UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Red social temática para la clasificación de libros, películas y videojuegos

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergio Pérez Andrés

Tutoras: Manoli Albert Albiol,
Victoria Torres Bosh

2018-2019

Resumen

El proyecto que se ha llevado a cabo consiste en el diseño e implementación de una red social temática que permita al usuario llevar a cabo el seguimiento y valoración de todos los libros, videojuegos y películas que haya consumido a lo largo del tiempo. A su vez, el usuario será capaz de compartir toda esta información con otros usuarios, así como descubrir nuevos títulos gracias al sistema de búsqueda o recursos compartidos por otros usuarios.

El proyecto se inicia con una fase de análisis, de la cual obtendremos los requisitos que debe cumplir nuestra aplicación y la planificación del trabajo a través de metodologías ágiles. A continuación, en la etapa de diseño se discutirán las diferentes arquitecturas y patrones utilizados. Estos patrones serán implementados en la siguiente etapa, donde a su vez, destacaremos los retos más importantes desarrollados. Después, realizaremos un estudio de las posibles opciones a la hora de desplegar o implantar nuestra aplicación. Por último, realizaremos una serie de pruebas que verifiquen la accesibilidad de la aplicación.

Palabras clave: Red social, API, Microservicios, React, Aplicación Web Responsiva.

Abstract

The project that has been carried out consists of the design and implementation of a thematic social network that allows the user to follow and review all the books, video games and movies that he has consumed over time. In turn, the user will be able to share all this information with other users, as well as discover new titles thanks to the search system or resources shared by other users.

The project starts with an analysis stage, where we will obtain the requirements that our application must satisfy and work planning through agile methodologies. Next, we will go to the design stage, where the different architectures and patterns will be discussed. The result of design stage will be implemented in the next stage. Then, we will carry out a study of different possible options when deploying or implementing our application, Finally, we will perform a set of tests to verify the accessibility of the application.

Keywords: Social Network, API, Microservices, React, Responsive Web Application.

Tabla de contenidos

1. Introducción 11

1.1. Motivación 11

1.2. Objetivos 12

1.3. Estructura 12

2. Estado del arte 14

2.1. IMDB 14

2.2. 3djuegos 15

2.3. Goodreads 17

2.4. Libib 18

3.1. Requisitos 19

3.1.1. Requisitos funcionales 19

3.1.2. Requisitos no funcionales 22

3.2. Casos de uso 23

3.3. Identificación y análisis de soluciones posibles 23

3.4. Metodología 25

3.5. Estudio de *APIs* 26

4. Fase de diseño 28

4.1. Arquitectura 28

4.2. Tecnologías 29

4.2.1. *Balsamiq* 29

4.2.2. *Draw.io* 29

4.2.3. *Javascript* 29

4.2.4. *React JS*, *Redux* y *React Router* 29

4.2.5. *Node JS* 30

4.2.6. *Mongo DB* y *MongoDB Compass Community* 31

4.2.7. *Postman* 32

4.2.8. *Visual Studio Code* 32

4.2.9. *Git*, *Github y Gitkraken* 32

4.3. Diseño detallado 34

4.3.1. Diseño del Modelo Conceptual 34

4.3.2. Diseño de la Lógica 35

4.3.3. Diseño de la Interfaz de Usuario 42

5. Desarrollo de la solución propuesta 46

5.1. Estructura de ficheros del repositorio 47

5.1.1. Etapa inicial 47

5.1.2. Segunda etapa 47

5.1.3. Etapa final 48

5.2. Cliente 49

5.2.1. Organización de ficheros 49

5.3. Servicio de autenticación 55

5.3.1. Organización de ficheros 55

5.3.2. Módulos 56

5.4. Servicio de la aplicación principal 58

5.4.1. Organización de ficheros 58

5.4.2. Módulos 59

5.5. Servicio de búsqueda 62

5.5.1. Organización de ficheros....63
5.5.2. Módulos....63
6. Implantación.... 68
6.1. Estudio de entornos para el despliegue 68
6.2. Conclusiones de la implantación 70
7. Pruebas 72
8. Conclusiones....74
9. Trabajos futuros....75
10. Referencias 76
11. Anexo 1: Despliegue en *Heroku*....78
12. Anexo 2: Despliegue en *Azure* 84

Tabla de figuras

Figura 1 Página principal de www.imdb.com 15
Figura 2 Página de bibliotecas en 3djuegos 16
Figura 3 Página principal en goodreads.com 17
Figura 4 Página de bibliotecas en www.libib.com 18
Figura 5 Casos de uso de la aplicación 23
Figura 6 Tablero Kaban del Sprint 1 26
Figura 7 Arquitectura básica de la aplicación 28
Figura 8 Herramienta de proyectos en Github 33
Figura 9 Ilustración de la interfaz gráfica de Gitkraken 34
Figura 10 Esquema final de la base de datos 35
Figura 11 Esquema aplicación web dinámica 36
Figura 12 Esquema aplicación SPA (Single Page Application) 37
Figura 13 Arquitectura de microservicios con autenticación centralizada 39
Figura 14 Arquitectura de microservicios con autenticación descentralizada 41
Figura 15 Petición con más de un microservicio involucrado 41
Figura 16 Mockup de la identificación y registro de usuario 42
Figura 17 Mockup página principal de la aplicación 43
Figura 18 Mockup de la página bibliotecas 43
Figura 19 Mockup ventana modal para añadir bibliotecas 44
Figura 20 Mockup página de búsqueda 44
Figura 21 Mockup página de recurso 45
Figura 22 Mockup página perfil de usuario con bibliotecas 46
Figura 23 Mockup página de perfil con seguidores y seguidos 46
Figura 24 Configuración de los espacios de trabajo 48
Figura 25 Scripts utilizados durante el desarrollo 49
Figura 26 Estructura de ficheros en el paquete cliente 49
Figura 27 Rutas de la aplicación con react-router 50
Figura 28 Estructura de las páginas de la aplicación 50
Figura 29 Diferencia entre la gestión del estado de React y React-Redux 51
Figura 30 Ejemplo de reducer para la autenticación 52
Figura 31 Posibles acciones en la autenticación 53
Figura 32 Ejemplo de acción dentro de Redux 54
Figura 33 Uso del estado y de sus acciones en un componente 55
Figura 34 Estructura de ficheros paquete de autenticación 55
Figura 35 Código con la generación del JWT 56

Figura 36 Método de autenticación 57
Figura 37 Ruta protegida mediante autenticación 57
Figura 38 Directorios y ficheros de la API principal 58
Figura 39 Configuración de la aplicación principal 59
Figura 40 Ejemplo de rutas en Express 60
Figura 41 Validación del cuerpo del mensaje 60
Figura 42 Función para obtener el estado de un recurso 61
Figura 43 Esquema del objeto Info 62
Figura 44 Estructura de ficheros servicio de búsqueda 63
Figura 45 Configuración de la caché 64
Figura 46 Método de búsqueda en todas las APIs 65
Figura 47 Implementación fábrica abstracta en Javascript 66
Figura 48 Interfaz de la estrategia usada en las búsquedas en Javascript 67
Figura 49 Página principal del panel de control en Heroku 68
Figura 50 Panel de control Azure 69
Figura 51 Configuración herramienta de auditoría 72
Figura 52 Registro en Heroku 78
Figura 53 Crear aplicación en Heroku 79
Figura 54 Crear configuración de Heroku 79
Figura 55 Configuración de la rama a desplegar en Heroku 80
Figura 56 Buildpacks para React 81
Figura 57 Variables de entorno de React en Heroku 81
Figura 58 Configuración del fichero static.js 82
Figura 59 Buildpacks NodeJs 82
Figura 60 Incluir MongoDB en el proyecto 82
Figura 61 Dynos activos 84
Figura 62 Página inicial del panel de control de Azure 84
Figura 63 Creación de un grupo de recursos en Azure 85
Figura 64 Configuración básica de la base de datos en Azure 86
Figura 65 Activar protocolo de transferencia de CosmosDB 86
Figura 66 Configuración de React en Azure 87
Figura 67 Desplegar React en Azure 88
Figura 68 Fichero process.json 89
Figura 69 Configuración extra de las APIs en Azure 89
Figura 70 Configuración básica de un App Service 90
Figura 71 Configuración de CORS en Azure 91

1. Introducción

Como es bien sabido, el ocio es uno de los grandes pilares de nuestra sociedad actual. No pasa un día en el que no leamos un libro, revista o artículo, veamos la televisión o juguemos con nuestro teléfono móvil, ordenador o consola.

Si nos dirigimos a las últimas encuestas de hábitos y prácticas culturales publicadas en 2014-2015[1] por el gobierno de España (página 61, figura R1) podremos observar que el crecimiento de 2-5% es positivo a lo largo de los años. Los ejemplos que más nos interesan para este proyecto según las actividades culturales realizadas en el último año son la lectura, donde aproximadamente un 62% de la población leyó algún tipo libro, el cine, donde un 54% vio alguna película y videojuegos, donde casi un 14% jugó a algún título durante el último año.

A medida que el tiempo avanza la cantidad de recursos aumenta y la mayoría de las veces solo recordamos de manera muy ligera cómo fue un libro, película o videojuego y algún pensamiento o idea principal del recurso consumido.

