Trabajo Fin de Máster
Máster en Ingeniería de Telecomunicación

Diseño, desarrollo y despliegue de plataforma web para gestión de actividades deportivas usando contenedores y computación en la nube

Autor: Francisco Ortiz Abril
Tutor: Isabel Román Martínez

Departamento de Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Máster
Máster en Ingeniería de Telecomunicación

Diseño, desarrollo y despliegue de plataforma web para gestión de actividades deportivas usando contenedores y computación en la nube

Autor:
Francisco Ortiz Abril

Tutor:
Isabel Román Martínez

Departamento de Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Trabajo Fin de Máster: Diseño, desarrollo y despliegue de plataforma web para gestión de actividades deportivas usando contenedores y computación en la nube.

Autor: Francisco Ortiz Abril

Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

Resumen

Este proyecto tiene como objetivo la implementación de una plataforma web para facilitar la organización de eventos deportivos, proporcionando una interfaz donde los usuarios pueden publicar actividades o unirse a las previamente disponibles.

Para el desarrollo de la plataforma se han construido varios servicios separados, usando contenedores y recursos de computación en la nube para su despliegue. Gracias a esta arquitectura y tecnologías se han podido cumplir algunos objetivos que habrían sido imposibles en un despliegue tradicional.

Índice

Resumen I

1 Introduccion 1

1.1 Motivación y objetivo 1

1.2 Estructura del documento 1

2 Estado del arte 3

2.1 Contenedores 3

2.1.1 Docker 4

2.1.2 Orquestación de contenedores 6

2.2 Aplicaciones web 8

2.3 Computación en la nube 9

3 Tecnologías utilizadas 11

3.1 En el desarrollo de la aplicación 11

3.1.1 Node.js 11

3.1.2 React 13

3.1.3 MongoDB 14

3.2 En el despliegue de la aplicación 14

3.2.1 Docker 14

Imágenes de Docker 15

Dockerfiles 15

Arquitectura de Docker 17

Persistencia de datos en Docker 17

3.2.2 Travis CI 18

3.2.3 Google Cloud Platform 19

4 Desarrollo del Proyecto 21

4.1 Desarrollo de la aplicación 21

4.1.1 Descripción general del sistema 21

Requisitos del sistema 21

Requisitos funcionales de la API 22

Requisitos funcionales de la interfaz 22

Arquitectura de la plataforma 23

4.1.2 Desarrollo de la API 24
Introducción 24
Definición de la API 24
Puntos de acceso 26
4.1.3 Desarrollo de la parte frontal 35
4.1.4 Despliegue de la aplicación 38
Despliegue de la base de datos 39
Despliegue de la API de SportHub 41
Despliegue de la parte frontal de SportHub 44
4.2 Testeo de la aplicación 44
5 Conclusiones 47

Índice de Figuras 49
Índice de Tablas 51
Índice de Códigos 53

1 Introduccion

1.1 Motivación y objetivo

La motivación de este proyecto surge a partir de dos factores: la escasez de plataformas donde consultar actividades deportivas dentro de una zona y poder unirte a ellas, así como del interés por investigar y aprovechar las ventajas que proporciona el despliegue de aplicaciones usando contenedores y recursos de computación en la nube. Entre estas ventajas se encuentran: elasticidad, menor inversión inicial o mantenimiento de la infraestructura a cargo del proveedor.

El objetivo de este proyecto es la creación de una plataforma web donde los usuarios podrán publicar diferentes actividades deportivas o unirse a las previamente disponibles, haciendo más fácil la organización de eventos deportivos que requieren un número mínimo de personas.

La plataforma constará de dos servicios separados, cada uno de ellos desplegado en un contenedor diferente:

- Servicio que aloja la parte frontal de la aplicación web.
- Servicio que proporciona una API de la que consumirá la parte frontal de la aplicación web. Este servicio se comunicará a su vez con una base de datos para abordar la persistencia.

Además, el proyecto incluye la creación de una infraestructura que permita la automatización del testeo y despliegue de dicha aplicación usando contenedores y máquinas virtuales en la nube. Para ello, se creará un proceso que permita el despliegue de cada servicio incluido en la plataforma ejecutando un solo comando. Los servicios se desplegarán en la infraestructura en la nube que proporciona Google a través del servicio **Google Cloud Platform**.

1.2 Estructura del documento

Después de esta introducción, en la siguiente sección se hará un recorrido por las diferentes tecnologías que se usarán en este proyecto, situándolas en el contexto temporal y explicando el motivo de su aparición. Además, se citarán varias implementaciones de dichas tecnologías que han ido surgiendo a lo largo del tiempo.

Una vez vistas las diferentes tecnologías a alto nivel, vendrá una sección en la que se explican con más profundidad las soluciones escogidas para el desarrollo de este proyecto (lenguajes de programación, aplicación para gestión de contenedores, etc.)

A continuación vendrá la sección en la que se describe el desarrollo de la plataforma, la cual será el núcleo de este proyecto. Esta sección detalla todos los pasos que se han ido siguiendo en el desarrollo de la plataforma.

Por último, el proyecto finaliza con unas conclusiones y una exposición de algunas líneas de continuación.

2 Estado del arte

En este capítulo se da a conocer el estado actual de los paradigmas y tecnologías relacionados con este proyecto. En la primera sección se presenta una introducción al concepto de contenedores. En la sección siguiente se menciona la evolución que están siguiendo las aplicaciones web en los últimos años, y por último, se hace una introducción a diferentes alternativas que hay actualmente para la utilización de infraestructura en la nube.

2.1 Contenedores

Hace unos diez años, la empresa VMware inventó una tecnología que permitía a un equipo anfitrión (*host*) con un sistema operativo Linux, ejecutar uno o más sistemas operativos clientes (como Windows). Esta tecnología consistía en un programa que creaba un entorno virtual (*máquina virtual*) donde se sintetizaba un entorno de computación real (NIC virtual, BIOS, tarjeta de sonido, vídeo, etc.). Ésto era posible gracias a una capa de software que gestionaba estos recursos reales en el equipo anfitrión y los repartía dinámicamente entre las distintas máquinas virtuales hospedadas en dicho equipo. A este software intermedio es a lo que se llamó *hypervisor* [18]. A partir de ese momento, cuando a menudo se habla de virtualización, automáticamente se piensa en productos como VMware, es decir, productos que nos permiten crear diferentes máquinas virtuales aisladas entre sí.

Ese concepto de virtualización está cambiando recientemente con el uso de los contenedores. Los contenedores son también una tecnología de virtualización, pero no necesitan hypervisor ni máquinas virtuales para ser ejecutados. En su lugar, se ejecutan directamente sobre el kernel del sistema operativo anfitrión.

Los contenedores son parte de un tipo de virtualización llamada *OS-level virtualization*. En este caso, el kernel es el encargado de ejecutar los distintos contenedores y de aislarlos unos de otros, por lo que no hay un hypervisor ni un sistema operativo huésped. Para entender mejor la virtualización a nivel de sistema operativo, es útil mencionar el comando *chroot*.

Código 2.1 Comando chroot.

```
chroot [OPTION] NEWROOTDIRECTORY [COMMAND [ARG]...]
```

Con este comando se puede cambiar el árbol de directorios de un proceso, de forma que el directorio raíz (/) para ese proceso será el directorio que se especifique en la ejecución

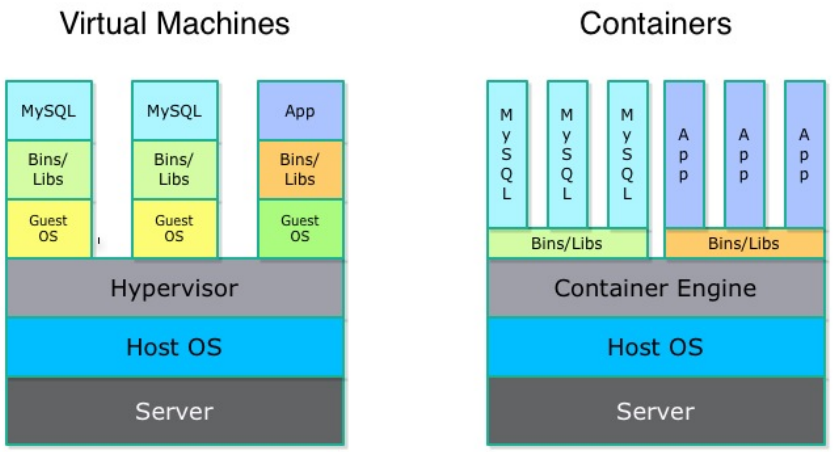

Figura 2.1 Contenedores vs Máquinas Virtuales.

del comando (en el ejemplo anterior, NEWROOTDIRECTORY). Este directorio debe pertenecer al árbol de directorios real. Una vez ejecutado el comando, los archivos/directorios que se encuentren fuera del directorio especificado serán inaccesibles para el proceso. Por tanto, el comando chroot invoca un proceso y cambia para éste y sus hijos el directorio raíz, creándose un nuevo árbol de directorios virtual a partir de éste. Aunque se está virtualizando un nuevo árbol de directorios, hay que tener en cuenta que los archivos son los mismos. Cuando se modifica un archivo dentro del proceso levantado con chroot (en este árbol virtual) se está modificando una parte del árbol de directorios real del host.

Al igual que se virtualiza el árbol de directorios, a partir de la versión 2.6.24 del kernel de Linux, es posible virtualizar el resto de recursos del host: red, procesos, usuarios, etc. Esto es posible gracias a la caracterísa del kernel llamada *namespaces*, de la que se hablará a continuación.

Un contenedor es simplemente un proceso que utiliza estas característas que se han mencionado en el kernel para tener una red, unos usuarios, un sistema de archivos, etc. virtuales que están enlazados con los recursos reales del host donde está hospedado.

El uso de contenedores nos proporciona una serie de ventajas y desventajas con respecto al uso de máquinas virtuales. Por un lado, un contenedor comparte sistema operativo con el host nativo y no tiene que recrear un sistema operativo completo. Gracias a esto el tamaño de los contenedores es mucho menor, haciéndolos eficientes, más fáciles de migrar, iniciar y recuperar. Por otro lado, alguna de las ventajas de las máquinas virtuales es la seguridad que proporciona la abstracción a nivel de hardware, ya que un ataque puede llegar unicamente a afectar la máquina virtual comprometida, aislando las demás que viven en el mismo hardware.

2.1.1 Docker

Como se puede ver, el concepto de contenedor utiliza funcionalidades del kernel de Linux que eran conocidas hace años. De hecho, empresas como Google utilizaban estas funcionalidades del kernel en proyectos internos desde el año 2004, y hay algunas otras soluciones además de Docker para interactuar con contenedores. Sin embargo, el concepto de contenedor ha adquirido muchísima más importancia a raíz del surgimiento de Docker, hasta tal punto que mucha gente habla simplemente de Docker cuando quiere hablar de contenedores. Docker es un proyecto de código abierto que ofrece una API de alto

nivel para proporcionar contenedores ligeros que ejecutan procesos de forma aislada. Para crear esa API, Docker utiliza funcionalidades del kernel como las que se han mencionado anteriormente, en especial:

- **CGroups**: limita, contabiliza y aisla el uso de recursos (CPU, memoria, disco, etc.) para un conjunto de procesos.

- **Namespaces**: utilizando namespaces, los cambios en un recurso solamente son visibles para los procesos dentro del mismo espacio de nombres, mientras que son invisibles para los demás. Existen 6 tipos básicos de namespaces relacionados con distintos aspectos del sistema:

  - **NETWORK namespace**: Aislamiento de red. Así, cada namespace de red tendrá sus propias interfaces, direcciones y puertos de red, tablas de enrutamiento, etc.

  - **PID namespace**: Aísla el espacio de identificadores de proceso. Un contenedor tendrá su propia jerarquía de procesos y su proceso padre o init (PID 1).

  - **UTS namespace**: Aísla el dominio y hostname, permitiendo a un contenedor poseer su propio dominio de nombres.

  - **MOUNT namespace**: Aísla los puntos de montaje de los sistemas de ficheros que puede ver un grupo de procesos. Este namespace fue el punto de partida, con chroot.

  - **USER namespace**: Aísla identificadores de usuarios y grupos. Así dentro un contenedor es posible tener un usuario con ID 0 (root) que se corresponda con un ID de usuario cualquiera en el host.

  - **IPC namespace**: Aísla la intercomunicación entre procesos dentro del espacio.

En el siguiente ejemplo se puede verificar cómo los procesos dentro de un contenedor tienen namespaces distintos, lo que significa que se mueven en espacios independientes y aislados entre sí. En el escenario mostrado en la imagen siguiente, se puede ver la ejecución de una *shell* dentro de un contenedor utilizando Docker. El PID del contenedor es 11344 desde el punto de vista del sistema anfitrión que, corresponde con el proceso 1 dentro del contenedor y que identifica al proceso *sh*, el primer comando ejecutado en el contenedor.

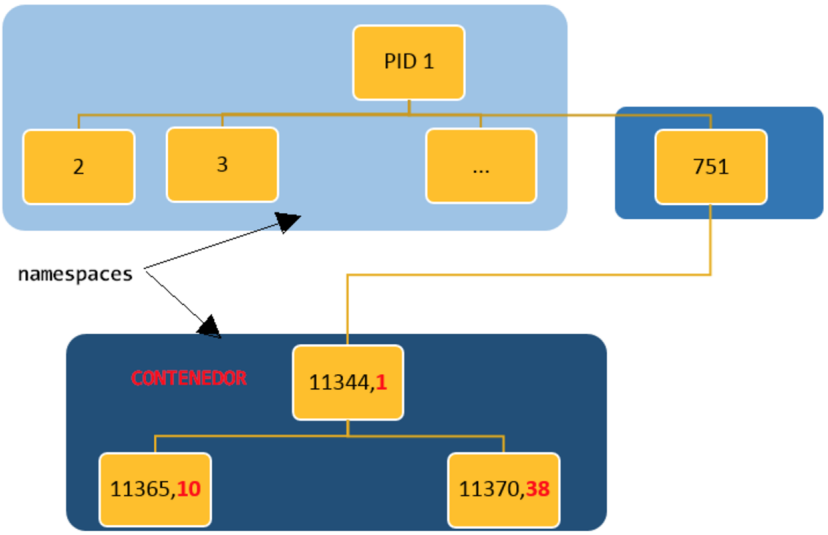

Figura 2.2 Namespace de proceso.

