario bruto medio que percibe un trabajador con estas características es de unos 21 306€ aproximadamente. [22]. Sabiendo esto, obtenemos los siguientes datos:

| Sueldo Bruto | Sueldo Neto | Sueldo por Hora | Duración Proyecto (horas) |
|---|---|---|---|
| 21.306€ | 17.044,8€ | 10,23€ | 300 |
| | | **Total** | 3609€ |

*Ilustración 23:Tabla de costes de personal*

Tras realizar una pequeña estimación observamos como el coste de personal es de unos 3,609€ aproximadamente, a esto hay que añadirle los costes que nos supone mantener el

trabajador para el estado en todo el tema de impuestos,sabiendo que el coste de cubrir los gastos sociales del trabajador es de un 30% de su sueldo bruto de forma que finalmente nos quedamos con un total de unos 4.691,7€ .[23]

#### 2.5.1.2 Costes en tecnologías

A continuación, vamos a dividir los costes tecnológicos en dos apartados: un primer apartado para la tecnología relacionada con el software y otro para la tecnología relacionada con los elementos hardware que necesitemos.

##### 2.5.1.2.1 Costes Software

Para poder estimar el coste software que vamos a tener en este proyecto debemos de tener en cuenta el tipo de suscripción que vamos a utilizar de las diferentes tecnologías que se usan y tener el cuenta el tiempo de uso de las mismas, ya que tendremos que hacer una correspondencia para adaptarla a la duración que va a tener este proyecto. Teniendo esto en cuenta y sabiendo que nuestro proyecto aproximadamente va a durar alrededor de unos 3 meses, los costes de software son los siguientes:

| Nombre | Distribuidor | Suscripción | Precio |
|---|---|---|---|
| **Windows 10** | Microsoft | Si | 48,90€ |
| **IntelliJ Community Edition** | JetBrains | No | 0€ |
| **Visual Studio Code** | Microsoft | No | 0€ |
| **H2 Database** | H2 | No | 0€ |
| **Adobe** | Adobe | No | 0€ |
| **Google Docs** | Google | No | 0€ |
| **OpenJDK** | Java | No | 0€ |
| **Angular** | Google | No | 0€ |
| **Spring** | SpringSource | No | 0€ |

| | |
|---|---|
| **Total** | 48,90€ |

*Ilustración 24:Tabla de costes software*

### 2.5.1.2.2 Costes hardware

Por último, se va a calcular el coste hardware de nuestra aplicación. Para ello tendremos en cuenta el ciclo de vida de los mismos, junto al número de meses de uso de los mismos para hacer los cálculos. En este caso se ha elegido 3 meses para poder realizarlos:

| **Nombre** | **Distribuidor** | **Estimación** | **Precio** |
|---|---|---|---|
| **Portátil Lenovo Ideal3** | Amazon | 6 años | 19,44€ |
| **Monitor Huawei AD80** | Amazon | 10 años | 4,33€ |
| | | **Total** | 23,77€ |

*Ilustración 25:Tabla de costes hardware*

### 2.5.1.3 Coste Total

Finalmente, tras realizar el estudio de los costes totales que supondrán este proyecto, obtenemos la siguiente tabla resumen con los diferentes costes de cada área

| **Nombre** | **Coste** |
|---|---|
| Costes de personal | 4.691,7€ |
| Costes de software | 48,90€ |
| Costes de hardware | 23,77€ |
| **Total** | 4.764,37€€ |

*Ilustración 26:Tabla resumen de costes del proyecto*

## 2.6 Modelo de dominio

Para finalizar el análisis del proyecto, se procede a realizar un modelo de dominio del sistema a partir de las Historias de Usuario descritas anteriormente (Punto 2.4.1)

El modelo de dominio es un diagrama que sirve para mostrar a los desarrolladores del proyecto las clases conceptuales que componen el problema junto a sus atributos, cómo se relacionan entre ellas y muestra el propio dominio del problema. [19].

Se puede entender como el inicio para el diseño de la aplicación. Este ofrece una visión general de las entidades que forman el proyecto, para así crear el sistema para el cliente.

En la siguiente ilustración se puede observar el modelo de dominio de la aplicación:

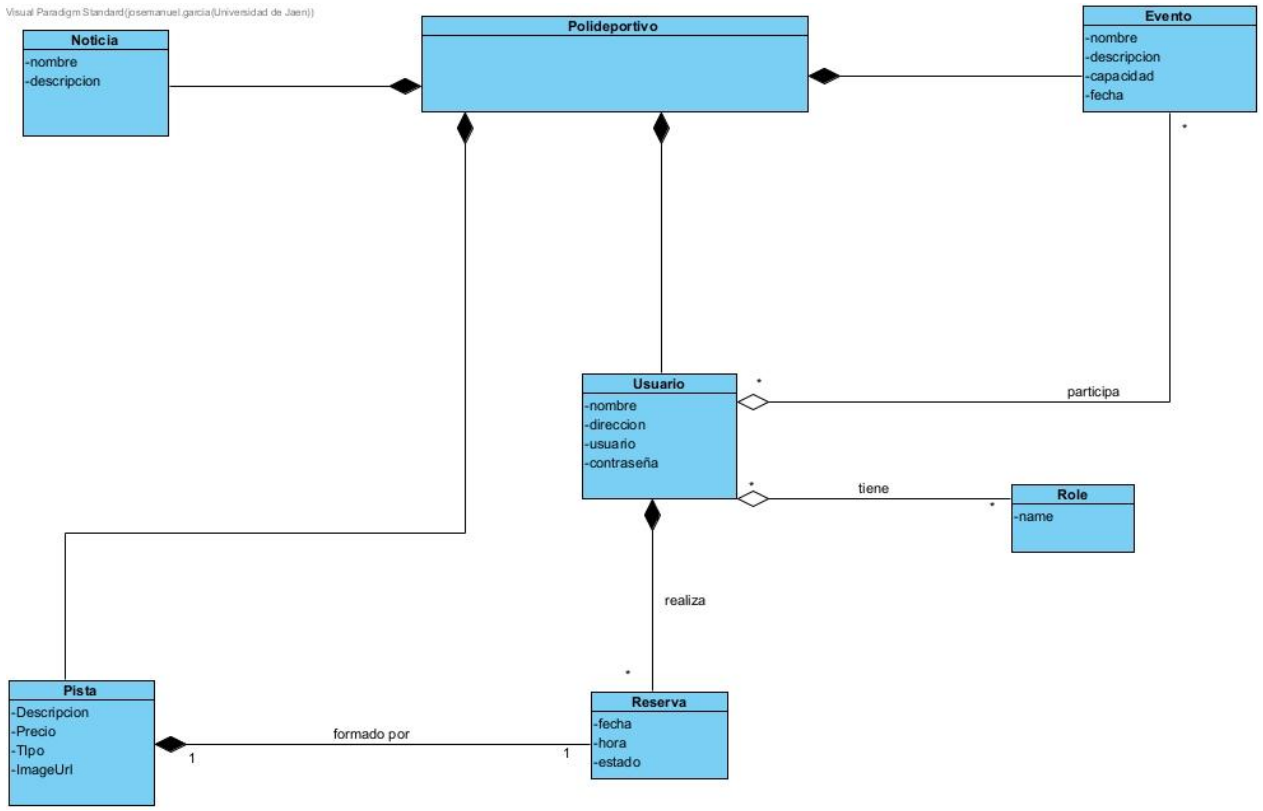



*Ilustración 27:Modelo de dominio*

Como se puede observar, la clase principal es Reserva, ya que se encargará de relacionar las entidades Pista y Usuario para poder realizar las mismas reservas. Cada reserva tendrá asociado un usuario que será quien la realiza y una pista que será la que se reserve.

A su vez, otra entidad que podemos observar es Evento. Un usuario podrá participar en muchos eventos y a su vez en un evento participan muchos usuarios.

Por último, existe la entidad noticia que no está relacionada con nadie puesto que solo servirá para mostrar información en la aplicación

# 3.- Diseño

En este apartado se va a proceder a realizar la etapa de diseño del proyecto. Una vez tenemos el análisis del proyecto procederemos a crear los diferentes diagramas a partir del modelo de dominio además de indicar el diseño de la interfaz gráfica, de la API Rest y por último un plan de pruebas.

## 3.1 Diagrama Entidad-Relación

En este apartado se va a mostrar y explicar el diagrama de entidad-relación obtenido a partir del modelo de dominio anterior.

Como podemos observar ha habido modificaciones en las tablas ya que noticias ha desaparecido y ha aparecido una tabla intermedia entre Usuario y Role. Esto ocurre ya que nos aseguramos de que así se aplica la 3º forma normal.

Además, como ORM hemos utilizado JPA ya que nos ayuda a realizar la implementación de forma más rápida y se integra muy bien. Destacar que gracias a esto, las tablas finales se generarán de forma automática a partir de las entidades y las relaciones presentes en el modelo que vamos a ver a continuación.

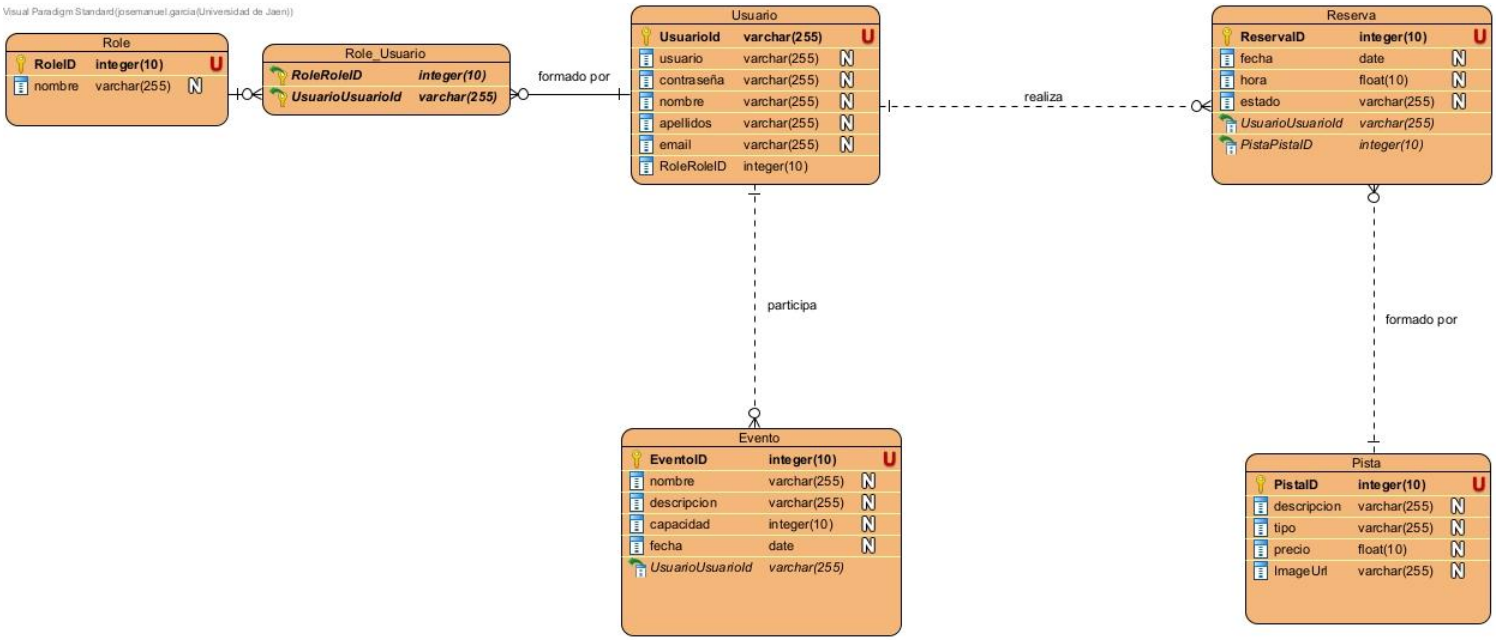



*Ilustración 28: Diagrama entidad-relación*

## 3.2 Diagrama de clases

En este apartado vamos a realizar el diseño de los diagramas de clases del proyecto.

Este punto se ha dividido en dos apartados, uno para la parte del servidor y otro para la parte del cliente ya que de esta forma podremos ver de forma más específica cómo se relacionan las clases entre ellas y cómo interaccionan.

### 3.2.1 Diagrama de clases del servidor

Para el desarrollo de la aplicación en la parte del servidor hemos utilizado una arquitectura hexagonal. Una arquitectura hexagonal es una arquitectura en la que se intenta asegurar que las partes que componen nuestro código se encuentren lo más desacopladas entre sí, es decir, que sean lo más independientes posible. Esta arquitectura se compone de tres niveles, infraestructura, aplicación y dominio.[27]

Esto posee una serie de ventajas entre las que destacan

- Facilita la inyección de dependencias
- Facilitar la realización de testing
- Permite diferenciar la lógica de negocio de la capa de infraestructura.

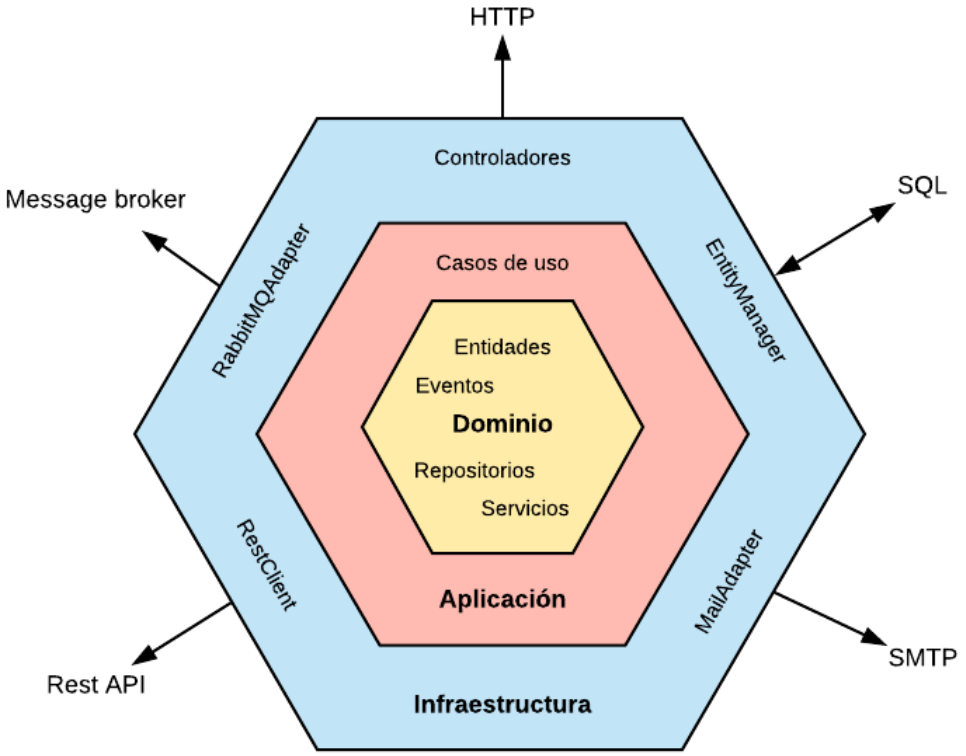



**Ilustración 29: Modelo de arquitectura hexagonal.**

Se va a mostrar la clases que componen tanto el *domain* como la *application*. En la arquitectura hexagonal se entiende application se corresponden con los servicios a los que va a acceder nuestra API y el Domain va a incluir la lógica de negocio de nuestra aplicación, concretamente las entidades.[23]

A continuación se va a mostrar las clases que componen los paquetes Domain y Application

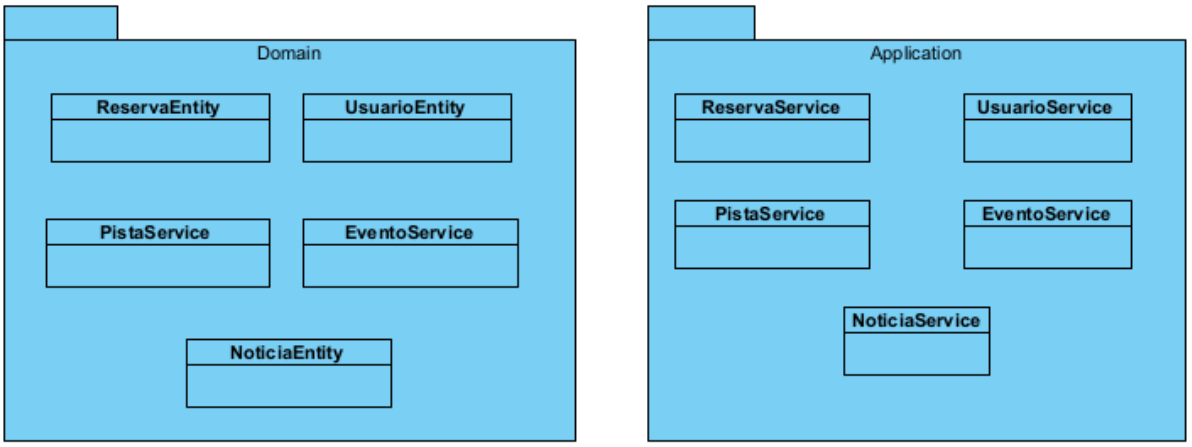



*Ilustración 30:Clases que componen los paquetes Domain y Application*

Una vez vistas las clases que componen los paquetes Domain e Infraestructure vamos a ver la distribución de paquetes de Infraestructure. La infraestructure se corresponde con los elementos externos a los que se va a conectar el sistema. En este caso estará compuesto por los controladores de cada entidad además de a su vez añadir los repositorios de cada una, que serán los que nos permitan realizar operaciones con la base de datos.

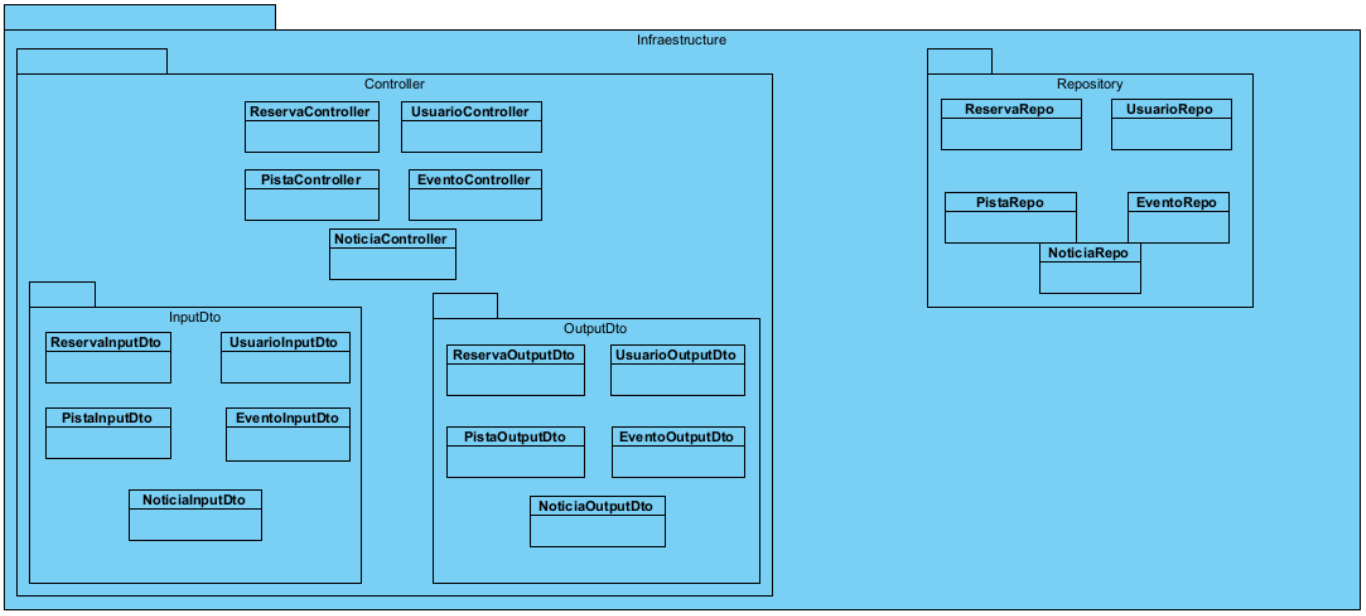



*Ilustración 31:Clases que componen el paquete infraestructure*

Para mostrar la relación con los controladores se van a considerar que solo tenemos uno por entidad pero realmente cada controlador está dividido en 4 controladores más pequeños para dividir entre los diferentes tipos de endpoints.

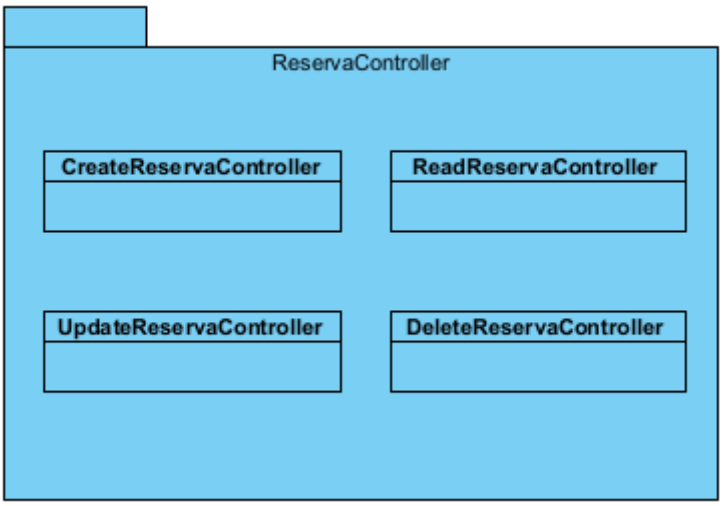



*Ilustración 32:Clase de controladores según endpoints*

En primer lugar vamos a ver cómo se relacionan los input y output con los controladores. Cada controlador tendrá como parámetro de entrada un input y a su vez devolverán un output con los datos de la entidad, de forma que en este caso cada controlador estará relacionado con los inputs de su propio objeto.

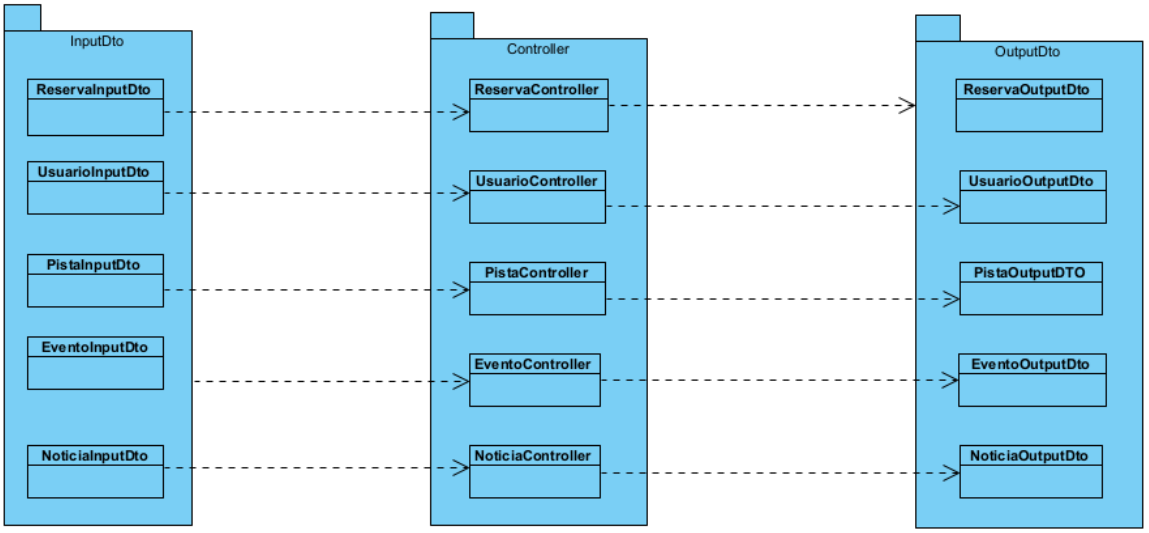



*Ilustración 33:Relación entre InputDto, Controllers y OutputDto*

A su vez y de la misma forma que en el diagrama anterior, los controladores llamarán a sus correspondientes servicios ya que serán los que tengan toda la lógica de cada uno y se encarguen de relacionarse con los modelos y los repositorios.

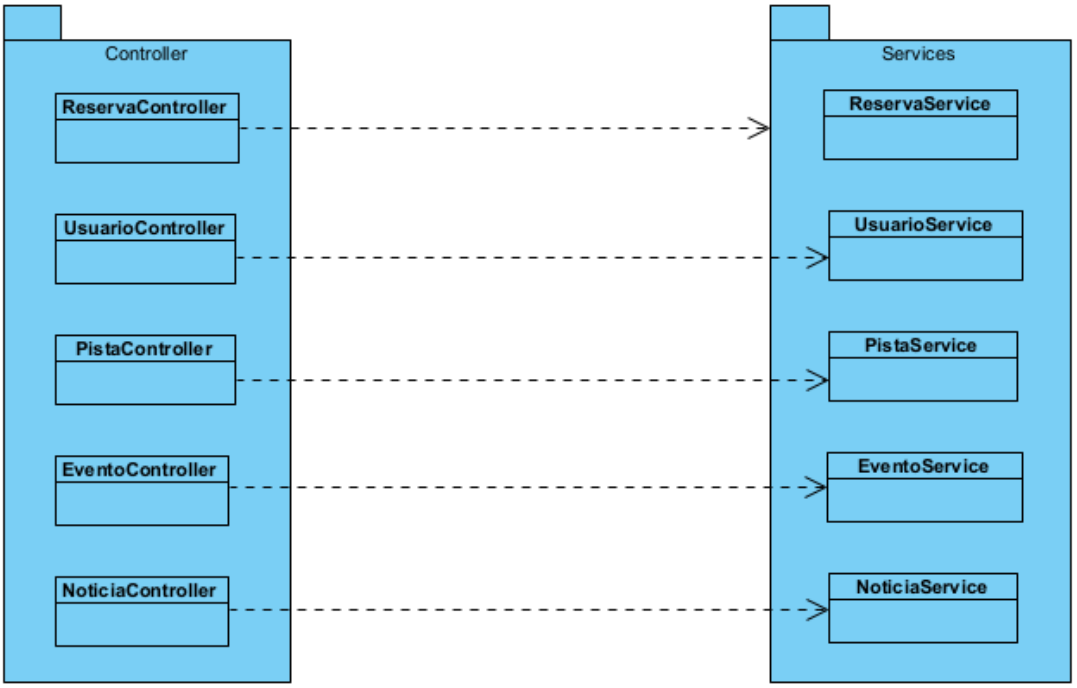



*Ilustración 34:Relación entre controllers y Services*

Una vez visto esto vamos a ver cómo se relacionan los servicios con los repositorios. Como veremos a continuación ReservaService es el servicio que se relaciona con más repositorios debido a que es el que contiene la lógica principal del proyecto, la realización de reservas.

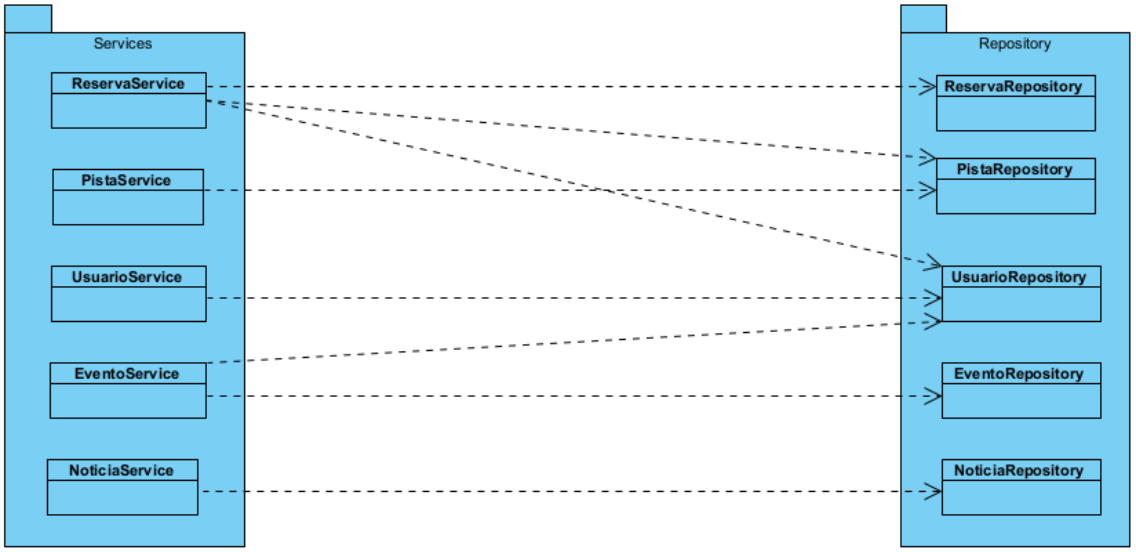



*Ilustración 35:Relación entre Services y Repository*

De la misma manera se relacionan los servicios con los modelos ya que a partir de los repositorios es cómo los obtenemos de la propia base de datos, salvo las pistas ya que dentro de ellas existen las propias reservas.