Por este motivo, y como dice la frase de García Márquez, *"El que no tiene memoria, se hace una de papel"* nuestra memoria tiene un límite y en muchas ocasiones no somos capaces de retener toda la información o pensamientos generados a raíz de un libro, película o videojuego. En este caso no utilizaremos un trozo de papel, pues gracias al avance de las tecnologías contamos con herramientas mucho más potentes como nuestros teléfonos móviles u ordenadores.

1.1. Motivación

La idea de la aplicación surge de todas las conversaciones que se tienen a diario sobre lo buena que ha sido una película recién estrenada, lo mucho que has jugado a un juego o lo enganchado que te tiene tu última lectura.

Junto a esas conversaciones nace la necesidad de llevar la cuenta de todos estos recursos. En un primer momento, uno busca por Internet buscando una plataforma que satisfaga sus necesidades.

[1] https://www.culturaydeporte.gob.es/dam/jcr:ad12b73a-57c7-406c-9147-117f39a594a3/encuesta-de-habitos-y-practicas-culturales-2014-2015.pdf

Sin embargo, la ausencia o no cumplimiento de todos los requisitos son los que inician el planteamiento y desarrollo de la idea inicial.

1.2. Objetivos

El principal objetivo que buscamos en este proyecto es desarrollar una aplicación que permita a sus usuarios llevar a cabo un seguimiento de su ocio, además de poder compartir dichas interacciones con otros usuarios.

Este objetivo puede dividirse a su vez en otros más pequeños. El primero de ellos es la diversidad de dispositivos. Un usuario debe ser capaz de utilizar la aplicación en cualquier dispositivo del mercado, o en su defecto, en el mayor número posible.

Partiendo de esta diversidad de dispositivos, queremos que la aplicación se accesible para todo el mundo y por ello se realizaran pruebas de accesibilidad al final del desarrollo. Y de la accesibilidad, damos el salto al apartado gráfico. Se desea también que la aplicación tenga un aspecto minimalista y adaptable a cualquier tipo de pantalla.

Entrando en el apartado técnico del proyecto, se propone utilizar una arquitectura de microservicios la cual nos permita escalar la aplicación en caso de necesidad. Acompañando a este objetivo, tenemos la implantación o despliegue, pues se estudiarán distintas posibilidades a la hora de llevar a cabo un despliegue con microservicios.

1.3. Estructura

En este apartado del proyecto hablaremos sobre la estructura general del mismo, con el objetivo de que el lector tenga una ligera idea a la hora de comenzar un nuevo apartado.

- En el capítulo uno, donde nos encontramos actualmente, hallaremos la introducción al proyecto, cuáles son sus objetivos y la estructura del mismo.
- En el capítulo dos presentamos una revisión de distintas aplicaciones web del mercado cuyas funcionalidades y objetivos son similares a la aplicación a desarrollar en este TFG.
- En el tercer capítulo desarrollamos la primera etapa del proceso la cual consiste en la toma de requisitos, generación de casos de uso y posibles opciones a desarrollar. Al final de este capítulo se realizará un análisis de las APIs utilizadas a la hora de obtener los datos.
- El cuarto capítulo, diseño, se explicará de una manera más teórica la arquitectura de la aplicación.

- La implementación la encontramos en el quinto capítulo, en ella hablaremos de los puntos más importantes de cada uno de los componentes de la aplicación.
- El sexto capítulo hablará sobre las posibles implantaciones o despliegues de nuestra aplicación.
- En el séptimo hablaremos sobre las pruebas de accesibilidad, rendimiento y buenas prácticas de la aplicación cliente.
- Los últimos dos capítulos corresponden a la conclusión del trabajo y las referencias utilizadas durante su desarrollo.

2. Estado del arte

En la actualidad existen gran cantidad de aplicaciones o sitios web dedicados al ocio y con un objetivo similar al del proyecto planteado. Generalmente podemos diferenciar estas aplicaciones en dos tipos.

El primer tipo, son páginas o aplicaciones dedicadas a un sector. *IMDB* o *3djuegos*, las cuales analizaremos más adelante, son un ejemplo de éstas. Además de su contenido de noticias, foros, etc., cuentan con su propio sistema de biblioteca y compartición para llevar una lista sobre las películas o series que estás viendo actualmente en el caso de *IMDB* o los videojuegos que has jugado en el caso de *3djuegos*.

El segundo tipo, y en el cual situamos a nuestra aplicación, encontramos las aplicaciones cuya función principal es almacenar de manera ordenada los recursos que vamos consumiendo. En este grupo, a su vez encontramos, de nuevo, dos tipos. Por un lado, un sector más especializado en una única materia, como son los libros en el caso de *GoodReads*. Por el otro lado, encontraríamos a un sector más generalista donde se pretende almacenar todos los posibles recursos. En este grupo encontramos la aplicación de *Libib* y donde se situará la aplicación desarrollada en este proyecto.

2.1. IMDB[2]

En primer lugar, analizaremos *IMDB*. La aplicación más longeva de la lista, que no la más vieja, pues ha sufrido distintos cambios a lo largo de su historia. Esta historia comienza un par de años antes de los años noventa como una red de usuarios sobre actrices. En 1992 se publica la primera versión de su sitio web. Desde 1998 y hasta la actualidad es propiedad de Amazon.

Los dueños de la página la describen como la mayor fuente autorizada de recursos relacionados con películas y el contenido televisivo. Diseñada para ayudar a los fanes a explorar en el mundo de las películas, espectáculos y así decidir qué poder ver [Figura 1].

[2] Página web: www.imdb.com

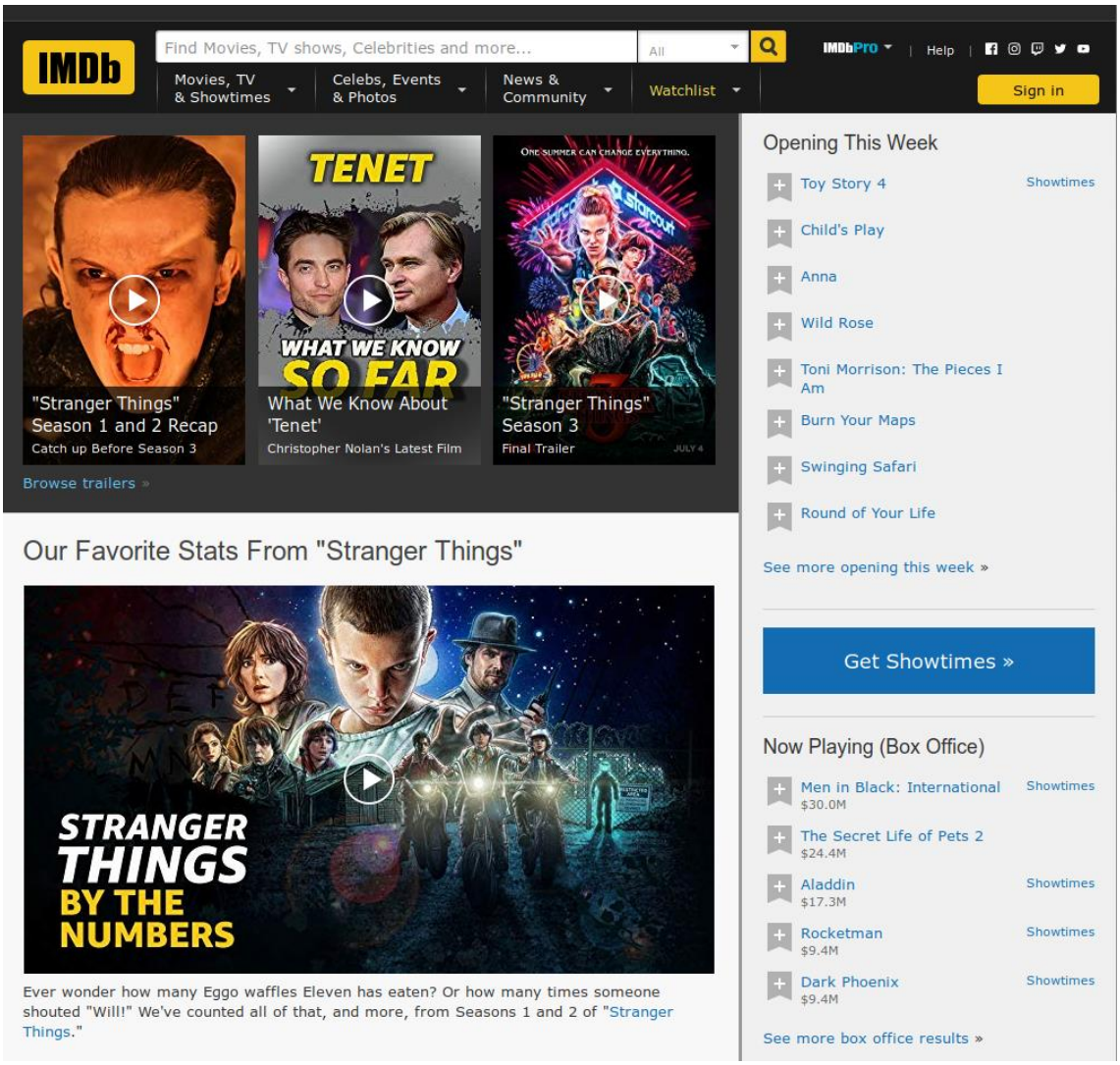

Figura 1 Página principal de www.imdb.com

Las características esta página, entre otras, son visualizar los tráileres de las películas, leer críticas o realizarlas, galerías de fotos y almacenar las películas en una sección llamada *Watchlist*.