Además de proporcionar la API para comunicarnos con las funcionalidades del kernel, la gran virtud de Docker es la facilidad con la que los usuarios pueden construir y compartir sus propias aportaciones, en forma de imágenes de Docker. Una imagen puede ser definida como un archivo comprimido (*tarball*), que contiene todos los ficheros necesarios para la aplicación que se desea lanzar en el contenedor. Cada imagen está compuesta por una serie de capas donde se almacenan ficheros. Para crear una nueva imagen se puede coger una imagen base, hacer las modificaciones que se estimen necesarias para el propósito de lo que se esté haciendo y guardar esas modificaciones como una nueva imagen. Para almacenar todas estas imágenes Docker construyó DockerHub, un repositorio donde los usuarios pueden guardar y compartir sus imágenes Docker.

Además de Docker, otras empresas han lanzado algunos productos para actuar como motor de ejecución de contenedores, aunque con menos éxito. Entre estas soluciones cabe destacar *Rocket*, una solución de la empresa *CoreOS*.

2.1.2 Orquestación de contenedores

Trabajar con contenedores individuales montados en un único nodo proporciona ventajas como las que se han mencionado anteriormente. Sin embargo, el verdadero potencial de los contenedores es el poder escalarlos y ejecutarlos utilizando múltiples *hosts* al mismo tiempo. La orquestación de contenedores es un término que abarca todas las tareas de programación, administración y mantenimiento de manera que podamos elegir en qué hosts desplegar uno o varios contenedores para que estos trabajen de forma paralela.

Hay diferentes herramientas para la orquestación de contenedores:

- **Docker**: antes de la versión 1.12 de Docker, para utilizar el propio Docker como servicio de orquestación había que recurrir a la plataforma Docker Swarm, con la que se podía crear un conjunto de máquinas que actuaban como hosts de Docker y que el usuario podía administrar. A partir de esta versión [41], Docker incluye sus propias funciones de orquestación en entornos de producción. Esta nueva característica se conoce como “modo enjambre“. Este modo opcional permite que todos los contenedores y nodos formen parte de una red descentralizada donde cada uno de

ellos trabaja de forma independiente, pero interviene en la administración del sistema general y comparte el almacenamiento de recursos. Todos los nodos se comunican a través del protocolo GRPC [39], un sistema RPC (Remote Procedure Call) [30] de código abierto creado por Google utilizando redes HTTP/2.

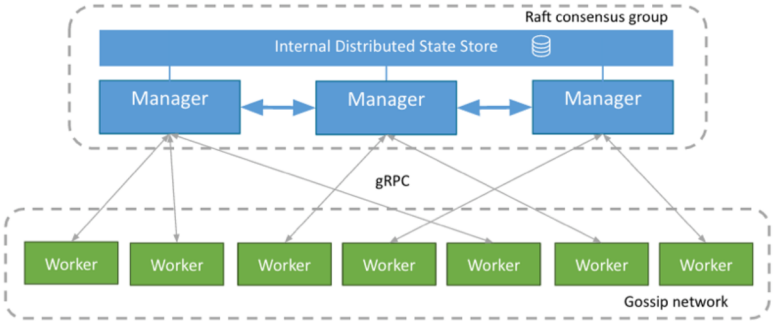

Figura 2.3 Orquestación en Docker.

- **Kubernetes**: Kubernetes es una plataforma creada para el despliegue automático, escalado y operación de aplicaciones en contenedores a través de un conjunto de nodos (*cluster*). Es un proyecto de código abierto escrito en Go y fue iniciado por Google en 2014. Algunas de sus característas más importantes son:
  - Despliegue automático: despliegue de aplicaciones de forma bastante rápida sin necesidad de excesivos pasos manuales.
  - Elasticidad: permite escalar las aplicaciones en vuelo. Esto quiere decir que es posible cambiar la capacidad y los recursos asignados a las aplicaciones cuando ya se están ejecutando. Por ejemplo, añadiendo o eliminando recursos en función del tráfico recibido.
  - Cubre las necesidades de producción: Kubernetes nació para solucionar algunos de los problemas iniciales de los contenedores, que provocaban que algunos usuarios fueran reacios a usar esta tecnología en entornos de producción. Algunos de los problemas más importantes eran la gestión de la escalabilidad y la seguridad. Kubernetes soluciona estos problemas añadiendo funcionalidades como replicación de contenedores, adición de balanceadores de carga o monitorización de salud de contenedores.

  Es posible utilizar Kubernetes en diferentes entornos:
  - Cluster local usando Docker o vagrant [40].
  - Cluster en Google Cloud Engine [38].
  - Solución personalizada: utilizar Kubernetes con otros proveedores de cloud y otros sistemas operativos.
- **Amazon ECS** [2]: servicio de administración de contenedores que proporciona Amazon AWS [35]. Proporciona una API compatible con Docker que permite ejecutar aplicaciones distribuidas en un clúster administrado de instancias de Amazon EC2 [1].

2.2 Aplicaciones web

La web está cambiando a un paso bastante rápido durante los últimos años. Cada vez existe una mayor expectativa por parte de los usuarios de que las aplicaciones web se mimeticen con las aplicaciones tradicionales de escritorio, ofreciendo su inmediatez, velocidad y fluidez en el uso. Conseguir estas capacidades con una aplicación web tradicional, con gran parte del procesamiento hecho en el servidor, es muy complicado. Cada recarga de página atestigua la diferencia que existe entre ambas soluciones.

Para tratar de solucionarlo surgió el concepto RIA (*Rich Internet Application*) [29], basado en tecnologías como Flash, AIR o Silverlight, pero el hecho de necesitar plugins y no funcionar en móviles fue determinante para su fracaso.

Por todo ello, la tendencia actual es llevar cada vez más características al navegador, al lado del cliente. En lugar de depender tanto del servidor o de plugins nos basamos en HTML5, CSS3 y JavaScript para crear las aplicaciones, y el servidor se limita a enviar y recibir los datos de manera rápida y eficiente. Así nacen las denominadas **Single Page Application o SPAs**.

Una SPA es un tipo de aplicación web donde todas las pantallas se muestran en la misma página, sin recargar el navegador. En terminos técnicos, una SPA es una aplicación donde existe un único punto de entrada, por ejemplo un archivo *index.html*. En estas aplicaciones no hay ningún otro archivo HTML al que se pueda acceder de manera separada y que nos muestre contenido o parte de la aplicación, toda la acción se produce dentro del mismo.

Aunque las SPA solamente tengan una página, lo que sí tienen son varias vistas, entendiendo por vista cada pantalla en una aplicación de escritorio. Por tanto, en la misma página se irán presentando distintas vistas produciendo el mismo efecto que una aplicación tradicional con distintas páginas. Este funcionamiento tiene un efecto principal: las pantallas se cargan más rápido al no haber recarga del navegador en cada cambio de página. De este modo se elimina uno de los principales cuellos de botella de las aplicaciones web con respecto a las aplicaciones de escritorio, el problema de la latencia. En el siguiente gráfico se puede ver la diferencia entre una SPA y una aplicación web tradicional a la hora de pedir un recurso en la web.

Como se puede ver en la figura, cuando el usuario da clic en un botón o enlace se dispara una acción que hace una petición asíncrona al servidor, cuya respuesta es devuelta en un tipo de formato JSON o XML. Este tipo de formato puede ser interpretado por el navegador sin necesidad de recargar todo el contenido.

Para conseguir esta interactividad en el lado del cliente, es necesario bastante código JavaScript. A medida que se va escribiendo más código es necesario tener dicho código más limpio y organizado. Para resolver este problema, hay varios *frameworks* de JavaScript que han ganado bastante popularidad a la hora de crear SPAs. Entre ellos están:

- **AngularJS** [5]: proyecto de código abierto mantenido por Google. AngularJS tiene plantillas que se basan en la tecnología bidireccional UI llamada *data binding*. Esta tecnología es una forma automática de actualizar la vista cuando el modelo cambia, así como lo recíproco. El proceso inicial de compilación crea un HTML, el cual el navegador interpreta y muestra. A partir de ahí, el paso se va repitiendo para las vistas subsecuentes.
- **EmberJS** [10]: al igual que Angular, se basa en la arquitectura Modelo-Vista-Controlador. Permite a los desarrolladores crear SPAs escalables a través de creación

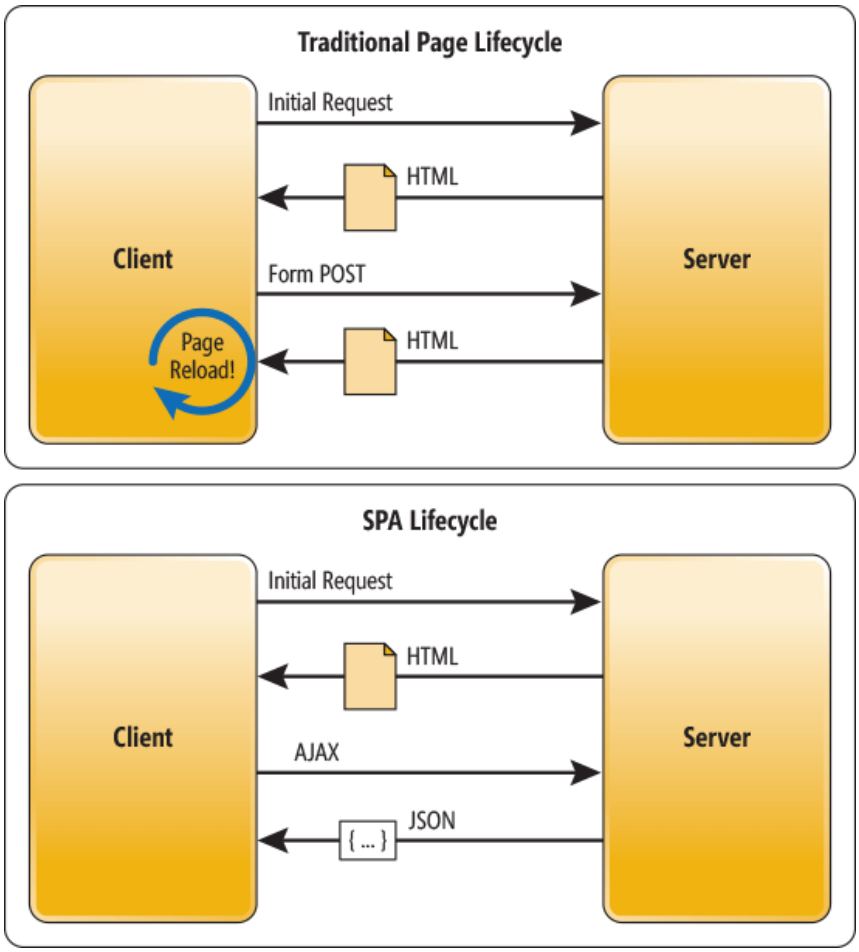

Figura 2.4 SPA vs Aplicación web tradicional.

de componentes con código JavaScript y plantillas de actualización automática impulsadas por Handlebars.js [43].

- **ReactJS** [28]: librería de código abierto creada e impulsada por Facebook. Ofrece grandes beneficios en rendimiento, modularidad y promueve un flujo muy claro de datos y eventos, facilitando la planificación y el desarrollo de las aplicaciones web.

2.3 Computación en la nube

La computación en la nube ha supuesto una revolución respecto al modelo tradicional de provisión de servicios TIC. Este modelo traslada la tradicional infraestructura de máquinas propias a un modelo donde un proveedor externo se encargará del mantenimiento de los recursos tales como servidores, almacenamiento, red, etc.. Dependiendo de su despliegue, hay distintos modelos de computación en la nube.

- Computación en la nube privada: una nube privada es aquella en la que solamente una organización tiene acceso a los recursos que se utilizan para implementar la nube. Es decir, una empresa dispone de un entorno en la nube en exclusiva, siendo esta empresa que ofrece los servicios finales la propietaria y gestora de dicha infraestructura cloud.

- Computación en la nube pública: un despliegue de *cloud* público se caracteriza por ofrecer recursos TIC sobre infraestructuras compartidas entre múltiples clientes. A estos recursos el cliente accede a través de internet o mediante conexiones VPN. Es decir, la empresa que ofrece los servicios finales no posee la infraestructura cloud, sino que está utilizando un proveedor.
- Computación en la nube híbrida: un despliegue de *cloud* híbrido es aquel que combina recursos de la computación en la nube privada con la pública. Este modelo surge a partir de la necesidad de clientes que aunque cuentan con infraestructura propia buscan aprovechar las ventajas de los servicios de un proveedor externo.

En los últimos años el modelo de computación en la nube que está descatando sobre los demás es el modelo de *cloud* público, ya que hay grandes empresas que están proporcionando recursos para usar este tipo de computación. Algunos de los productos más destacados son:

- Amazon Web Services (AWS) [35]: plataforma donde se recogen los servicios ofrecidos por Amazon. Ofrece una gran cantidad de servicios donde se pueden destacar: Amazon EC2 (capacidad de computación en la nube), Amazon S3 [4] (capacidad de almacenamiento) y Amazon RDS [3] (servicio de base de datos).
- Google Cloud Platform (GCP) [38]: plataforma de Google. Aunque llegó un poco más tarde que Amazon, poco a poco ha ido consiguiendo gran protagonismo en este campo. Algunos de los productos más destacados son *Google Compute Engine* [15] (servicio de computación), *Google Cloud Storage* [16] (servicio de almacenamiento) o *Google Container Engine* [17] (servicio de gestión y orquestación de contenedores).
- Microsoft Azure [25]: plataforma de Microsoft. Al igual que en Amazon Web Services y Google Cloud Platform, sus servicios más populares son los de computación y almacenamiento en la nube.