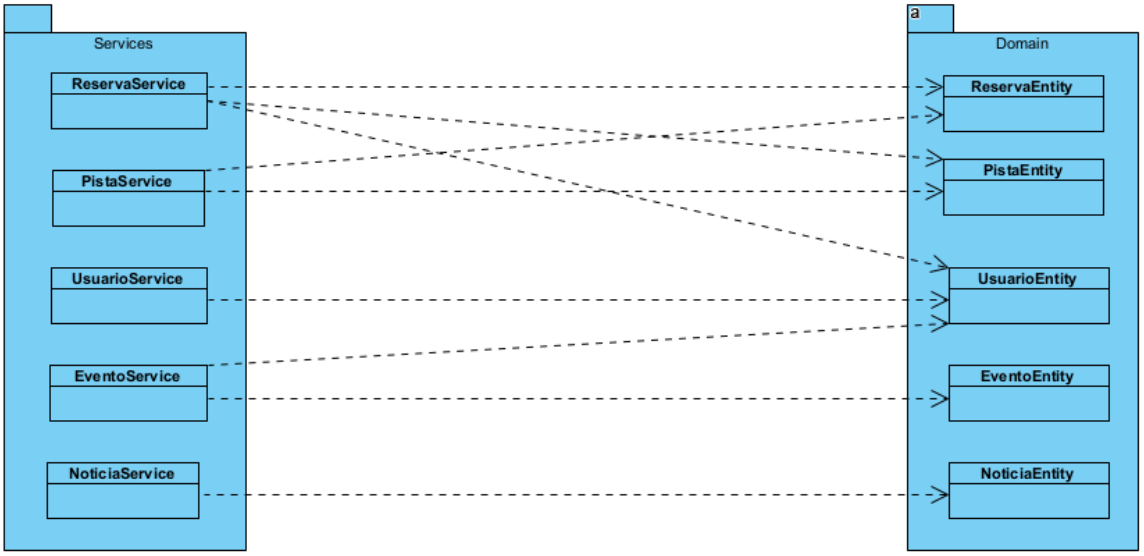



*Ilustración 36 Relación entre Services y Domain*

Por último, vemos cómo se relacionan los repositorios con los entity de la carpeta domain. Todos se relacionan consigo mismos excepto ReservaRepository que además de si mismo también lo hace con PistaEntity.

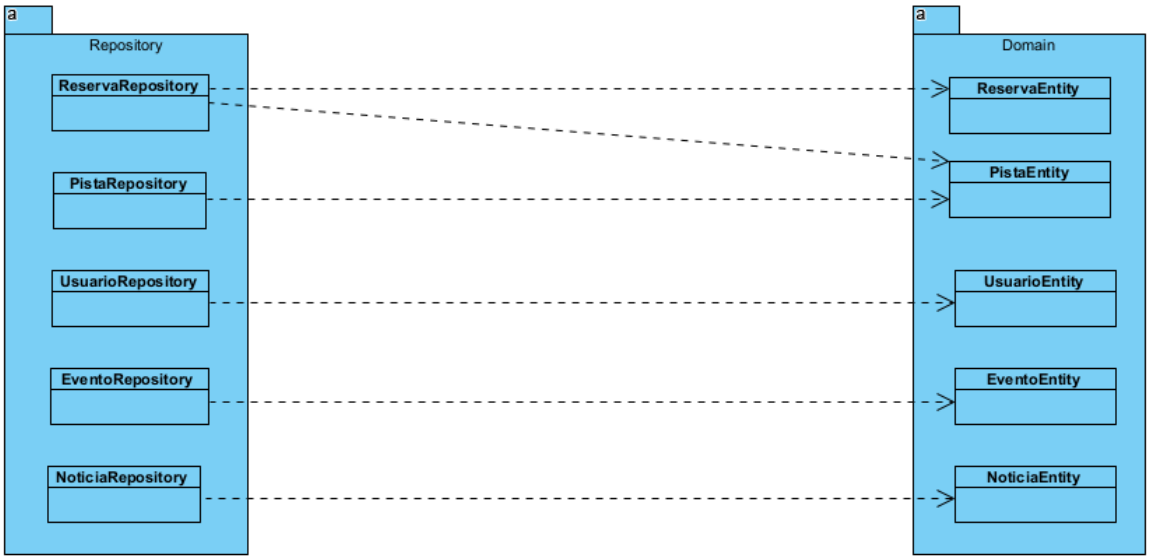



*Ilustración 37:Relación entre Repository y Domain*

### 3.2.1 Diagrama de clases del cliente

En este apartado se va a explicar y mostrar el diagrama de clases de la parte del cliente.El modelo seguido ha sido una programación basada en componentes de forma que cada uno actuará como una clase, lo que permite una mejor encapsulación de las mismas. En primer lugar se va a explicar qué representa cada cosa y posteriormente se mostrará el diagrama.

En este diagrama se van a poder distinguir clases de dos colores y que heredan de una principal, las de color naranja y las de color azul. Las de color naranja se van a corresponder con servicios y van a extender de Injectable mientras que las azules son aquellas que extienden de component.

- **Servicios (naranja)**: Se corresponden con los servicios de la aplicación y van a tener la anotación Injectable. Son los que se van a encargar de compartir información por toda la aplicación y relacionar la comunicación con la parte del cliente.

- **Componentes (azul)**: Se corresponden con componentes y tienen la anotación Component. Estos van a ser la vista de nuestra aplicación. A su vez van a contener toda la lógica de la misma junto a su fichero personalizado de estilos y van a acceder a los servicios para adquirir información. Cabe destacar que aunque no esté puesto todos los componentes están relacionados con Router que es un servicio que nos va a permitir realizar la navegación entre las diferentes vistas aunque posteriormente explicaremos su funcionamiento.

A su vez, la programación orientada a componentes facilita mucho el desarrollo y la inyección de librerías necesarias, como en AuthService ya que contiene todo lo relacionado con la gestión de la autenticación de la aplicación. A su vez gracias a la independencia entre componentes LoginComponent puede usarlo y junto al servicio anterior gestionar el almacenamiento del token en toda la aplicación.

Una vez explicado esto vamos a ver a continuación como ha quedado el diagrama de clases de la parte del cliente:

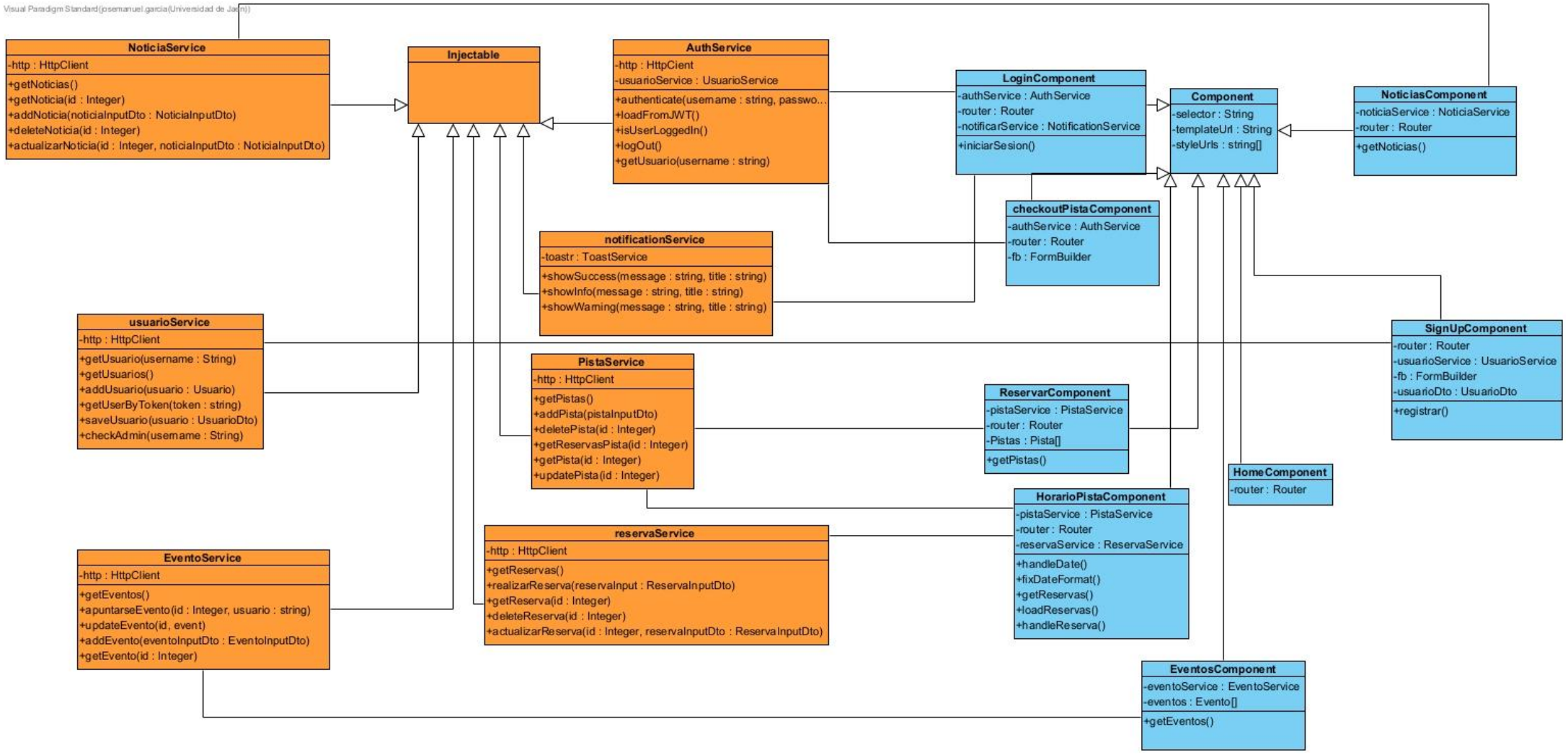



*Ilustración 38: Diagrama de clases del cliente*

## 3.3 Diagramas de secuencia

La principal función de esta sección es entender cómo funciona la aplicación explicada a partir de los diagramas que se han diseñado anteriormente. Para ello se han elegido aquellos que establecen las bases del proyecto y además proporcionan una visión general de forma que se puede deducir el funcionamiento de las demás partes del proyecto.

El proceso de explicación se basa en mostrar en primer lugar el diagrama de secuencia que se va a desarrollar para posteriormente comentar el mismo.

### 3.3.1 Inicio de sesión

Hay que tener en cuenta que la ventana de iniciar sesión será visible cuando se pulse en la opción Iniciar sesión, o se intente realizar una reserva o apuntarse a un evento y no se esté logueado.

En primer lugar el usuario que use la aplicación introducirá su usuario y contraseña, y si tiene el formato correcto se almacenará en el localStorage para usarlo en la aplicación. A su vez se enviará la petición al servidor para que haga la validación del usuario.

En el servidor se comprueba el contenido recibido y se llama a la parte de autenticación para generar el token JWT y que lo devuelva en caso de que la autenticación sea correcta. En caso de que esto ocurra se mandará un código de petición correcta junto al token para poder utilizarlo en las demás llamadas dentro de la aplicación.

Para poder utilizar el Json Web Token se va a almacenar dentro del LocalStorage de la aplicación para poder realizar la autenticación directamente. Una vez realizado el inicio de sesión, en caso de no estar ni en reservar ni en eventos, se redirigirá a la página de inicio de la aplicación y si no, a reservar o eventos respectivamente.

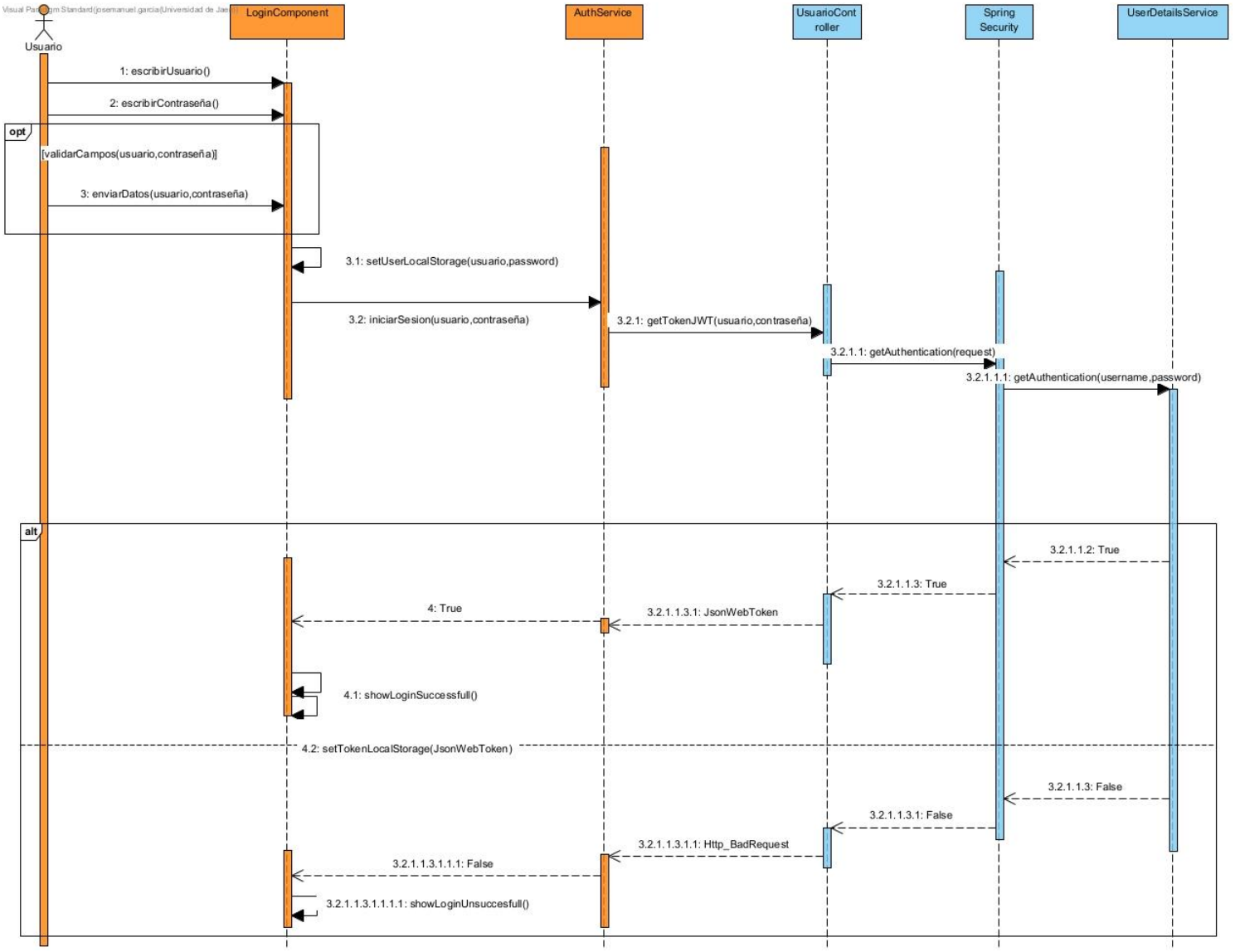



**Ilustración 39:Diagrama de secuencia de Iniciar Sesión**

## 3.3.2 Registrar un usuario

Una vez hemos visto el diagrama de secuencia de iniciar sesión en nuestra aplicación, vamos a ver como sería el de registrarse.

En este caso, rellenaremos un formulario con los datos de la cuenta del usuario. Una vez rellenados se llamaría al servicio del usuario para que mande la petición al servidor. El controlador de usuario la recibirá, y este a su vez llamará al servicio que será el encargado de insertar el usuario en caso de que todos los datos sean correctos y lo devolverá hacia la vista en orden inverso.

Como podemos observar la funcionalidad de este es bastante sencilla a diferencia de iniciar sesión puesto que no requiere de clases relacionadas con la seguridad.

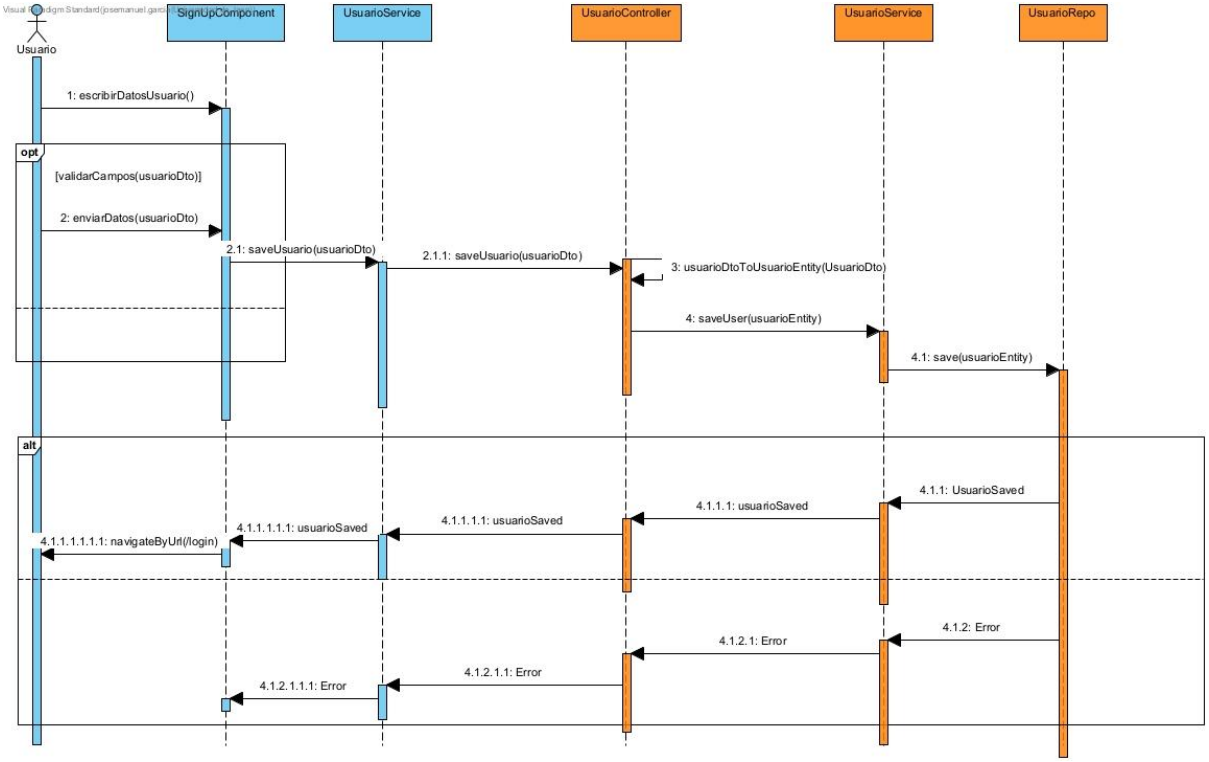



**Ilustración 40:Diagrama de secuencia de Registrar un usuario**

## 3.3.3 Realizar una reserva

Por último, el diagrama de secuencia que vamos a mostrar es el de realizar reserva, ya que es la funcionalidad principal de la aplicación.

Para este diagrama hay que tener en cuenta que el usuario ya habrá iniciado sesión y habrá elegido la pista junto a la hora que quiere alquilarla. El funcionamiento será bastante sencillo.

En primer lugar el usuario pulsará en reservar por lo que enviará los datos necesarios para poder realizar una reserva. Una vez hecho esto, el servicio de reservas se relaciona con el endpoint de realizarReservas del servidor para realizar la lógica necesaria para reservar. En caso de que la reserva pueda realizarse se enviará el objeto de la reserva y en caso contrario un error 422 con un mensaje indicando lo sucedido y por tanto no dejará realizar la reserva.

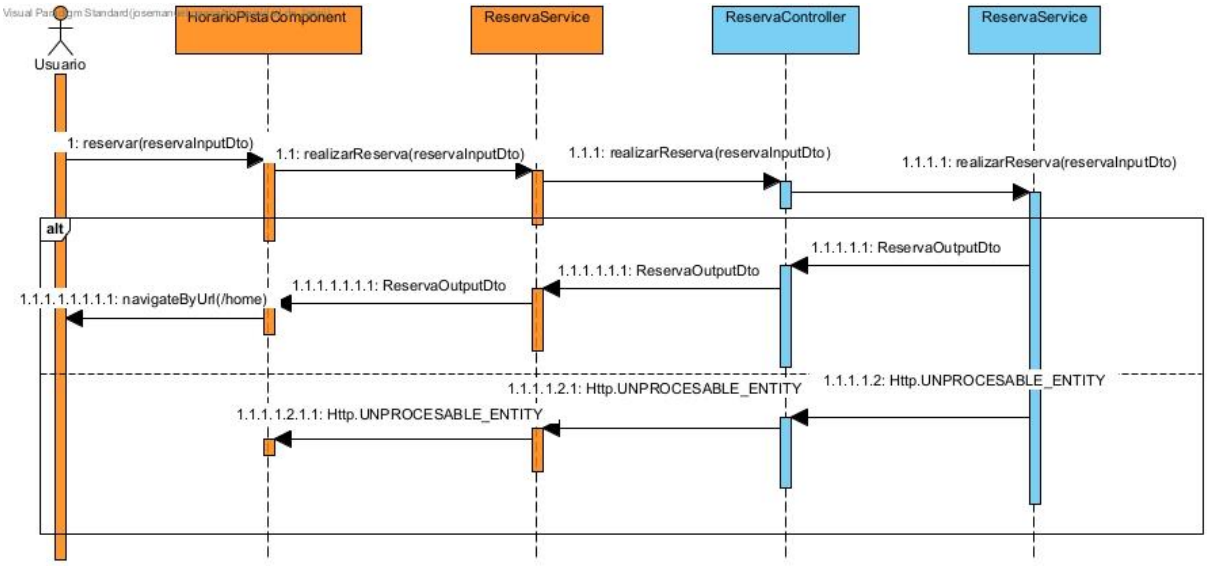



**Ilustración 41:Diagrama de secuencia de Realizar Reserva**

## 3.4 Diseño de la Interfaz

En este punto se va a explicar mediante la creación de varios wireframes el objetivo que se pretende conseguir en la implementación de la interfaz gráfica. Es decir, mostrar cómo se quiere que se distribuya. Además se va a explicar el diseño que se va a seguir para la API REST de nuestra aplicación.

### 3.4.1 Diseño de interfaz Gráfica

En primer lugar se va a proceder con la página principal de la aplicación. Esta consta de un menú vertical a la izquierda con las diferentes secciones que formarán parte de la página web, junto a una imagen que se corresponderá con el logo de la misma. En la parte derecha y central de la pantalla será donde se muestre y se gestione la mayor parte de la actividad por parte del usuario. En este caso pondrá un mensaje de bienvenida para los usuarios.


Ilustración 42:Wireframe del home de la aplicación

Una vez vista la pantalla principal, se mostrará la vista de iniciar sesión. Está compuesta por un pequeño formulario en el el usuario realizará el login para entrar en la aplicación. Además, se dará la opción de crear una cuenta en caso de que el usuario no tenga ninguna creada.

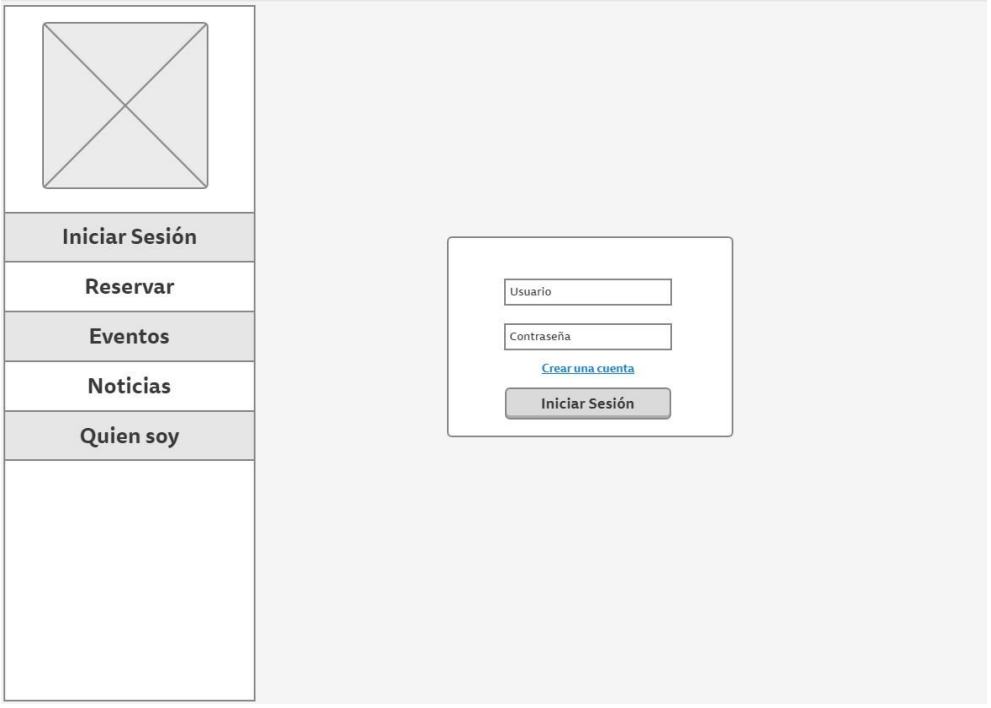



Ilustración 43:Wireframe de Inicio de sesión

Dentro de la sección reservar se encuentran las diferentes pistas. Cada pista es representada con un ítem que se desplegará cuando se tenga el cursor sobre ella, pero inicialmente mostrará el nombre de la pista, una imagen de la misma y un botón para reservar.

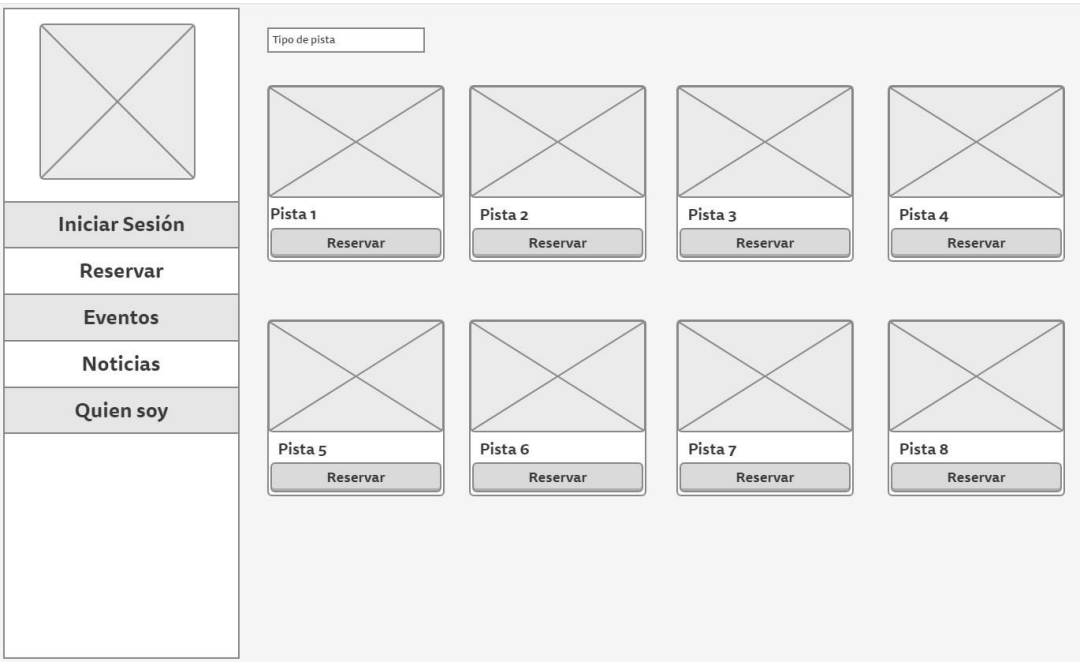



Ilustración 44:Wireframe de Reservar Pista

A su vez, una vez que se ha elegido la pista que se va a reservar, se tendrá un calendario para seleccionar el día que se quiere reservar junto a una pequeña tabla que se dividirá en tantas filas como horas de apertura tenga el polideportivo junto a si está reservado o no.