La principal diferencia que encontramos con respecto al proyecto es la forma de descubrir. Pues su página principal nos recuerda a un blog de noticias, mientras que en nuestro proyecto nos enfocamos más a las películas que han podido ver las personas a las que seguimos.

2.2. 3djuegos[3]

Lo primero que tenemos que decir de 3djuegos, es que es un portal de noticias relacionado con la industria de videojuegos, por lo que la tarea de crear listas de

[3] Página web: www.3djuegos.com

videojuegos pasa a un segundo plano. Fue lanzado en junio del 2005 y los datos más recientes sobre usuarios activos datan de 2010 con casi cuatro millones de usuario únicos.

Al no encontrar mejor alternativa en lo que respecta al mundo de los videojuegos en forma de página web o aplicación móvil, se ha seleccionado ésta [Figura 2].

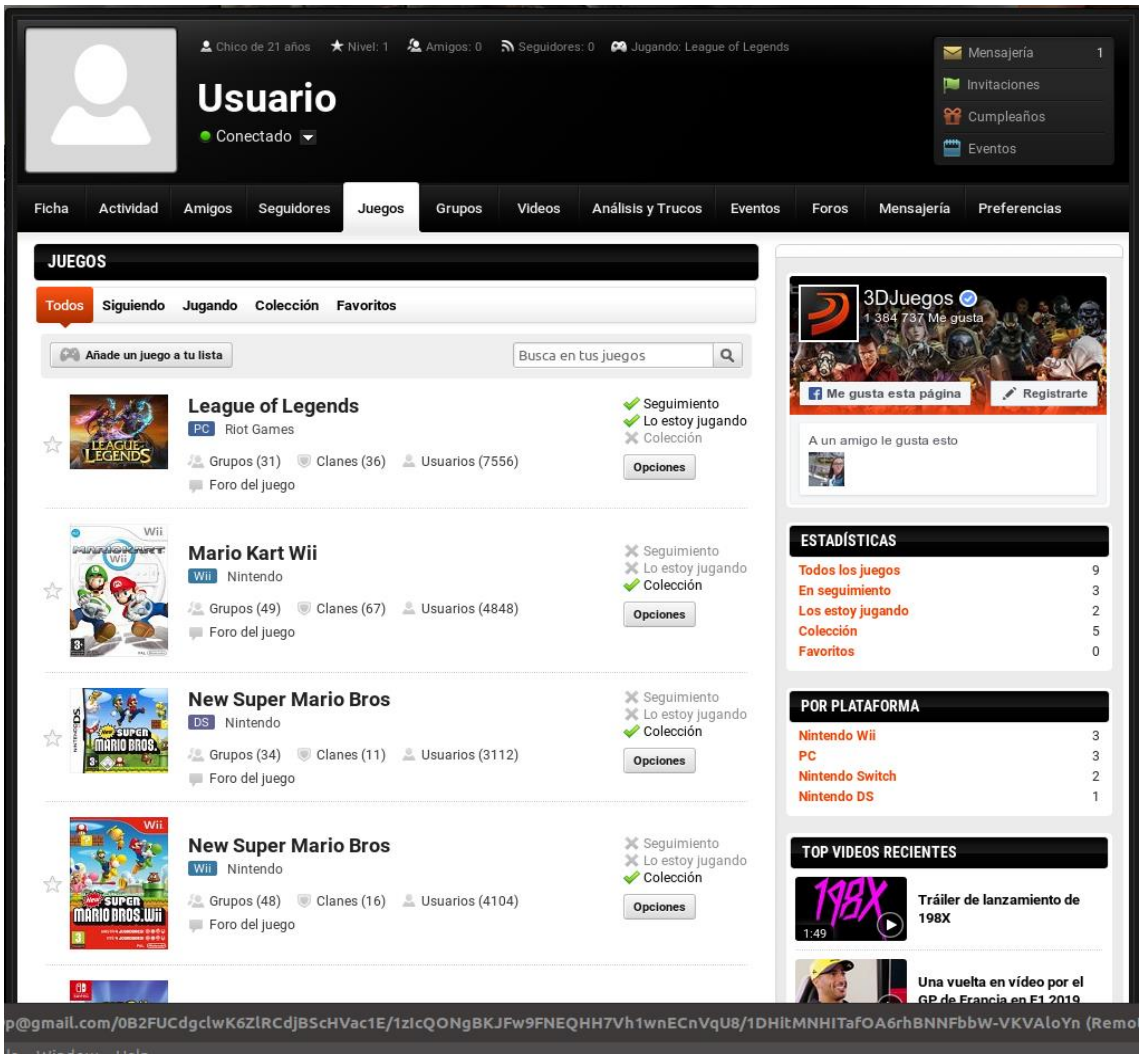

Figura 2 Página de bibliotecas en 3djuegos

Las diferencias con nuestra aplicación son claras. En 3djuegos todo gira en torno a un blog de noticias donde los usuarios participan con sus comentarios. De manera adicional, dichos usuarios pueden mantener una lista con los videojuegos que o bien han jugado, están jugando en este momento o que siguen, para que sean informados de las últimas novedades.

2.3. Goodreads[4]

A continuación, damos paso a la aplicación en la que se inspiró el proyecto en un primer momento. La página web fue lanzada en diciembre del 2006, en 2007 ya contaba con más de 650.000 miembros. Posteriormente, en 2012 reportó diez millones de miembros. En 2013, Amazon anunció la compra de Goodreads, y en el momento actual de la publicación de este trabajo, sigue siendo propiedad de dicha empresa.

Podríamos describirla de una manera muy parecida al título del proyecto, pues estamos ante una red social de catalogación que gira en torno al mundo de la literatura.

Cuenta con muchas opciones en todas sus funcionalidades y pretende darnos gran cantidad de atajos para usuarios expertos, sin embargo, muchas veces podemos encontrar una sobre información, sobre todo en una etapa temprana del uso de la aplicación [Figura 3].

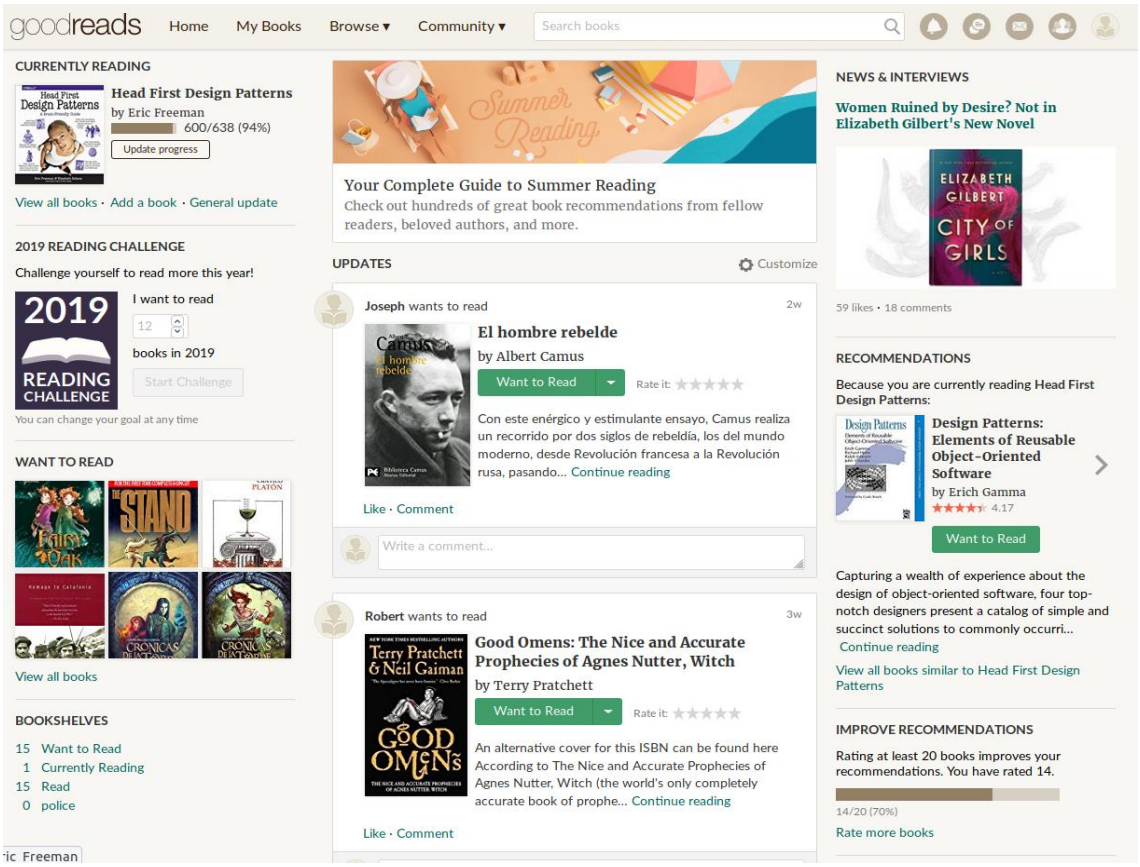

Figura 3 Página principal en goodreads.com

[4] Página web: www.goodreads.com

Las principales diferencias que encontraremos respecto a nuestro proyecto serán la temática, puesto que en nuestro proyecto no nos centraremos en una única temática, y la complejidad pues intentaremos mantener una interfaz más limpia, así como un repaso de funcionalidades que puedan llegar a ser de poca utilidad o se salgan del ámbito de estudio del proyecto.