3 Tecnologías utilizadas

3.1 En el desarrollo de la aplicación

Como se ha mencionado en la sección 1.1, uno de los objetivos de este proyecto es la creación de una plataforma web donde los usuarios puedan crear, ver y unirse a diferentes actividades deportivas dentro de su zona geográfica. En esta sección se detallan las tecnologías utilizadas para el desarrollo de dicha plataforma.

3.1.1 Node.js

Node.js ha sido utilizado para desarrollar el servicio que proporciona la API que consumirá la parte frontal de la aplicación.

Node.js es un entorno de ejecución de código JavaScript del lado del servidor. Está basado en el motor v8 de JavaScript de Google [42], diseñado para correr en un navegador y ejecutar el código JavaScript de forma extremadamente rápida.

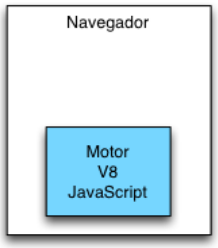

Figura 3.1 Motor v8 en el navegador.

La tecnología que está detrás de Node.js, a través del desarrollo de un conjunto amplio de librerías, permite ejecutar este motor en el lado del servidor, abriendo un nuevo abanico de posibilidades en cuanto al mundo de desarrollo se refiere.

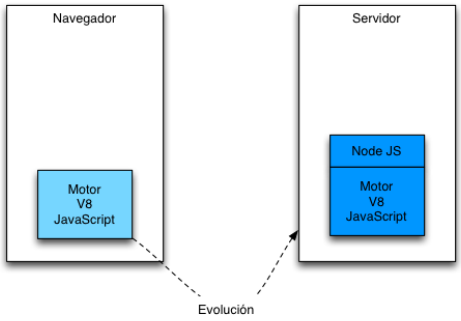

Figura 3.2 Motor v8 en el servidor.

Node.js trabaja con un único hilo de ejecución que es el encargado de organizar todo el flujo de trabajo que se debe realizar.

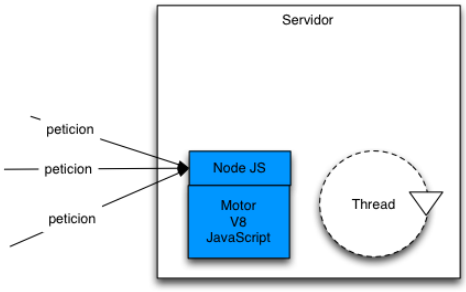

Figura 3.3 Hilo de ejecución de Node.js.

¿Qué sucedería si ese hilo se pone a realizar una tarea que requiere X tiempo y queda bloqueado? Para trabajar de una forma óptima Node.js delega gran parte del trabajo a un conjunto de hilos (*threads*). Este conjunto de hilos está construido con la libreria *libuv* [23], que dispone de su propio entorno multi-hilo asíncrono. Cuando a Node.js le llega una petición, le envía el trabajo que hay que realizar al conjunto de hilos.

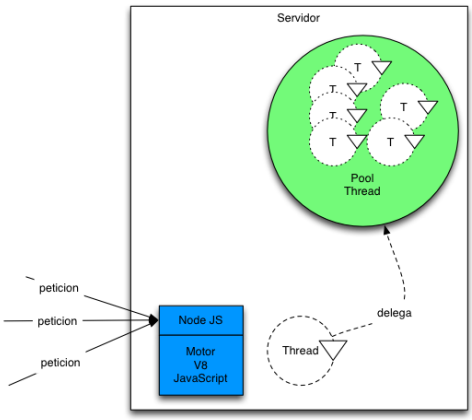

Figura 3.4 Delegación del trabajo al conjunto de hilos.

Cuando Libuv recibe el trabajo, alguno de sus hilos se encargará de abordarlo. Una vez el trabajo ha sido completado, libuv se encarga de emitir un evento que será recibido por el hilo principal de Node.js. Este hilo principal se encargará de terminar de procesarlo.

Como se puede ver, casi todas las operaciones en Node.js se realizan de forma asíncrona. Esto quiere decir que hay que manejar bien esa asincronía para crear un código que funcione de manera óptima.

La elección de Node.js como lenguaje para desarrollar el servicio de la plataforma web ha sido debida a diferentes factores:

- Rapidez de Node.js: el código JavaScript es ejecutado bastante rápido en el motor V8.
- Gran repositorio de paquetes: al ser un proyecto con código abierto, Node.js tiene un repositorio de paquetes compartido de un gran tamaño y calidad. El número de módulos en el *Node Package Manager (NPM)* ha incrementado de una forma considerable en poco tiempo.
- Conocimiento del lenguaje JavaScript en el momento del desarrollo del proyecto.

3.1.2 React

Como se ha expuesto en la sección 2.2, React es un framework creado por Facebook que facilita la creación de *Single Page Applications (SPAs)*. En este proyecto React es utilizado para la creación de la parte frontal de la plataforma web.

Antes de empezar a explicar React conviene explicar el concepto de diseño basado en componentes. Este paradigma consiste en dividir la interfaz web completa en pequeñas partes con un propósito específico y fácilmente actualizable/reusable. Cada pequeña parte de la interfaz es lo que se conoce como un componente, y cada componente tiene una serie de características:

- Tiene un estado y unas propiedades.
- Son independientes: es decir, cada pieza de la interfaz no debe depender de las demás.
- Son pequeños y específicos: es decir, cada componente tiene una función muy concreta que lo distingue del resto.

React es una librería de alto rendimiento para el desarrollo de interfaces de usuario. React mantiene un Virtual DOM [32] con la definición, estado y propiedades de cada componente. Este Virtual DOM está conectado con el DOM [9] real, de manera que cada elemento tiene una representación visual. A su vez, el Virtual DOM es una de las claves para el alto rendimiento de React. Este Virtual DOM es utilizado en el proceso que calcula los cambios de la interfaz web en cada interacción del usuario. Aparentemente, cada interacción del usuario afecta a toda la interfaz, pero no es así. Cuando se produce una interacción, React crea un Virtual DOM con los nuevos cambios y calcula las diferencias entre el Virtual DOM actual y el siguiente. De esta forma, solamente se modifican sobre el DOM real ciertos componentes, reutilizando todos lo máximo posible. Esta es la clave del alto rendimiento que proporciona React.

La aplicación de React en este proyecto tuvo al principio una desventaja, ya que el diseño de la aplicación en pequeños componentes y el aprendizaje de este tipo de diseños

requirió más tiempo. Sin embargo, se ha elegido esta tecnología con el propósito de ir añadiendo nuevas funcionalidades a la plataforma con un esfuerzo mucho más reducido.

3.1.3 MongoDB

MongoDB [26] es un sistema de base de datos NoSQL [27] multiplataforma de licencia libre. MongoDB está orientado a documentos de esquema libre, lo que implica que cada registro puede tener un esquema de datos distinto.

En MongoDB cada registro o conjunto de datos se denomina documento, y estos documentos pueden ser agrupados en colecciones (equiparables con las tablas de las bases de datos relacionales pero sin estar sometidas a un esquema fijo).

La siguiente figura representa la jerarquía principal de MongoDB.

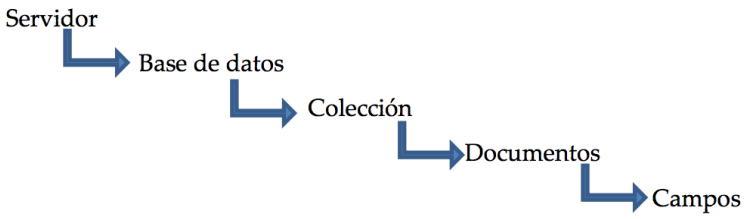

Figura 3.5 Arquitectura MongoDB.

La diferencia con las bases de datos relacionales es que MongoDB no utiliza tablas, filas ni columnas, sino que utiliza documentos con distintas estructuras. Un conjunto de Campos formarán un documento, que en caso de asociarse con otros formará una Colección. Las bases de datos están formadas por Colecciones, y a su vez, cada servidor puede tener tantas bases de datos como el equipo donde se aloja permita.

Para el intercambio y almacenamiento de documentos en MongoDB se utiliza el formato BSON (Binary JavaScript Object Notation). Se trata de una representación binaria de estructuras de datos y mapas, diseñada para ser más ligera y eficiente que JSON (JavaScript Object Notation).

MongoDB cuenta con una serie de herramientas que permiten trabajar con la base de datos:

- **Mongod**: servidor de bases de datos de MongoDB.
- **Mongo**: cliente para la interacción con la base de datos MongoDB.
- **Mongofiles**: herramienta para trabajar con ficheros directamente sobre la base de datos MongoDB.

3.2 En el despliegue de la aplicación

3.2.1 Docker

Aunque en la 2.1 se han nombrado algunas alternativas en el uso de contendores, como Rocket, la tecnología elegida para desarrollar el proyecto ha sido Docker. Como se ha

expuesto en dicha sección, Docker proporciona un API bastante potente para la interacción con los contenedores, y tiene detrás una extensa comunidad debido a su creciente uso.

En las siguientes secciones se hace una introducción a los conceptos más importantes en Docker.

Imágenes de Docker

Las imágenes son la base de los contenedores. Como se menciona en la sección 2.1.1, una imagen puede ser definida como un fichero comprimido (*tarball*) que contiene todos los ficheros necesarios para la aplicación que se desea lanzar en el contenedor. Cada imagen está compuesta por una serie de capas donde se almacenan los ficheros. Esta división en capas incrementa la reusabilidad, disminuye el uso de disco y acelera el proceso de construcción de la imagen. Esto es debido a que cuando se ejecuta el comando de construcción de una imagen con Docker (usando el comando *docker build*), éste tiene en memoria caché las diferentes capas, por lo que solamente tendrá que construir de nuevo las capas que se hayan modificado, en lugar de construir la imagen completa.

Una vez se construye una imagen, se puede ejecutar dicho contenedor usando el comando básico de Docker, *docker run*. Cuando se lanza un contenedor lo que se hace es lanzar esa imagen creada y se añade una nueva capa (con permiso de escritura) a la misma: la capa de contenedor (*container layer*). Todos los cambios que se realizan al contenedor cuando está corriendo (escribir nuevos ficheros, modificar archivos existentes, etc.) son añadidos a esta última capa. Además de los comandos de construcción y lanzamiento de un contenedor, hay otros comandos para interaccionar con los contenedores [6].

Un ejemplo de lanzamiento de un contenedor sería utilizando el siguiente comando:

Código 3.1 Comando docker run.

```
docker run hello-world
```

El comando anterior ejecuta un contenedor a partir de la imagen llamada *hello-world*. Es posible crear nuevas imágenes o utilizar una ya construida. Las imágenes ya construidas se pueden encontrar en diferentes repositorios, públicos o privados. La propia plataforma de Docker tiene un registro de imágenes, DockerHub [8]. En este registro se encuentran imágenes subidas tanto por usuarios de Docker como otras subidas por grandes compañias que desean proporcionar un uso libre de esas imágenes.

Dockerfiles

Como se ha mencionado anteriormente, una imagen de Docker está formada por todos los ficheros que necesita la aplicación que se va a lanzar en el contenedor. Para construir esta imagen, Docker hace una división en capas, donde cada una de estas capas consiste en una serie de instrucciones. Esas instrucciones pueden ser, por ejemplo:

- Lanzar un comando
- Añadir un nuevo directorio
- Crear una variable de entorno

Estas instrucciones realizadas son guardados en un fichero llamado *Dockerfile*. En la figura 3.6 se puede ver un ejemplo básico de *Dockerfile* donde, a una imagen base de

Ubuntu 14.04, se le añade el servidor web Apache, se crean variables de entorno necesarias y se abre el puerto 80 del contenedor. Cada una de estas instrucciones es a lo que llamamos capa.

```
# A basic apache server
FROM ubuntu:14.04

MAINTAINER Chuck Norris: 0.1

RUN apt-get update
RUN apt-get install -y apache2
RUN apt-get clean

ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80

CMD ["/usr/sbin/apache2", "-D",
"FOREGROUND"]
```

Figura 3.6 Dockerfile.

A partir del Dockerfile anterior, es posible construir una imagen con el comando *docker build*.

Código 3.2 Comando construcción contenedor.

```
docker build -t ubuntu/apache-server /path/to/directory/
   with/Dockerfile
```

El parámetro *-t* se usa para indicar el nombre que tendrá la imagen, y a continuación se indica la ruta hasta el directorio que contiene el Dockerfile. A continuación, es posible ver las imágenes creadas usando el comando *docker images*. En la figura 3.7 se puede ver un ejemplo.

```
→ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED             SIZE
ubuntu/apache-server    latest              b7d489408534        8 minutes ago       225 MB
```

Figura 3.7 Docker Images.

Una vez construida la imagen, el siguiente paso es iniciar el contenedor. Cuando se ejecute el comando *docker run* se iniciará el contenedor ejecutando el comando de inicialización indicado en el Dockerfile (*/usr/sbin/apache2 -D FOREGROUND*).