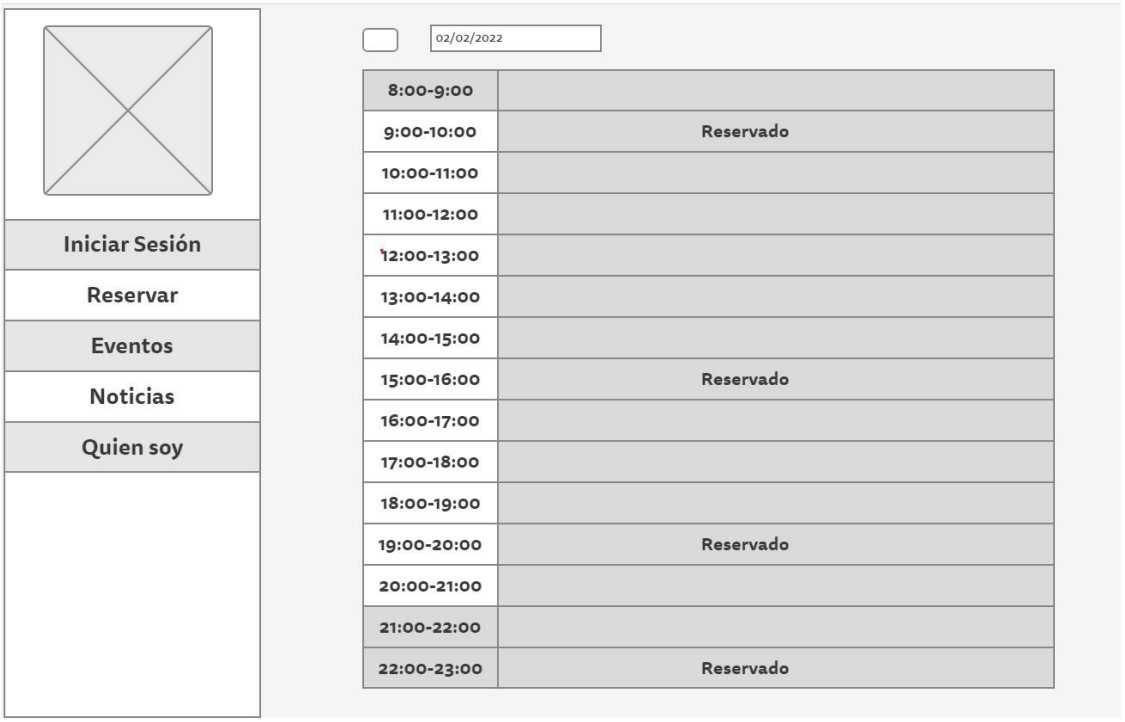



Ilustración 45:Wireframe de Horario de Pista

Por último, para el proceso de realizar una reserva se mostrará un pequeño formulario con la información relacionada con la reserva y un botón para confirmar la reserva.


Ilustración 46:Wireframe de Confirmar Reserva

Además, se dispone de una sección para apuntarse a los eventos que se organicen en el polideportivo. Por defecto solo se verá el nombre del evento junto al botón de apuntarse al mismo. Cuando el cursor esté encima del ítem del evento se verá una breve descripción del mismo junto al número de plazas disponibles del mismo.

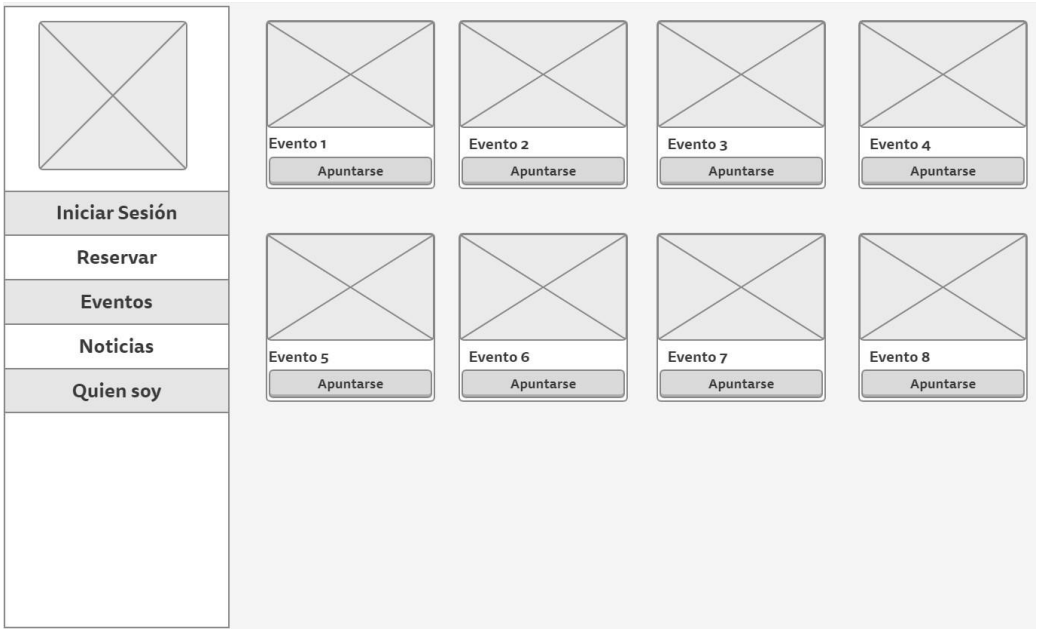



Ilustración 47: Wireframe de Eventos

### 3.4.2 Diseño de la API-REST

En este apartado se va a explicar el diseño seguido para la implementación de la API REST del proyecto.

Para ello, debemos de tener en cuenta que es un API REST y en qué consiste. Un API REST principalmente es una interfaz de aplicación que sigue las directrices de la arquitectura REST, diseñadas por Roy Fielding. [24]

Las API REST funcionan mediante peticiones HTTP para principalmente realizar funciones básicas relacionadas con las operaciones en bases de datos:

- **POST**: Crear un registro
- **GET**: Recuperar un registro
- **PUT**: Actualizar un registro
- **DELETE**: Eliminar un registro.

Cada petición a una API REST suele utilizar cabeceras de entrada en la petición y cabeceras de salida junto a códigos de respuesta para facilitar la comprensión tanto de usuario como de desarrollador.

A nivel de diseño podemos diferenciar varias entidades con una funcionalidad más compleja como son las de Usuario y Reserva. Estas serán las principales en nuestra aplicación debido a que al ser una aplicación de gestión de reservas forman parte de la funcionalidad principal de la misma. Posteriormente las entidades restantes poseerán las funcionalidades de un CRUD básico, excepto los eventos ya que el usuario podrá apuntarse a ellos

A continuación, se va a realizar una tabla con los diferentes endpoints de la API REST junto con información de los mismos. La tabla estará compuesta por un número que será simplemente para enumerar los endpoints, una columna para grupo que se corresponderá con la entidad a la que pertenece el endpoint. Las siguientes cuatro columnas serán la ruta del endpoint, una breve descripción del mismo y por último los parámetros de entrada y salida del mismo.
Finalmente encontraremos una última columna en el que se indicarán los códigos de respuesta del endpòint.

| Número | Grupo | Tipo | Ruta | Descripción | Parámetros de entrada | Parámetros de salida | Códigos HTTP |
|---|---|---|---|---|---|---|---|
| 1 | Usuario | POST | /api/empresa/user/ | Endpoint para la creación de un usuario | UsuarioInput Dto | UsuarioOutputDto | Correcto: 200<br>Error:422 |
| 2 | Usuario | POST | /api/empresa/login | Endpoint para realizar login de un usuario | Username y password | Bearer token | Correcto: 200<br>Error:422 |
| 3 | Usuario | GET | /api/empresa/user/{nombre Usuario | Endpoint para obtener los datos de un usuario | Nombre de usuario | UsuarioOutputDto | Correcto: 200<br>Error:422 |
| 4 | Usuario | POST | /api/empresa/user/role/ | Endpoint para asignar un rol a un usuario | username y rolename | Código HTTP 200 | Correcto: 200<br>Error:422 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **5** | **Reserva** | **POST** | /api/empresa/reservas | Endpoint para realizar una reserva | ReservaInputDto | ReservaOutputDto | Correcto: 200<br>Error:422 |
| **6** | **Reserva** | **GET** | /api/empresa/reservas | Endpoint para listar todas las reservas | - | List<ReservaOutputDto> | Correcto: 200 |
| **7** | **Reserva** | **GET** | /api/empresa/reservas/{id} | Endpoint para obtener los datos de una reserva | Id de la reserva | ReservaOutputDto | Correcto: 200<br>Error:404 |
| **8** | **Reserva** | **DELETE** | /api/empresa/reservas/{idReserva} | Endpoint para eliminar una reserva | IdReserva | Código HTTP 200 | Correcto: 200<br>Error:404 |
| **9** | **Pista** | **POST** | /api/empresa/pistas | Endpoint para crear una pista | - | PistaOutputDto | Correcto: 200<br>Error:422 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **10** | **Pista** | **GET** | /api/empresa/pistas | Endpoint para listar todas las pistas | - | List<PistaOutputDto> | Correcto:200 |
| **11** | **Pista** | **GET** | /api/empresa/pistas/{id} | Endpoint para obtener los datos de una pista | Id de la pista | PistaOutputDto | Correcto: 200<br>Error:404 |
| **12** | **Pista** | **GET** | /api/empresa/pistas/{id}/reservas | Endpoint para obtener las reservas de una pista | Id de la pista | List<ReservaOutputDto> | Correcto: 200<br>Error:404 |
| **13** | **Pista** | **DELETE** | /api/empresa/pistas/{id} | Endpoint para eliminar una pista | Id de la pista | Código HTTP 200 | Correcto: 200<br>Error:404 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **14** | **Pista** | **PUT** | /api/empresa/pistas/{id} | Endpòint para editar una pista | PistaInputDto | PistaOutputDto | Correcto: 200<br>Error:404 |
| **15** | **Noticia** | **POST** | /api/empresa/noticias | Endpoint para crear una noticia | NoticiaInputDto | NoticiaInputDto | Correcto:200 |
| **16** | **Noticia** | **GET** | /api/empresa/noticias | Endpoint para listar todas las noticias | - | List<NoticiaOutputDto> | Correcto: 200<br>Error:404 |
| **17** | **Noticia** | **GET** | /api/empresa/noticias/{id} | Endpoint para obtener los datos de una noticia | Id de la noticia | NoticiaOutputDto | Correcto: 200<br>Error:404 |
| **18** | **Noticia** | **DELET** | /api/empresa/noticias/{id} | Endpoint para eliminar una pista | Id de la noticia | Código HTTP 200 | Correcto: 200<br>Error:404 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | E | | | | | |
| **19** | **Noticia** | **PUT** | /api/empresa/noticias/{id} | Endpoint para editar una pista | PistaInputDto | NoticiaOutputDto | Correcto: 200<br>Error:404 |
| **20** | **Evento** | **POST** | /api/empresa/eventos | Endpoint para listar todas las pistas | - | List<EventoOutputDto> | Correcto:200 |
| **21** | **Evento** | **GET** | /api/empresa/eventos/ | Endpoint para listar todas los eventos | Id del evento | EventoOutputDto | Correcto: 200 |
| **22** | **Evento** | **GET** | /api/empresa/eventos/{id} | Endpoint para obtener los datos de un evento | Id del evento | EventoOutputDto | Correcto: 200<br>Error:404 |
| **23** | | | /api/empresa/eventos/{id} | Endpoint para eliminar | Id del evento | Código HTTP 200 | Correcto: 200 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Evento** | **DELETE** | | un evento | | | Error:404 |
| **24** | **Evento** | **PUT** | /api/empresa/eventos/{id} | Endpòint para editar un evento | EventoInputDto | EventoOutputDto | Correcto: 200<br>Error:404 |

**Ilustración 48: Diseño de API Rest**

## 3.5 Plan de pruebas

En este apartado se va a realizar una explicación sobre cómo se van a realizar las diferentes pruebas de la aplicación, principalmente en la parte del servidor puesto que es donde se va a encontrar la funcionalidad de la misma.

Se ha optado por utilizar pruebas de caja negra,caracterizadas porque sólo nos interesa saber tanto la entrada como las salida de las mismas, a partir de unos datos de entrada debemos obtener la salida que se espere [25]. Además, utilizaremos los criterios de aceptación como modo de pruebas de caja negra para el funcionamiento de la aplicación en el cliente.

Para ello se va a utilizar PostMan, ya que es un programa que nos permite realizar pruebas a nuestra API mediante HTTP Request de una forma muy sencilla e intuitiva.[26].


**Ilustración 49:Logo de PostMan.**

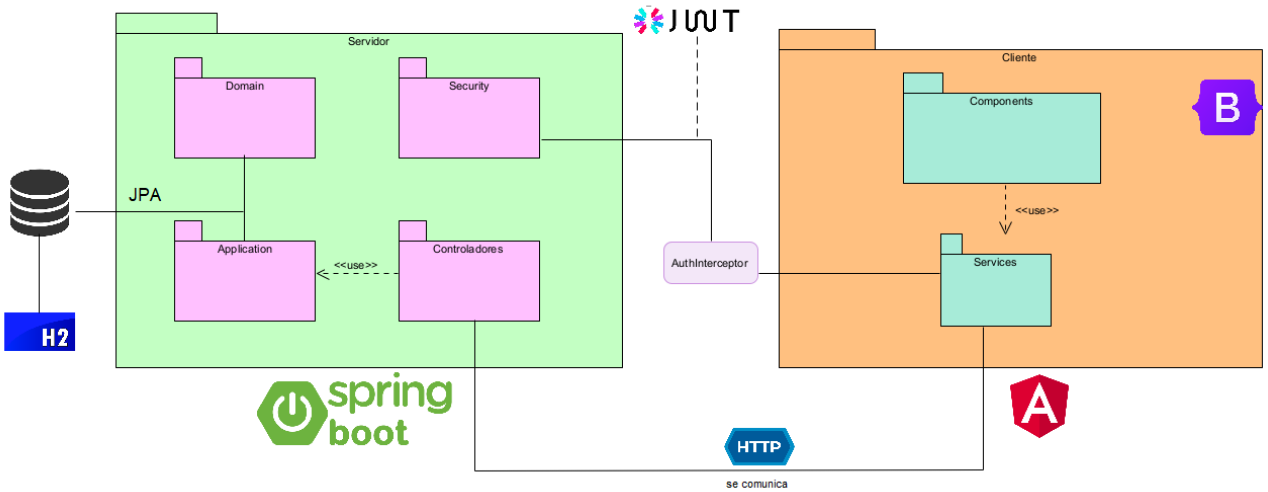
El proceso a seguir es bastante sencillo y se detallará posteriormente en el apartado de implementación. Someteremos la API de forma que esperaremos que se devuelva el código de respuesta junto con el objeto que se espera, de forma que así garantizaremos que funciona de forma correcta.

# 4.- Implementación

## 4.1 Arquitectura

En este apartado se va a explicar la relación existente entre la parte del cliente y del servidor que va a existir en el desarrollo de la aplicación, tratando los componentes por los que están formados y la funcionalidad del mismo.

Como se ha observado durante el desarrollo de este Trabajo de Fin de Grado, hay dos aplicaciones independientes, una para la parte del servidor y otra para la parte del cliente. En primer lugar se va a realizar una explicación por separado de cada parte para




finalmente ver cómo se realiza la interacción entre ambas.

**Ilustración 50: Diagrama de paquetes del Backend con arquitectura hexagonal.**

### 4.1.1 Aplicación en el servidor

En nuestra aplicación, para cada entidad existirá esta distribución de forma que estarán compuestos por tres paquetes:

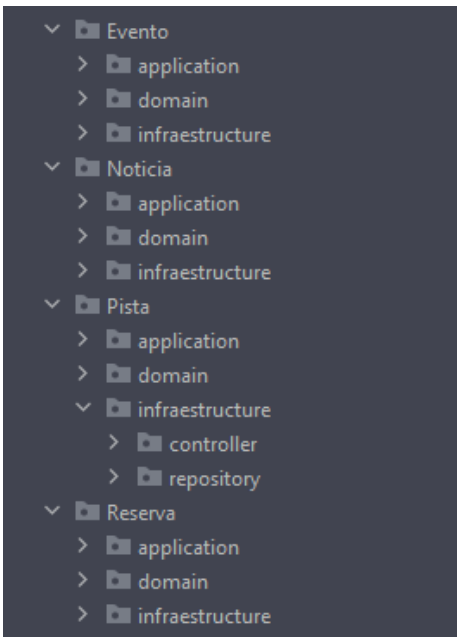



Ilustración 51: Diagrama de paquetes del Backend con arquitectura hexagonal.

- **Application**: Estará compuesto por los servicios de la entidad (casos de uso)
- **Domain**:Estará compuesto por la entidad
- **Infraestructure**: Estará formado por los controladores del objeto

A continuación vamos a ver un pequeño ejemplo sobre la distribución de los ficheros de un paquete:

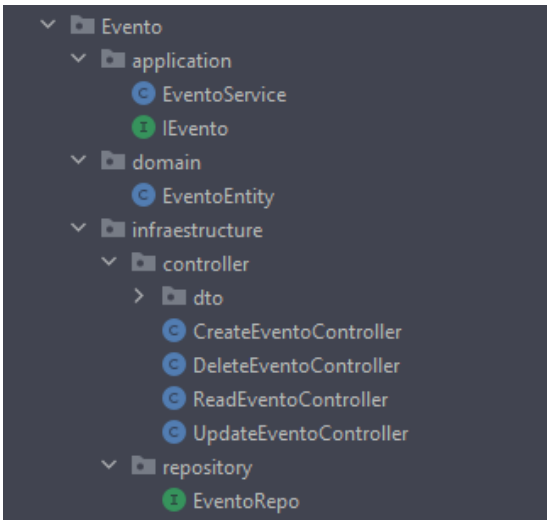



Ilustración 52: Fragmento de paquetes de arquitectura hexagonal en Evento

### 4.1.2 Aplicación en el cliente

Para la programación en la aplicación del cliente se ha optado por realizar una programación basada en componentes. A su vez cada componente inyecta al servicio

asociado y llama a la funcionalidad que este quiera usar de él. La distribución ha sido lo siguiente:

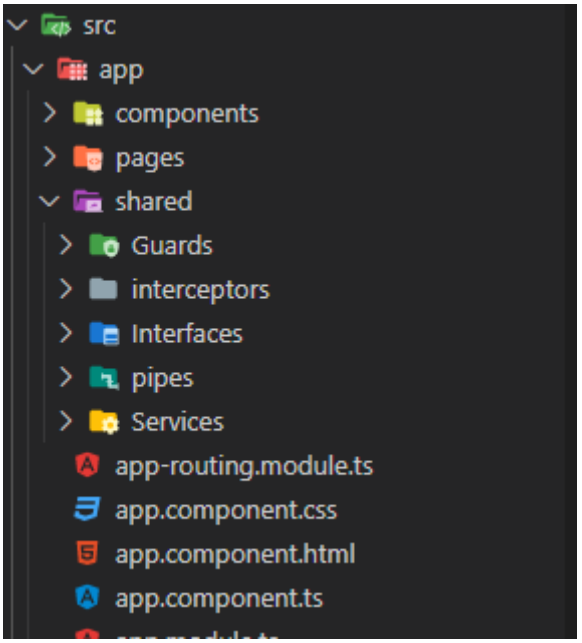



Ilustración 53: Fragmento de paquetes en parte del cliente

- **Components**: Componentes sencillos que compondrán las páginas
- **Pages**: Se corresponde con las páginas con las que va a interactuar el usuario-
- **Shared:** Servicios que usaremos para comunicarnos con el servidor junto a otros componentes que serán compartidos para todos los componentes.
    - **Guards**: Salvaguardas para gestionar el acceso a páginas.
    - **Interceptors**: Ayuda a gestionar peticiones a nuestra API Rest.
    - **Interfaces**:Representaciones de los objetos.
    - **PIpes**: Elementos para implementar filtrado.
    - **Services**: Servicios con la funcionalidad y comunicación con los endpoints de la API Rest.

## 4.2 Detalle sobre implementación

Tras ver las diferentes arquitecturas usadas tanto en la parte del servidor como en la parte del cliente se van a exponer las tecnologías utilizadas, librerías, dependencias necesarias y configuración del proyecto para su correcto funcionamiento.

### 4.2.1 Aplicación API REST

Para el desarrollo del código en el lado del servidor se ha utilizado Java. Como entorno de desarrollo se ha utilizado IntellIJ junto a Apache Maven para poder gestionar todo lo relacionado con la ejecución de la aplicación y las dependencias de forma eficiente. Además del propio Spring como framework para la programación del mismo.

#### 4.2.1.1 Persistencia

Para gestionar la aplicación a nivel de persistencia se ha utilizado Jpa en Hibernate [28] para crear la respectiva base de datos con las tablas a partir de las clases que hemos desarrollado. Gracias a las anotaciones de Jpa nos permite configurar la aplicación de Spring de forma sencilla. El mapeo de las entidades se realizará mediante diferentes anotaciones en los diferentes atributos o en la propia cabecera de la clase según sea necesario.

Las principales anotaciones de Jpa son las siguientes:

- **@Entity:** Sirve para indicar que esa clase se va a persistir

```
@Entity
@Data
@Table(name = "Reserva")
@NoArgsConstructor
@Transactional
public class ReservaEntity {
```

Ilustración 54: Ejemplo de anotación @Entity.

- **@Id:** Indica cual es la clave primaria de esa entidad.
- **@GeneratedValue:** Autogenerar la clave primaria.

```
public class ReservaEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Integer id;
```

Ilustración 55: Ejemplo de anotación @Id y @GeneratedValue.

- **@OneToMany, @ManyToOne** para indicar el tipo de relación entre las diferentes entidades. A su vez tiene distintas anotaciones para configurar cada una de ellas entre las que destacan:

```java
@JsonBackReference
@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "pista_asignada_id")
PistaEntity pistaAsignada;
```

Ilustración 56: Ejemplo de anotación @ManyToOne.

- **FetchType**: Indica la estrategia para recuperar datos desde la base de datos. Existen dos tipos: EAGER y LAZY. La principal diferencia entre ambos es que EAGER carga todos los datos aunque no se estén usando y LAZY solo los carga en el momento de uso.
- **Cascade:** Nos permite indicar como afectan las acciones que hacemos sobre entidades relacionadas. Los principales tipos de CASCADE son ALL,PERSIST,MERGE,REMOVE,REFRESH,DETACH.

```java
@JsonBackReference
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
        name = "EVENTOS_USUARIOS",
        joinColumns = @JoinColumn(name = "FK_EVENTO", nullable = false),
        inverseJoinColumns = @JoinColumn(name = "FK_USUARIO",nullable = false))
List<UsuarioEntity> usuariosApuntados;
```

Ilustración 57: Ejemplo de anotación @ManyToMany.

- **OrphanRemove:** Esa anotación sirve para borrar las entidades asociadas entre sí.

```java
@JsonManagedReference
@OneToMany(mappedBy = "pistaAsignada", cascade = CascadeType.ALL, orphanRemoval = true)
List<ReservaEntity> reservasAsignadas;
```

Ilustración 58: Ejemplo de FetchType, Cascade y OrphanRemove.

A su vez, para realizar la implementación de las clases de la parte del servidor se han usado Beans para facilitar la inyección de dependencias entre clases. Un Bean en Spring es una clase normal que es tratada por el propio Spring y facilita las dependencias necesarias que esta necesite para que este funcione correctamente.

Para ayudarnos a gestionar los beans de forma más sencilla existen anotaciones para indicar que tipo de bean es y saber cómo se comporta. En esta aplicación las anotaciones que se van a utilizar son @Repository para la persistencia y @Service para los diferentes servicios. Para realizar la inyección de dependencias entre los diferentes beans se ha utilizado la anotación @Autowired.

```
@Repository
public interface ReservaRepo extends JpaRepository<ReservaEntity,Integer> {
    ReservaEntity findByFechaReservaAndHoraReservaAndPistaAsignada(Date fechaR
```

Ilustración 59: Ejemplo de anotación @Repository.

```
@Service
@EnableScheduling
@Proxy(lazy = false)
public class ReservaService implements IReserva {
```

Ilustración 60: Ejemplo de anotación @Service.

#### 4.2.1.2 Controladores

Para poder comunicar nuestras aplicaciones de servidor y cliente se usan controladores para gestionar la API REST que estamos implementando. La anotación utilizada para ello es @RestController junto a @RequestMapping para indicar la ruta que se usará en el Path para conectar con el cliente.

```
@RestController
@RequestMapping("api/empresa/reservas")
public class CreateReservaController {
```

Ilustración 61: Ejemplo de anotación @RestController y @RequestMapping.

Dependiendo del tipo de método que sea la llamada que queramos hacer, se usarán las anotaciones @GetMapping, @PostMapping, @PutMapping y @DeleteMapping ya sea para obtener, crear, actualizar o borrar respectivamente. Además se pueden obtener parámetros de diferentes formas, desde el path con la anotación @PathVariable hasta

desde un objeto Json con la anotación @RequestBody para indicarle que desde ese objeto será donde se almacenará la información que necesita para la petición.

```
@PostMapping()
public ResponseEntity<ReservaOutputDTO> addReserva(@RequestBody @Valid ReservaInputDTO reservaInputDTO){
    return ResponseEntity.ok().body(reservaService.realizarReserva(reservaInputDTO));
}
```

Ilustración 62: Ejemplo de anotación @PostMapping.

```
@DeleteMapping("{id}")
public void deleteReservaById(@PathVariable Integer id) { reservaService.deleteById(id); }
```

Ilustración 63: Ejemplo de anotación @DeletetMapping.

```
@GetMapping
public ResponseEntity<List<ReservaOutputDTO>> findAll(){
    return ResponseEntity.ok().body(reservaService.findAll());
}
```

Ilustración 64: Ejemplo de anotación @GetMapping.

```
@PutMapping("{id}")
public PistaOutputDto actualizarPista(@PathVariable Integer id, @RequestBody ActualizarPistaDto pista) {
    return pistaService.actualizarPista(id, pista);
}
```

Ilustración 65: Ejemplo de anotación @PutMapping.

Finalmente, para realizar la respuesta de la petición se realizará mediante una ResponseEntity de tipo 200 para saber que todo ha funcionado de forma correcta devolviendo un objeto DTO con los datos correspondientes a la petición. En caso de no realizarse la petición de forma correcta, se ha creado una clase personalizada para la gestión de errores mostrando el mensaje que queramos dependiendo del error que se lance.

#### 4.2.1.3 Autenticación

Como se ha visto a lo largo del proyecto, se ha implementado un sistema de inicio de sesión para así poder restringir el acceso a diferentes secciones y distinguir entre usuarios y administrador. Para ello, se ha basado el sistema de autenticación en Json Web Token [BIBLIO] para poder llevarlo a cabo.

```
User user = (User)authentication.getPrincipal();
Algorithm algorithm = Algorithm.HMAC256("secret".getBytes());
String access_token = JWT.create()
        .withSubject(user.getUsername())
        .withExpiresAt(new Date(System.currentTimeMillis()+ 10 * 60 * 10000))
        .withIssuer(request.getRequestURL().toString())
        .withClaim( name: "roles",user.getAuthorities().stream().map(GrantedAuthority::getAuthority)
                .collect(Collectors.toList()))
        .sign(algorithm);
```

Ilustración 66:Fragmento de código de generación de token JWT.

El funcionamiento es sencillo. Cuando se inicia sesión con el usuario se crea un token mediante una encriptación HS256. Este token está compuesto por tres partes, Header, Payload y Verify Signature. En nuestro caso vamos a trabajar con la parte de Payload.

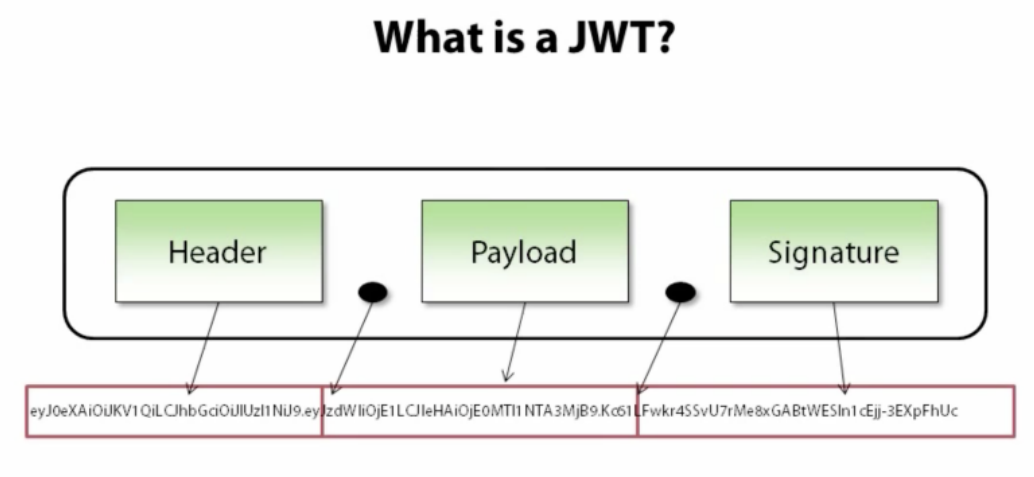



Ilustración 67:Partes de un token JWT

Aquí dentro podemos enviar el dato que necesitemos, en este caso se envía el usuario junto con el rol que este tiene dentro de la aplicación, tanto usuario como administrador. Una vez que se tiene el token en la parte de la aplicación web este se almacena de forma que se mandará cada vez que este realice una petición a la parte del servidor. Este lo enviará mediante Headers, en el campo Authorization y en token seguirá el siguiente patrón: Bearer Token.