2.4. Libib[5]

En último lugar encontramos una aplicación web con los objetivos muy similares a los nuestros. Pues en su interior encontramos el almacenaje de los de los recursos en bibliotecas. Estas bibliotecas están claramente diferenciadas por tipo de recurso [Figura 4].

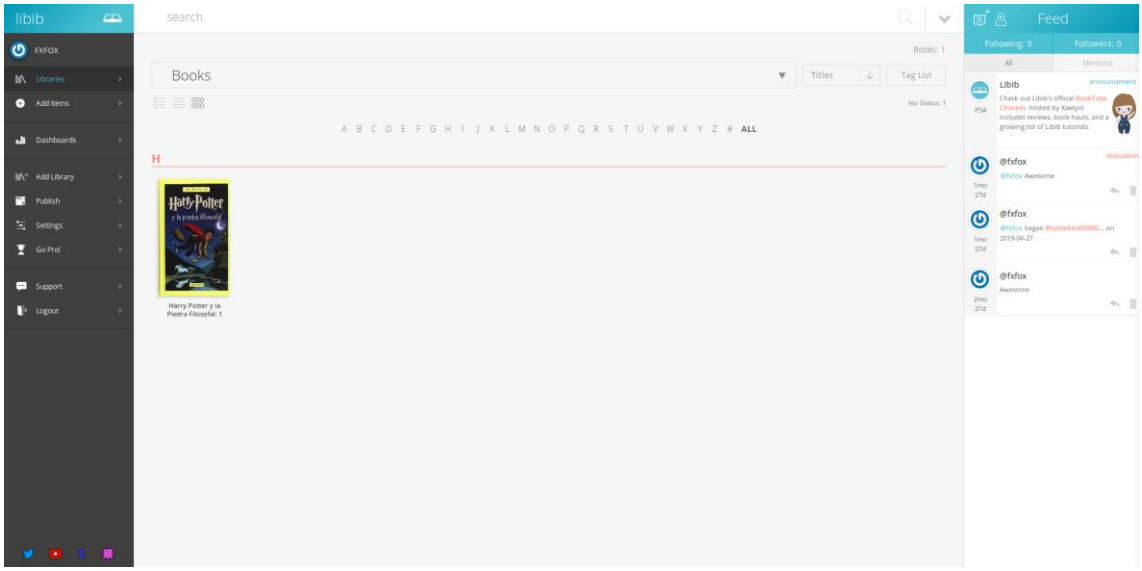

Figura 4 Página de bibliotecas en www.libib.com

Además, contamos con un sistema de seguimiento, lo que otorga una funcionalidad más social a la aplicación. Esta funcionalidad siempre está presente en el lado derecho de la pantalla.

Los usuarios cuentan con una página de perfil donde queda registrada su actividad en la página. En su página también podemos encontrar las bibliotecas que el usuario decida colocar en estado público.

A la hora de utilizar la aplicación se han encontrado posibles mejoras en la gestión de los menús, así como la diferenciación entre recursos o el perfil personal de los usuarios.

[5] Página web: www.libib.com

3. Fase de análisis

La primera etapa con la que comenzaremos el proyecto será la de análisis. En este apartado trataremos los requisitos del proyecto [3.1 Requisitos], casos de uso [3.2 Casos de uso], posibles soluciones [3.3 Identificación y análisis de soluciones posibles], metodología empleada [3.4 Metodología] y un estudio de las API que nos otorgaran la información necesaria de los recursos [3.5 Estudio de *APIs*].

3.1. Requisitos

Este apartado contiene los principales requerimientos del proyecto agrupados por tipo y ámbito de actuación. En primer lugar, se indicarán los requerimientos de tipo funcional y no funcional.

Los requisitos funcionales son aquellos que describen el comportamiento del sistema, mientras que los no funcionales describen las características de dicho comportamiento, es decir, la calidad del sistema.

3.1.1. Requisitos funcionales

Tras el primer análisis centrado en las otras aplicaciones existentes en el mercado, se presenta a continuación el listado con los requisitos mínimos funcionales que nuestro sistema debe satisfacer.

CU1: Registro

1. El sistema debe permitir al usuario registrar sus datos a través de un formulario de registro.
2. El sistema deberá de validar los datos de la petición de registro.
3. El sistema mostrará un mensaje de error en caso de que algunos de los datos proporcionados sean incorrectos o se haya producido cualquier otro tipo de error.
4. En caso de que el proceso de registro se haya realizado exitosamente, el sistema almacenará los datos de usuario en una base de datos.
5. A continuación, el sistema redirigirá al usuario a la página principal de la aplicación.

CU2: Inicio de sesión

1. El sistema debe permitir llevar a cabo el inicio de sesión mediante la introducción de los datos indicados en el registro, en particular, de su correo electrónico y la contraseña introducida.
2. En caso de producirse algún tipo de error, ya sea por credenciales incorrectas o por error del servidor, se deberá mostrar un mensaje de error al usuario.
3. En el caso de que la validación sea correcta, se redirigirá al usuario a la página principal de la aplicación.

CU3: Cerrar sesión

1. Cualquier usuario con la sesión iniciada debe ser capaz de finalizarla mediante un botón de “Cierre de sesión”.
2. El usuario debe ser capaz de cerrar sesión desde cualquier página de la aplicación.
3. Una vez cerrada la sesión, el sistema redirigirá al usuario a la página inicial de la aplicación.

CU4: Listar bibliotecas

1. Cualquier usuario con la sesión iniciada debe ser capaz de listar bibliotecas.
2. Cada usuario tendrá reservado un listado de bibliotecas que el resto de los usuarios podrá visitar.
3. Al realizar la carga de las bibliotecas, el sistema debe mostrar que el sistema no se ha quedado colgado.
4. Si se produce algún error al llevar a cabo la carga de las bibliotecas, el sistema deberá mostrar un mensaje de error.

CU5: Añadir bibliotecas

1. Un usuario con la sesión iniciada debe ser capaz de crear una biblioteca nueva dentro de su lista de biblioteca.
2. El sistema debe proporcionar un formulario donde el usuario pueda introducir los datos de la biblioteca.
3. El sistema deberá validar los datos introducidos por el usuario.
4. Si la validación ha sido satisfactoria, la nueva biblioteca se guardará en la base de datos.

5. Si se produce algún error, el sistema debe mostrar un mensaje de error.

CU6: Modificar bibliotecas

1. Un usuario con sesión iniciada debe ser capaz de modificar las bibliotecas que él haya creado.
2. El sistema debe proporcionar un formulario con los datos anteriores de la biblioteca, permitiendo al usuario realizar las modificaciones que desee.
3. El sistema deberá validar los datos introducidos y comprobar que el usuario que está modificando la biblioteca tiene permiso para ello.
4. Si todo ha ido correctamente, los datos se modificarán en la base de datos.
5. En caso de error, se deberá mostrar un mensaje que avise de ello.

CU7: Eliminar bibliotecas

1. El sistema debe proporcionar la capacidad de eliminar las bibliotecas que el usuario haya creado previamente.
2. El sistema validará si el usuario que envía la petición está autorizado. Posteriormente, eliminará dicha biblioteca.
3. En caso de error, se mostrará un mensaje que lo notifique.

CU8: Listar contenido de bibliotecas

1. Listar el contenido de una biblioteca.
2. El sistema debe mostrar una barra de carga mientras los datos son cargados.

CU9: Añadir recurso a biblioteca

1. El usuario debe poder añadir un recurso a cualquiera de sus bibliotecas.

CU10: Eliminar recuso de bibliotecas

1. El usuario debe ser capaz de eliminar cualquier recurso de sus bibliotecas.

CU11: Búsqueda de recursos

1. El sistema debe ofrecer al usuario un sistema de búsqueda por título o autor.
2. La búsqueda rápida debe ser visible en cualquier pantalla de la aplicación.
3. El sistema debe dejar filtrar el resultado por tipo de recurso.

CU12: Últimas actividades

1. El sistema debe proporcionar una lista con la actividad reciente de los usuarios o personas que un usuario sigue.

CU13: Perfiles de usuario

1. El sistema proporcionará al usuario un perfil donde podrá:
   a. visualizar sus datos básicos de usuario y editarlos
   b. visualizar sus bibliotecas, seguidores y personas seguidas
2. El usuario podrá buscar por nombre los perfiles de otros usuarios y visualizar la misma información que en punto anterior.
3. El usuario podrá seguir o dejar de seguir a otros usuarios desde sus perfiles.