Tras este paso se tiene lanzado el contenedor y se tendría un servidor Apache corriendo en el puerto 80 de dicho contenedor. Sin embargo, si se prueba a acceder al puerto 80 desde el equipo local, este puerto no estará abierto por este servidor. Esto es debido a que hay que hacer un mapeo entre el puerto 80 del contenedor y el puerto del equipo local en el que se quiere que esté disponible Apache. Esto es posible mediante el parámetro *-p* de *docker run*, cuyo formato es *-p puerto-host:puerto-contenedor*. Por ejemplo:

Código 3.3 Comando construcción contenedor.

```
docker run -p 8080:80 -d ubuntu/apache-server
```

Arquitectura de Docker

El objetivo de esta sección es explicar la arquitectura del sistema explicado en las secciones anteriores. Como se ha presentado previamente, Docker proporciona una API de alto nivel con la que se pueden hacer operaciones con contenedores. Para esto se utiliza una arquitectura cliente-servidor donde:

- **Cliente**: es un programa en línea de comandos (binario de Docker). Se encarga de hablar con el servidor de Docker a través de la API.
- **Servidor**: proceso corriendo como demonio. Recibe y procesa las peticiones de la API. Se encarga de ejecutar los contenedores y guardar las imágenes.

El esquema es el mostrado en la figura 3.8:

- A través de la línea de comandos se hacen las operaciones deseadas con los contenedores (pull, run, etc.).
- Estas operaciones son recibidas y procesadas por el servidor (Docker Daemon), que está corriendo en el host.
- Este servidor se comunica con Docker Registry, donde están situadas todas las imágenes de Docker.

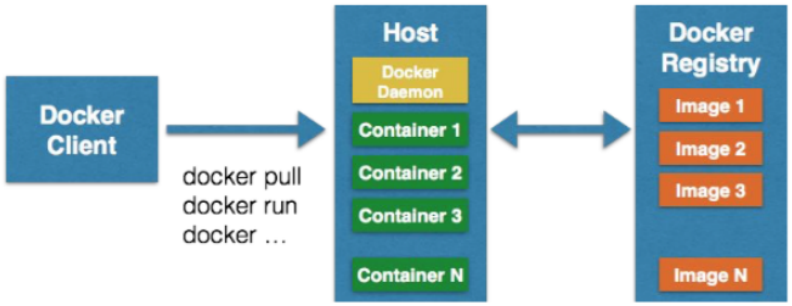

Figura 3.8 Estructura de Docker.

Persistencia de datos en Docker

A primera vista, Docker nos sirve para crear contenedores que se encargan de mantener vivo un servicio mientras se encuentren en ejecución, pero en cuanto éstos son destruidos, todo el contenido es eliminado.

Este comportamiento es ideal para crear contenedores temporales usados en entornos de pruebas o de desarrollo que no necesiten persistir más allá de la duración de las pruebas unitarias requeridas, pero se vuelve un impedimento en entornos de producción, donde los datos son parte muy importante y a veces la parte vital del programa.

Para poder mantener los datos a través de sesiones e incluso poder usarlos en diferentes contenedores existen dos opciones:

- Usar directorios reales en el sistema de archivos dentro del contenedor.
- Usar la API de volúmenes de Docker [36].

En cuanto a la primera opción, es posible usar un directorio real en el sistema de archivos dentro de uno o más contenedores de Docker. Para eso, se puede ejecutar el comando *docker run* con los siguientes parámetros:

Código 3.4 Comando construcción contenedor.

```
docker run -v /ruta/del/directorio/local:/punto/de/
   montaje imagen
```

La opción *-v* indica el montaje de un volumen dentro del contenedor que se crea, y tiene el formato de [ruta local]:[ruta del contenedor]. Ambas rutas deben ser absolutas y existir previamente a la iniciación del contenedor. Si la ruta no existe en la imagen es recomendado crearla desde el Dockerfile para poder montar el directorio externo. La primera ventaja que proporciona esta técnica es que se pueden usar proyectos y programas ya existentes de manera transparente, permitiendo desplegar en Docker aplicaciones creadas sin tener en cuenta Docker. En cuanto a desventajas, la administración de los permisos en los directorios se vuelve manual y propensa a errores.

La otra opción, como se ha mencionado, es utilizar la API de volúmenes de Docker. Esta opción tiene la ventaja de que Docker se encarga de crear y administrar los volúmenes de manera que no puedan ser eliminados de manera accidental en el momento de eliminar los contenedores o imágenes. Una de las desventajas de utilizar la API es que los directorios son complicados de administrar fuera de Docker, por lo que es recomendable usarlos únicamente dentro de los contenedores y no tratar de modificar el contenido externamente.

3.2.2 Travis CI

Antes de explicar Travis CI merece la pena hacer una pequeña introducción a la integración y entrega continua.

El proceso de integración continua es una práctica de desarrollo de software propuesta por Martin Fowler [24] que consiste en integrar constante y automáticamente el código de un proyecto con el objetivo de detectar fallos lo antes posible.

En su forma más básica la integración continua involucra solo una herramienta que debe monitorizar el repositorio del código fuente para detectar cambios o en su defecto ejecutarse bajo cierta programación establecida. Cuando se ejecuta esta herramienta, se descarga la última versión del código, lo compila y ejecuta un conjunto de pruebas. En caso de error la herramienta debe notificar a los desarrolladores, mientras que si todo sale bien nos permite ir más allá pudiendo hacer tareas como la automatización del proceso de despliegue.

La entrega continua es una práctica de ingeniería de software mediante la cual, se crean, prueban y preparan automáticamente los cambios en el código y se entregan para la fase de producción. Amplía la integración continua al implementar todos los cambios en el código en un entorno de pruebas y/o de producción después de la fase de creación. Cuando la entrega continua se implementa de manera adecuada, los desarrolladores dispondrán siempre de un artefacto que se ha sometido a un proceso de pruebas estandarizado y listo para su despliegue.

Travis CI es un servicio distribuido de integración y generación continua. Permite conectar con un repositorio de GitHub [37] y probar el código desarrollado una vez publicado en el servidor.

Travis CI dispone de distintos *runtimes* con distintas configuraciones de librerías, lenguajes de programación y bases de datos, de forma que es posible probar un proyecto sobre distintas arquitecturas sin tener que tener todas ellas montadas. Estas configuraciones son detalladas en un fichero llamado *.travis.yml* en formato YAML [33] que irá colocado en el directorio raíz de nuestro proyecto.

3.2.3 Google Cloud Platform

Como se ha expuesto en la sección 2.3, *Google Cloud Platform* [38] es la solución ofrecida por Google para proporcionar servicios de computación en la nube. Dentro de esta plataforma, en este proyecto se hace uso de *Google Container Engine (GKE)*, que se puede describir como la plataforma sobre Google Cloud Platform para manejar de forma distribuida (en la nube) contenedores de Docker. *Google Container Engine* [17] está basado en Kubernetes, que como se ha visto en la sección 2.1.2 es un sistema open source publicado por Google para la gestión de contenedores.

Google Container Engine está compuesto por un conjunto de instancias de Google Compute Engine en las que se ejecuta Kubernetes para gestionar los contenedores de Docker. Además, a través de una API corriendo en la propia máquina es posible hacer operaciones como reescalado de nuestro cluster de contenedores.

4 Desarrollo del Proyecto

4.1 Desarrollo de la aplicación

En esta sección se presenta el desarrollo de la aplicación, a la que se ha llamado *SportHub*. En primer lugar se presenta una descripción general de la plataforma, con algunos requisitos que ésta debe cumplir. A continuación se detalla el desarrollo de la parte que actúa como *backend* de la aplicación, la cual consta de un API desarrollado en Node.js. En la tercera sección se presenta la parte frontal, desarrollada usando la librería de JavaScript React. Por último, se detallan los pasos necesarios para desplegar las distintas partes de la plataforma en Google Cloud Platform.

4.1.1 Descripción general del sistema

Requisitos del sistema

- **Infraestructura necesaria**

  Toda la infraestructura utilizada en la plataforma debe estar en la nube, evitando tener que gestionar y mantener máquinas físicas.

- **Requisitos de disponibilidad**

  La plataforma debe tener alta disponibilidad y ser elástica. En un principio no se prevee un gran volumen de tráfico, pero se debe diseñar una infraestructura donde pueda crecer el número de peticiones y no suponga ningún problema de rendimiento.

- **Testeo de la plataforma**

  Cada vez que un desarrollador hace un cambio en el código de cualquier servicio de la plataforma, se debe lanzar un proceso que se encargue de comprobar que el funcionamiento es el esperado.

- **Despliegues de la plataforma**

  El despliegue de las diferentes partes de la plataforma debe ser totalmente automático. Cada vez que un desarrollador sube un cambio en el código, se debe crear un automáticamente un artefacto (en este caso, una imagen de Docker) que contenga el servicio que se quiera desplegar. Este proceso de despliegue debe ser un simple comando, de forma que cualquier persona que desarrolle en el código pueda ser capaz de subir un cambio a producción.

Requisitos funcionales de la API

- **Versionado de la API**

  Es necesario un control de versiones para que los clientes que se integren con la API no pierdan soporte ante cambios en los diferentes puntos de acceso. Cuando aparezca una nueva versión de la API, será necesario dar soporte a la versión anterior durante un período de 6 meses. El patrón de versionado de la API será de libre elección.

- **Inserción de usuario**

  Un usuario debe poder ser insertado haciendo una petición que incluya la información de: Nombre, Email y Contraseña con la que iniciará sesión en la plataforma.

- **Autenticación**

  Un usuario debe poder autenticarse utilizando el email y la contraseña con los que se registra. En el paso de autenticación de un usuario, la API debe devolver un token con el que el usuario podrá acceder al resto de rutas autenticadas.

- **Autenticación de rutas**

  Todos los puntos de acceso de la API deben requerir autenticación, excepto los puntos de acceso de registro de usuario e inicio de sesión. Esta autenticación debe llevarse a cabo mediante el token que se entrega al usuario cuando inicia sesión.

- **Creación de deportes**

  Se debe ofrecer un punto de acceso para la inserción de deportes. Cuando se inserta un deporte se debe especificar el nombre y el número mínimo de jugadores que acepta.

- **Creación de actividades**

  Se debe ofrecer soporte a creación de actividades en un punto de acceso de la API. Cuando una actividad sea creada, ésta debe ser visible para el resto de usuarios de la plataforma. Los diferentes usuarios podrán unirse a las actividades disponibles hasta que se alcance el número mínimo de jugadores asociados al deporte correspondiente.

- **Gestión de actividades**

  El usuario que crea la actividad debe tener permiso para poder eliminarla. Los demás usuarios solamente tendrán acceso a verla y a unirse a ella.

- **Respuesta de los puntos de acceso**

  Los diferentes puntos de acceso deben incluir el código HTTP correspondiente en la respuesta, en función de si la petición ha sido procesada con éxito o hay algún error.

Requisitos funcionales de la interfaz

- **Creación de usuarios en la plataforma**

  Un usuario que quiera registrarse en la plataforma debe tener la capacidad de registrarse a través de un formulario donde se le requieran los siguientes datos: Nombre, Email y Contraseña con la que iniciará sesión en la plataforma.

- **Inicio de sesión en la plataforma**

  Un usuario que quiera iniciar sesión en la plataforma debe poder hacerlo insertando el Email y la Contraseña con los que se registró en el sistema.

- **Entrada a la plataforma**

  Cuando un usuario inicia sesión en la plataforma, debe encontrar una pantalla donde pueda elegir entre ver las actividades disponibles en la plataforma o crear nueva. Ambas elecciones deben estar accesibles al usuario a través de un único click.

- **Vista de las diferentes actividades**

  Será necesario mostrar las diferentes actividades junto con la información asociada a ellas: nombre, deporte lugar y fecha. En dicha página, el usuario propietario de la actividad tendrá soporte para poder eliminarla, mientras que el resto de usuarios podrán verla, unirse, o borrarse de ella.

- **Creación de una actividad**

  Si el usuario elige el paso de crear una actividad cuando está dentro de la plataforma, se le debe mostrar un formulario donde poder insertar los datos de la misma: nombre, deporte, lugar, día y hora. Cuando un usuario crea una actividad a través de la plataforma, automáticamente será unido a la misma.

Arquitectura de la plataforma

En este apartado se presenta una arquitectura del sistema en forma de diagrama de bloques.

- **Parte frontal**: esta parte será un contenedor con la imagen de Docker que se muestra en la figura: *sporthub-frontend*. Como se ha mencionado, la tecnología utilizada en esta parte ha sido React.

- **API**: contenedor con la imagen de Docker *sporthub-backend*. La tecnología utilizada ha sido Node.js

- **Base de datos**: instancia de MongoDB.

En siguientes apartados se presentará la descripción y despliegue de cada una de estas partes, pero la comunicación entre ellas se puede ver en la siguiente figura:

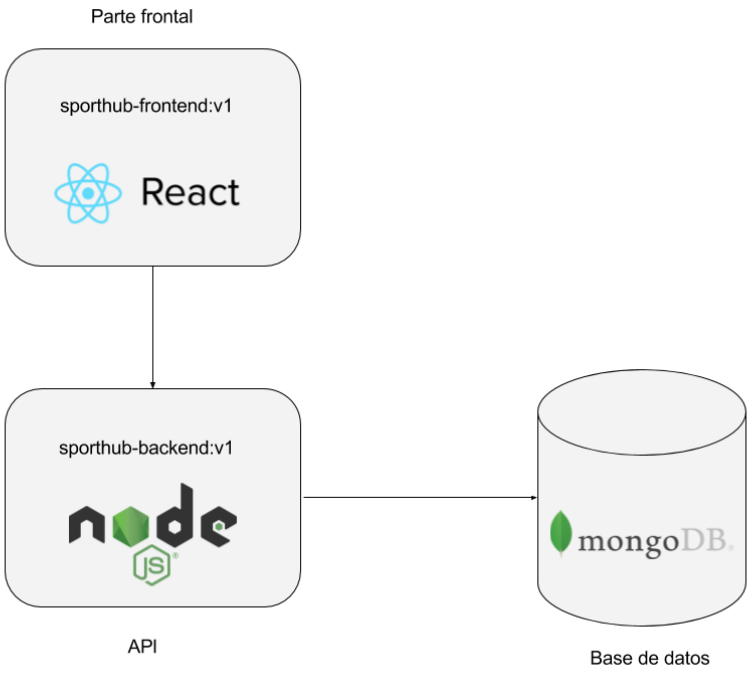

Figura 4.1 Arquitectura de SportHub.