Headers 8 hidden

| | KEY | VALUE |
|---|---|---|
| ☑ | Authorization | Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJzdWIi... |
| | Key | Value |

Ilustración 68: Headers necesarios para peticiones con token JWT.

De esta forma cuando se mande una petición desde el cliente, el servidor se encargará de desencriptar el token recibido y se encargará de ver si se tienen los permisos necesarios para realizar la petición que se ha enviado.

#### 4.2.1.4 Lombok

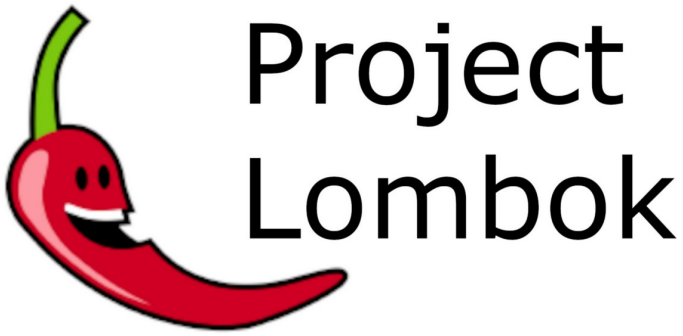



Ilustración 69: Logo de Lombok.

Lombok es una librería que nos permite implementar getters, setters y constructores de forma automática sin necesidad de escribirlos en el código.

Para poder implementarla en el código en primer lugar pondremos la dependencia correspondiente con la librería en el pom.xml del proyecto. Una vez instalada la dependencia bastará con usar las anotaciones definidas para cada caso [29]:

- @Getter: Implementar Getters de forma automática.
- @Setter: Implementar Setters de forma automática.
- @AllArgsContructor: Implementar constructor parametrizado de forma automática.
- @NoArgsContructor: Implementar constructor por defecto de forma automática.

- @Data: Implementar getters,setters y obliga a implementar un constructor parametrizado.

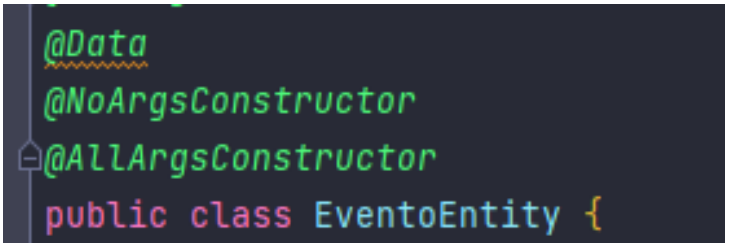



Ilustración 70: Ejemplo de anotaciones @Data, @NoArgsContructors y @AllArgsContructors

Una vez puestas las anotaciones en función a aquello que queremos implementar de forma automática podremos usarlos en los objetos que creemos de las correspondientes clases.

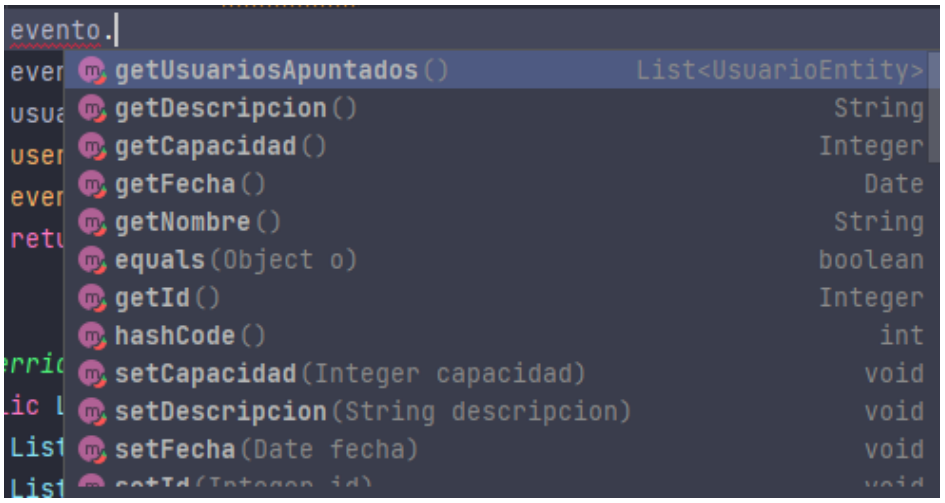



Ilustración 71: Ejemplo de get autogenerados.

## 4.2.2 Aplicación Cliente

Para el desarrollo de la aplicación en la parte del cliente se ha optado por utilizar Angular como framework. Concretamente la versión 13.3.2 de Angular CLI y la versión 16.14.2 de Node. Como IDE de desarrollo se ha utilizado Visual Studio Code ya que junto a los plugins que este posee para el desarrollo de código en Angular ayudaba bastante. Al ser una comunidad muy amplia, Visual Code tiene mucho soporte y en caso de problemas con el IDE tienen una rápida resolución. Inicialmente Angular posee 3 paquetes principales de Angular-devkit, que son architect, core y schematics.

Como se ha explicado anteriormente se ha seguido una programación basada en componentes para la creación de la aplicación web. Podemos distinguir la parte de componentes,la parte de pages y la parte de shared. Cada componente está dividido en un fichero HTML, un fichero CSS y un fichero en formato TypeScript para aportar funcionalidad a nuestra web y comunicar todas las páginas entre sí.

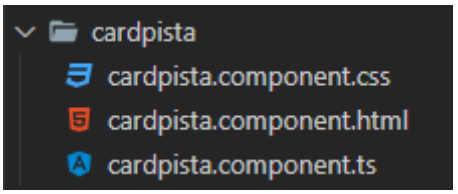



Ilustración 72: Ejemplo de un componente

El uso de librerías es muy sencillo y el proceso siempre es el mismo para todas las librerías que se han usado en el proyecto. En primer lugar, se instalan usando el comando ng install y el nombre de la librería que queramos. Hecho esto bastará con importar dentro del fichero de módulos el que se corresponda con el de la librería que hemos instalado y automáticamente ya podremos usarlo en toda nuestra aplicación las veces que necesitemos.

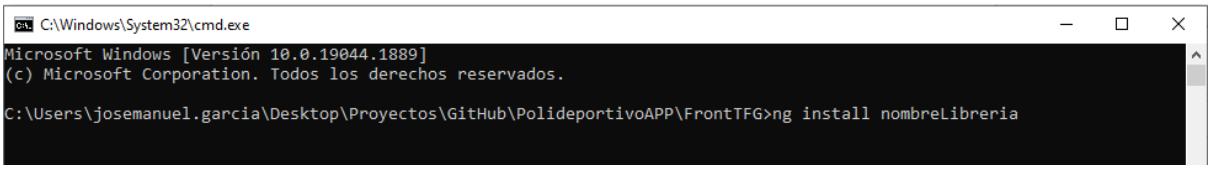



Ilustración 73: Ejemplo de sintaxis para instalar una librería.

Para comunicar las páginas entre sí se ha utilizado Router Link ya que es sencillo e intuitivo. Aparte, se han usado varias librerías como la de ngx-toast para poder mostrar notificaciones, aunque se explicarán posteriormente.

También se ha optado por usar BootStrap 5 para el desarrollo de HTML puesto que al haber trabajado antes ha ayudado bastante a hacer más familiar la implementación de la web. Para cargarlo bastará con instalar la librería oficial y añadir la siguiente linea en el fichero index.html del proyecto

```
<!-- Bootstrap core CSS -->
<link href="/docs/5.1/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
```

Ilustración 74: Linea de código para usar Bootstrap en Angular.

### 4.2.2.1 Navegación web

Para realizar la comunicación entre las diferentes páginas de la web se ha optado por utilizar Router Link que es una librería del propio Angular [30]. Esta permite desplazarte entre las diferentes páginas de la aplicación de forma sencilla. El funcionamiento consiste en poner el selector <router-outler> dentro del index principal del proyecto junto a la importación del módulo RouterModule para poder usarlo.

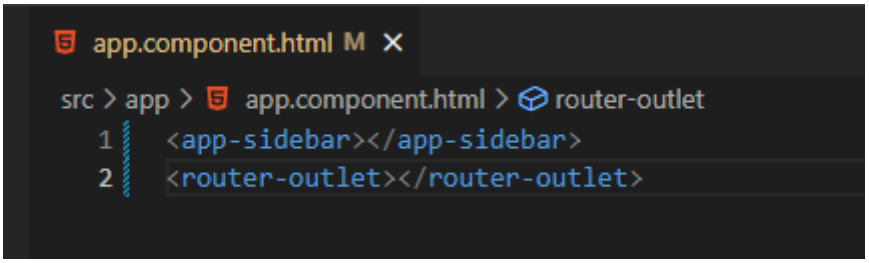



Ilustración 75:Selector routerOutler para gestionar navegación

A su vez, dentro del fichero app-routing.ts se podrán definir las rutas haciendo referencia al componente que al que se quiere desplazar. Una vez hecho esto, bastará con poner la propiedad routerLink y asignarle como valor el nombre de la ruta que le hayamos especificado en el fichero anterior.

```
const routes: Routes = [
  {path:'home', component: HomeComponent},
  {path:'login', component:LoginComponent},
  {path:'prueba', component:SidebarComponent},
  {path:'eventos', component:EventosComponent},
  {path:'noticias', component:NoticiasComponent},
```

Ilustración 76:Fragmento de código con rutas en App-Routing-Module

### 4.2.2.2 Autenticación/Autorización en el cliente

A su vez, para gestionar el control de acceso a páginas restringidas hemos utilizado Guards. Los Guards son interfaces que nos permiten gestionar el control de acceso a páginas determinando si se puede acceder a estas rutas o no.[31]

```
export class GuardsGuard implements CanActivate {

  constructor(private authService:AuthenticationService,private router:Router){

  }
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
  boolean | UrlTree | Observable<boolean | UrlTree> | Promise<boolean | UrlTree> {
    if(!this.authService.isUserLoggedIn()){
      this.router.navigateByUrl("/login");
      return false;
    }
    return true;
  }
}
```

Ilustración 77: Fragmento de código con clase Guards

Además, para gestionar la autenticación hemos usado un Interceptor. Un interceptor sirve para cambiar el contenido de peticiones entre nuestra aplicación y la parte del servidor [32]. De esta forma podremos controlar petición a petición las peticiones hacia el servidor y en este caso añadir el header authorization con el token a las peticiones de forma sencilla.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {

  constructor() {}

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {
    if('http://localhost:8080/api/empresa/login'===request.url){
      return next.handle(request);
    }
    const jwt = localStorage.getItem("authorization");

    if(jwt){

      request = request.clone({
        setHeaders: { Authorization: jwt }});


    }
    return next.handle(request);

  }
```

Ilustración 78: Fragmento de código de clase interceptor.

### 4.2.2.3 Ngx-Toastr para realizar notificaciones al usuario

Para poder mostrar mensajes al usuario hemos decidido usar una librería llamada Ngx-Toastr para ello ya que nos permite definir diferentes notificaciones dependiendo de para que lo queramos utilizar, ya sea como información o como algo que ha salido. [33]

Para hacernos una idea a nivel visual de como quedarían en nuestro proyecto hemos usado una página que permite generar las notificaciones a partir de todas sus propiedades [34]

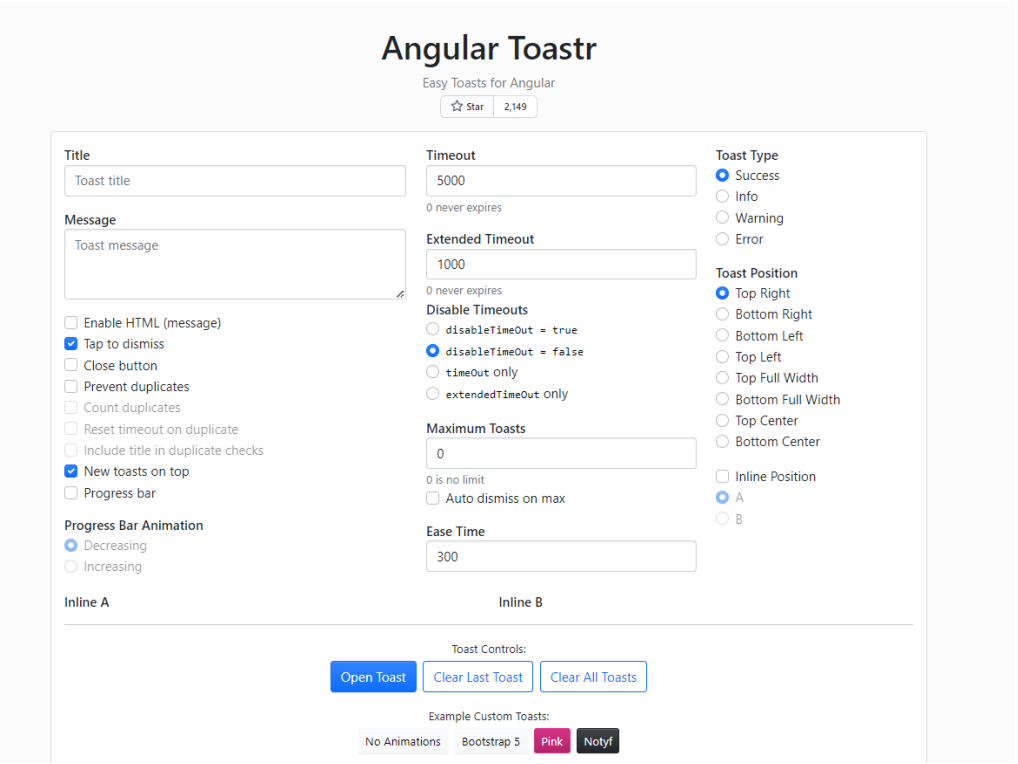



**Ilustración 79: Web de Toastr.Verce**

Una vez visto esto, la implementación en nuestro en nuestro proyecto es bastante sencilla, bastará con seguir los pasos básicos para una librería cualquiera junto a los pasos que encontramos en la documentación de la misma.

Tras instalar la librería en nuestro proyecto tendremos que implementar un servicio que inyecte el servicio ToastService ya que será el que nos permita configurar la notificación al usuario de forma personalizada.

```
import { Injectable } from '@angular/core';
import { ToastrService } from 'ngx-toastr';

@Injectable({
  providedIn: 'root'
})
export class NotificationService {

  constructor(private toastr: ToastrService) { }
```

Ilustración 80: Inyección de servicio ToastrService

Dependiendo del tipo de notificación que queramos implementar crearemos un método con las propiedades asociadas a cada una.

```
showSuccess(message:string, title:string){
  this.toastr.success(message, title, {
    extendedTimeOut: 1000,
    timeOut: 5000,
    enableHtml: true,
    tapToDismiss: false,
    progressBar: true,
    closeButton:true,
    progressAnimation: 'increasing',
    positionClass:'toast-top-right'
  })
  console.log(this.toastr.success)
}

showInfo(message:string, title:string){
  this.toastr.info(message, title, {
    extendedTimeOut: 1000,
    timeOut: 5000,
    enableHtml: true,
    tapToDismiss: false,
    progressBar: true,
    closeButton:true,
    progressAnimation: 'increasing',
  })
}

showError(message:string, title:string){
  this.toastr.error(message, title, {
    extendedTimeOut: 1000,
    timeOut: 5000,
    enableHtml: true,
    tapToDismiss: false,
    progressBar: true,
    closeButton:true,
    progressAnimation: 'increasing'
  })
}
```

Ilustración 81: Funciones de configuración de notificaciones.

Una vez definido el servicio para crear las notificaciones bastará con inyectar el servicio en los componentes en los que queramos usar las notificaciones y llamar a los métodos creados en el mismo.

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css'],
})
export class LoginComponent implements OnInit {
  username!: string;
  password!: string;

  constructor(private authService: AuthenticationService, private router:Router,private notificarService:NotificationService) {}
```

Ilustración 82: Inyección de servicio creado.

```
iniciarSesion() {
  this.authService
    .authenticate(this.username, this.password)
    .subscribe({ complete: () => {
                  this.authService.userSubject$.subscribe( (v) => {console.log(v),
                  this.notificarService.showSuccess("Se ha logueado correctamente","Login realizado"),
                  this.router.navigateByUrl(environment.actualUrl)
    } )
    }, error: (e) => {this.notificarService.showError("Inicio de sesión incorrecto.","Login incorrecto")
  }, next: (v) => {console.log(v)} });
```

Ilustración 83: Ejemplo de uso de notificación en Login.

### 4.2.2.3 Comunicaciones REST con API Rest

Para gestionar la comunicación con nuestra API Rest debemos de hacerlo de una forma bastante similar a como hemos explicado en el apartado anterior, con la diferencia de que dentro de los métodos realizaremos diferentes llamadas dependiendo del tipo de llamada de la API.

Para guardar los objetos que obtengamos de respuesta de llamar a nuestra API usaremos interfaces. Una interfaz sirve para definir las características del objeto para posteriormente poder acceder a ellas. [35]

```
export interface Pista{
    id:number;
    descripcion:string;
    precio:number;
    imageUrl:string;

}
```

Ilustración 84: Interface de Pista

En primer lugar, lo que haremos será importar e inyectar el módulo HttpClient ya que será el que usaremos para gestionar todo lo relacionado con la comunicación entre la parte de front y la API.

```
export class PistaService {


private apiServerUrl=environment.apiBaseUrl;

constructor( private http:HttpClient) { }
```

Ilustración 85: Inyección de HttpClient para peticiones a API Rest

Una vez hecho esto, tendremos que crear tantos métodos como queramos usar de la API. En este caso todas las funciones han seguido el mismo esquema. Llamamos al servicio http inyectado y dependiendo del tipo de petición (GET,POST,PUT,DELETE) que hayamos definido en la API junto al tipo de dato que devuelve acompañado de la ruta del endpoint en la API y los argumentos que cada uno necesite.

```
public getPistas():Observable<Pista[]>{
    return this.http.get<Pista[]>(`${this.apiServerUrl}/api/empresa/pista`);
}


public addPista(pistaInputDTO:pistaInputDTO):Observable<Pista>{
  return this.http.post<Pista>(`${this.apiServerUrl}/api/empresa/pista`,pistaInputDTO);
}

public borrarPista(id:number):Observable<void>{
  return this.http.delete<void>(`${this.apiServerUrl}/api/empresa/pista/${id}`);
}
public getReservasPista(id:number):Observable<Reserva[]>{
  return this.http.get<Reserva[]>(`${this.apiServerUrl}/api/empresa/pista/${id}`);
}
public getPista(id:number):Observable<Pista>{
  return this.http.get<Pista>(`${this.apiServerUrl}/api/empresa/pista/obtenerpista/${id}`);
}
public actualizarPista(id:number,pista:ActualizarPistaDTO):Observable<Pista>{
  return this.http.put<Pista>(`${this.apiServerUrl}/api/empresa/pista/${id}`,pista);
}
```

Ilustración 86:Fragmento de código de PistaService con peticiones a endpoints de API.

# 5 Pruebas

En este apartado se van a detallar de forma más extensa y concreta las pruebas indicadas en el apartado de diseño de pruebas.

Para recapitular, se va a utilizar PostMan en su versión 7.36.7 para la realización de las mismas ya que es bastante intuitivo y nos servirá para obtener muy buena información sobre la API. Postman es una herramienta muy buena ya que nos permite guardar todos los datos de prueba en un fichero para poder realizar las pruebas de forma rápida y sencilla.

## 5.1 Pruebas en el servidor

Como se ha explicado anteriormente, para realizar las pruebas en el servidor se ha usado PostMan y se ha desarrollado una colección para poder hacerlas de forma rápida y sencilla.

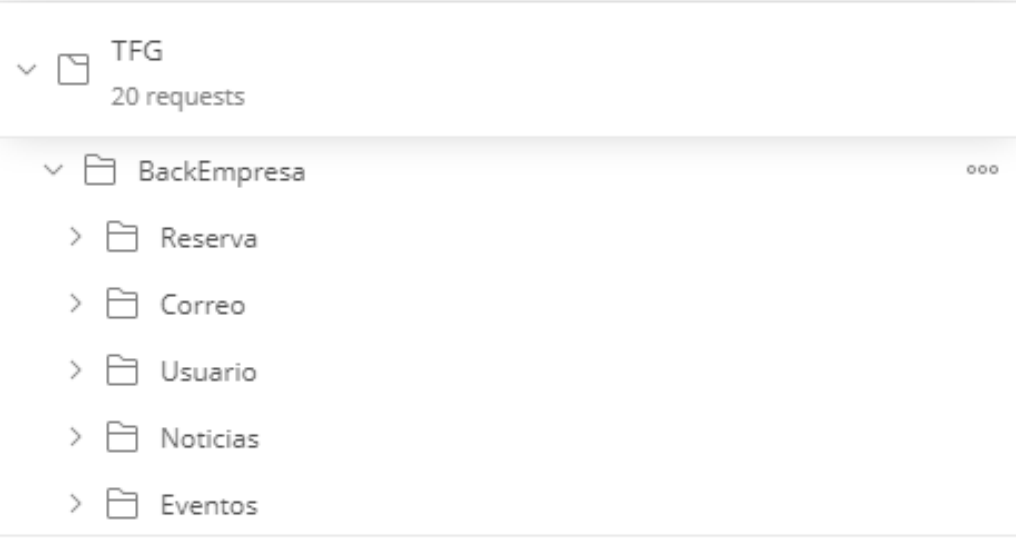
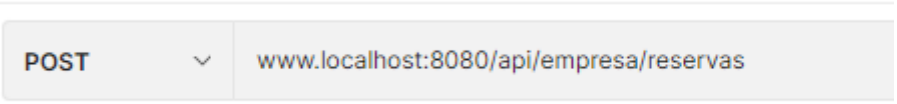



Ilustración 87: División de paquetes en PostMan

A continuación, vamos a mostrar cómo se realiza una prueba de un endpoint con esta herramienta. Todas las pruebas seguirán el mismo proceso, aunque todas las pruebas están explicadas en el apéndice IV.

Como ejemplo de prueba vamos a ver como realizar una reserva, ya que es el principal objetivo de la aplicación. Debemos de tener en cuenta que en este caso los parámetros y body que necesitan se generan en el cliente pero para realizar las pruebas lo haremos de forma manual. Para ello tendremos en cuenta los siguientes pasos:

- En primer lugar introduciremos en la ruta del PostMan el endpoint a probar.




- Una vez introducida la ruta, introduciremos los parámetros necesarios, en este caso un body con los datos requeridos para realizar la reserva.


Ilustración 88: División de paquetes en PostMan

- Tras esto, introduciremos en Headers el token de la siguiente forma: Bearer token para que el sistema se asegure de que el usuario tiene permisos para hacerlo

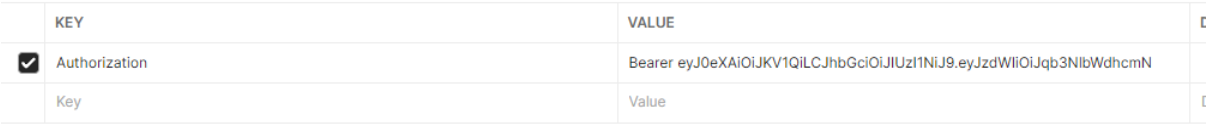


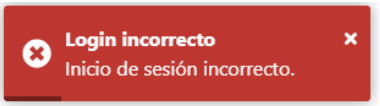

Ilustración 89: División de paquetes en PostMan

- Una vez hecho esto comprobaremos la salida y veremos si se ha realizado de forma correcta o no.

## 5.2 Pruebas en el cliente

Para realizar las pruebas en el cliente el procedimiento seguido será realizar una prueba a mano sobre las diferentes funcionalidades que queremos probar y ver si se cumple el criterio de aceptación.

Como prueba sencilla vamos a probar el login de un usuario. En caso de que el usuario sea incorrecto devolverá una notificación de que los datos introducidos no son correctos y la aplicación no hará nada.



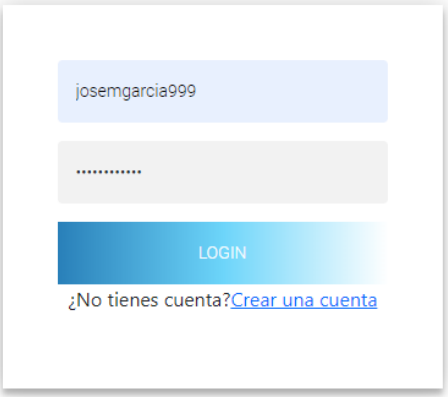




Ilustración 90: Inicio de sesión erroneo

En caso de que el inicio de sesión sea correcto, nos devolverá un mensaje indicando que el login ha sido correcto.

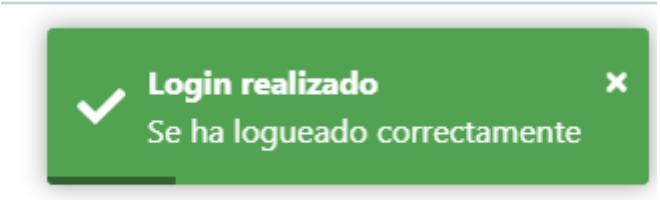



Ilustración 91: Notificación de inicio de sesión correcto.

## 5.3 Conclusiones de plan de pruebas

Una vez realizado el plan de pruebas completo para toda nuestra API Rest, nos damos cuenta de la importancia de las mismas ya que nos ayudan mucho a conseguir una mejor integridad y consistencia en los datos con los que trabajamos.

Este plan de pruebas me ha ayudado para poder pulir detalles que inicialmente se escapan como puede ser la introducción de datos nulos cuando no deberían de serlo, a la hora de eliminar una pista no eliminar las reservas asociadas a la misma. De la misma forma ocurre con los eventos.
Además, con las pruebas de usuario, nos hemos asegurado de una correcta generación de los tokens junto a la encriptación de la contraseña. También destacar que de esta forma aseguramos que el funcionamiento de la API Rest es lo más correcto posible porque hemos podido comprobar y establecer los códigos de respuesta que hemos considerado adecuados junto a validar que cada endpoint devuelve lo que debe devolver.

Teniendo en cuenta que hemos realizado 7 pruebas para la gestión de usuarios y aparte tenemos 21 endpoints. A su vez se dividen en dos pruebas:

a) Funcionamiento correcto .
b) Funcionamiento incorrecto.

Lo cual hace un total de 49 pruebas realizadas que junto a las pruebas
que se hacen a la vez que vamos avanzando en el desarrollo, se convierten en un número considerable para mostrar el funcionamiento de los endpoints de la API.

Para la parte del cliente teniendo en cuenta que se han realizado un total de 18 pruebas y han sido satisfactorias, hemos realizado un total de 67 pruebas.

Como conclusión, este plan de pruebas nos ha ayudado a detectar errores tanto a nivel conceptual como a nivel de implementación, además de asegurarnos de que todos los endpoints de la API como las llamadas desde el cliente funcionan correctamente y confirma una correcta validación de los mismos en los datos junto a la existencia de los suficientes mecanismos para hacerlo de forma efectiva y que no se complete la petición en caso de ser inválida.

# 6.- Conclusiones

Este apartado se corresponde con el punto final del proyecto.
En él realizaremos una pequeña retrospectiva basándonos en los resultados conseguidos junto con el cumplimiento de los objetivos inicialmente planteados. Además, se

complementará con una valoración del mismo de las expectativas y objetivos personales conseguidos con el mismo.
Finalmente en el punto 6.1 se incluirán posibles mejoras para el futuro.

En primer lugar, se ha realizado una pequeña investigación sobre las diferentes aplicaciones que cubrían las reservas de instalaciones deportivas de una ciudad. También se ha hecho un estudio sobre las diferentes tecnologías aportando sus ventajas e inconvenientes para posteriormente decidirnos.
Hecho esto se han definido los requisitos del sistema y una planificación temporal para el mismo finalizando con la creación del modelo de dominio del sistema.