CU15: Sistema de amistad

1. El sistema de amistad está basado en seguidores y seguidos.
2. El sistema debe permitir seguir y dejar de seguir a otros usuarios.

3.1.2. Requisitos no funcionales

RNF1: Recursos

1. El usuario deberá disponer de un navegador y conexión a internet.

RNF2: Seguridad

1. Para poder realizar cualquier acción en la aplicación el usuario tiene que estar autenticado.
2. La sesión de usuario se deberá cerrar cada hora para evitar dejar una sesión iniciada.
3. El sistema deberá utilizar conexión HTTPS para todas las peticiones.

RNF3: Interfaz y usabilidad

1. El usuario deberá poder realizar todos los requisitos funcionales descritos en la sección anterior en un número máximo de cinco clics, exceptuando los formularios.
2. El sistema tendrá un sistema de paginación mediante desplazamiento infinito.

RNF4: Rendimiento

1. El sistema debe de responder en menos de dos segundos.
2. El lado cliente debe ser lo más ligero posible para que la descarga inicial se haga lo más rápido posible.
3. Todas las peticiones de datos deben ser realizadas mediante *AJAX* para no ralentizar la carga inicial del sistema.

3.2. Casos de uso

A continuación, se presenta el diagrama de caso de uso que recoge todos los requisitos del apartado anterior [Figura 5].

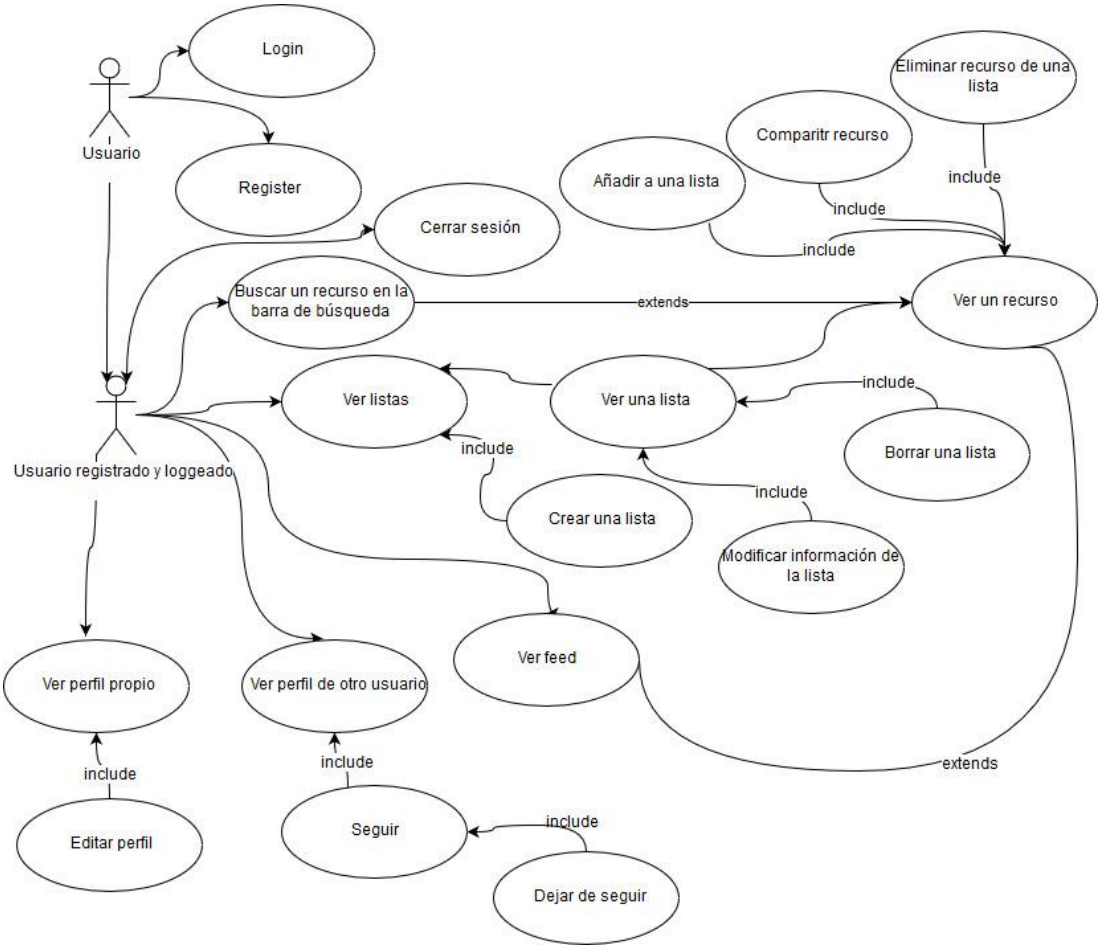

Figura 5 Casos de uso de la aplicación

3.3. Identificación y análisis de soluciones posibles

Una vez se ha realizado el trabajo de toma y análisis de requisitos, es turno de llevar a cabo un estudio de las diferentes opciones tecnológicas que ofrece el mercado a la hora

de desarrollar una aplicación, valorando así las posibles ventajas y desventajas de cada una de ellas.

La primera opción que encontraremos es el desarrollo de una aplicación web *responsiva*[6] que sea adapte a dispositivos de diferente tamaño como son los móviles, tabletas, portátiles y ordenadores de sobre mesa. Este tipo de aplicaciones se van a ejecutar sobre el navegador del dispositivo, por ello su mayor ventaja reside en la compatibilidad con prácticamente todos los dispositivos del mercado, puesto que en todos ellos encontramos un navegador. Sin embargo, el rendimiento de la aplicación es inferior a la de una aplicación nativa puesto que tenemos que realizar una carga continua de datos.

La siguiente opción que encontramos es el desarrollo de una aplicación móvil nativa. En el caso de que quisiésemos abarcar todo el mercado móvil, tendríamos que desarrollar dos aplicaciones, ya que existen dos grandes ecosistemas (Android y iOS). Sin embargo, en este caso dejaríamos a un gran número de usuarios sin servicio, pues no tendríamos soporte para ordenador.

La opción de una aplicación nativa de ordenador es parecida al caso anterior, no obstante, solo tendríamos que desarrollar una única aplicación. Dejando sin servicio en este caso a todos los usuarios de dispositivos móviles.

La ventaja de las aplicaciones nativas son su gran velocidad y posible uso sin Internet. A pesar de esta ventaja, la diferencia que encontramos no es lo suficientemente grande como para paliar la segmentación del mercado.

El desarrollo de múltiples aplicaciones, podría ser una opción adecuada. A pesar de ello en este proyecto nos encontramos un factor de tiempo que nos ayuda a decantarnos por la aplicación web. La cuál nos ofrece una experiencia de usuario aceptable englobando todo el mercado de dispositivos. Además, de poder desarrollar aplicaciones nativas posteriormente [Tabla 1].

[6] https://www.w3schools.com/html/html_responsive.asp

Tabla 1: Comparativa de los distintos enfoques del desarrollo

	Aplicación web	Aplicación nativa móvil	Aplicación nativa ordenador	Aplicación nativa móvil y ordenador	Aplicación web y aplicación nativa móvil
Experiencia de usuario en móvil	Media	Alta	-	Alta	Alta
Experiencia de usuario en ordenador	Media-Alta	-	Alta	Alta	Media-Alta
Tiempo de desarrollo	Medio	Alta	Medio	Muy alto	Muy alto

3.4. Metodología

Una vez finalizada la fase de análisis, da comiendo la parte de diseño y posterior implementación. Para llevar un control del desarrollo utilizaremos metodologías ágiles.

El proyecto está dividido en *sprints*. Cada *sprint* tiene una duración de diez días, es decir, dos semanas de lunes a viernes. Cada día de trabajo se traduce en seis horas de trabajo aproximadamente.

Los distintos casos de uso que se han obtenido durante la fase de análisis serán divididos en tareas más pequeñas, y a su vez, se estimará el tiempo que lleva desarrollar dicha tarea. Estas tareas se almacenarán en la sección de casos pendientes o *backlog*.

Al principio de cada *sprint* se decidirán las tareas que se van a desarrollar. La prioridad de las tareas vendrá dada por la necesidad de una cierta funcionalidad. En caso de no tener una prioridad clara, se seleccionarán tareas relacionadas entre sí.

El proyecto está dividido en cinco *sprints* empezando por el *sprint* cero. El *sprint* cero corresponde a la fase de requisitos y diseño. Del *sprint* uno hasta la mitad del cuarto nos

encontramos con la fase de implementación. Lo que queda de *sprint* cuatro hasta el final del *sprint* cinco estará destinado al despliegue de la aplicación y resolución de errores.

Todo el seguimiento se ha llevado a cabo a través de la herramienta de proyectos de *Github*. Para cada *sprint* se ha creado un tablero de *Kaban* donde las tareas han ido pasando entre los distintos "Para hacer", "En progreso" y "Terminado" [Figura 6 Tablero Kaban del Sprint 1].