4.1.2 Desarrollo de la API

Introducción

El propósito de esta sección es documentar la API de *SportHub*, que en un principio será consumido por la parte frontal de la aplicación pero que podrá ser usado por algún otro servicio externo, ya que es totalmente independiente de dicha parte frontal.

SportHub es una aplicación en la que diferentes personas podrán compartir actividades deportivas con el propósito de que otras personas se unan a ellas, haciendo más fácil la organización de eventos deportivos que necesitan un número mínimo de personas. A partir de ese objetivo, esta API debe proporcionar una interfaz con la que hacer esas operaciones de forma sencilla.

Definición de la API

En este apartado se presentan algunas definiciones comunes usadas en la API.

- **Versionado de la API**: una de las cosas que hay que tener en cuenta a la hora de desarrollar un API es que éste, al igual que otro tipo de aplicaciones web, va a cambiar. Si la API es pública o es consumida por una gran cantidad de clientes, es casi seguro que muchos de estos clientes queden obsoletos o no sigan al día las actualizaciones que el equipo que desarrolla la API va haciendo en la misma. La solución a este problema es dar compatibilidad a versiones anteriores mediante un versionado. A la hora de versionar un API, hay diferentes enfoques:

  – **Versionado por URL**:

    https://api.dominio.com/v1/recurso/

Esto sería un ejemplo de URL en la que se indica la versión de la API. Como ventaja de este tipo de enfoque se encuentran: la sencillez con la que se ve la versión de la API o la facilidad a la hora de desarrollar, ya que no requiere un desarrollo en el punto de acceso para distinguir lo que devuelve dependiendo de la versión. Como ejemplo de APIs usando este sistema se encuentren la API de Youtube [34] o de Uber [44].

– **Versionado en la cabecera de las peticiones**:

  https://api.dominio.com/recurso/

  "API version": 1

  En este caso, la versión de la API se añade como cabecera en la petición. Los partidarios de este uso se apoyan en que es el caso más académico cuando estás desarrollando una API REST, ya que con cabecera por URL se viola uno de los principios: *un recurso debe estar identificado por un y solo un punto de acceso*.

– **Versionado por parámetro en la URL**

  https://api.dominio.com/recurso/?v=1

  Otro tipo de versionado común. El gran defecto de esta opción es que se mezcla la versión de la API con los demás parámetros que soporte el mismo, como opciones de filtrado o paginación.

En este proyecto, dada la sencillez y que no se espera un gran número de clientes consumiendo la API, se ha elegido la opción de indicar la versión de la API en la URL.

- **Patrones comunes**: la API sigue una serie de patrones comunes:
  – Uso de códigos HTTP estándar:

Tabla 4.1 Códigos HTTP.

200 ok	201 created
202 accepted	203 not authorized
204 no content	205 reset content
206 partial content	
300 multiple choice	301 moved permanently
302 found	303 see other
304 not modified	307 temporary redirect
400 bad request	401 unauthorized
402 payment required	403 forbidden
404 not found	405 method not allowed
410 gone	411 length required
412 preconditions failed	413 request entity too large
414 requested URI too long	415 unsupported media
416 bad request range	417 expectation failed
500 server error	501 not implemented
502 bad gateway	503 service unavailable
504 gateway timeout	505 bad HTTP version

– En caso de error, la API devuelve el código HTTP correspondiente (por ejemplo, 400) y un mensaje en formato JSON con información adicional sobre el error. Un ejemplo de mensaje de error es el siguiente:

Código 4.1 Mensaje de error API.

```
{
    "success": "false",
    "message": "name and numPlayers fields are mandatory"
}
```

– La autenticación se hace mediante JSON Web Tokens (JWTs) [19]. Una vez el usuario se autentica contra el servidor, recibirá un token con el que podrá hacer las operaciones que desee. Cuando el usuario cierra la sesión, el token es borrado y ya no volverá a ser válido. Este token se pasa por cabecera en las peticiones, y no se podrá acceder a ninguna ruta que requiera autenticación sin proveerlo. Las rutas que requieren autenticación son todas las rutas de la API excepto las de registro de usuario e inicio de sesión.

Un ejemplo válido sería el siguiente:

Código 4.2 Petición Autenticada API.

```
curl
  -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIs
9.eyJzdWIiOiI1OGIxY2FlZjU1YTVmMzNiMWFkYTY4MD" \
  -H "Content-Type: 'application/json'"
https://api.sporthub.com/api/v1/activities
```

Con el comando anterior se recuperarían las actividades disponibles. En el apartado siguiente se explicará con más detalle la función de cada punto de acceso.

Puntos de acceso

- **Recurso usuarios**
  – Obtener todos los usuarios del sistema

Tabla 4.2 Obtener todos los usuarios del sistema.

URL	https://sporhub-api.com/api/v1/users
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta de un objeto JSON conteniendo el nombre y el email de cada uno de los usuarios del sistema. Este punto de acceso estará

disponible solamente para un usuario administrador, cuya creación es una de las líneas de continuación de este proyecto.

Por ejemplo:

Código 4.3 Obtener usuarios.

```
{
    "testUser": "user@example.com",
    "user2": "user@mail.com"
}
```

– Obtener todos los emails registrados

Tabla 4.3 Obtener todos los emails registrados.

URL	https://sporhub-api.com/api/v1/users/emails
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta de un array conteniendo una entrada por cada email registrado en el sistema. Este punto de acceso estará disponible solamente para un usuario administrador, cuya creación es una de las líneas de continuación de este proyecto.

Por ejemplo:

Código 4.4 Obtener emails.

```
[
    "user1@example.com",
    "user@mail.com"
]
```

– Obtener información de usuario a través de su email

Tabla 4.4 Obtener información de usuario.

URL	https://sporhub-api.com/api/v1/users/email/:email
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta de un objeto JSON que contiene la información de dicho usuario.

Por ejemplo:

Código 4.5 Obtener usuario.

```
{
    "_id":"590f62adfe81737asf85af7",
    "email":"user@mail.com",
    "name":"User Name",
    "__v":0
}
```

– Obtener usuario actual

Tabla 4.5 Obtener usuario actual.

URL	https://sporhub-api.com/api/v1/users/me
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta de un objeto JSON que contiene la información del usuario al que pertenece el token con el que se está haciendo la petición al API.

Por ejemplo:

Código 4.6 Obtener usuario.

```
{
    "name":"User Name",
    "email":"user@mail.com"
}
```

- **Recurso actividades**
  – Obtener todas las actividades registradas en el sistema

Tabla 4.6 Obtener todas las actividades.

URL	https://sporhub-api.com/api/v1/activities
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta por un array donde cada elemento es un objeto JSON que contiene la información de cada actividad.

Por ejemplo:

Código 4.7 Obtener actividades.

```
[
```

```
    {
        "name":"Tenis Madrid",
        "sport":"tennis",
        "numPlayers":2,
        "place":"Madrid",
        "date":"2017-05-17T22:00:00.000Z",
        "hour":"2017-05-07T18:24:47.000Z",
        "users":["UserName, UserName2"]
    },
    {
        "name":"Basket Madrid",
        "sport":"basketball",
        "numPlayers":10,
        "place":"Madrid",
        "date":"2017-05-15T22:00:00.000Z",
        "hour":"2017-05-07T19:20:08.000Z",
        "users":["User"]
    }
]
```

– Añadir nueva actividad

Tabla 4.7 Añadir actividad.

URL	https://sporhub-api.com/api/v1/activities
Método HTTP	POST
Códigos de respuesta	[401, 400, 201]
Formato de respuesta	Descrito posteriormente

Tabla 4.8 Parámetros de la petición.

Nombre del campo	Requerido	Tipo	Descripción
name	Si	String	Nombre de la actividad
sport	Si	String	Deporte que se practicará
place	Si	String	Lugar de la actividad
users	No	Array	Usuarios que formarán parte de la actividad
date	Si	Date	Fecha de la actividad
hour	Si	Date	Hora de la actividad

Ejemplo de peticiones y respuestas:

Petición y respuesta con carencia de campos

Código 4.8 Añadir actividad.

```
curl -X POST \
  https://api.sporthub.com/api/v1/activities \
```

```
  -H 'authorization: bearer eyJhbGciOiJIUzI1NiIsIn
  R5cCI6IkpXVCJ9 \
  -H 'content-type: application/json' \
  -d '{
  "name":"Futbol en Madrid",
  "sport":"soccer",
  "place": "Madrid",
}'

{
    "success":false,
    "message":"Name, sport, place, date and hour
    fields are mandatory"
}
```

Petición y respuesta correcta

Código 4.9 Añadir actividad.

```
curl -X POST \
  'https://api.sporthub.com/api/v1/activities' \
  -H 'authorization: bearer eyJhbGciOiJIUzI1NiIsInR5c
  CI6IkpXVCJ9' \
  -H 'content-type: application/json'
  -d '{
      "name": "Test",
      "sport": "soccer",
      "place": "Madrid",
      "users": ["user@mail.com"],
      "date": "Sun Jul 16 2017 00:00:00 GMT+0200 (CEST)",
      "hour": "Sat Jul 15 2017 19:30:52 GMT+0200 (CEST)"
      }'

{
    "success":true,
    "message":"You have successfully
    added a new activity"
}
```

– Unir usuario a actividad

Tabla 4.9 Unir usuario a actividad.

URL	https://sporhub-api.com/api/v1/activities/:name/join
Método HTTP	POST
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

Tabla 4.10 Parámetros de la petición.

Nombre del campo	Requerido	Tipo	Descripción
user	Si	String	Email del usuario que se unirá a la actividad

Ejemplo de petición y respuesta:

Código 4.10 Unirse a actividad.

```
curl -X POST \
  https://api.sporthub.com/api/v1/activities/Test/join \
  -H 'authorization: bearer eyJhbGciOiJIUzI1NiIsInR5cC.
  eyJzdWIiOiI1OGZiOKJgOZA1' \
  -H 'content-type: application/json' \
  -d '{
  "user": "user@mail.com"
}'

{
    "success":true,
    "message":"You have successfully join to Test activity"
}
```

– Borrar actividad

Tabla 4.11 Borrar actividad.

URL	https://sporhub-api.com/api/v1/activities/:name/join
Método HTTP	DELETE
Códigos de respuesta	[401, 400, 204]
Formato de respuesta	Descrito posteriormente

El usuario con permisos para borrar una actividad será el usuario que la ha creado.

Ejemplo de petición y respuesta:

Código 4.11 Borrar actividad.

```
curl -X DELETE \
  https://api.sporthub.com/api/v1/activities/Test \
  -H 'authorization: bearer eyJhbGciOiJIUzI1NiIsInR5cCI6
  eyJzdWIiOiI1OGZiOKJgOZA1' \

{
    "success":true,
    "message":"Activity Test successfully removed"
}
```

- **Autenticación**

  – Registro de usuario

Tabla 4.12 Registro de usuario.

URL	https://sporhub-api.com/api/v1/users
Método HTTP	POST
Códigos de respuesta	[409, 400, 201]
Formato de respuesta	Descrito posteriormente

Tabla 4.13 Parámetros de la petición.

Nombre del campo	Requerido	Tipo	Descripción
name	Si	String	Nombre del usuario
email	Si	String	Email del usuario
password	Si	String	Password del usuario

Ejemplo de petición y respuesta:

Código 4.12 Añadir usuario.

```
curl -X POST \
  https://api.sporthub.com/api/v1/users \
  -d '{
  "name":"User Name",
  "email":"user@mail.com",
  "password": "mypassword",
}'

{
    "success":true,
    "message":"You have successfully signed up!"
}
```

  – Inicio de sesión

Tabla 4.14 Inicio de sesión.

URL	https://sporhub-api.com/api/v1/login
Método HTTP	POST
Códigos de respuesta	[400, 200]
Formato de respuesta	Descrito posteriormente

Tabla 4.15 Parámetros de la petición.

Nombre del campo	Requerido	Tipo	Descripción
email	Si	String	Email del usuario
password	Si	String	Password del usuario

Ejemplo de petición y respuesta:

Código 4.13 Inicio sesión.

```
curl -X POST \
  https://api.sporthub.com/api/v1/login \
  -d '{
  "email":"user@mail.com",
  "password": "mypassword",
}'

{
    "success": true,
    "message": "You have successfully logged in!",
    "an_user_token",
    "user": {"email": "user@email.com"}
}
```

- **Recurso Deportes**
  - Obtener los deportes registrados en la plataforma

Tabla 4.16 Obtener deportes.

URL	https://sporhub-api.com/api/v1/sports
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta por un array donde cada elemento contiene el nombre de un deporte.

Por ejemplo:

Código 4.14 Obtener deportes.

```
["tennis","basketball","soccer"]
```

  - Añadir un deporte

Tabla 4.17 Añadir deporte.

URL	https://sporhub-api.com/api/v1/sports
Método HTTP	POST
Códigos de respuesta	[401, 400, 201]
Formato de respuesta	Descrito posteriormente

Tabla 4.18 Parámetros de la petición.

Nombre del campo	Requerido	Tipo	Descripción
name	Si	String	Nombre del deporte
numPlayers	Si	Number	Número mínimo de jugadores

Esta funcionalidad estará solamente disponible para un usuario administrador del sistema.