A continuación, se ha realizado una fase de diseño en el que se han definido el diagrama entidad-relación y los diagramas de clase para ver cómo se relacionan entre ellas dentro del sistema. Además de indicar el diseño de la interfaz gráfica y de la API Rest que se va desarrollar junto a la descripción del plan de pruebas a seguir para verificar el correcto funcionamiento del sistema.

Finalmente está la fase de implementación en la que se dan detalles sobre el desarrollo realizado tanto en la parte del servidor como en la parte del cliente aportando ejemplos de cada uno junto con la metodología seguida para realizar el plan de pruebas de la aplicación.

A nivel de desarrollo, se han aplicado las directrices de las ingeniería del software. Se ha comenzado con un análisis del proyecto en el que se han concretado y definido todos los requisitos que debe de cumplir el sistema junto con el modelo de dominio del software. Una vez hecho esto se ha proseguido con una etapa de diseño en la que se ha intentado aplicar las técnicas y soluciones más adecuadas en base a lo obtenido en el análisis. Por último se ha realizado una última fase en la que se ha hecho mención a detalles de la implementación realizada junto a un plan de pruebas del software para validar y asegurar el correcto funcionamiento del sistema.

A nivel académico este proyecto me ha ayudado adquiriendo muchos conocimientos y aplicando todas las metodologías y habilidades que he aprendido durante el estudio de este grado, aportándome así, una nueva visión sobre un mercado que cada vez está más en auge, ya que a día de hoy, cada vez más son las empresas que permiten realizar la reserva de sus instalaciones de manera online.

Desde un punto de vista personal, la realización de este proyecto ha supuesto un gran desafío ya que nunca antes había afrontado un proyecto de tal magnitud y sentí bastante miedo al empezarlo. Conforme iba pasando el tiempo y fui investigando las tecnologías que iba a utilizar, algunas de las cuales nunca había usado, me sirvió como motivación para poder conseguir realizar este proyecto. Si tuviera que comentar una de la mayor dificultad que he tenido en el desarrollo de este proyecto ha sido el aprender a desarrollar Angular. Principalmente esto es debido al desconocimiento en TypeScript y la programación orientada en componentes, por lo que ha sido un gran reto
Estos meses han conllevado un gran esfuerzo a nivel personal pero a su vez la experiencia me ha fortalecido y enriquecido como persona.

Como conclusión, ha sido un proyecto muy bueno y que seguro que me será de gran utilidad de cara al desarrollo de nuevos trabajos en el ámbito laboral.

## 6.1 Mejoras y trabajos futuros

El trabajo realizado permite tener un buen proyecto inicial sobre el que seguir profundizando. Este proyecto posee una funcionalidad básica y correcta pero debido a la duración y extensión del proyecto hay otras que no se han podido abordar. Debido a ello, en este apartado se quiere indicar posibles nuevas mejoras de cara al futuro. Además, considero que el diseño, arquitectura y tecnologías elegidas en mi propuesta de solución contribuirán en el futuro a que sea relativamente sencillo incluir estas mejoras.

Ahora se van a indicar un conjunto de posibles mejoras que se podrían intentar realizar en un futuro:

- **Implementación de aplicación móvil**. Actualmente los usuarios realizan la mayor parte de sus gestiones mediante sus propios smartphones por lo que aunque la aplicación web funcione correctamente, una aplicación móvil mejoraría la UX.

- **Mejorar el front-end de la aplicación**. Debido al desconocimiento inicial de Angular como framework para el desarrollo de aplicaciones junto a la inexperiencia con TypeScript el diseño que se ha dado ha sido bastante pobre y gracias a la programación basada en componentes y el gran soporte que tiene Angular no sería muy complicado.

- Realizar **alojamiento web** en un servidor para así poder realizar reservas de las pistas de un polideportivo desde cualquier lugar sin necesidad de tener que desplegar la parte de cliente y servidor por separado.

# 7.- Bibliografía

[1] sandra.hall@wearesocial.net. (2022, 14 febrero). Digital Report 2022: El informe sobre las tendencias digitales, redes sociales y mobile. We Are Social Spain. https://wearesocial.com/es/blog/2022/01/digital-report-2022-el-informe-sobre-las-tendencias-digitales-redes-sociales-y-mobile/

[2] Las mejores App para la gestión de reservas de pistas de pádel e instalaciones deportivas. (2021, julio 5). Myturn; App gestion de aforo y piscinas - Myturn. https://www.myturn.es/las-mejores-app-para-la-gestion-de-reservas-de-pistas-de-padel-e-instalaciones-deportivas/

[3] Software de Gestión de Instalaciones Deportivas, Clubes Deportivos y Polideportivos | R24h. (s. f.). Reservas 24h. https://reservas24h.es/portal/default.aspx

[4] Software de Gestión de Reserva de Instalaciones - ClicAc. (s/f). Clicac.com., de http://clicac.com/ClicAcWeb/

[5] App de reservas y control de aforo. (2021, November 16). Myturn; App gestion de aforo y piscinas - Myturn. https://www.myturn.es/

[6] Qué es el Backend de una web y por qué es tan importante. (2017, April 8). Rafa Arjonilla. https://rafarjonilla.com/que-es/backend/

[7] Softtek. (2021, December 22). Los mejores Frameworks de backend para 2022. Softtek. https://softtek.eu/tech-magazine/software-trends/los-mejores-frameworks-de-backend-para-2022/

[8] Installation. (s/f). Laravel.com. Recuperado el 7 de junio de 2022, de https://laravel.com/docs/9.x

[9] Django documentation. (s/f). Djangoproject.com. 7 de junio de 2022, de https://docs.djangoproject.com

[10] Express - Infraestructura de aplicaciones web Node.js. (s/f). Expressjs.com. Recuperado el 7 de junio de 2022, de https://expressjs.com/es/

[11] W. Wheeler. Spring in Practice. Manning, 2013

[12] A. J. Rueda Ruiz, «Apuntes de la asignatura Desarrollo de Aplicaciones Empresariales,» Jaén.

[13] Spring Framework. (n.d.). Spring.I, from https://spring.io/projects/spring-framework/

[14]Spring framework overview. (n.d.). Spring.Io.From https://docs.spring.io/spring-framework/docs/current/reference/html/overview.html

[15] (S/f). Arimetrics.com. Recuperado el 7 de junio, de https://www.arimetrics.com/glosario-digital/frontend

[16] Empezando. (s/f). Reactjs.org. Recuperado el 1 de septiembre de 2022, de https://es.reactjs.org/docs/getting-started.html

[17] ¿Qué es Vue? (s/f). Lenguajejs.com. Recuperado el 7 de junio, de https://lenguajejs.com/vuejs/introduccion/que-es-vue/

[18]Angular. (s/f). Angular.Io. Recuperado el 7 de junio, de https://angular.io

[19]Handa, U. (2021, octubre 27). 7 razones para usar angular para su aplicación web en 2022. Cynoteck. https://cynoteck.com/es/blog-post/reasons-to-use-angular-for-your-web-app/

[20] Rehkopf, M. (n.d.). Historias de usuario. Atlassian. from https://www.atlassian.com/es/agile/project-management/user-stories

[21] Victor Manuel Rivas Santos, «Apuntes de la asignatura Desarrollo Ágil,» Jaén.

[22] Salario de Programador/a junior en España. (s/f). Indeed.com. Recuperado el 23 de junio de 2022, de https://es.indeed.com/career/programador-junior/salaries

[23] Retención IRPF 2022: ¿Cuánto me tienen que retener en mi nómina? (2022, mayo 9). Blog Oficial de Bankinter. https://www.bankinter.com/blog/finanzas-personales/como-calcular-retenciones-irpf-nomina

[24] Introducción a DDD y arquitectura hexagonal con un ejemplo de aplicación en Java. (2021, February 7). Blog Bitix. https://picodotdev.github.io/blog-bitix/2021/02/introduccion-a-ddd-y-arquitectura-hexagonal-con-un-ejemplo-de-aplicacion-en-java/

[25] rest-apis. (2021, abril 6). Ibm.com. https://www.ibm.com/es-es/cloud/learn/rest-apis

[26] ¿Qué son las pruebas de caja negra? (2022, junio 3). KeepCoding Tech School. https://keepcoding.io/blog/que-son-las-pruebas-de-caja-negra

[27] Postman. (s/f). Postman.com. Recuperado el 28 de julio de 2022, de https://www.postman.com

[28] Calle, N. R. (2020, November 8). Ejemplo de Arquitectura Hexagonal con Spring Data. Refactorizando; Noel Rodríguez Calle. https://refactorizando.com/ejemplo-de-arquitectura-hexagonal/

[29] Mihalcea, V., Ebersole, S., Boriero, A., Morling, G., Badner, G., Cranford, C., Bernard, E., Grinovero, S., Meyer, B., Ferentschik, H., King, G., Bauer, C., Andersen, M. R., Maesen, K., Vansa, R., & Jacomet, L. (s/f). Hibernate ORM 6.1.2.Final user guide. Jboss.org. Recuperado el 28 de Julio, de https://docs.jboss.org/hibernate/orm/6.1/userguide/html_single/Hibernate_User_Guide.html

[30] de Raúl, V. M. C. (s/f). Proyecto Lombok, ¡facilítame la vida! Paradigmadigital.com. Recuperado el 1 de septiembre de 2022, de https://www.paradigmadigital.com/dev/proyecto-lombok-facilitame-la-vida/

[31]Angular. (n.d.-b). Angular.Io. Recuperado el 28 de Julio, de https://angular.io/api/router/RouterLink

[32] Guards en Angular, ¿Cómo funcionan? (s/f). Binary Coffee. Recuperado el 28 de julio de 2022, de https://binary-coffee.dev/post/guards-en-angular-como-funcionan

[33] (N.d.-b). Medium.com. Recuperado el 28 de julio de 2022, from https://medium.com/@insomniocode/angular-autenticación-usando-interceptors-a26c167270f4

[34] Usar clases e Interfaces en los servicios Angular. (n.d.). Desarrolloweb.com. Recuperado el 28 de julio de 2022, de https://desarrolloweb.com/articulos/clases-interfaces-servicios-angular.html

[35] Npm: Ngx-toastr. (s/f). Npm. Recuperado el 28 de julio de 2022, de https://www.npmjs.com/package/ngx-toastr

[36] Angular toastr. (s/f). Angular Toastr. Recuperado el 28 de julio de 2022, de https://ngx-toastr.vercel.app

# APÉNDICES

## Apéndice I. Descripción de contenidos suministrados

En este apéndice vamos a indicar los ficheros que adjuntamos con el contenido del proyecto. En la raíz de la carpeta se encontrarán 3 ficheros, la memoria del Trabajo de fin de grado con nombre TFG_Garcia_Ronda_Jose_Manuel.pdf. Una parte llamada Anexos donde se incluirá la documentación necesaria para proceder con los trámites para la entrega y la carpeta Material Adicional donde incluiremos los ficheros implementados y diagramas. Partiremos de la carpeta Material Adicional que habrá dentro del zip TFG_Garcia_Ronda_Jose_Manuel.zip.

- **sources**: Contendrá todos los ficheros fuente tanto de la parte del servidor como del cliente

- **diagramas:** Tendrá los ficheros fuente de los diagramas de clase y secuencia mostrados en el informe.

# Apéndice II. Manual de instalación del sistema.

## 7.1 BackEnd

En primer lugar lo que haremos será descomprimir la carpeta TFG_Garcia_Ronda_Jose_Manuel.zip. A su vez dentro de ella accederemos a la carpeta Material Adicional, Sources y Backend. Tendremos que hacer esto para ejecutarlo.

- Tendremos que instalar IntelliJ y cargar todos los ficheros. Tras ello tendremos que un realizar un mvn clean install para instalar correctamente todas las dependencias y finalmente ejecutarlo desde el propio IDE.

Esta alternativa desplegará el proyecto en el puerto 8080.

## 7.2 FrontEnd

### BackEnd

En primer lugar lo que haremos será descomprimir la carpeta TFG_Garcia_Ronda_Jose_Manuel.zip. A su vez dentro de ella accederemos a la carpeta Material Adicional, Sources y Backend.. Para ejecutarlo tendremos que tener node instalado en nuestro pc.

Para instalar el front abriremos una cmd dentro de la raíz del proyecto de front y ejecutaremos los siguientes comandos:

- npm install –force :Instalará todas las librerías y dependencias necesarias.
- ng serve: Desplegará el proyecto en el puerto 4200.

# Apéndice III. Manual de uso de usuario.

En este apéndice se va a mostrar como realizar la instalación de nuestro proyecto en el sistema. Para ellos lo dividiremos en dos puntos, uno para la parte del servidor y otro para la parte del cliente Tener en cuenta que hay creados dos usuarios cuyas credeciales son las siguientes

- usuario: josemgarcia999 contraseña: 1234. Es administrador.
- usuario:juanmy999; contraseña: 1234. Es usuario.

Una vez desplegamos la aplicación, como podemos observar en la siguiente ilustración veremos el home de la aplicación.

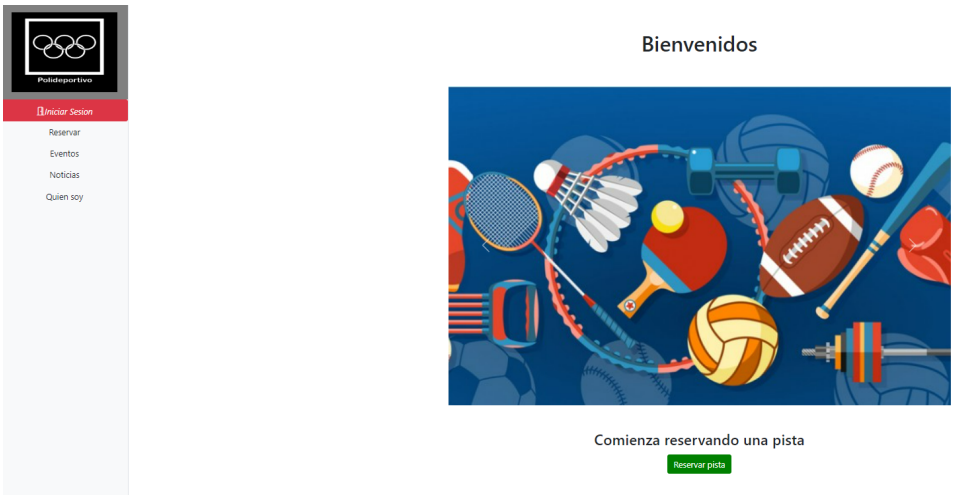



**Ilustración 92: Home de la aplicación.**

A continuación, para facilitar al usuario la interacción, se ha añadido un botón para que se pulse sobre él para comenzar a reservar una pista.

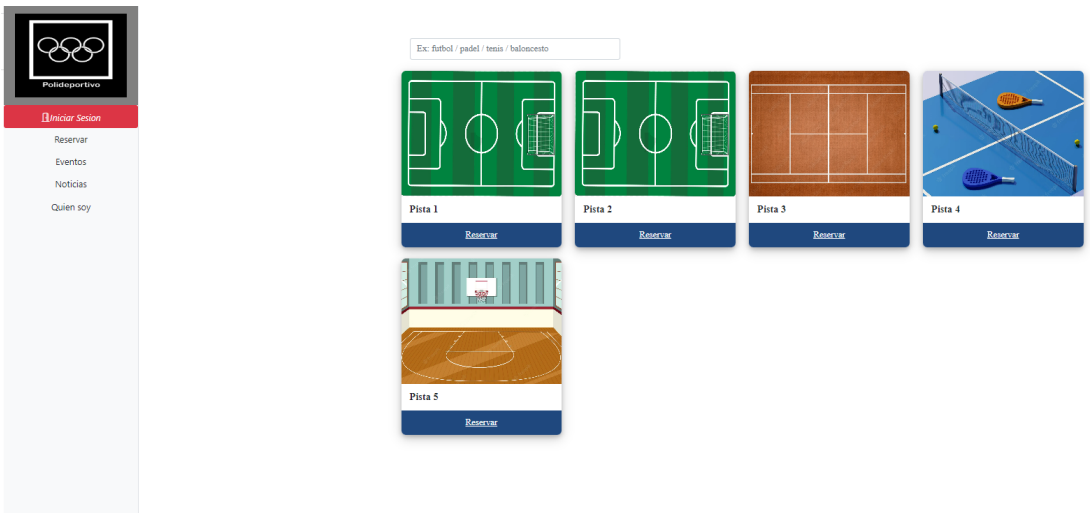



**Ilustración 93: Pistas existentes.**

Al pulsar sobre las pistas veremos el detalle de la misma, tanto la descripción como el precio. Una vez que elijamos la pista que queremos reservar le daremos a reservar y como aparecerá en la siguiente ilustración nos llevará a la vista de inicio de sesión.

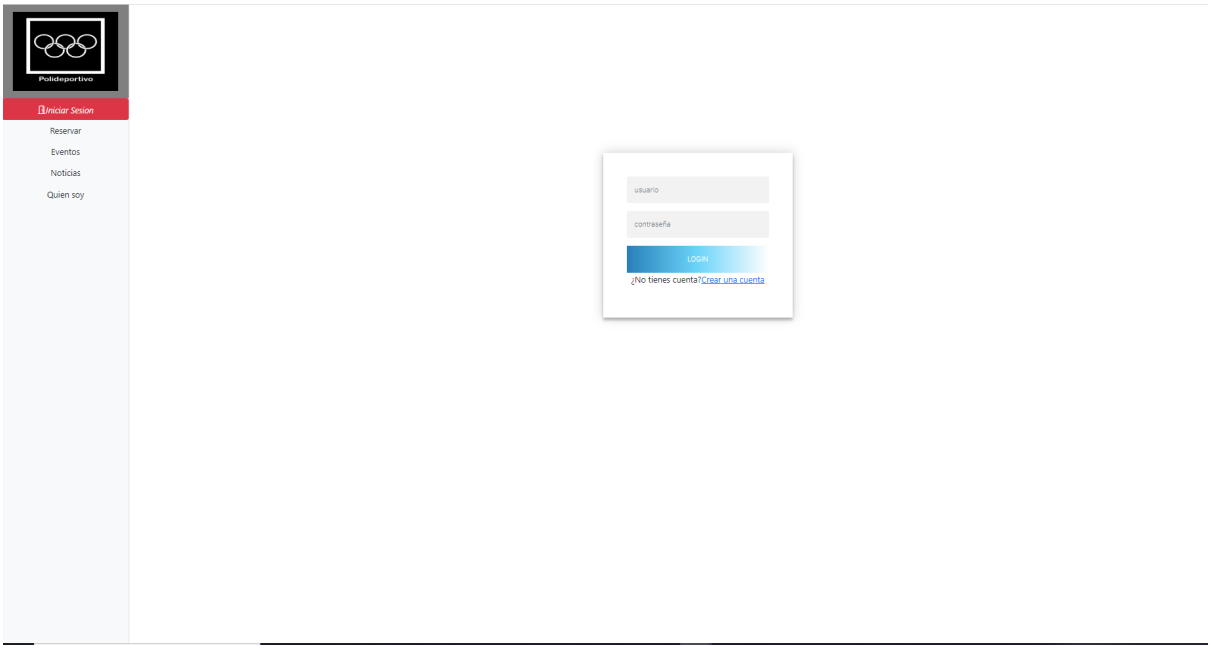



**Ilustración 94: Vista Login.**

En caso de ser un usuario nuevo, pulsaremos sobre Crear una cuenta y nos llevará a un formulario con los campos a rellenar.

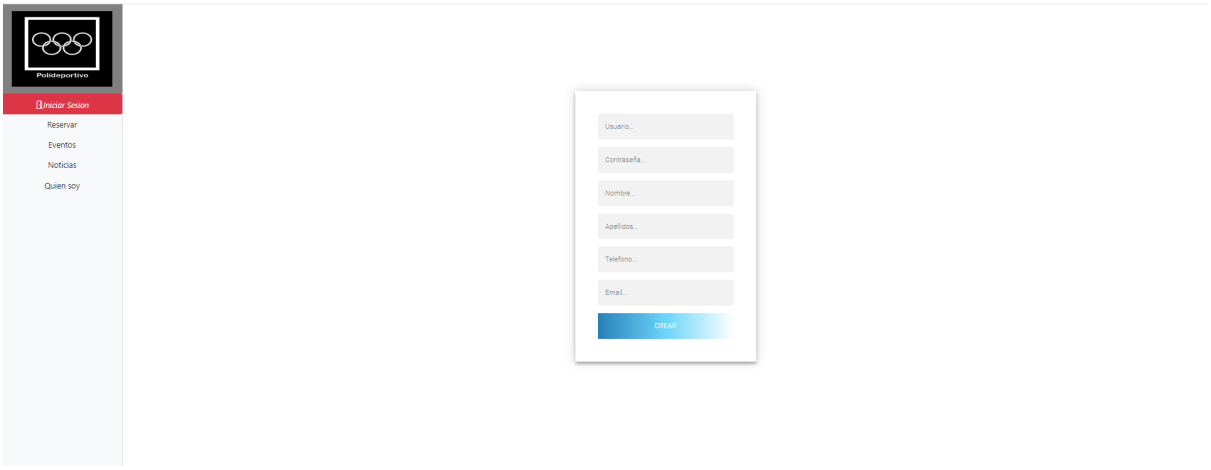



Ilustración 95: Vista Crear Cuenta.

Una vez creada la cuenta nos redireccionará hacia la página de inicio de sesión y una vez hecho esto nos redireccionará al home. En caso de ya tener cuenta nos llevará a la vista donde se encuentran todas las pistas.

Una vez elegimos la pista a reservas nos llevará a otra vista en la que veremos hora por hora la disponibilidad de la pista. Una franja roja indicará que está reservada y nada indicará que está libre.

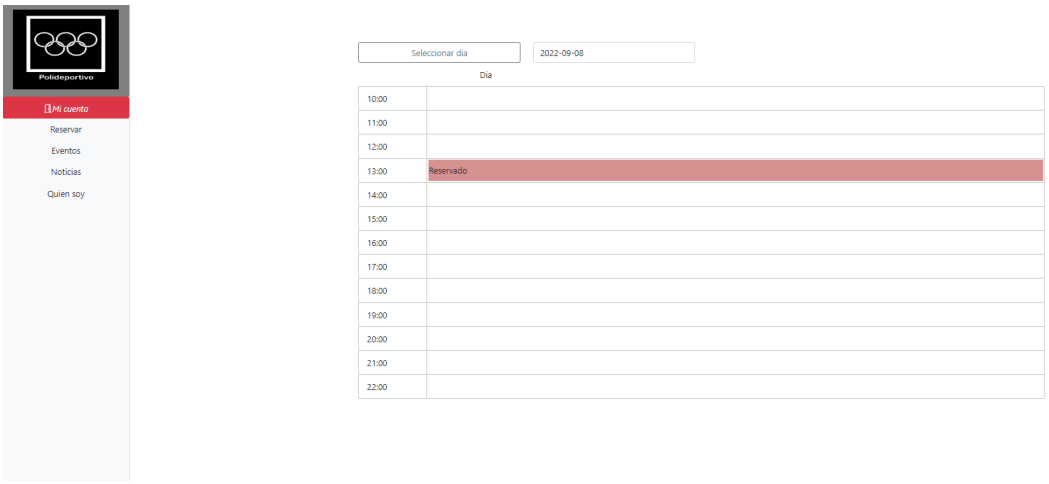



Ilustración 96: Vista Horario de pista.

Tras elegir la hora se nos desplegará un pequeño pop-up para confirmar la reserva para poder elegir otra hora en caso de haber elegido mal.

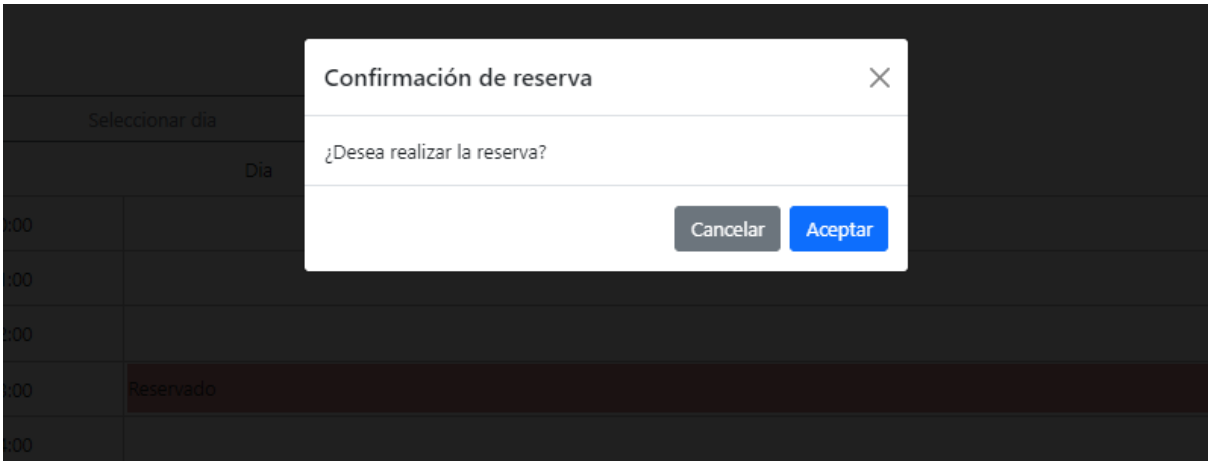



Ilustración 97: Pop-up de confirmación de reserva.

Una vez confirmada la reserva se pondrá esa hora como reservado y se nos mostrará una notificación de reserva realizada.

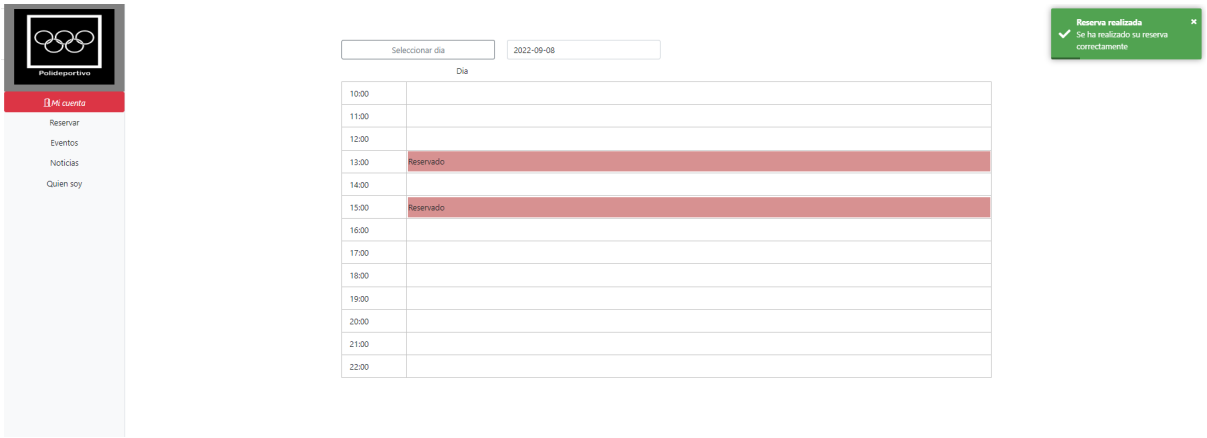



Ilustración 98: Pop-up de confirmación de reserva.

Si pulsamos sobre Mi cuenta podremos ver las reservas que hemos realizado, los eventos a los que nos hemos apuntado y cerrar sesión de la misma.

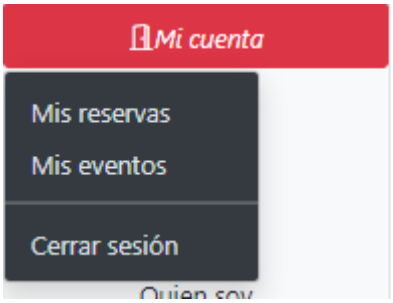



Ilustración 99: Opciones de usuario.

Al pulsar sobre Mis reservar veremos tarjetas que se corresponden con cada reserva y los datos de la misma de la misma forma que en la siguiente ilustración.