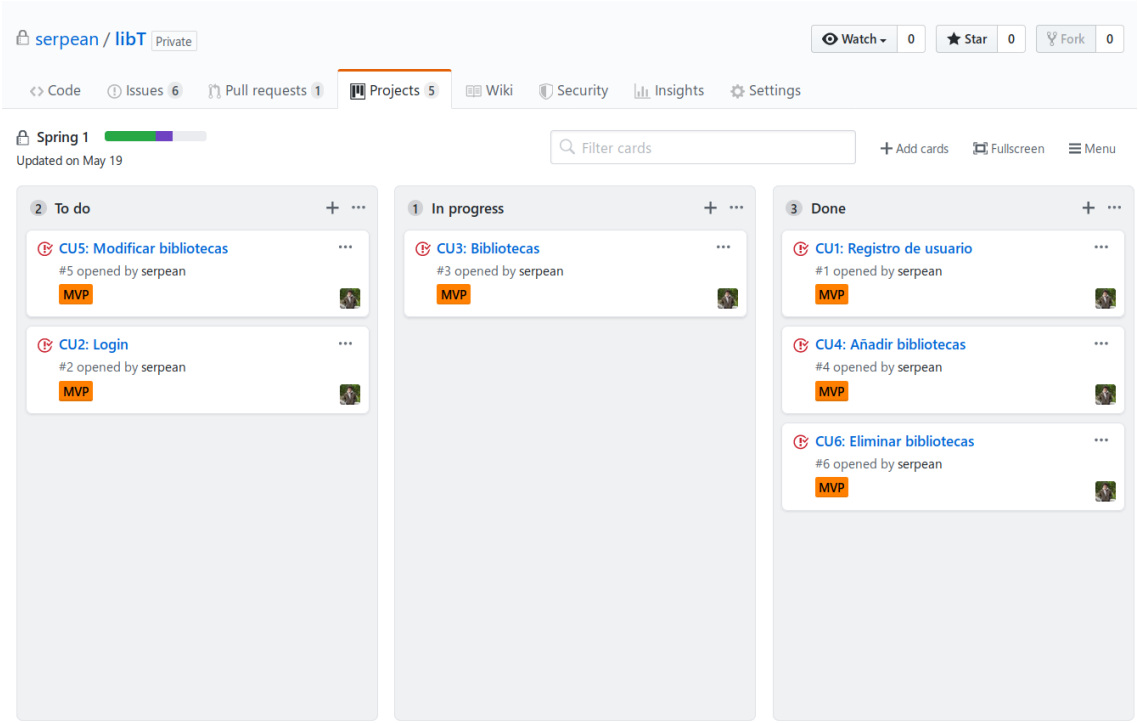

Figura 6 Tablero Kaban del Sprint 1

3.5. Estudio de *APIs*

Como hemos tratado ya en el apartado de toma de requisitos, existen tres tipos de recursos en este proyecto, libros, películas y videojuegos. Para obtener los datos necesarios para el correcto funcionamiento de la aplicación se han utilizado dos *APIs* diferentes.

- *Google Books API*[7]: Usada en la búsqueda de libros, es un servicio *RESTful* sin coste alguno y con muchas opciones configurables. Las opciones que utilizaremos en el proyecto serán la búsqueda de libros por peticiones sencillas y el visor de información.

[7] https://developers.google.com/books/docs/v1/using

- *OMDb API*[8]: Usada en la búsqueda de películas y videojuegos. Es un servicio web *RESTful* como el anterior, sin embargo, más limitado que el anterior en lo que respecta a configuración de peticiones. Para poder realizar peticiones esta *API* tendremos que adjuntar una clave secreta o *key* en cada petición.

Las principales diferencias entre las dos *APIs* anteriores además de su contenido serán la cantidad de parámetros configuradores a la hora de realizar una petición de búsqueda, el volumen de información proporcionado (mucho mayor en el caso de la *Google Books API*) y la documentación, sencilla en ambos casos, sin embargo, en *OMDb API* está mucho más simplificada y podemos probarla directamente en la página web de la documentación.

En ambos casos podemos encontrar una versión gratuita de la aplicación, aunque en la versión de *OMDb* tendremos que contratar un plan de pago si el volumen de peticiones aumenta. Se han implementado algunas soluciones que se explicaran más adelante en el capítulo 4.3.2.2 Lógica de la API de búsqueda.

[8] http://www.omdbapi.com/

4. Fase de diseño

En esta fase se lleva a cabo el diseño completo de la aplicación. En ella se tratan temas como la elección de la arquitectura, guías de estilos y tecnologías utilizadas. Los puntos que se tratarán en este capítulo serán 4.1 Arquitectura, 4.2 Tecnologías y 4.3 Diseño detallado.

4.1. Arquitectura

A continuación, entramos en la arquitectura de nuestra aplicación. Este punto está dedicado a la organización las diferentes piezas de software utilizadas en este trabajo.

La arquitectura de la aplicación estará basada en microservicios. Esto quiere decir que la lógica de nuestra aplicación está formada por un conjunto de servicios más pequeños comunicados por mecanismos ligeros. Cada uno de estos servicios llevará a cabo una funcionalidad distinta dentro de la aplicación.

Dicha arquitectura podemos observarla en la Figura 7. En ella vemos cómo está formada por una aplicación web en el lado del cliente y tres API REST en el lado del servidor. Todas estas aplicaciones se comunican mediante peticiones HTTP.

La separación del lado servidor en dos APIs diferentes tiene como objetivo principal aliviar la carga que puedan crear los usuarios al buscar recursos continuamente en las diferentes APIs externas de información, ya que será una de las actividades más frecuentes de la aplicación. Más adelante se profundizará en el mecanismo de autenticación usado en esta arquitectura.

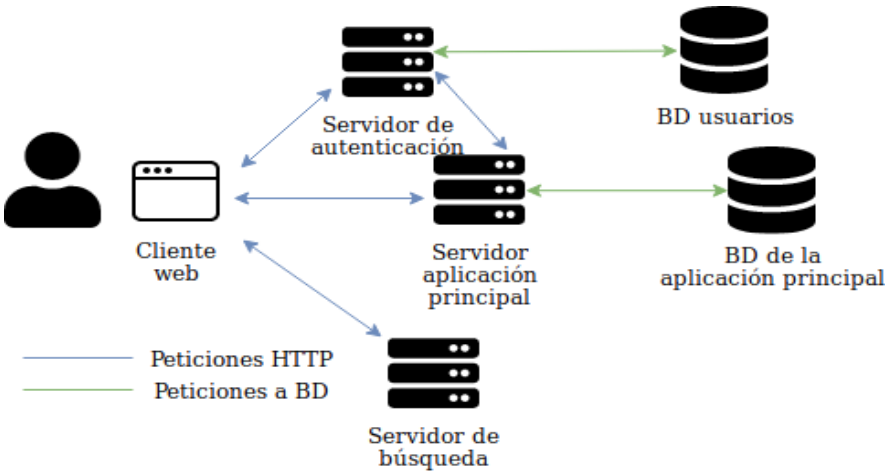

Figura 7 Arquitectura básica de la aplicación

4.2. Tecnologías

En este apartado hablaremos de las diferentes tecnologías y herramientas utilizadas a lo largo del proyecto.

4.2.1. *Balsamiq*[9]

Balsamiq es una potente herramienta de maquetado, la cual nos permite reproducir en el ordenador la experiencia de esbozar modelos en una libreta o pizarra.

Esta herramienta permite centrarnos en la ubicación de los elementos en nuestra aplicación gracias a su gran catálogo de iconos y figuras prediseñados. Además, nos permite cambiar rápidamente entre sus dos modos *Sketching* y *Wireframe*.

4.2.2. *Draw.io*[10]

Draw.io es una aplicación *web online* para realizar diagramas. Nos hemos apoyado en ella, sobre todo, para realizar los diagramas de arquitectura mostrados en la memoria, que han servido de ayuda a lo largo de todo el desarrollo del proyecto.

4.2.3. *Javascript*

Javascript es un lenguaje interpretado de alto nivel en el cual se basan el resto de las tecnologías de programación de este proyecto. El motivo principal para utilizar *Javascript* es su versatilidad, puesto que lo podemos utilizar como lenguaje en todas las aplicaciones que vamos a desarrollar para este proyecto.

Utilizar siempre el mismo lenguaje permite al desarrollador evitar cambios de contexto entre lenguajes, es decir, posibles errores de sintaxis producidos al cambiar entre dos o más lenguajes en un corto periodo de tiempo, pudiendo centrarse únicamente en el desarrollo de la lógica de la aplicación.

4.2.4. *React JS*[11], *Redux*[12] y *React Router*[13]

React JS ha sido la tecnología elegida para desarrollar la parte cliente de nuestra aplicación web. *React JS* o simplemente *React* es una biblioteca diseñada por Facebook para desarrollar interfaces gráficas.

[9] balsamiq.com
[10] www.draw.io
[11] reactjs.org
[12] redux.js.org
[13] reacttraining.com

Con la ayuda de *React* podemos crear aplicaciones de una página única (Single Page Application, *SPA*), sin embargo, llevar a cabo ciertas tareas, como el enrutamiento o el manejo del estado, en *React* puro pueden ser muy tedioso, sobretodo cuando la aplicación aumenta de tamaño.