Ejemplo de petición y respuestaa:

Código 4.15 Añadir deporte.

```
curl -X POST \
  https://api.sporthub.com/api/v1/sports \
  -H 'authorization: bearer eyJhbGciOiJIUzI1NiIsIn
  R5cCI6IkpXVCJ9 \
  -H 'content-type: application/json' \
  -d '{
  "name":"soccker",
  "numPlayers":22,
}'

{
    "success":true,
    "message":"You have successfully added a new
    sport"
}
```

– Obtener información de un deporte en concreto

Tabla 4.19 Obtener información de deporte.

URL	https://sporhub-api.com/api/v1/sports/name/:name
Método HTTP	GET
Códigos de respuesta	[401, 400, 200]
Formato de respuesta	Descrito posteriormente

La respuesta estará compuesta por un objeto JSON con la información correspondiente del deporte: id en la base de datos, nombre y número de jugadores

Por ejemplo:

Código 4.16 Obtener información de deporte.

```
{
    "_id":"590f65feef618c49853c1cf4",
    "name":"soccer",
    "numPlayers":22
}
```

4.1.3 Desarrollo de la parte frontal

En esta sección se explica el desarrollo de la parte frontal de la aplicación *SportHub*. Como se ha mencionado en la sección 3.1.2, la tecnología utilizada para desarrollar esta parte ha sido la librería de JavaScript React JS. En la misma sección se explica que uno de los conceptos claves de React JS es el desarrollo basado en componentes.

Siguiendo los patrones de diseño de la librería, para esta aplicación se han desarrollado una serie de componentes que conformarán las distintas vistas. Además, estos componentes irán incluídos en otros artefactos llamados contenedores. Estos contenedores son simplemente componentes de React JS cuya función será renderizar otros componentes en la vista de la aplicación y comunicarse con el backend para traer la información necesaria (estos contenedores no son contenedores de Docker). Esta renderización consiste en incluir el componente de React JS dentro de un nodo del DOM que maneja el navegador.

Los componentes desarrollados han sido los siguientes:

- **Componente Base**: este componente, llamado *Base*, contiene las cabeceras que estarán incluídas en todas las vistas de la aplicación. Contiene una pequeña lógica con la que hace algunos cambios en las cabeceras dependiendo de si un usuario está autenticado.
- **Componente para la pantalla de llegada a la aplicación**: este componente es llamado *HomePage* y contiene la vista de la pantalla de llegada a la aplicación, cuando el usuario no está autenticado.
- **Componente para el inicio de sesión**: llamado *LoginForm*, incluye el formulario de inicio de sesión.
- **Componente para el registro de usuario**: llamado *SignUpForm*, incluye el formulario de registro de usuario.
- **Componente para vista principal tras la autenticación**: componente llamado *Dashboard* que contiene la vista que ve el usuario cuando se autentica en la aplicación. En esta vista tendrá diferentes opciones, como ver las actividades disponibles o añadir una nueva actividad o deporte.
- **Componente para ver las actividades registradas**: llamado *Activities*, componente que despliega un panel (*dashboard*) con las diferentes actividades registradas en la plataforma.
- **Componente para añadir nueva actividad**: *NewActivityForm*, contiene un formulario para añadir una actividad.

Los componentes descritos anteriormente son simplemente elementos visuales, donde no se incluye comunicación con el servicio para mostrar la información correspondiente o hacer las operaciones deseadas. Para lograr esta comunicación se han creado diferentes elementos, a los que se llama contenedores, que están formados por un componente (parte visual) y diferentes operaciones donde se determina la información que se muestra (comunicación con la parte del backend). Basicamente, la función de los contenedores es renderizar un componente con la información que ha adquirido a través de su comunicación con el servicio. Los contenedores de la aplicación son los siguientes:

- **Contenedor para la página de actividades**: llamado *ActivitiesPage*, este elemento se encarga de comunicarse con la API para proporcionar operaciones relacionadas con las diferentes actividades (eliminación, creación, unirse...) y además se encarga de renderizar el componente *Activities* para mostrar la información.

- **Contenedor para la vista principal al autenticarse**: su nombre es *DashboardPage*, y se encarga de renderizar el componente *Dashboard*.

- **Contenedor para el inicio de sesión**: *LoginPage*, su función es comunicarse con el backend para realizar la autenticación y renderizar el componente *LoginForm* para proporcionar la interfaz al usuario.

- **Contenedor para el registro de usuario**: *SignUpPage*, se encarga de la misma función que el contenedor del login para el registro de usuarios.

- **Contenedor para añadir una nueva actividad**: llamado *NewActivityPage*, se comunica con la parte de la API para añadir una actividad y renderiza el componente *NewActivityForm*.

En las siguientes figuras se puede ver la estructura de la aplicación y la función de cada contenedor.

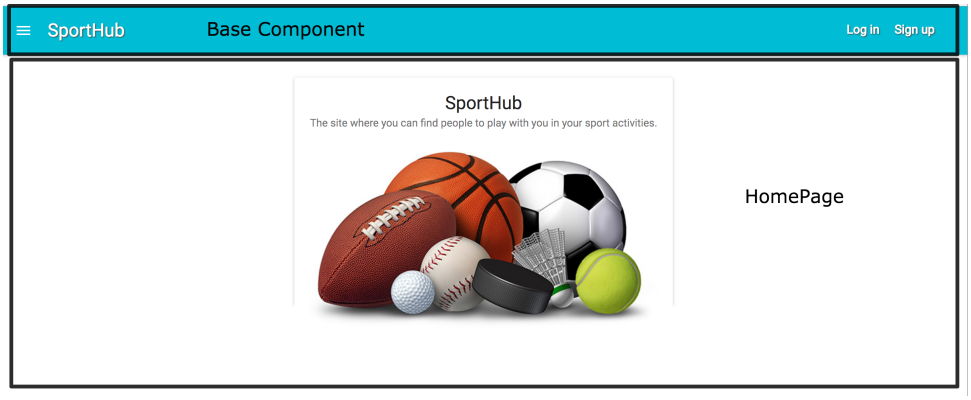

Figura 4.2 Pantalla de llegada a la aplicación.

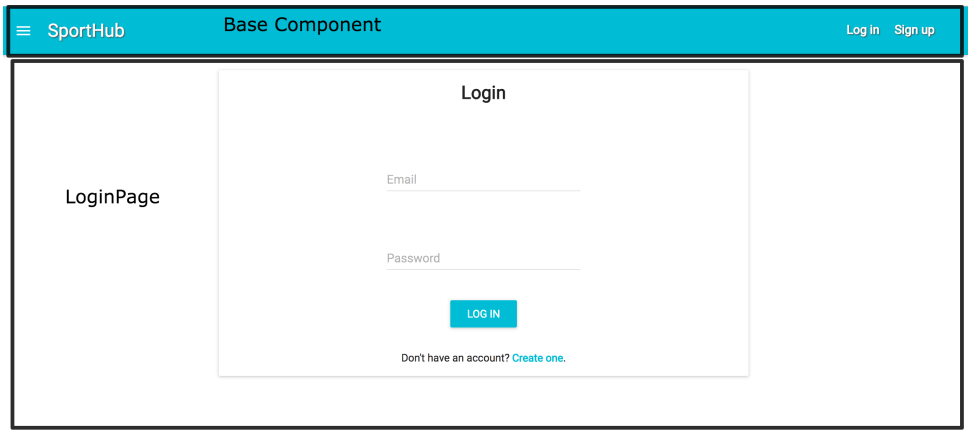

Figura 4.3 Pantalla Inicio sesión.

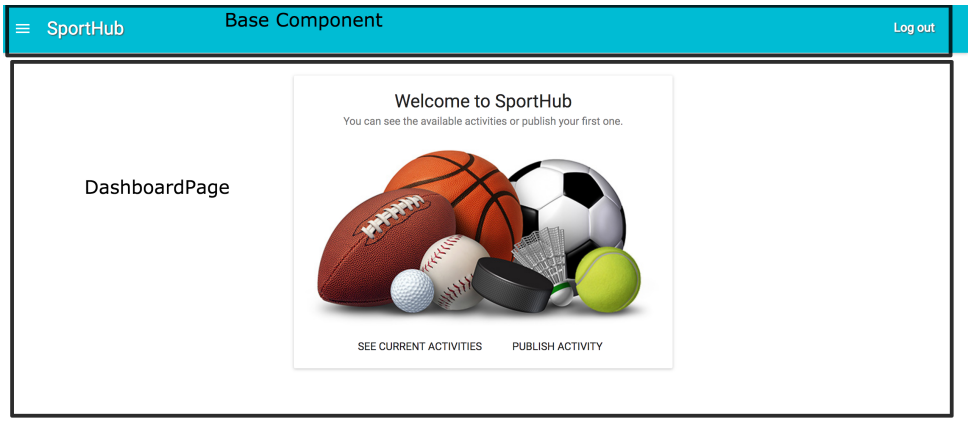

Figura 4.4 Pantalla principal al iniciar sesión.

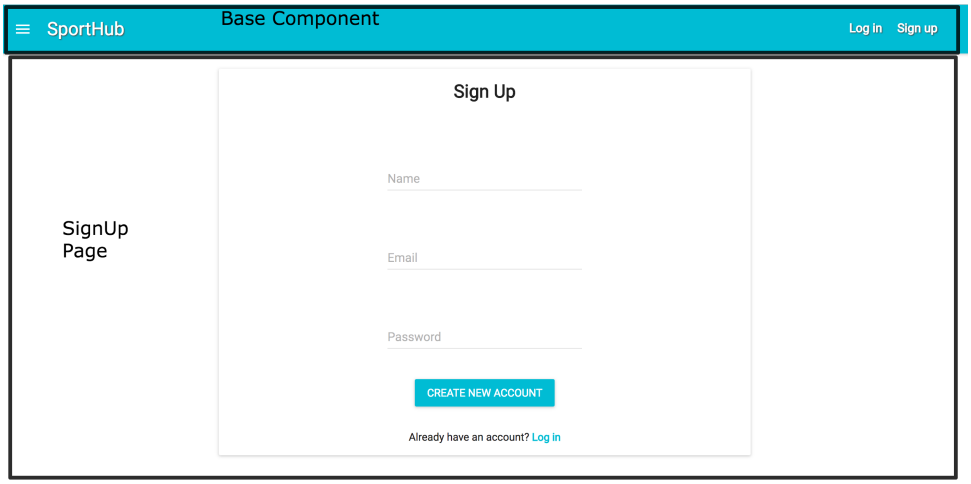

Figura 4.5 Pantalla Registro Usuario.

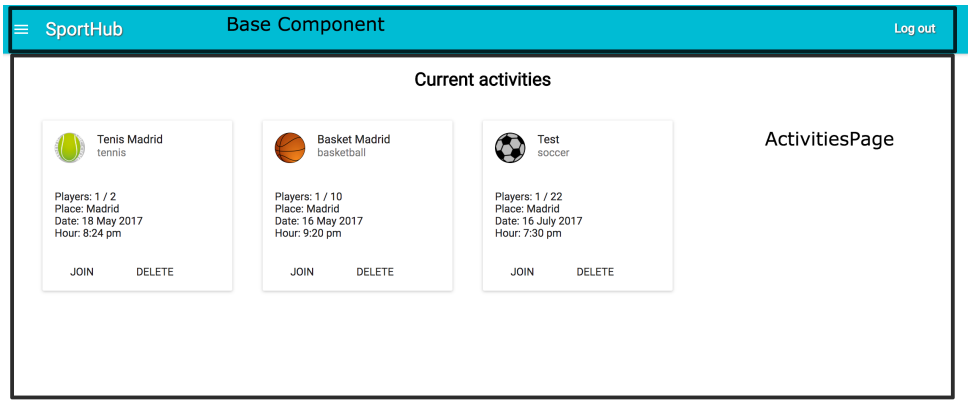

Figura 4.6 Pantalla Actividades Disponibles.

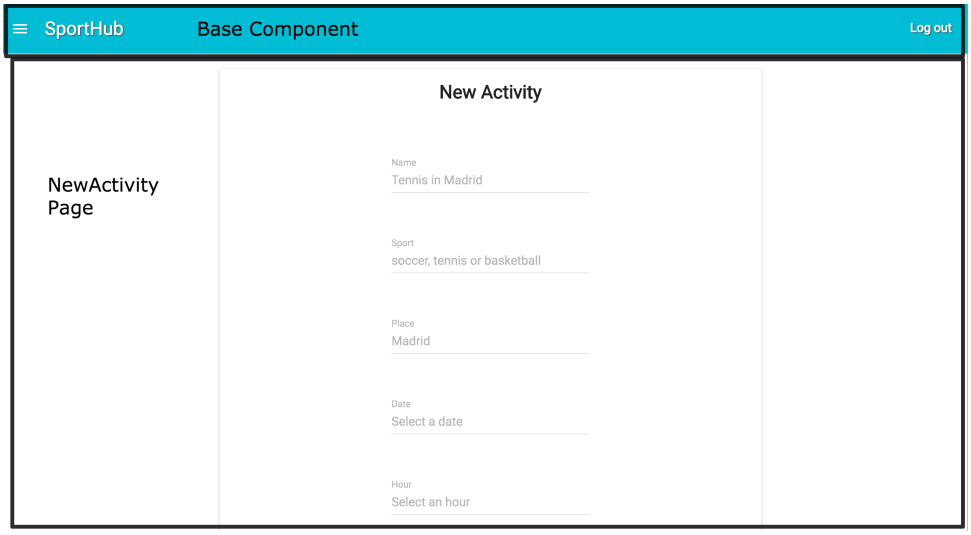

Figura 4.7 Pantalla Añadir Actividad.

4.1.4 Despliegue de la aplicación

Como se ha nombrado en la sección 3.2.3, el despliegue de la aplicación en la nube se va a efectuar utilizando *Google Cloud Platform*.