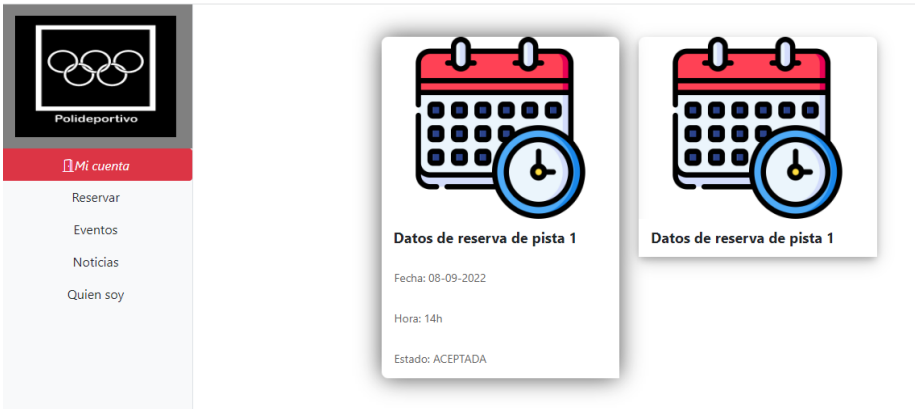



Ilustración 100: Vista de mis reservas.

De igual forma procederemos con los eventos, accederemos a su vista y le daremos a apuntarse. Se nos indicará que todo ha ido bien con una notificación.

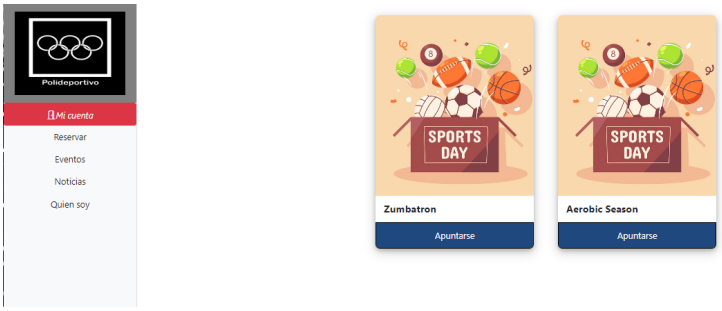



Ilustración 101: Vista de eventos.

De igual forma que con las reservas, si accedemos a mis eventos se mostrarán los eventos a los que está apuntado el usuario.

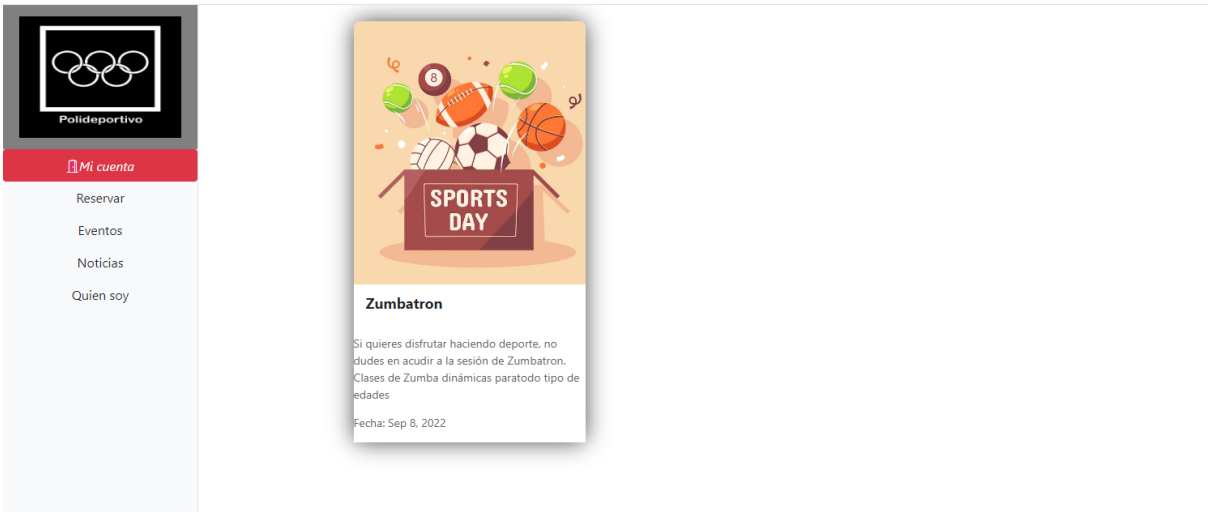



Ilustración 102: Vista de mis eventos.

El apartado noticias simplemente será de información para leer noticias que haya publicadas con información del polideportivo.


Ilustración 103: Vista de noticias.

En el apartado de Quien soy será meramente informativa con datos e información sobre mi y del proyecto desarrollado.

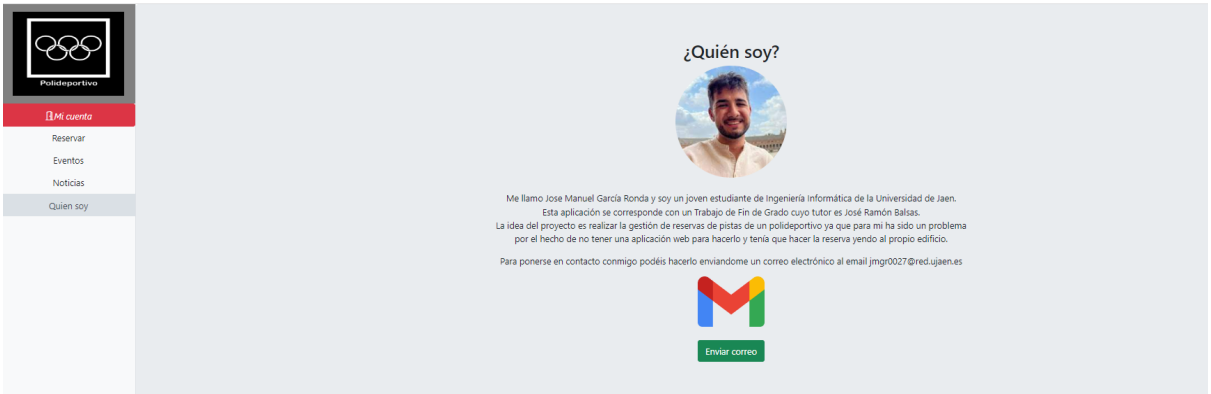



Ilustración 104: Vista de Quien soy.

En caso de iniciar sesión con un administrador se mostrará el apartado de Admin en el que incluirán las opciones de administración.


Ilustración 105: Vista de administrador.

Tras ello veremos la Home del administrador en la que podremos ver las diferentes opciones que tiene.

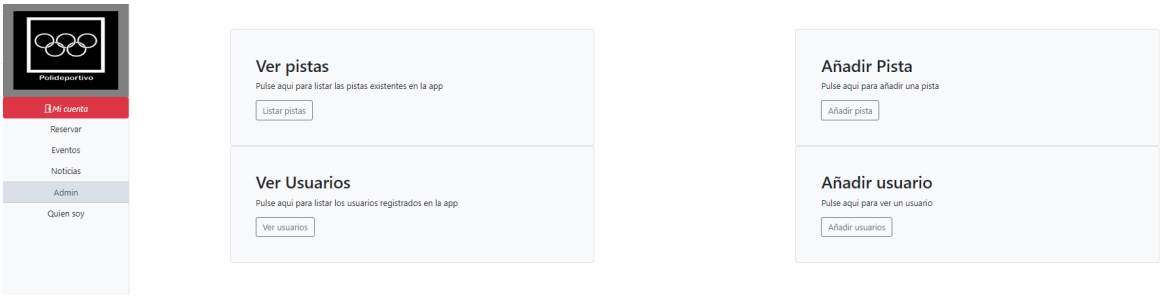



Ilustración 106: Vista de home de admin.

Para hacernos una idea veremos el ejemplo de ver pistas. Tendremos una tabla con los datos que existan junto con botones para cada una para eliminar o editar. Además podremos añadir una pista.

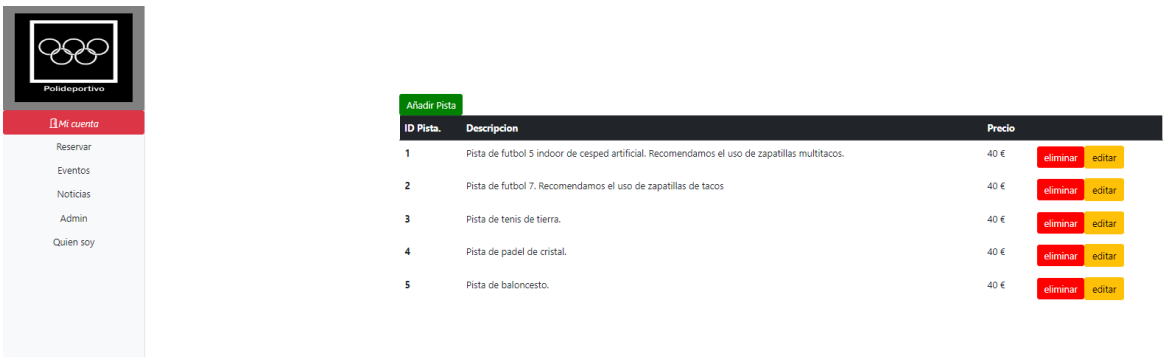



Ilustración 107: Vista listar pistas.

# Apéndice IV. Pruebas de API Rest

## 8.1 Gestión de usuarios

En primer lugar se va a proceder con las pruebas para la gestión de usuarios, de esta forma se comprueba que la funcionalidad necesaria para la aplicación responde de forma correcta. Para ello vamos a ir endpoint por endpoint donde empezaremos por registrar un usuario cualquiera, pasaremos por hacer login para ver que se genera el token JWT correctamente y finalmente comprobaremos dos funciones a nivel de administrador como son añadir rol y comprobar administrador para cada uno.

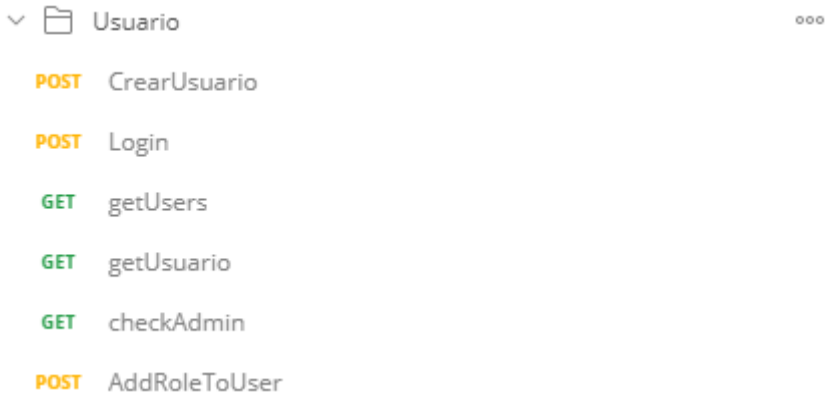



Ilustración 108: Colección EndPoints de Usuario en PostMan

- Creación de usuario. Se corresponde con una petición de tipo POST y el endpoint del mismo es el siguiente: www.localhost:8080/api/empresa/user/save. Este requerirá los campos de un usuario en formato JSON siguiendo el siguiente formato:

```
{
  "username":"username",
```

```
  "password":"password",
  "nombre":"nombre",
  "apellidos":"apellidos",
  "telefono":”telefono",
  "email":"email@email.com",
  "roles":"roles"
}
```


En caso de que la creación sea correcta, se devolverá un código HTTP 201 que se corresponde con CREATED, junto a todos los campos del usuario.

Los datos del usuario que vamos a crear son los siguientes:


```
{
  "username":"prueba123",
  "password":"1234",
  "nombre":"Prueba",
  "apellido":"Usuario de prueba",
  "telefono":"666666666",
  "email":"prueba@gmail.com",
  "roles":"ROLE_USER"
}
```


Al enviar la petición lo que obtenemos es lo siguiente:

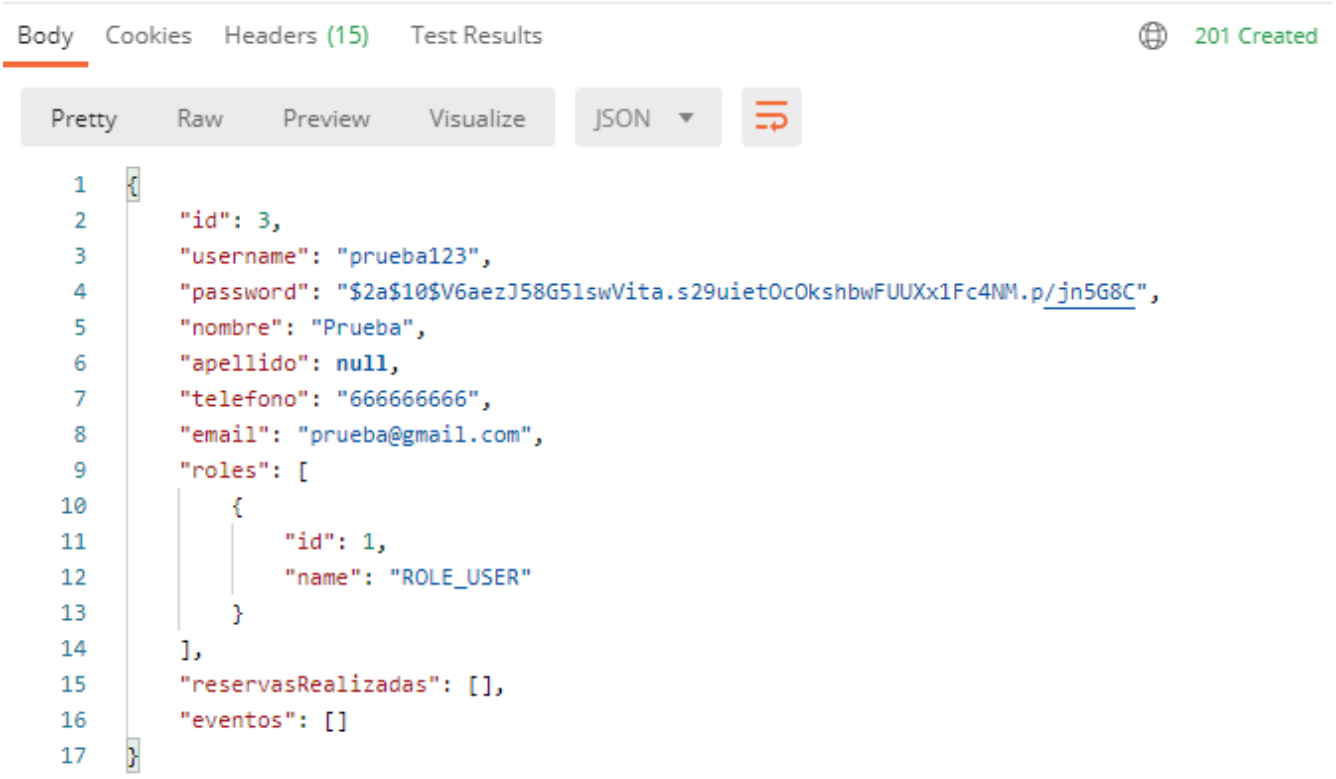



*Ilustración 109: Salida petición a endpoint de crear usuario.*

Como se puede observar, se ha devuelto un código 201 con los datos del usuario que hemos introducido y se corresponden con los datos de entrada, además de ver las reservasRealizadas y eventos apuntados, que como es lógico ahora mismo no tienen nada puesto que el usuario está recién creado.

- Login de Usuario. Se corresponde con una petición de tipo POST y la url es www.localhost:8080/api/empresa/login. En este caso, para realizar la petición el body no será un objeto de tipo JSON si no que usaremos un formato de x-www-form-urlencoded para introducir los datos.En caso de que exista el usuario que hemos introducido se devolverá un código HTTP 200 junto al token de acceso para el usuario que hemos introducido. El formato de los campos del usuario es el siguiente

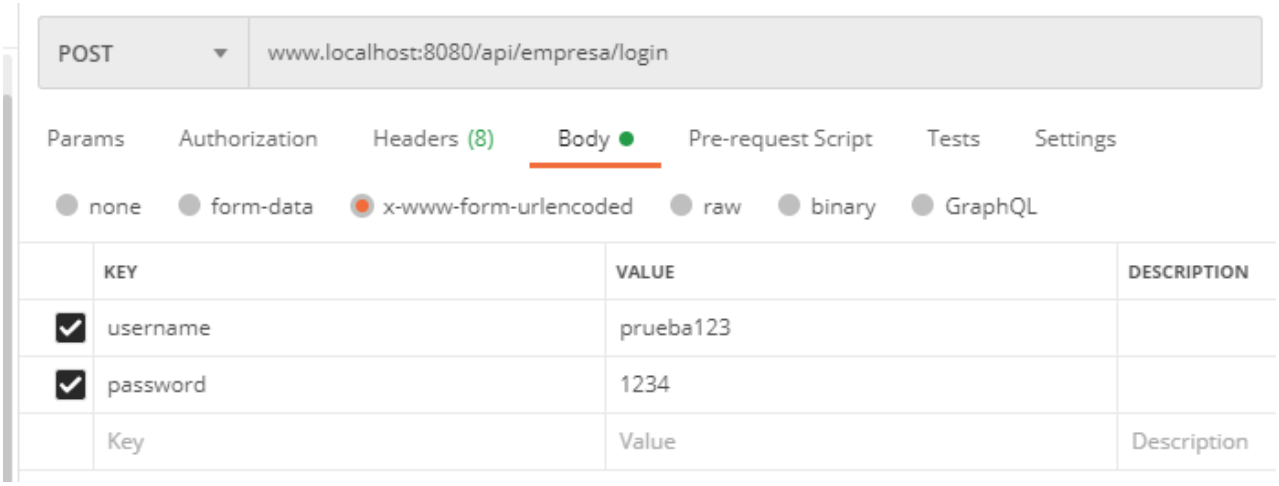



*Ilustración 110: Parámetros de entrada para endpoint de login.*

Una vez introducidos los datos, cómo se corresponden con el valor que tiene el usuario de prueba que se ha creado anteriormente, se genera el token de acceso para el mismo junto al de refresco en caso de que el de acceso expire.

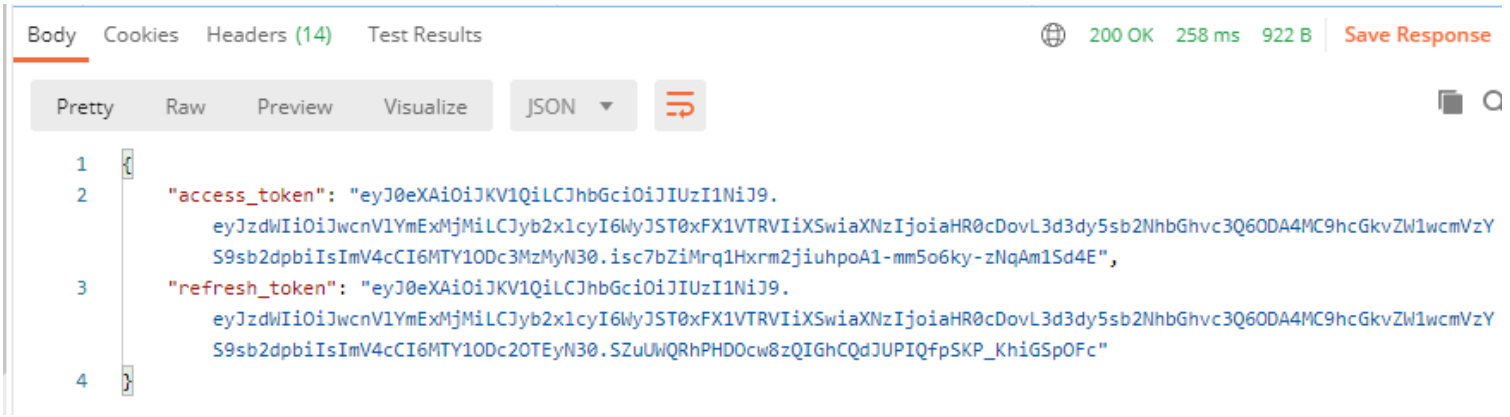



*Ilustración 111: Token JWT generado para el usuario creado.*

En caso de que hubiéramos introducido un par usuario-contraseña incorrectos se habría devuelto un código 401 para indicar que el login no ha sido correcto.

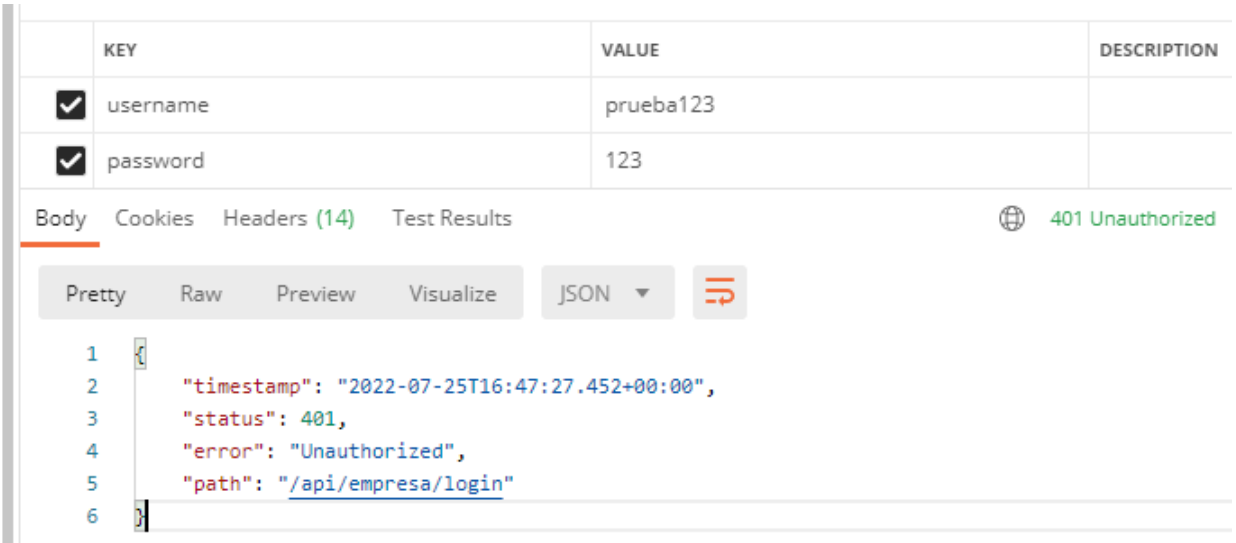


*Ilustración 112: Login incorrecto.*

- GetUsers: Se corresponde con una petición de tipo GET y la url de este endpoint es www.localhost:8080/api/empresa/users. Este endpoint necesitará el usuario y la contraseña por el body siguiendo el formato del login (usuario y la contraseña) junto el token en formato Bearer token en la parte de Header. Nos devolverá un código 200 junto con la lista de usuarios creados en la aplicación en caso correcto y en caso contrario un 403 si no está permitido o el token expira.

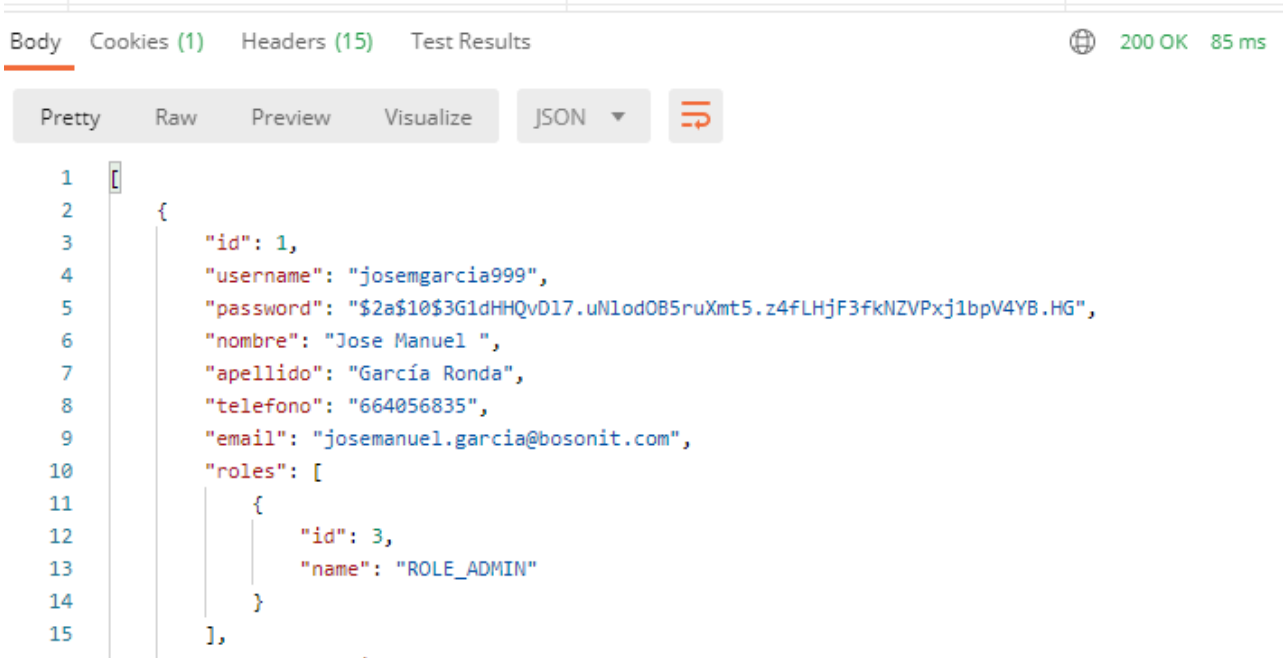



*Ilustración 113: Fragmento de respuesta de endpoint de listar los usuarios de la aplicación.*

- GetUsuario: Se corresponde con un petición de tipo GET y la url de este endpoint es: www.localhost:8080/api/empresa/users/{nombreUsuario}. Este endpoint devuelve un código 200 junto con los datos del usuario pasado por el path. Este endpoint no necesita parámetros de ningún tipo.

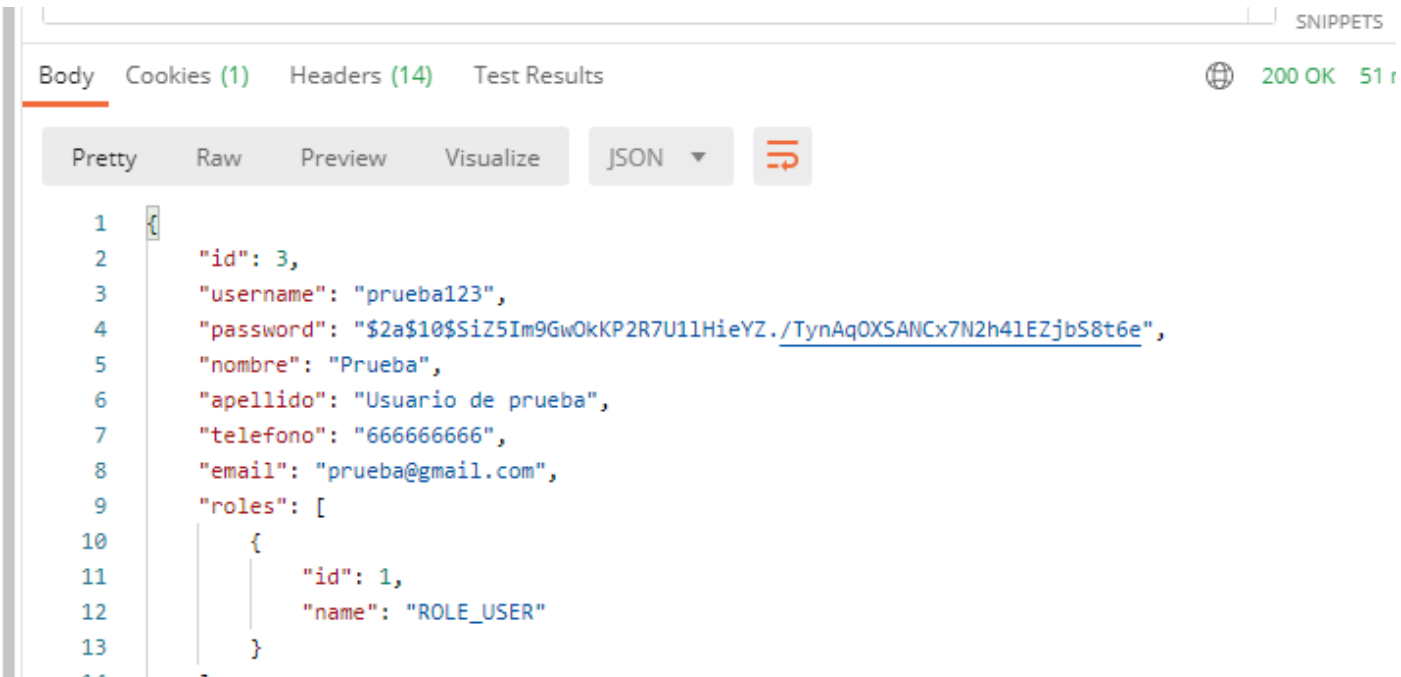


*Ilustración 114:Respuesta de endpoint de obtener datos de un usuario.*

- CheckAdmin: Se corresponde con un petición de tipo GET y la url de este endpoint es: www.localhost:8080/api/empresa/checkAdmin/{nombreUsuario} . Este endpoint devuelve un código 200 junto a un booleano que será true si el usuario es administrador o false en caso contrario.