Por ello, se ha decidido acompañar a *React* con otras dos bibliotecas como son *Redux* y *React Router*.

4.2.4.1. *Redux*

Redux cuenta con las siguientes características:

- Predecible. El comportamiento de nuestra aplicación será consistente, a pesar de que sea ejecutado en distintos entornos además de ser fácil a la hora de probar.
- Centralizado: todo el estado de la aplicación se concentrará en un único lugar, controlando la persistencia del estado.
- Depurable: Gracias a otra herramienta llamada *Redux Dev Tools* podemos seguir la traza de cualquier cambio dentro de la aplicación.
- Flexible: Redux trabaja con cualquier capa de interfaz de usuario además de tener un gran ecosistema de *add-ons* (como *Thunk*[14]).

4.2.4.2. *React Router*

Para completar nuestro ecosistema en el lado cliente, utilizaremos *React Router*. De nuevo se trata de una biblioteca que mejora funcionalidades de *React*. En este caso nos ayudará a mantener un sistema de enrutamiento dinámico dependiendo de si el usuario ha iniciado o no sesión.

4.2.5. *Node JS*[15]

Node JS es un entorno de ejecución para *Javasciprt* fuera del navegador. Esto quiere decir que es la tecnología que nos permite utilizar *Javascirpt* para desarrollar nuestro servidor.

[14] github.com/reduxjs/redux-thunk
[15] https://nodejs.org

Junto a *Node* utilizaremos *Express*[16], que es un *framework* el cual nos facilita la tarea de crear un servidor HTTP.

Otras bibliotecas que cabe destacar en el desarrollo son *bcryptjs*[17], utilizada para encriptar las contraseñas; *jsonwebtoken*[18], quien genera los *tokens* de sesión para los distintos usuarios; *morgan*[19], una biblioteca que registra las peticiones HTTP que recibe nuestro servidor o *multer*[20], que nos ayuda con la gestión de subida de imágenes al servidor.

Por defecto, al instalar *Node*, instalamos un gestor de paquetes llamado *NPM* (*Node Package Manager*). Al inicio del proyecto se utilizó este gestor de paquetes para instalar todas las dependencias de la aplicación. Posteriormente, se implementó el uso de *YARN* como gestor de paquetes, del cual se habla más en profundidad en el capítulo cinco.

4.2.6. *Mongo DB*[21] y *MongoDB Compass Community*

MongoDB es una base de datos orientada a documentos, multiplataforma y de código abierto. Se trata de una base de datos *NoSQL*, usando el formato *JSON* dentro de sus documentos.

La elección frente a una base de datos *SQL* o relacional es un tema de compatibilidad. *MongoDB* es una de las bases de datos más usadas en proyectos de *Node JS*. Entre sus principales características tenemos indexación de índices primarios y secundarios, replicación y balanceo de carga. Aunque podamos usar una base de datos relacional, estaríamos desaprovechando el potencial que encontramos juntando *Node*, *Express* y *MongoDB*.

A lo largo del proyecto se han utilizado diferentes servicios en la nube que ofrecen una base de datos Mongo de manera gratuita. Estos servicios son *Mongo Atlas, Cosmos DB* y *mLab*.

En este proyecto encontraremos a *Mongoose*[22] un *ORM* de *MongoDB*, el cual proporciona la validación de datos automáticos gracias a la declaración de esquemas.

Como herramienta adicional o espacio de trabajo para *MogoDB*, encontramos *MongoDB Community*. Aunque tengan nombres casi idénticos no debemos

[16] expressjs.com
[17] www.npmjs.com/package/bcrypt
[18] https://github.com/auth0/node-jsonwebtoken
[19] https://github.com/expressjs/morgan
[20] https://github.com/expressjs/multer
[21] www.mongodb.com
[22] https://github.com/Automattic/mongoose

confundirlas. *MongoDB Community* es una herramienta que utilizaremos para consultar el estado actual de nuestra base de datos.

Nos permite visualizar de una manera rápida y sencilla las colecciones y documentos que la forman. Además, podemos utilizarla para realizar consultas directas sobre la base de datos.

4.2.7. *Postman*[23]

Postman es un entorno de desarrollo de *APIs* el cual nos ofrece una gran variedad de herramientas. En nuestro proyecto lo utilizaremos para probar, de una manera más rápida y sencilla, todos los puntos de entrada de los servicios. Además, contamos con un sistema de historial, muy útil para no perder el tiempo reescribiendo una y otra vez la misma petición.

4.2.8. *Visual Studio Code*[24]

Visual Stuido Code es un editor de código multiplataforma desarrollado por Microsoft. Por defecto incluye soporte para *debugging*, *git*, marcado de sintaxis y autocompletado entre otras.

Sin embargo, el auténtico potencial de *Visual Studio Code* son las extensiones, las cuales permiten aumentar las funcionalidades de este editor de texto conviertiéndolo en un IDE de programación.

Algunas extensiones instaladas para el proyecto han sido *GitLens*, *Git*, *Azure Account*, *Azure CosmosDB*, *Azure Storage* y *Prettier*.

4.2.9. *Git, Github y Gitkraken*[25]

Estas tres herramientas están muy relacionadas entre sí. La principal de ellas es *Git*, un sistema de control de versiones que permite llevar a cabo un registro de todos los ficheros que conforman el proyecto y los cambios que han ido surgiendo a lo largo del desarrollo.

Junto a *Git*, contamos con *Github*, un sistema de hosting para nuestro repositorio *Git*. Además de almacenar nuestro proyecto, utilizaremos otras herramientas que nos ofrece

[23] https://www.getpostman.com/
[24] https://code.visualstudio.com/
[25] https://www.gitkraken.com/

como *Github Projects* [Figura 8], donde hemos llevado a cabo el seguimiento del proyecto a través de *sprints*.

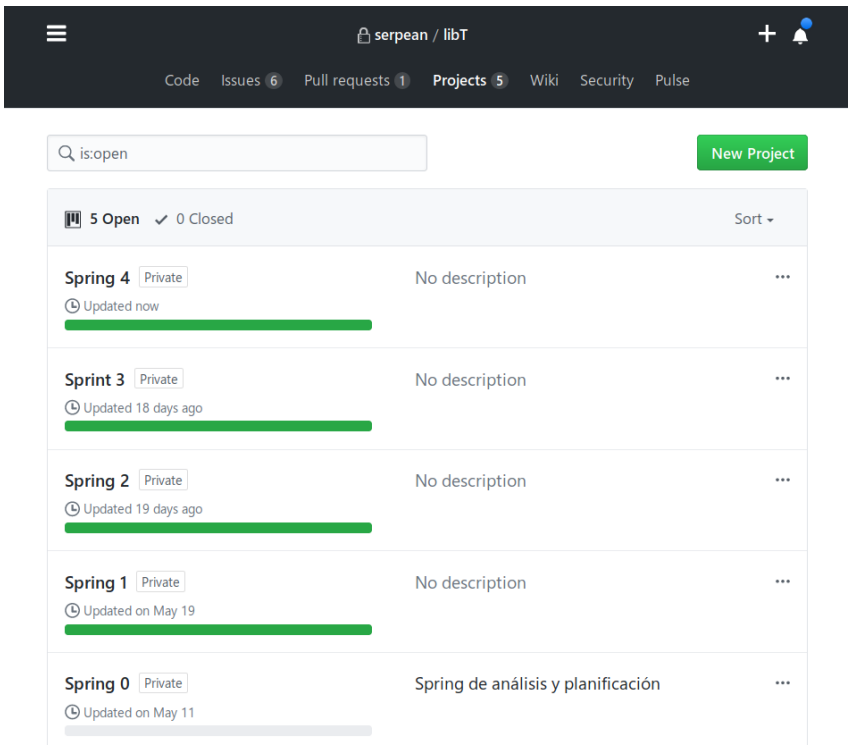

Figura 8 Herramienta de proyectos en Github

Por último, tenemos *Gitkraken*, una interfaz gráfica para *Git*, la cual nos facilita la tarea de trabajar con *Git* [Figura 9].