El requisito para poder hacer el despliegue de la aplicación en Google Cloud Platform es tener una cuenta en Google y tener un proyecto de Google Cloud Platform configurado. Es posible hacer ambas cosas siguiendo los siguientes pasos:

- Crear cuenta en Google en el siguiente enlace: *https://accounts.google.com/SignUpWithoutGmail?hl=en*
- Una vez con la cuenta, iniciar sesión e ir al siguiente enlace para crear un proyecto en Google Cloud Platform: *https://console.cloud.google.com/project*
- Una vez creado el proyecto, es posible verlo en la consola de Google:

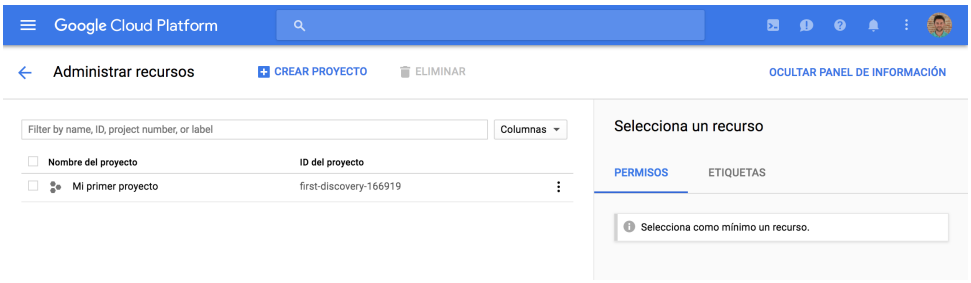

Figura 4.8 Google Cloud Console.

El despliegue completo de la plataforma consistirá en desplegar tres partes por separado:

- Despliegue de la base de datos
- Despliegue de la API de SportHub
- Despliegue de la parte frontal de SportHub

Despliegue de la base de datos

Como se expuso en la sección 3.1.3, la tecnología de base de datos elegida para este proyecto ha sido MongoDB. Para la base de datos, en lugar de usar un contenedor, se va a hacer uso directamente de una máquina que contenga MongoDB. Esto es debido a que la base de datos va a escalar a otro ritmo que el resto de la plataforma, y se va a separar del clúster de contenedores. Para desplegar una instancia de Google Cloud Platform con MongoDB de forma rápida se puede hacer uso de Google Launcher [13], una herramienta de Google para facilitar el proceso de lanzar instancias en la nube. Los pasos necesarios son los siguientes:

- Ir a Google Cloud Launcher habiendo iniciado sesión en Google y teniendo creado el proyecto de Google Cloud Platform.

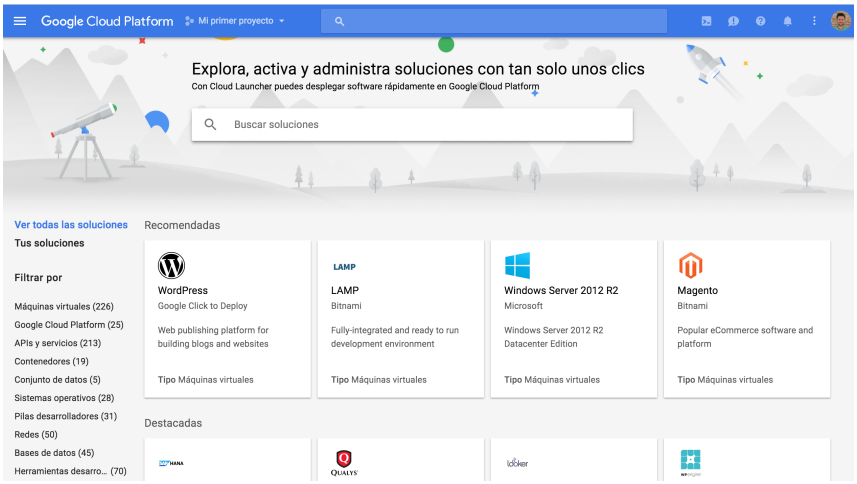

Figura 4.9 Google Cloud Launcher.

- Seleccionar MongoDB

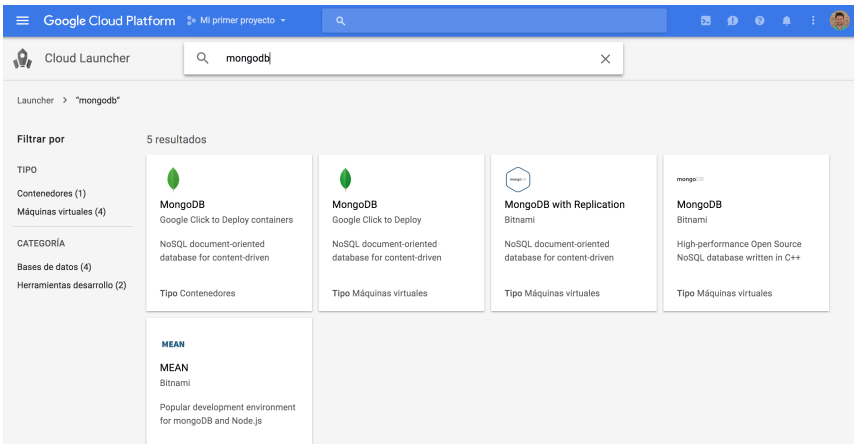

Figura 4.10 Seleccionar MongoDB.

- Se puede seleccionar cualquiera de las soluciones propuestas y aparecerá un botón para ejecutar la máquina directamente en Google Compute Engine.

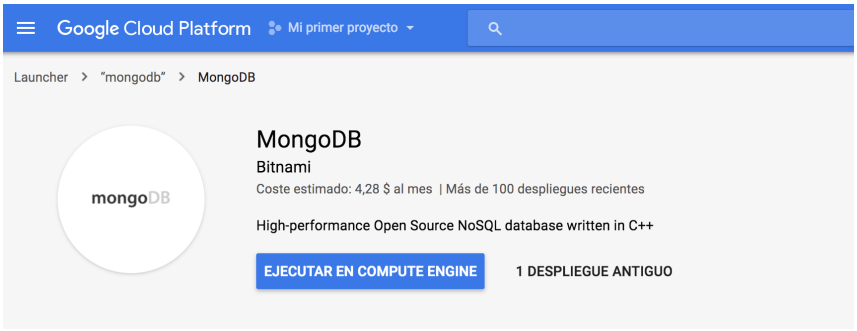

Figura 4.11 Opción de desplegar.

- En ese paso, automáticamente se creará una instancia en Google Compute Engine con MongoDB instalado. Por defecto, permitirá comunicación desde fuera en el puerto 27017, el puerto por defecto de MongoDB.

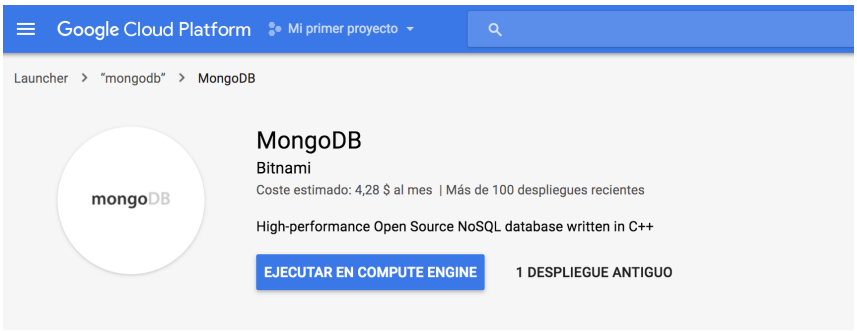

Figura 4.12 Opción de desplegar.

- A partir de ese momento, es posible conectarse a la base de datos a través de la aplicación usando ese puerto.

Despliegue de la API de SportHub

En este apartado se presenta el proceso de empaquetar la API de la aplicación SportHub en una imagen de Docker y, a continuación, desplegar esa imagen en un cluster de contenedores en Google Cloud Platform.

Antes de comenzar es necesario instalar algunas herramientas en la máquina local con las que poder establecer una comunicación con la API de Google Cloud Platform para contenedores y realizar las operaciones correspondientes.

- El primer paso para comenzar es instalar el SDK de Google Cloud [14], que incluye la herramienta de línea de comandos *gcloud*.
- Usando la herramienta *gcloud*, el siguiente paso es instalar la herramienta de línea de comandos de Kubernetes, *kubectl*. *kubectl* proporciona comunicación con Kubernetes, que es el sistema de orquestación de contenedores de Google Cloud Platform. Para instalar esta herramienta es necesario ejecutar el siguiente comando:

```
gcloud components install kubectl
```

- Instalar Docker Community Edition (CE) [7]. Docker será usado para construir la imagen de la aplicación.
- Instalar la herramienta de control de versiones Git [11] para conseguir el código del proyecto.

Establecer valores por defecto en la herramienta gcloud

Para no tener que escribir el ID de nuestro proyecto y la zona de nuestras máquinas [12] cada vez que utilicemos *gcloud*, es posible establecer una configuración por defecto ejecutando los siguientes comandos:

```
gcloud config set project PROJECT_ID

gcloud config set compute/zone YOUR_ZONE
```

Empaquetado y despliegue de la aplicación

Una vez configurada la herramienta de comunicación con Google Cloud Platform e instalado el software necesario para empaquetar la imagen, es posible comenzar con el proceso. Los pasos a seguir van a ser los siguientes:

- Construir una imagen de Docker con la aplicación.
- Subir la imagen al registro de imágenes de Docker de Google Cloud.
- Crear un clúster de contenedores.
- Desplegar la aplicación en el clúster.
- Exponer la aplicación a internet.
- Escalar el despliegue.
- Desplegar nuevas versiones del código de cualquier parte de la aplicación.

1. Construir imagen de Docker

Google Container Engine acepta imágenes de Docker como formato de despliegue de aplicación, por lo que será necesario construir una imagen de Docker con el código del servicio de SportHub para poder desplegarlo. El primer paso a seguir será obtener el código del repositorio [31] ejecutando el siguiente comando:

```
git clone git@github.com:Fortiz2305/sporthub-backend.git
cd sporthub-backend
```

Este repositorio contiene un Dockerfile con las instrucciones necesarias para construir la imagen. A continuación se va a guardar una variable de entorno con el nombre del proyecto, ya que será usada para hacer referencia a la imagen en el repositorio de imágenes. Para crear la variable de entorno basta con ejecutar el siguiente comando en una línea de comandos:

```
PROJECT_ID="$(gcloud config get-value project)"
```

Para construir la imagen de Docker:

```
docker build -t gcr.io/${PROJECT_ID}/sporthub-backend:v1 .
```

2. Subir la imagen al registro de Google Cloud

El siguiente paso es subir la imagen creada al repositorio de imágenes en Google Cloud Platform, de manera que Google Container Engine pueda descargarla. Es posible hacer este paso con el comando mostrado a continuación:

```
gcloud docker -- push gcr.io/${PROJECT_ID}/sporthub-backend:v1
```

3. Crear un clúster de contenedores

Una vez que la imagen está lista, el siguiente paso es crear un clúster de contenedores donde ejecutar la imagen. Un clúster consiste en un conjunto de instancias donde está ejecutándose Kubernetes, el orquestador de contenedores que proporciona Google Container Engine.

Una vez creado el cluster, se usará Kubernetes (a través de *kubectl*) para desplegar y gestionar la aplicación.

Para crear el clúster:

```
gcloud container clusters create sporthub-cluster --num-nodes=3
```

4. Despliegue de la aplicación

Como se ha mencionado anteriormente, ahora es necesario utilizar *kubectl* para establecer comunicación con Kubernetes y gestionar la aplicación. Kubernetes representa las aplicaciones como *Pods* [21], que pueden ser vistos como unidades que representan un contenedor o un grupo de contenedores que guarden relación. En este caso, cada *Pod* contiene solamente la imagen de la aplicación. Con el comando que se muestra a continuación se crea un Pod que estará escuchando en el puerto 3000, el puerto que expone la API de SportHub.

```
kubectl run hello-world --image=gcr.io/${PROJECT_ID}/sporthub-
   backend:v1 --port 3000
```

Es posible pasar como parámetro otros valores como variables de entorno de la misma forma que se le está pasando el puerto. El comando anterior crea un recurso llamado *Deployment* [20] en el clúster. El objeto Deployment gestiona diferentes copias de la aplicación (llamadas réplicas) y se encarga de que estas copias se ejecuten en nodos del clúster. En este caso, el objeto Deployment tiene solamente una replica de la aplicación.

5. Exponer la aplicación a Internet

Por defecto, los contenedores ejecutándose en Google Container Engine no son accesibles desde internet, ya que no tienen asignada una dirección IP pública. Es posible exponer la aplicación a internet utilizando el comando *kubectl expose*:

```
kubectl expose deployment sporthub-backend --type=LoadBalancer --
   port 3000
```

Con el comando anterior Kubernetes crea un recurso llamado Servicio [22], que da soporte de red a los pods de la aplicación. Cuando se crea un servicio, Kubernetes asigna automáticamente una dirección IP externa y crea un balanceador de carga para la aplicación, con el que se da soporte para poder añadir más réplicas. Una vez que Kubernetes asigna una dirección IP pública a la aplicación, es posible acceder a ella desde fuera.

```
→ kubectl get service
NAME         CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
kubernetes   10.31.240.1   <none>          443/TCP          6m
sporthub     10.31.245.9   35.187.119.12   3000:30270/TCP   46s
```

Figura 4.13 Comando kubectl get service.

6. Escalar el despliegue

Es posible escalar la aplicación y añadir más recursos ejecutando un solo comando. Utilizando el comando *kubectl scale* se pueden asignar todas las réplicas que se deseen para cualquier despliegue.

```
kubectl scale deployment sporthub-backend --replicas=3
```

En el caso de que se tengan numerosas réplicas, el balanceador de carga que se creó en el paso de creación del servicio reparte de forma automática el tráfico entre las diferentes réplicas.