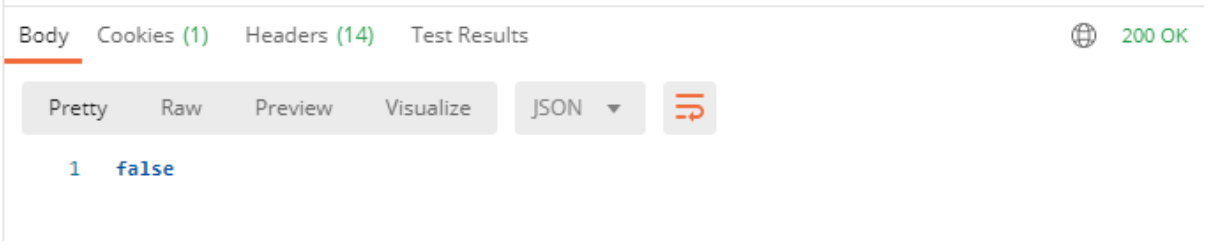



*Ilustración 115:Respuesta de endpoint para comprobar si un usuario posee rol de administrador.*

- AddRoleToUser: Se corresponde con una petición de tipo POST y la url de este endpoint es: www.localhost:8080/api/empresa/role/addtouser. Requiere por body un objeto de tipo JSON con el nombre de usuario y nombre del rol que se quiere asignar.

  El formato de objeto es el siguiente:

  ```
  {
    "username":"username"
    "roleName":"rolename"
  }
  ```

  Como ejemplo, se va a añadir al usuario que hemos creado anteriormente el rol de administrador. En caso correcto devolverá un código 200.

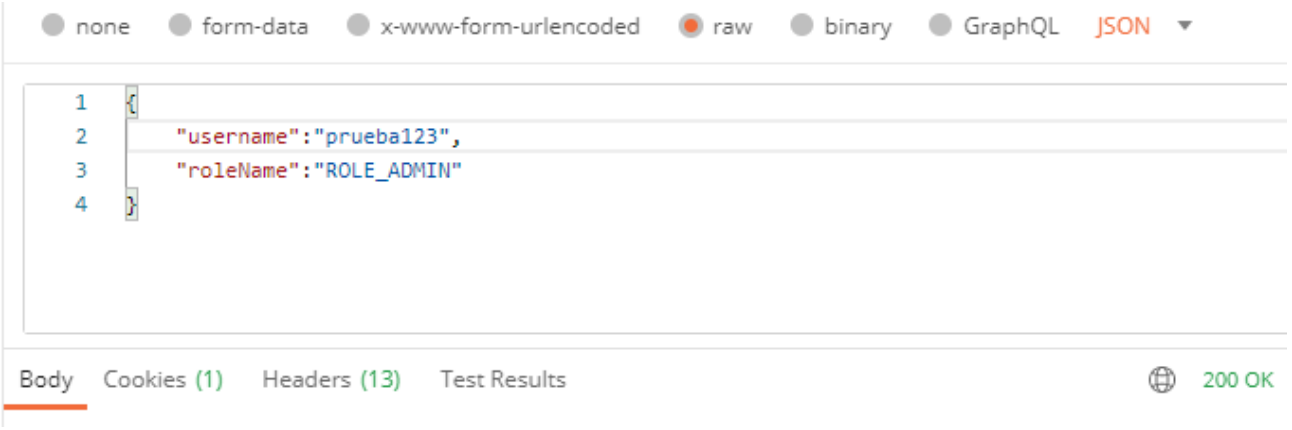



*Ilustración 116:Respuesta de endpoint añadir rol a un usuario.*

## 8.2 Prueba para gestión de reservas

### 8.2.1 Realizar una reserva

En este punto se va a realizar una prueba sobre la realización de reservas,ya que es la parte más importante y en la que se basa nuestra aplicación.En primer lugar tendremos que tener en cuenta las pistas existentes en la aplicación para saber el id de la pista que se va a reservar, ya que será necesario para poder hacer la reserva. A su vez será necesario introducir el nombre de usuario y la fecha y hora en la que se va a realizar la reserva de la pista. Estos datos se obtendrán del front pero en este caso lo haremos de forma manual para ver su correcto funcionamiento.

Este endpoint tendrá una petición de tipo POST y la url del endpoint es: www.localhost:8080/api/empresa/reservas.En caso de que la reserva se realice correctamente, esta devolverá un código 200 junto con los datos de la reserva. Además enviará un correo electrónico con los datos de la reserva al correo que tenga asociado el usuario que realiza la misma. Este endpoint requiere como entrada un objeto de tipo JSON con el siguiente formato:

```
{
  "fechaReserva":"YYYY-MM-DD",
  "horaReserva":"0.00",
  "username":"nombreUsuario",
  "idPista":"idPista"
}
```

Como ejemplo, vamos a suponer que el usuario que hemos creado quiere reservar la pista 1 para el día 14 de julio a las 17:00 de la tarde.


```
{
  "fechaReserva":"2022-07-14",
  "horaReserva":"17.00",
  "username":"prueba123",
  "idPista":"1"
}
```


Como podemos observar, este es el correo generado cuando la reserva se realiza de forma correcta, en la que se muestra la fecha y la hora junto al importe de la pista y el estado de la misma, que ha sido aceptada.

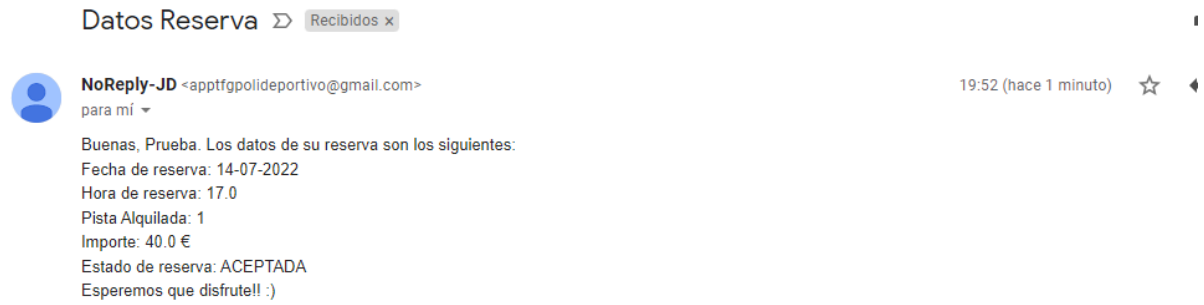



*Ilustración 117: Correo generado con los datos de la reserva realizada.*

Además, este endpoint devuelve los datos de la propia reserva en formato JSON junto a un código 200.


*Ilustración 118:Respuesta de endpoint de realizar reserva.*

En caso de que la reserva sea incorrecta se devolverá un código 422 ya que es una excepción que se ha creado para tratar este caso con el mensaje de pista reservada. Esto es lo que ocurre en caso de que reservemos la pista a la misma fecha y hora.


*Ilustración 119:Respuesta con código 422 de endpoint de realizar reserva.*

## 8.2.2 Eliminar una reserva

Una vez realizadas las pruebas para realizar las pruebas para realizar una reserva vamos a realizar las pruebas necesarias para eliminarlas. Este endpoint es más sencillo que el anterior ya que solamente necesitaremos pasarle por path el id de la reserva que deseemos eliminar.

Este endpoint será de tipo DELETE y la ruta del mismo es la siguiente: www.localhost:8080/api/empresa/reservas/{id}. En caso de eliminar correctamente la reserva devolverá un código 200 y de intentar eliminar una reserva que no existe devolverá un error 404 con un mensaje de error para el usuario indicando lo ocurrido.

Para realizar una prueba vamos a tener en cuenta la reserva realizada en la prueba anterior. En primer lugar vamos a intentar eliminar una reserva que no hayamos creado aún, como por ejemplo la 3.

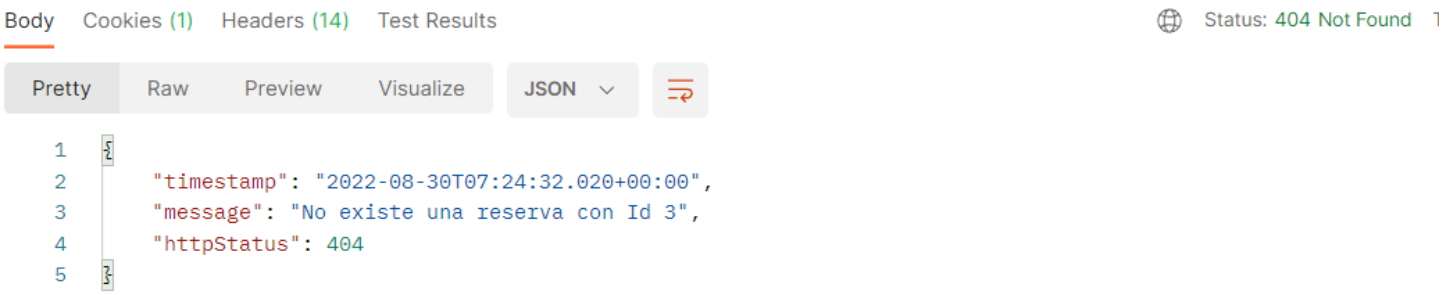



*Ilustración 120: Respuesta con código 404 de endpoint de eliminar reserva.*

Ahora si intentamos eliminar la reserva con el id de la reserva creada anteriormente la respuesta que obtendremos será un código 200 para indicarnos que se ha eliminado correctamente.

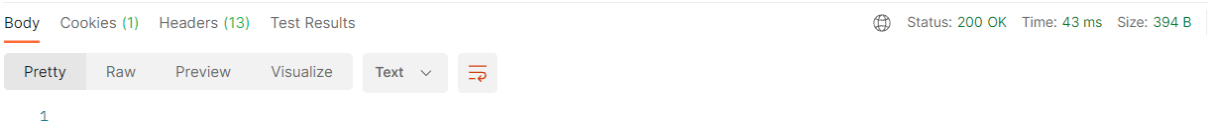



*Ilustración 121: Respuesta con código 200 de endpoint de eliminar reserva.*

### 8.2.3 Mostrar todas las reservas

En este punto se va a realizar la prueba para mostrar todas las reservas que se han realizado en la aplicación.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/reservas. En todos los casos devolverá un código 200 con todas las reservas realizadas. Si ahora mostramos la lista de reservas nos devolverá una lista vacía.

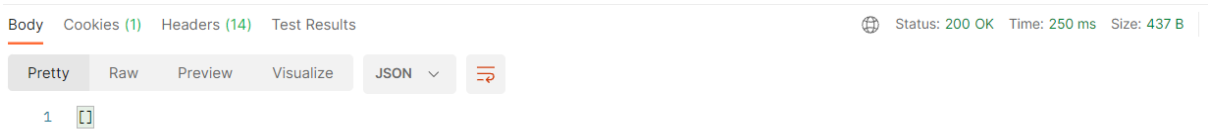



*Ilustración 122: Respuesta con código 200 de endpoint de mostrar reserva.*

Para comprobar el funcionamiento hemos añadido dos reservas nuevas con los usuarios josemgarcia999 y juanmy999.

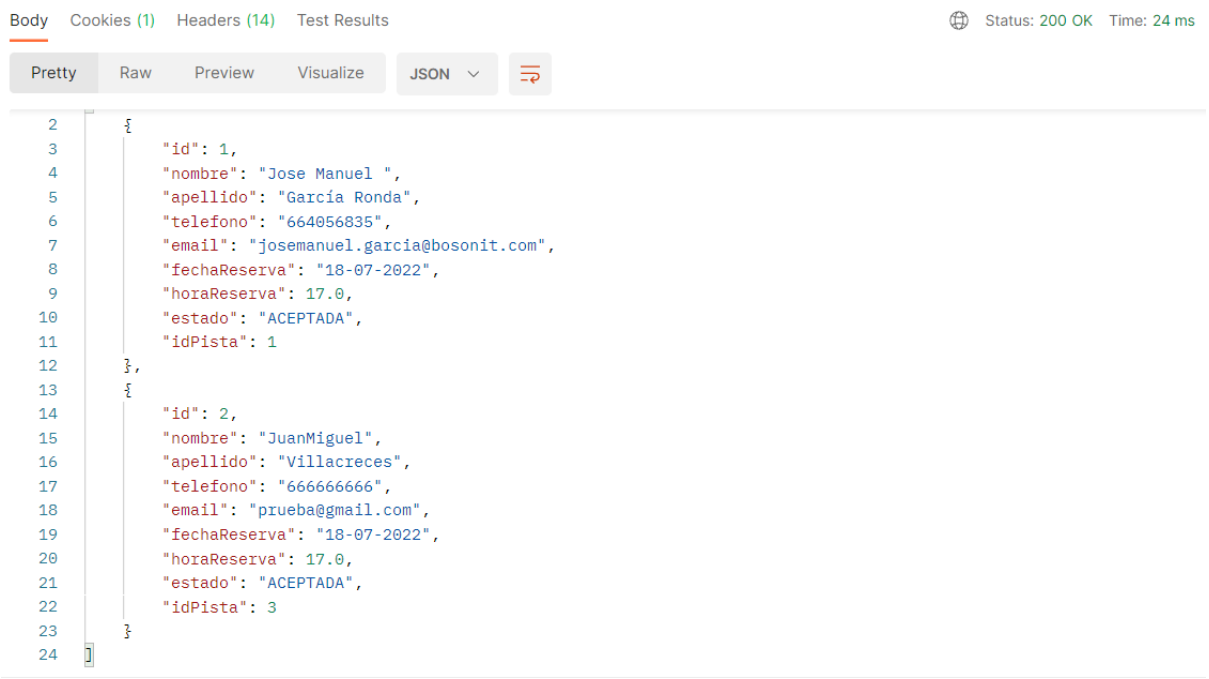



*Ilustración 123: Respuesta con código 200 de endpoint de mostrar reserva.*

### 8.2.4 Mostrar datos de una reserva

En este punto se va a realizar la prueba para mostrar los datos de una reserva.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/reservas/id. Para comprobar las dos posibles respuestas de este endpoint buscaremos dos reservas, una que exista y otra que no. En el caso de que no exista nos devolverá un mensaje de error con un código 404.


*Ilustración 124: Respuesta con código 404 de endpoint de mostrar datos de una reserva.*

Si ahora queremos ver los datos de una de las reservas que tenemos ya creadas nos devolverá un DTO con los datos de la misma junto a un código 200.

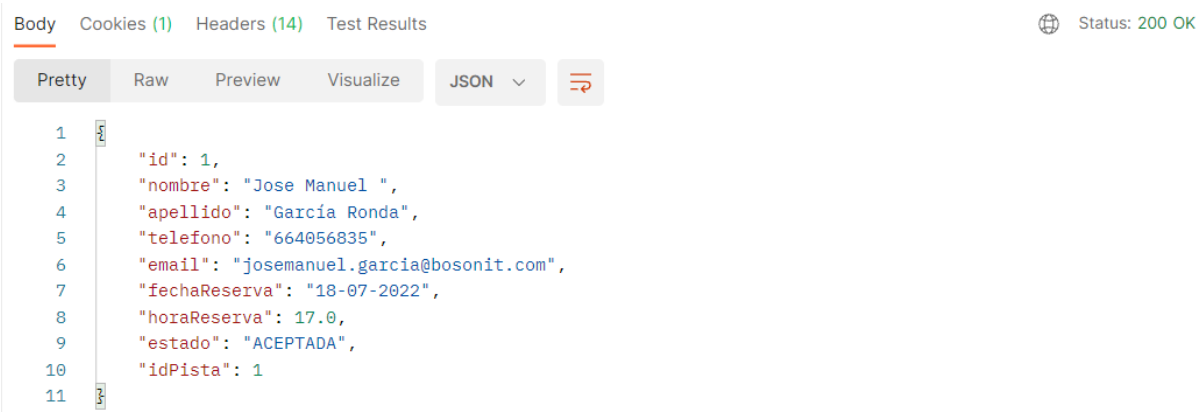



*Ilustración 125: Respuesta con código 200 de endpoint de mostrar datos de una reserva.*

## 8.3 Prueba para gestión de pistas

En este punto vamos a realizar las pruebas relacionadas con la gestión de pistas en la aplicación.

### 8.3.1 Mostrar todas las pistas

Vamos a realizar las pruebas relacionadas como obtener todas las pistas que existan.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/pista. En todos los casos devolverá un código 200 con todas las pistas existentes. Si ahora mostramos la lista de pistas nos devolverán las cinco pistas que se crean inicialmente.

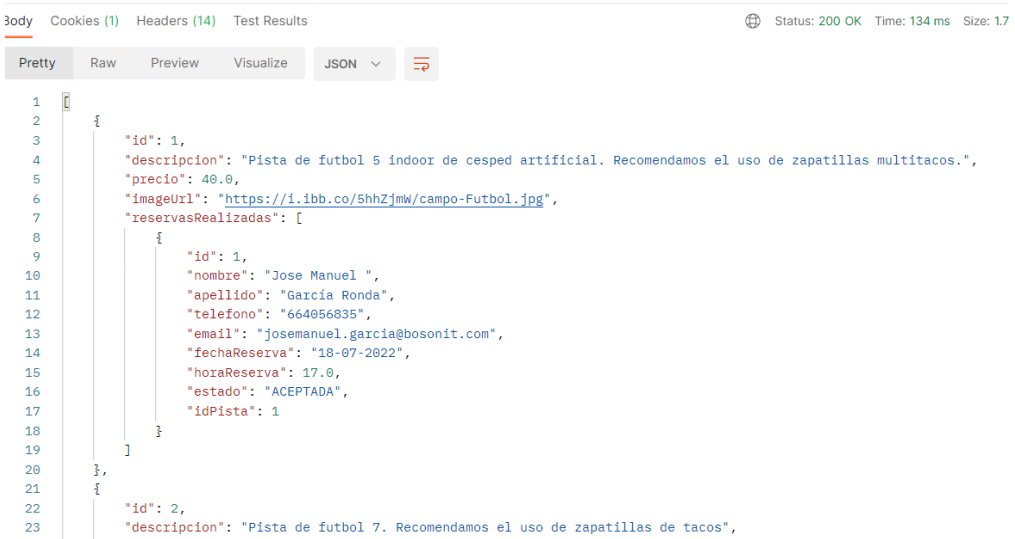



*Ilustración 126: Respuesta con código 200 de endpoint de mostrar todas las pistas.*

### 8.3.2 Mostrar datos de una pista

En este punto se va a realizar la prueba para mostrar los datos de una pista.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/pista/id. Para comprobar las dos posibles respuestas de este endpoint buscaremos dos pistas, una que exista y otra que no. En el caso de que no exista nos devolverá un mensaje de error con un código 404.

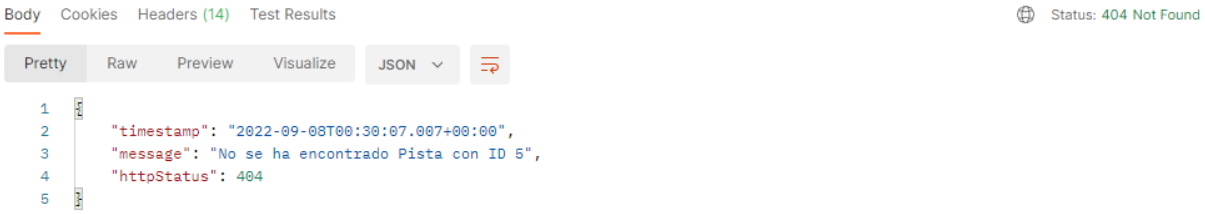



*Ilustración 127: Respuesta con código 404 de endpoint de mostrar datos de una pista.*

Si ahora queremos ver los datos de una de las pista que tenemos ya creadas nos devolverá un DTO con los datos de la misma junto a un código 200.

### 8.3.3 Actualizar una pista

En este punto vamos a comprobar el endpoint para actualizar los datos de una pista.

Este endpoint es de tipo PUT y la ruta del endpoint es www.localhost:8080/api/empresa/pista/id junto a un body en el que le pasaremos un inputDto con los datos de la pista que queramos modificar. Un ejemplo de input es el siguiente:

```
{
  "descripcion":"descripcion de prueba",
  "precio":"99.5"
}
```

Sabiendo esto, en caso de que la actualización se haya realizado correctamente devolveremos un código 200 con los campos de la reserva actualizados.Para ello veremos el valor de la pista 1 antes de modificar sus atributos y que devuelve tras hacer la llamada al mismo.

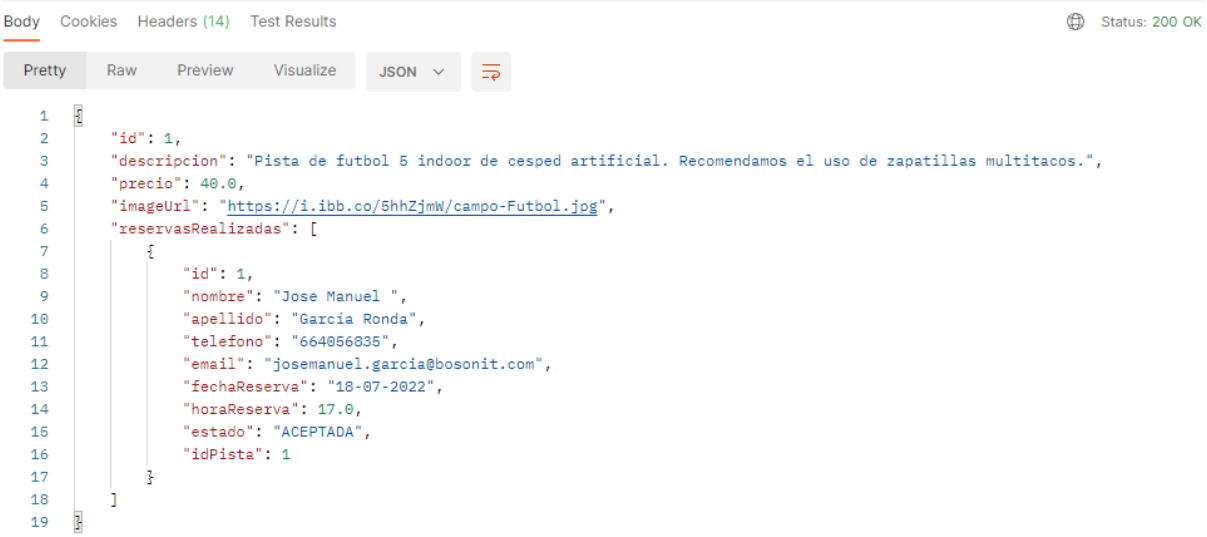



*Ilustración 128: Respuesta con código 200 de endpoint de mostrar datos de pista 1.*

Una vez hemos visto el valor de los atributos de la pista, vamos a actualizarle el precio y la descripción con los mismos valores que se han puesto en el input de ejemplo.

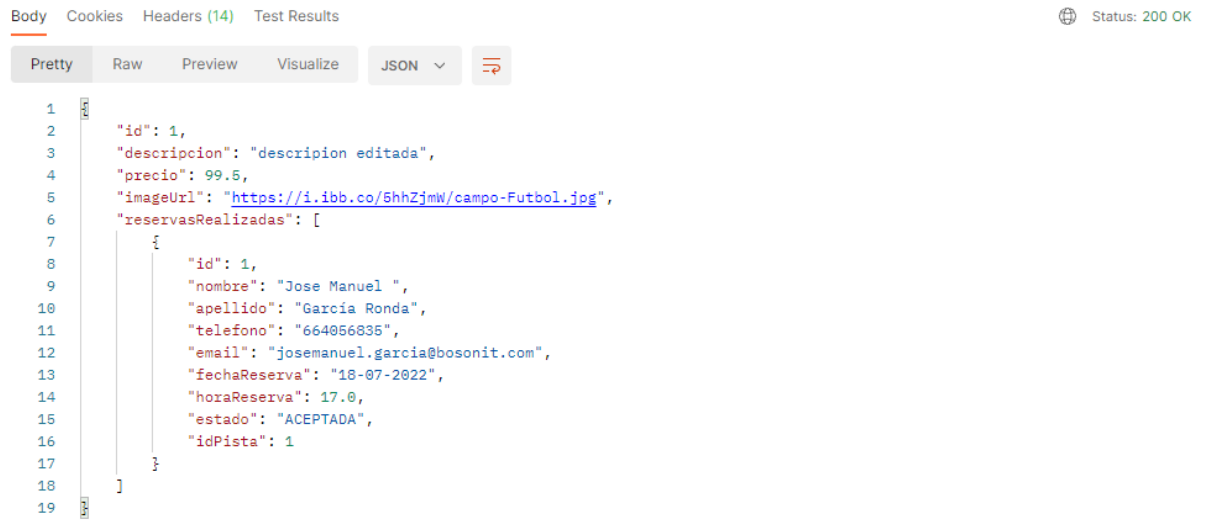



*Ilustración 129: Respuesta con código 200 de endpoint de actualizar una pista.*

En el caso de que no exista la pista a editar nos devolverá un mensaje de error con un código 404. junto a un mensaje de error. En este caso, probaremos a editar la pista 9, ya que no existe.

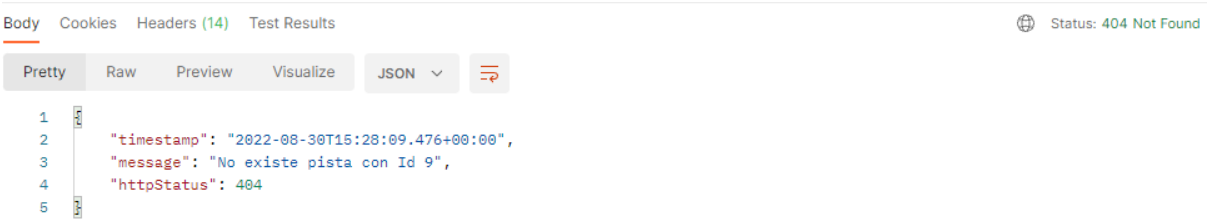


*Ilustración 130: Respuesta con código 404 de endpoint de actualizar una pista.*

### 8.3.4 Crear una pista

En este apartado vamos a comprobar el endpoint que hemos implementado para crear una pista.

El endpoint será de tipo POST y la ruta del mismo será www.localhost:8080/api/empresa/pista/ y requerirá pasarle por body un input con los datos de la pista siguiendo este formato:

```
{
   "descripcion":"prueba de descripcion",
   "tipo":"tipo",
   "precio":10.0
}
```

En caso de que la pista se cree correctamente nos devolverá un código 200 junto con un objeto con todos los datos de la pista. Dependiendo del tipo de pista que introduzcamos, pondrá una imagen u otra que se corresponderá al tipo. Los tipos disponibles son futbol, tenis, baloncesto y padel. Este dato vendrá desde el cliente pero existe una validación para este campo para prevenir errores. La pista que vamos a crear tiene los siguientes datos:

```
{
   "descripcion":"Pista de fútbol 7 de cesped artificial.",
   "tipo":"Futbol",
   "precio":10.0
}
```


*Ilustración 131: Respuesta con código 200 de endpoint de crear una pista.*

En caso de que el tipo introducido no sea válido, devolveremos un error 422 junto con el mensaje de tipo incorrecto.

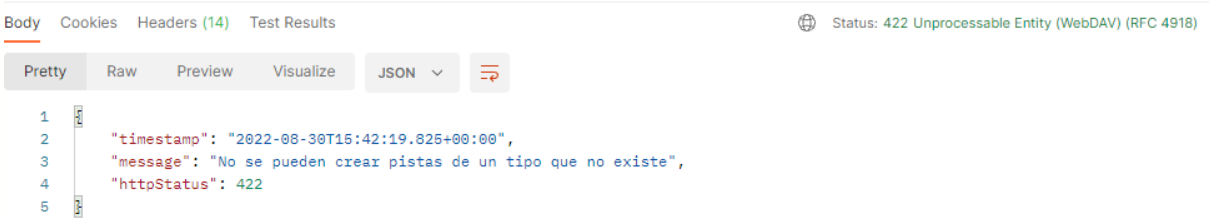



*Ilustración 132: Respuesta con código 422 de endpoint de crear una pista .*

### 8.3.5 Eliminar una pista

En este punto vamos a comprobar el endpoint para eliminar una pista. Este endpoint será de tipo DELETE y la ruta del mismo es www.localhost:8080/api/empresa/pista/id. En caso correcto devolverá un código 200. En este caso vamos a eliminar la pista 6 que hemos creado anteriormente.