7. Desplegar nuevas versiones del código de cualquier servicio de la aplicación

Google Container Engine proporciona un despliegue progresivo de las aplicaciones, de forma que no hay tiempo sin servicio entre diferentes despliegues. Si se quiere actualizar la aplicación con una nueva imagen de Docker, es posible hacerlo siguiendo los siguientes pasos:

Primero se crea una versión de la aplicación ejecutando el comando *docker build*.

```
docker build -t gcr.io/${PROJECT_ID}/sporthub-backend:v2 .
```

El siguiente paso es subir la nueva imagen al registro.

```
gcloud docker -- push gcr.io/${PROJECT_ID}/sporthub-backend:v2
```

Por último, actualizar la versión de la imagen en nuestro despliegue:

```
kubectl set image deployment/sporthub-backend sporthub-backend=gcr.
   io/${PROJECT_ID}/sporthub-backend:v2
```

Con el comando anterior, Google Container Engine desplegará la nueva versión del código de la aplicación antes de eliminar la versión del código que estaba anteriormente desplegada, de forma que no haya tiempo sin servicio durante el despliegue. Una vez la nueva versión está desplegada de forma estable, será cuando elimine la versión obsoleta.

Despliegue de la parte frontal de SportHub

Es posible desplegar la parte frontal de la aplicación siguiendo los mismos pasos expuestos anteriormente para el despliegue de la API. El código de la parte frontal está disponible en el siguiente repositorio: *https://github.com/Fortiz2305/sporthub-frontend*

4.2 Testeo de la aplicación

Como se ha expuesto en la sección anterior, será necesario que cada vez que se añada un cambio al código de la plataforma, se genere de forma automática un artefacto listo para ser desplegado a producción (imagen de Docker). Además, se ha desarrollado la infraestructura necesaria para que esto sea posible.

Para poder seguir estas prácticas sin que haya errores en la plataforma, es necesario un proceso de testeo cada vez que se añade un cambio. Este proceso de testeo se realiza en el entorno de integración continua, Travis CI, justo antes de generar la nueva imagen de Docker. Las etapas que se siguen cada vez que se hace un cambio en el código son las siguientes:

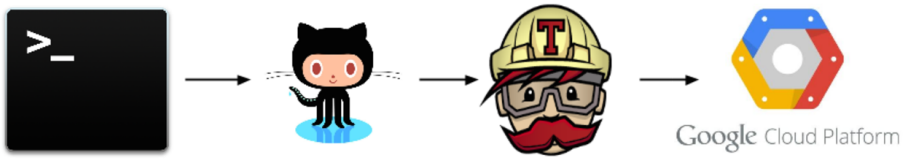

Figura 4.14 Etapas seguidas cuando se cambia el código.

- **1. Cambio en el código**.
- **2. Subir el código a GitHub**: plataforma donde está alojado el código del proyecto.
- **3. Ejecutar tests en Travis CI**: cada vez que se actualiza el código del proyecto en GitHub, se ejecuta un nuevo proceso en el entorno de integración continua automáticamente. En este proceso se realizará un testeo de la plataforma y, si los tests pasan correctamente, se generará una nueva imagen de Docker con la última versión del código de la aplicación.
- **4. Incluir imagen en Google Cloud Platform**: este proceso de integración continua también será encargado de incluir esta nueva imagen en el registro de imágenes de Docker de Google Cloud Platform. De esta forma, la imagen queda lista para el despliegue.

La configuración necesaria para que se hagan estos pasos en Travis CI se encuentra en el fichero de configuración de Travis CI de cada proyecto:

- Configuración API: *https://github.com/Fortiz2305/sporthub-backend/blob/master/.travis.yml*
- Configuración Parte frontal: *https://github.com/Fortiz2305/sporthub-frontend/blob/master/.travis.yml*

5 Conclusiones

En este proyecto se parte de la idea de hacer una plataforma web con unos requisitos que incluían, entre otros, despliegue automático y continuo de las diferentes partes de la plataforma. Para facilitar la implementación de la plataforma con los requisitos expuestos en la sección 4.1.1, se ha utilizado una arquitectura de microservicios donde cada servicio tiene una determinada función y no depende de las demás partes de la aplicación. Cada uno de estos servicios ha sido desplegado en un contenedor diferente utilizando recursos de computación en la nube.

Con este escenario se han conseguido diferentes ventajas:

- Al tener separadas las diferentes partes de la aplicación, se pueden administrar de forma individual, asignando los recursos que necesite cada parte en función de su demanda.
- El uso de contenedores para alojar la aplicación ha permitido poder testear las funcionalidades cuando estamos en desarrollo en un entorno que es exactamente igual que el entorno de producción, por lo que desaparecen los errores típicos al cambiar de entorno. Además, se tiene un sistema mucho más ligero incorporando solamente las librerías que requiere la aplicación.
- Añadir una nueva funcionalidad o el proceso de incorporación de un nuevo desarrollador a la plataforma será bastante más sencillo con una arquitectura dividida en funcionalidades que si se hubiera optado por desarrollar toda la aplicación monolítica. Si se quiere añadir una nueva parte a la plataforma, se pueden elegir de nuevo parámetros como el lenguaje de programación más convenientee, etc.
- Al usar recursos de computación en la nube, se ha hecho posible montar la infraestructura necesaria para el despliegue de la aplicación sin tener adquirir ningún equipamiento ni el conocimiento necesario para gestionarlo.

Uno de los problemas encontrados durante el desarrollo del proyecto, ha sido la persistencia de los datos utilizando contenedores, los cuales están pensados para ser reemplazables sin afectar al comportamiento del sistema. Por ello, la solución más sencilla ha sido utilizar una base de datos en una máquina en la nube, a la cual se conecta la API de la plataforma. Al no tener los mismos requisitos para la base de datos que para otras partes de la plataforma, no se han utilizado contenedores ni herramientas de orquestación para el despliegue de la misma. Esto ha proporcionado sencillez y un ahorro económico, cumpliendo con

creces los requisitos que se le exigen a esta parte en este momento. Además, esta máquina también puede ir escalando en un futuro.

En cuanto a mejoras para el futuro, hay pensadas algunas líneas de avance que no se han abordado en este proyecto:

- Como se ha mencionado en algunas secciones del proyecto, se creará un usuario administrador y será el único que tenga acceso a determinadas funciones de la API: listado de todos los usuarios, creación y eliminación de deportes o borrado de cualquier actividad.
- Actualmente, cuando un usuario entra en la plataforma, aparecen todas las actividades que se hayan incorporado a la plataforma. En un futuro, se puede incorporar la localización favorita del usuario para que la plataforma muestre las actividades de interés al usuario en función de su situación geográfica.
- En el desarrollo de este proyecto no se ha puesto mucho empeño en el diseño gráfico de la aplicación, ya que no era el principal objetivo.

Por último, también apuntar que el mundo de los contenedores y los recursos de computación en la nube está en constante evolución y las grandes compañías que hay detrás están incorporando nuevas funcionalidades con una gran rapidez. Este es un buen momento para el desarrollo de este proyecto ya que se han asentado algunas de las tecnologías y paradigmas de computación que se usan actualmente en empresas modernas y que se irán incorporando poco a poco en los próximos años a un mayor número de usuarios.

Índice de Figuras

2.1	Contenedores vs Máquinas Virtuales	4
2.2	Namespace de proceso	6
2.3	Orquestación en Docker	7
2.4	SPA vs Aplicación web tradicional	9
3.1	Motor v8 en el navegador	11
3.2	Motor v8 en el servidor	12
3.3	Hilo de ejecución de Node.js	12
3.4	Delegación del trabajo al conjunto de hilos	12
3.5	Arquitectura MongoDB	14
3.6	Dockerfile	16
3.7	Docker Images	16
3.8	Estructura de Docker	17
4.1	Arquitectura de SportHub	24
4.2	Pantalla de llegada a la aplicación	36
4.3	Pantalla Inicio sesión	37
4.4	Pantalla principal al iniciar sesión	37
4.5	Pantalla Registro Usuario	37
4.6	Pantalla Actividades Disponibles	38
4.7	Pantalla Añadir Actividad	38
4.8	Google Cloud Console	39
4.9	Google Cloud Launcher	39
4.10	Seleccionar MongoDB	40
4.11	Opción de desplegar	40
4.12	Opción de desplegar	40
4.13	Comando kubectl get service	43
4.14	Etapas seguidas cuando se cambia el código	44

Índice de Tablas

4.1	Códigos HTTP	25
4.2	Obtener todos los usuarios del sistema	26
4.3	Obtener todos los emails registrados	27
4.4	Obtener información de usuario	27
4.5	Obtener usuario actual	28
4.6	Obtener todas las actividades	28
4.7	Añadir actividad	29
4.8	Parámetros de la petición	29
4.9	Unir usuario a actividad	30
4.10	Parámetros de la petición	31
4.11	Borrar actividad	31
4.12	Registro de usuario	32
4.13	Parámetros de la petición	32
4.14	Inicio de sesión	32
4.15	Parámetros de la petición	33
4.16	Obtener deportes	33
4.17	Añadir deporte	34
4.18	Parámetros de la petición	34
4.19	Obtener información de deporte	34

Índice de Códigos

2.1 Comando chroot 3

3.1 Comando docker run 15
3.2 Comando construcción contenedor 16
3.3 Comando construcción contenedor 16
3.4 Comando construcción contenedor 18

4.1 Mensaje de error API 26
4.2 Petición Autenticada API 26
4.3 Obtener usuarios 27
4.4 Obtener emails 27
4.5 Obtener usuario 27
4.6 Obtener usuario 28
4.7 Obtener actividades 28
4.8 Añadir actividad 29
4.9 Añadir actividad 30
4.10 Unirse a actividad 31
4.11 Borrar actividad 31
4.12 Añadir usuario 32
4.13 Inicio sesión 33
4.14 Obtener deportes 33
4.15 Añadir deporte 34
4.16 Obtener información de deporte 35

Bibliografía

[1] *Amazon ec2*, https://aws.amazon.com/es/ec2/.

[2] *Amazon ecs*, https://aws.amazon.com/es/ecs/.

[3] *Amazon rds*, https://aws.amazon.com/es/rds/.

[4] *Amazon s3*, https://aws.amazon.com/es/s3/.

[5] *Angular js*, https://angularjs.org/.

[6] *Docker commands*, https://docs.docker.com/engine/reference/commandline/docker/.

[7] *Docker installation*, https://docs.docker.com/engine/installation/.

[8] *Dockerhub*, https://hub.docker.com/.

[9] *Document object model*, https://www.w3.org/TR/DOM-Level-2-Core/introduction.html.

[10] *Ember js*, https://www.emberjs.com/.

[11] *Git installation*, https://git-scm.com/downloads.

[12] *Google cloud compute zones*, https://cloud.google.com/compute/docs/zones#available.

[13] *Google cloud launcher*, https://cloud.google.com/launcher/?hl=es.

[14] *Google cloud sdk*, https://cloud.google.com/sdk/docs/quickstarts.

[15] *Google compute engine*, https://cloud.google.com/compute/?hl=es.

[16] *Google compute storage*, https://cloud.google.com/storage/?hl=es.

[17] *Google container engine*, https://cloud.google.com/container-engine/?hl=es.

[18] *Hypervisor*, http://searchservervirtualization.techtarget.com/definition/hypervisor.

[19] *Json web tokens*, https://jwt.io/.

[20] *Kubernetes deployment*, *https://kubernetes.io/docs/user-guide/deployments/*.

[21] *Kubernetes pods*, *https://kubernetes.io/docs/user-guide/pods/*.

[22] *Kubernetes service*, *https://kubernetes.io/docs/user-guide/services/*.

[23] *Libuv*, *https://github.com/libuv/libuv*.

[24] *Martin fowler*, *https://martinfowler.com/*.

[25] *Microsoft azure*, *https://azure.microsoft.com/es-es/*.

[26] *Mongo db*, *https://www.mongodb.com/es*.

[27] *Nosql databases explained*, *https://www.mongodb.com/nosql-explained*.

[28] *React*, *https://facebook.github.io/react/*.

[29] *Rich internet applications (ria)*, *http://searchmicroservices.techtarget.com/definition/Rich-Internet-Application-RIA*.

[30] *Rpc*, *http://searchmicroservices.techtarget.com/definition/Remote-Procedure-Call-RPC*.

[31] *Sporthub backend*, *https://github.com/Fortiz2305/sporthub-backend*.

[32] *Virtual dom*, *http://www.arquitecturajava.com/que-es-el-virtual-dom-y-como-funciona/*.

[33] *Yaml*, *http://yaml.org/*.

[34] *Youtube api reference*, *https://developers.google.com/youtube/v3/sample_requests*.

[35] Amazon, *Amazon web services*, *https://aws.amazon.com/es/*.

[36] Docker, *Docker volumes*, *https://docs.docker.com/engine/tutorials/dockervolumes/*.

[37] GitHub, *Github*, *https://github.com/*.

[38] Google, *Google cloud platform*, *https://cloud.google.com/?hl=es*.

[39] Inc. Google, *Grpc*, *http://www.grpc.io/*.

[40] HashiCorp, *Vagrant*, *https://www.vagrantup.com/*.

[41] Docker Inc., *Docker blog*, *https://blog.docker.com/2016/06/docker-1-12-built-in-orchestration/*.

[42] Google Inc., *Chrome v8*, *https://developers.google.com/v8/*.

[43] Yehuda Katz, *Handlebars js*, *http://handlebarsjs.com/*.

[44] Uber, *Uber api reference*, *https://developer.uber.com/docs/riders/references/api*.