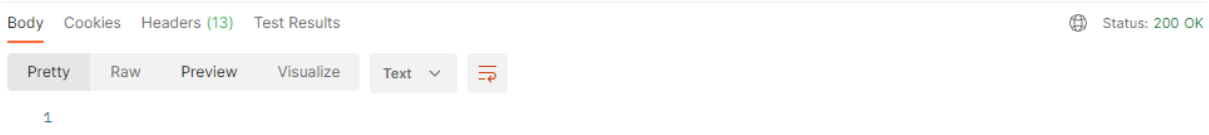



*Ilustración 133: Respuesta con código 200 de endpoint de eliminar una pista.*

En caso de intentar eliminar una pista que no existe devolverá un código 404 junto con un mensaje de error.

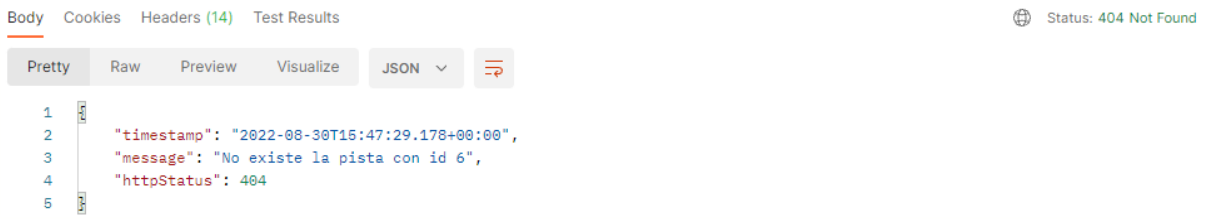



*Ilustración 105: Respuesta con código 404 de endpoint de eliminar una pista.*

### 8.3.6 Obtener reservas de una pista

En este punto vamos a comprobar el endpoint para obtener las reservas que tiene una pista. Este endpoint será de tipo GET y la ruta del mismo es www.localhost:8080/api/empresa/pista/id. En caso correcto devolverá un código 200. Si no tiene reservas devolverá una lista vacía y en caso contrario una lista con las reservas realizadas.En este caso vamos a obtener las reservas de la pista 1, en la que debe de aparecer una.

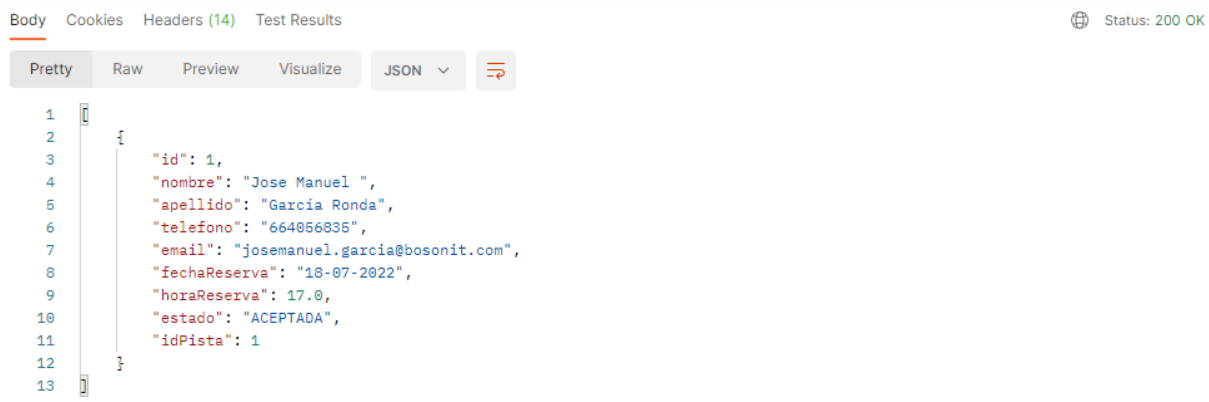



*Ilustración 135: Respuesta con código 404 de endpoint de mostrar datos de una pista.*

## 8.4 Prueba para gestión de eventos

### 8.4.1 Mostrar todas los eventos

En primer lugar vamos a realizar las pruebas relacionadas como obtener todas las pistas que existan.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/evento. En todos los casos devolverá un código 200 con todas los eventos existentes. Si ahora mostramos la lista de eventos nos devolverán con los eventos que haya creados.


*Ilustración 135: Respuesta con código 200 de endpoint de mostrar todos los eventos.*

### 8.4.2 Mostrar datos de un evento

En este punto se va a realizar la prueba para mostrar los datos de un evento.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/evento/id. Para comprobar las dos posibles respuestas de este endpoint buscaremos dos eventos, uno que exista y otra que no. En el caso de que no exista nos devolverá un mensaje de error con un código 404.

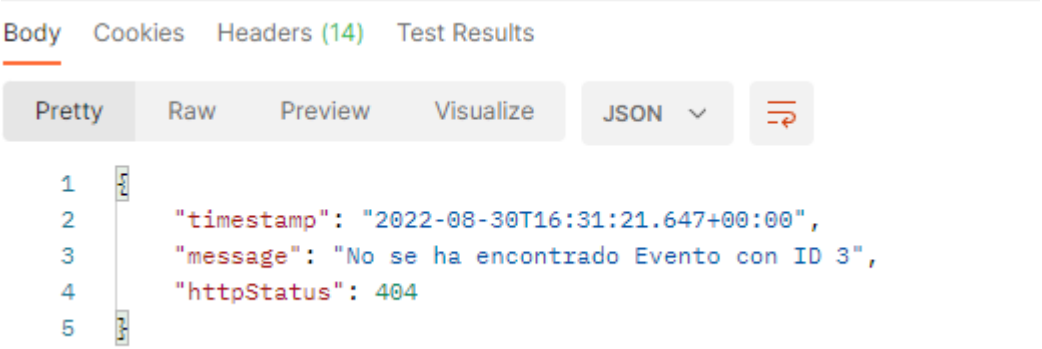



*Ilustración 136: Respuesta con código 404 de endpoint de mostrar datos de un evento.*

Si ahora queremos ver los datos de un evento que tenemos ya creado nos devolverá un DTO con los datos del mismo junto a un código 200.


*Ilustración 137: Respuesta con código 200de endpoint de mostrar datos de un evento.*

### 8.4.3 Actualizar un evento

En este punto vamos a comprobar el endpoint para actualizar los datos de un evento.

Este endpoint es de tipo PUT y la ruta del endpoint es www.localhost:8080/api/empresa/evento/id junto a un body en el que le pasaremos un inputDto con los datos de la pista que queramos modificar. Un ejemplo de input es el siguiente:


```
{
  "nombre":"nombre de prueba",
  "descripcion":"descripcion de prueba",
  "capacidad": 25,
  "fecha":"07-07-2022"
}
```


Sabiendo esto, en caso de que la actualización se haya realizado correctamente devolveremos un código 200 con los campos del evento actualizados.Para ello veremos el valor de la pista 1 antes de modificar sus atributos y que devuelve tras hacer la llamada al mismo.


*Ilustración 138: Respuesta con código 200 de endpoint de mostrar datos del evento 1.*

Una vez hemos visto el valor de los atributos del evento, vamos a actualizarle la capacidad a 30 personas y la fecha a día 12 de agosto.


```
{
    "id": 1,
    "nombre": "Zumbaton",
    "descripcion": "Mueve tus caderas con nosotros! Aprovecha la oportunidad y apúntate a Zumbaton.",
    "capacidad": 30,
    "fecha": "2022-08-10T00:00:00.000+00:00",
    "idUsuarios": []
}
```


*Ilustración 139: Respuesta con código 200 de endpoint de actualizar un evento.*

En el caso de que no exista el evento a editar nos devolverá un mensaje de error con un código 404. junto a un mensaje de error. En este caso probaremos a editar el evento 5, ya que no existe.

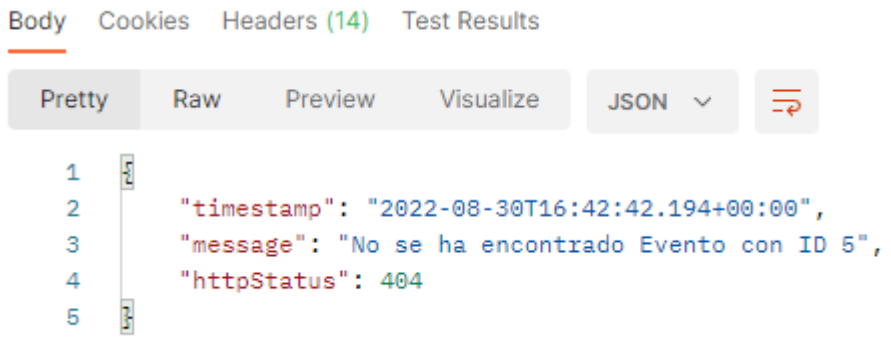



*Ilustración 140: Respuesta con código 404 de endpoint de mostrar actualizar un evento.*

### 8.4.4 Prueba para crear un evento

En este apartado vamos a comprobar el endpoint que hemos implementado para crear un evento.

El endpoint será de tipo POST y la ruta del mismo será www.localhost:8080/api/empresa/evento y requerirá pasarle por body un input con los datos del evento siguiendo este formato, el mismo que para actualizar.:

```
{
  "nombre":"nombre de prueba",
  "descripcion":"descripcion de prueba",
  "capacidad": 25,
  "fecha":   "fecha":"07-07-2022"
}
```

En caso de que la pista se cree correctamente nos devolverá un código 200 junto con un objeto con todos los datos del evento. El evento que vamos a crear tiene los siguientes datos:

```
{
  "nombre":"Evento de prueba",
  "descripcion":"Creamos este evento para comprobar que el endpoint funciona correctamente",
```

```
  "capacidad": 10,
  "fecha":"2022-08-09”
"
}
```

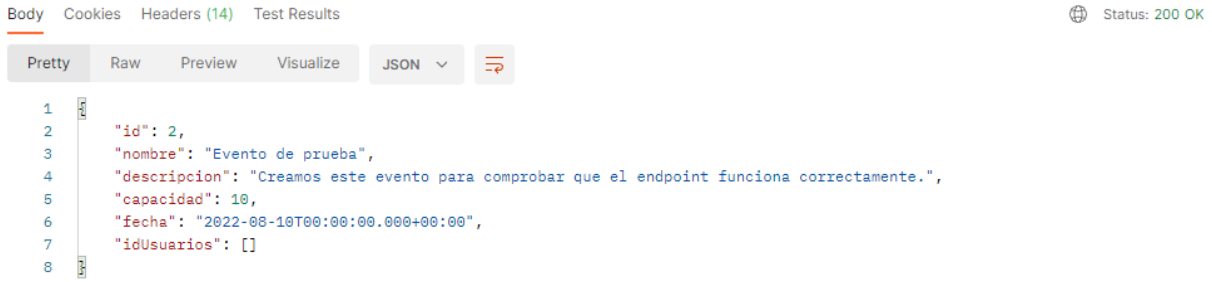




*Ilustración 141: Respuesta con código 200 de endpoint de crear un evento.*

### 8.4.5 Eliminar un evento.

En este punto vamos a comprobar el endpoint para eliminar un evento. Este endpoint será de tipo DELETE y la ruta del mismo es www.localhost:8080/api/empresa/evento/id. En caso correcto devolverá un código 200. En este caso vamos a eliminar el evento 2 que hemos creado anteriormente.

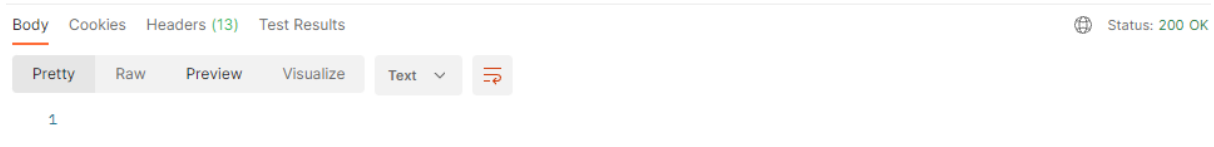



*Ilustración 142: Respuesta con código 200 de endpoint de eliminar un evento.*

En caso de intentar eliminar un evento que no existe devolverá un código 404 junto con un mensaje de error.


*Ilustración 143: Respuesta con código 404 de endpoint de eliminar un evento.*

Como podemos comprobar, si llamamos al endpoint de mostrar todos los eventos solo nos aparecerá el evento con id 1.


*Ilustración 144: Respuesta con código 200 de endpoint de mostrar todos los eventos.*

### 8.4.6 Apuntarse a un evento.

En este punto vamos a comprobar el funcionamiento del endpoint para que un usuario se apunte a un evento. Este endpoint será de tipo post y la ruta del mismo es www.localhost:8080/api/empresa/evento/idEvento/nombreUsuario. En caso de que el usuario se pueda apuntar al evento porque haya plazas disponibles se devolverá un código 200 junto con los datos del evento además de disminuir la capacidad en una unidad.

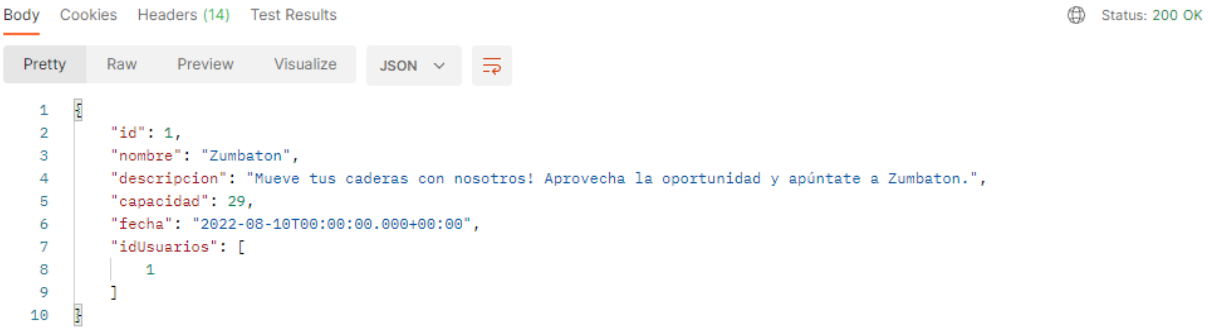



*Ilustración 145: Respuesta con código 200 de endpoint de apuntarse a evento.*

Si el evento ya no tiene plazas y un usuario desea apuntarse, se lanzará un código 422 con un mensaje de error indicando.

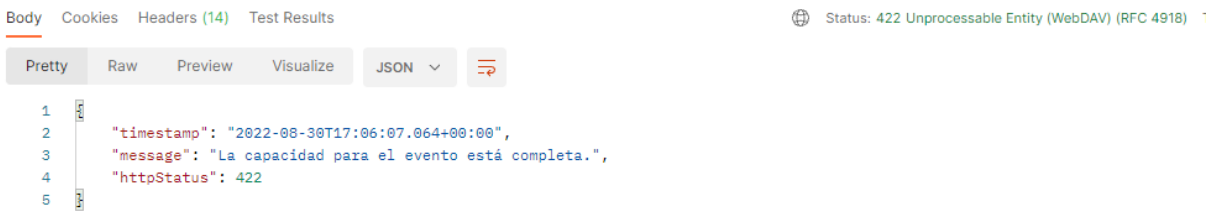



*Ilustración 146: Respuesta con código 422 de endpoint apuntarse a evento.*

## 8.5 Prueba para gestión de noticias

### 8.5.1 Mostrar todas las noticias

En primer lugar vamos a realizar las pruebas relacionadas como obtener todas las noticias que existan. Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/noticia. En todos los casos devolverá un código 200 con todas las noticias existentes. Si ahora mostramos la lista de noticias nos devolverán con las noticias que haya creadas.

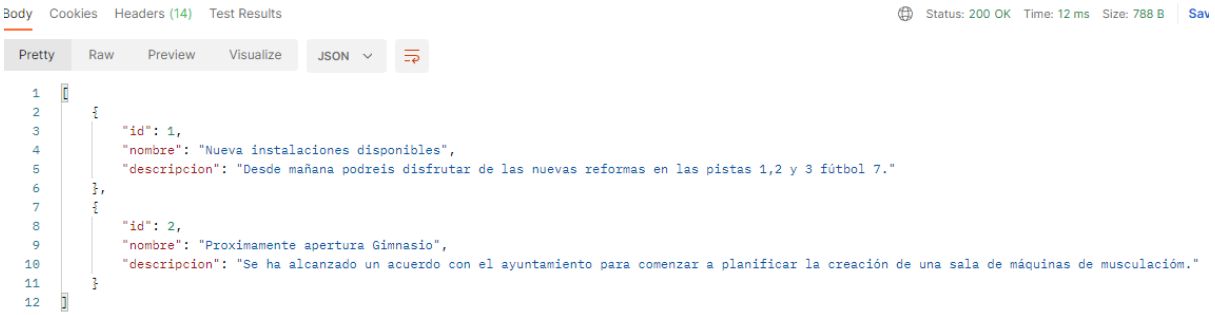



*Ilustración 147: Respuesta con código 200 de endpoint de mostrar todas las noticias.*

### 8.5.2 Mostrar datos de una noticia

En este punto se va a realizar la prueba para mostrar los datos de una noticia.

Este endpoint es de tipo GET y la ruta del endpoint es www.localhost:8080/api/empresa/noticia/id. Para comprobar las dos posibles respuestas de este endpoint procederemos igual que en los otros casos. Buscaremos dos noticias, una que exista y otra que no. En el caso de que no exista nos devolverá un mensaje de error con un código 404.

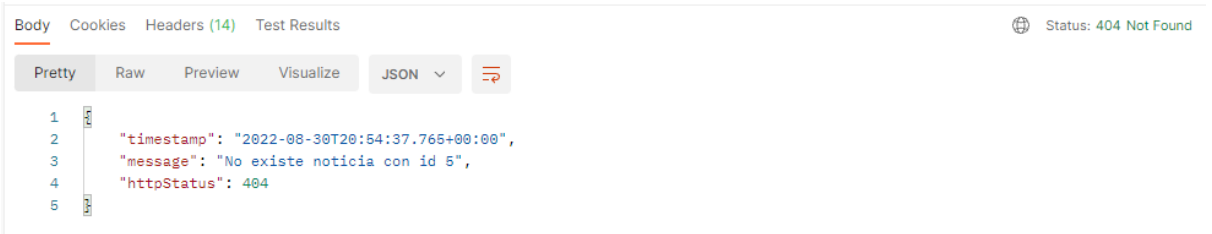



*Ilustración 148: Respuesta con código 404 de endpoint de mostrar datos de ua noticia.*

Si ahora queremos ver los datos de un evento que tenemos ya creado nos devolverá un DTO con los datos del mismo junto a un código 200.

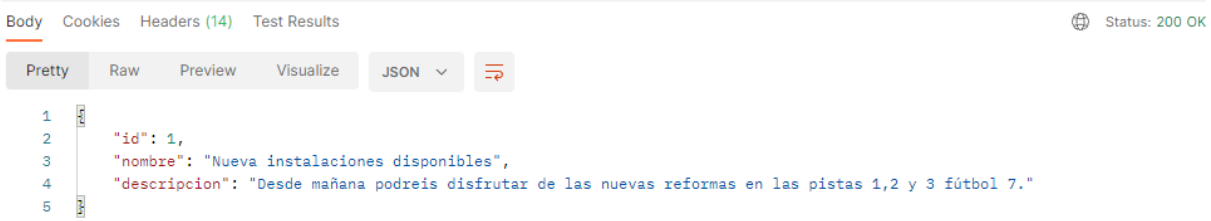



*Ilustración 149: Respuesta con código 200 de endpoint de mostrar datos de una noticia.*

### 8.5.3 Actualizar una noticia

En este punto vamos a comprobar el endpoint para actualizar los datos de un evento. Este endpoint es de tipo PUT y la ruta del endpoint es www.localhost:8080/api/empresa/noticia/id junto a un body en el que le pasaremos un inputDto con los datos de la pista que queramos modificar. Un ejemplo de input es el siguiente:

```
{
    "nombre":"Nombre de prueba",
    "descripcion":"Descripción de prueba"
}
```

Sabiendo esto, en caso de que la actualización se haya realizado correctamente devolveremos un código 200 con los campos de la noticia actualizada.Para ello veremos el valor de la noticia 1 antes de modificar sus atributos y que devuelve tras hacer la llamada al mismo.

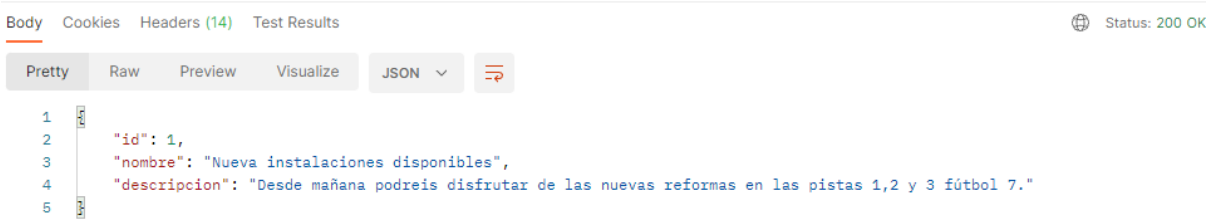



*Ilustración 150: Respuesta con código 200 de endpoint de mostrar datos de noticia con id 1.*

Una vez hemos visto el valor de los atributos del evento, vamos a actualizarle el nombre a “*Reforma de instalaciones de fútbol 7”.*

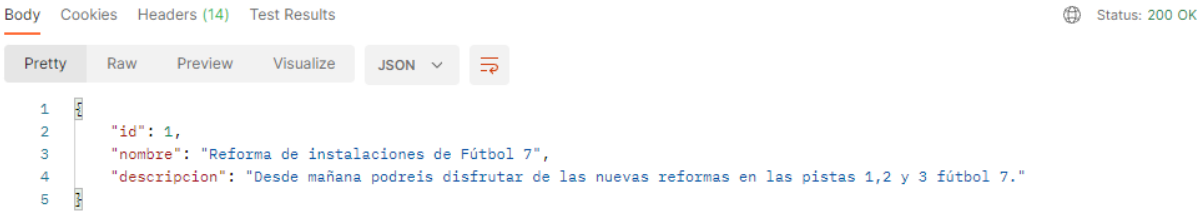



*Ilustración 123: Respuesta con código 200 de actualizar una pista.*

En el caso de que no exista el evento a editar nos devolverá un mensaje de error con un código 404. junto a un mensaje de error. En este caso, probaremos a editar la noticia 7, ya que no existe.

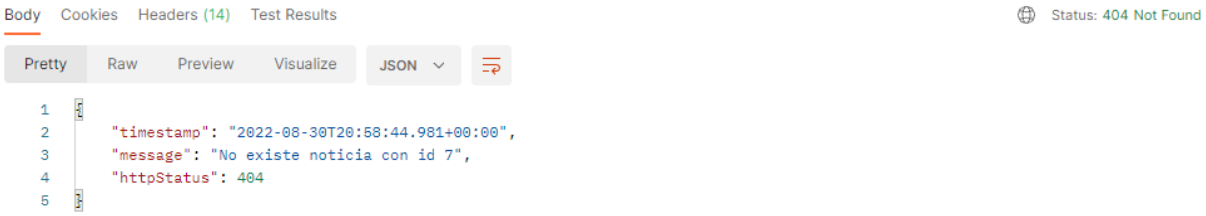



*Ilustración 124: Respuesta con código 404 de endpoint de actualizar una pista.*

## 8.5.4 Crear una noticia

En este apartado vamos a comprobar el endpoint que hemos implementado para crear una noticia. El endpoint será de tipo POST y la ruta del mismo será www.localhost:8080/api/empresa/noticia y requerirá pasarle por body un input con los datos de la noticia siguiendo este formato, el mismo que para actualizar.:

```
{
  "nombre":"nombre de prueba",
  "descripcion":"descripcion de prueba",

}
```

En caso de que la pista se cree correctamente nos devolverá un código 200 junto con un objeto con todos los datos de la noticia. El evento que vamos a crear tiene los siguientes datos:

```
{
  "nombre":"Noticia de prueba",
"descripcion":"Como podemos ver, se ha creado la noticia correctamente."
}
```

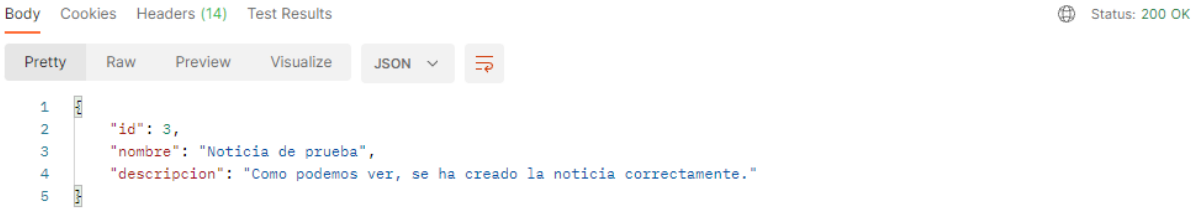




*Ilustración 125: Respuesta con código 200 de endpoint de crear una pista.*

### 8.5.5 Eliminar una noticia.

En este punto vamos a comprobar el endpoint para eliminar una noticia. Este endpoint será de tipo DELETE y la ruta del mismo es www.localhost:8080/api/empresa/noticia/id. En caso correcto devolverá un código 200. En este caso vamos a eliminar la noticia 3 que hemos creado anteriormente.

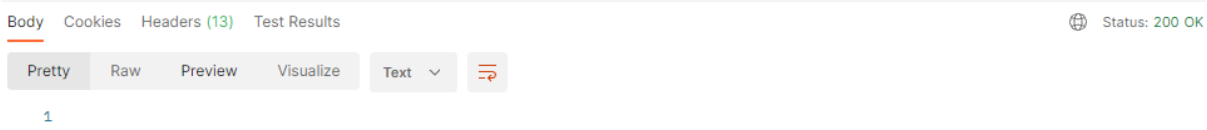



*Ilustración 126: Respuesta con código 200 de endpoint de eliminar una noticia.*

En caso de intentar eliminar un evento que no existe devolverá un código 404 junto con un mensaje de error.

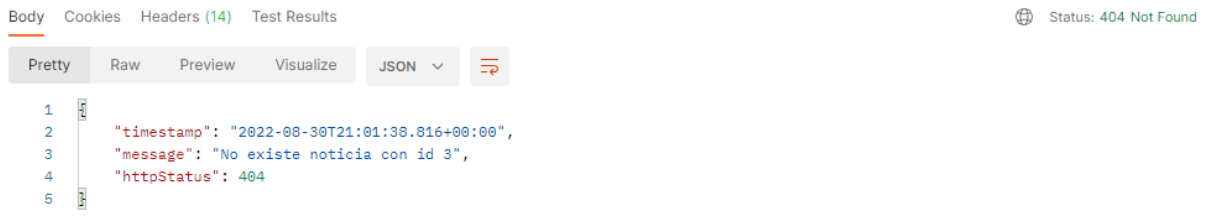



*Ilustración 127: Respuesta con código 404 de endpoint de eliminar una noticia.*

Como podemos comprobar, si llamamos al endpoint de mostrar todos las las noticias nos apareceran las noticias con id 1 y 2

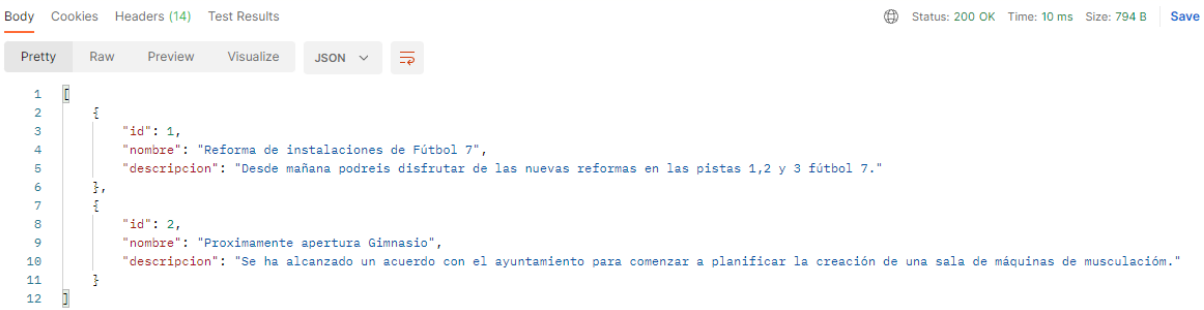



*Ilustración 128 : Respuesta con código 404 de endpoint de mostrar todas las noticias.*