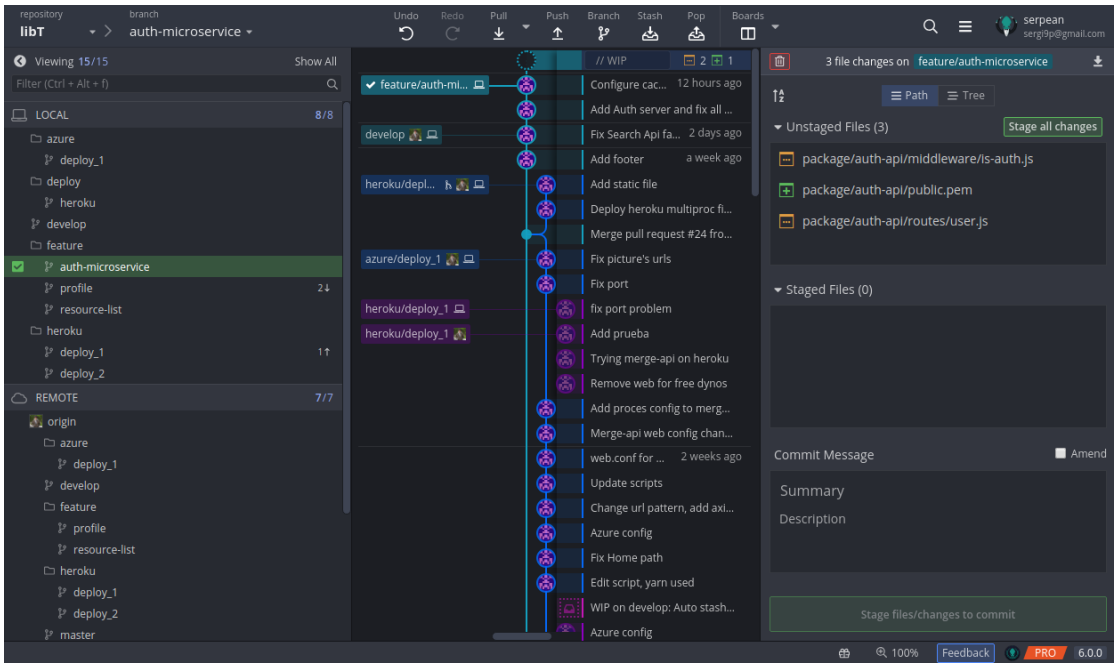

Figura 9 Ilustración de la interfaz gráfica de Gitkraken

4.3. Diseño detallado

Una vez tenemos clara la arquitectura principal de la aplicación y tecnologías utilizadas, damos paso a una descripción más detallada del diseño. Para llevar a cabo dicha explicación, dividiremos esta sección en diseño de la base de datos [4.3.1 Diseño del Modelo Conceptual], diseño de la lógica de la aplicación [4.3.2 Diseño de la Lógica] y diseño de la interfaz de usuario [4.3.3 Diseño de la Interfaz de Usuario].

4.3.1. Diseño del Modelo Conceptual

En este apartado se muestran los aspectos relacionados con la base de datos del servidor. Los diagramas han sido creados en a herramienta *Draw.io*.

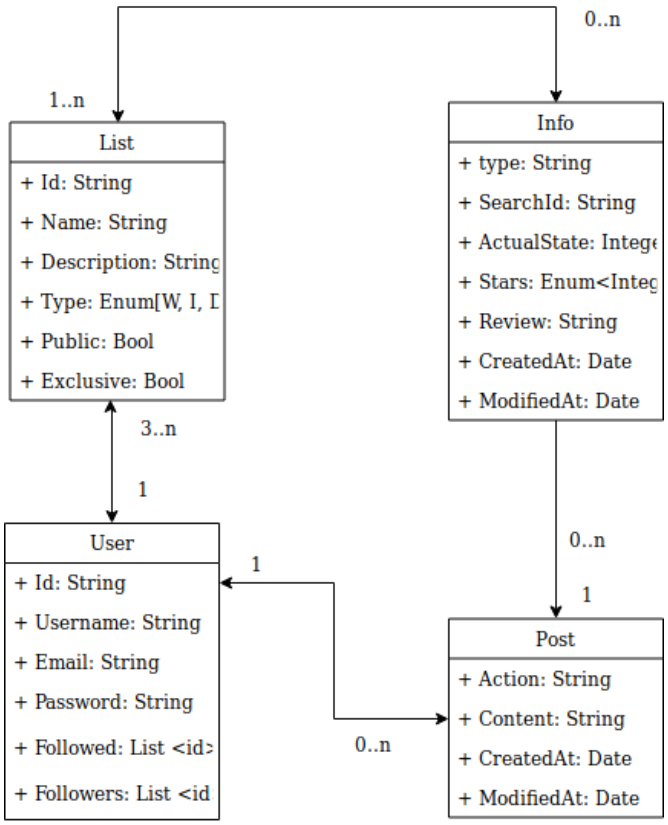

Figura 10 Esquema final de la base de datos

4.3.2. Diseño de la Lógica

En este apartado hablaremos sobre las distintas decisiones acerca de la aplicación. Este apartado está dividido de manera que cada aplicación cuenta con su propio subapartado.

4.3.2.1. Lógica del lado cliente

Cuando trabajamos en una aplicación web, tenemos datos dinámicos los cuales son necesarios obtenerlos de un servidor externo al dispositivo donde se está ejecutando el navegador.

Para llevar a cabo este intercambio de datos, el navegador envía una petición y el servidor le responde. Este intercambio se debe producir cada ver que necesitamos cualquier tipo de información, sin embargo, como desarrolladores tenemos diferentes maneras de manejar este comportamiento.

4.3.2.1.1. Páginas dinámicas

La página HTML es creada de manera dinámica [Figura 11], en el servidor con la ayuda de un lenguaje de programación del lado del servidor y normalmente con la ayuda de un motor de plantillas o t*emplating engine*.

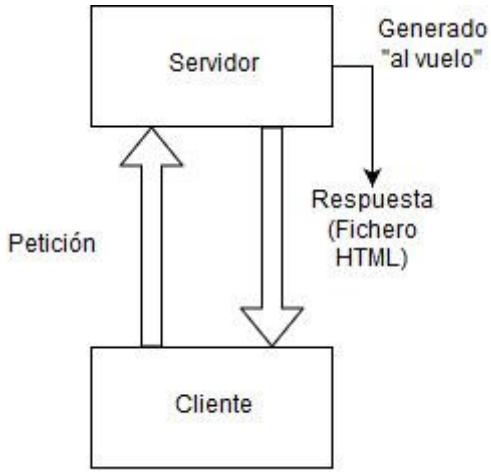

Figura 11 Esquema aplicación web dinámica

Este diseño ha sido el más utilizado en los últimos años, y en la actualidad, sigue siendo el más usado a la hora de construir una aplicación web. Su principal ventaja es el posicionamiento en motores de búsquedas, pues el servidor envía la página completa al finalizar la carga inicial. Otra ventaja podría ser el rendimiento necesario en el lado cliente, puesto que solo sería necesario un navegador web para llevar a cabo el renderizado de la página.

Su principal desventaja es la de tener la necesidad de recargar la página cada vez que se realice un cambio, por ejemplo, si queremos cambiar cierto texto en un párrafo de la aplicación dejando el resto de manera idéntica, tendríamos que reiniciar la página.

Con el paso de los años y antes de saltar a la siguiente solución, estas páginas dinámicas podían tratar esos pequeños cambios a través de *AJAX* [9]. Con la ayuda de *Javascript* y la aparición de *Jquery*, no era necesario volver a recargar la página cada vez.

4.3.2.1.2. SPA (*Single Page Application*)

En el otro extremo encontramos *SPA (Single Page Appication)* o aplicaciones de una única página [Figura 12] si lo traducimos al castellano. En este tipo de páginas el cliente solo recibe un único fichero HTML simple al inicio de la conexión, sin importar la URL, y es el propio navegador quien se dedica a generar el HTML con el contenido final de la página.

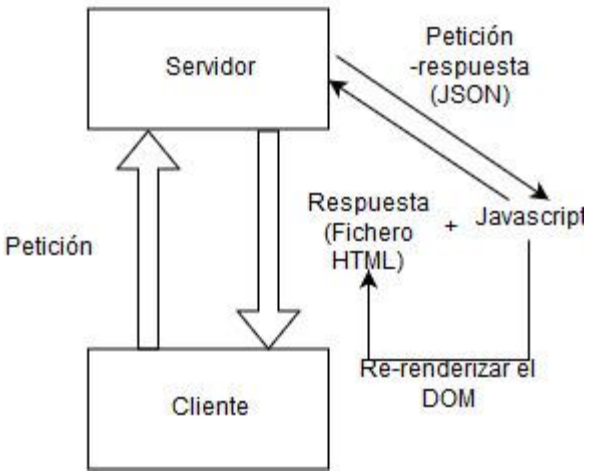

Figura 12 Esquema aplicación SPA (Single Page Application)

En esta implementación sigue existiendo un servidor, al cual mandamos todas consultas a la hora de obtener los datos. Al contrario que en la anterior, el usuario solo tendrá que esperar una vez a que cargue la aplicación entera, dando un comportamiento más propio de una aplicación móvil o nativa.

Esto no significa que la aplicación no necesite más datos, probablemente tenga que realizar más peticiones. Sin embargo, en lugar de recargar la página, mostraremos un elemento en la interfaz gráfica que permita saber al usuario que los datos están tardando en cargar (como una barra de carga). Las peticiones serán realizadas, como ya comentamos antes vía *AJAX*.

Como tenemos que realizar la renderización en el lado cliente, necesitaremos mucho código *Javascirpt*. Es por ello, que normalmente necesitaremos la ayuda de un *framework* o biblioteca para llevar a cabo esta tarea. En este proyecto, y como ya se adelantó en la sección 4.2 Tecnologías, *React* será la biblioteca elegida para este proyecto. Otras opciones pueden ser *Angular*, *Vue* o *Svelte*.

Por un lado, como principal ventaja tenemos la experiencia de usuario, la carga de los elementos iniciales es casi instantánea y posteriormente el usuario nunca ve que la pantalla se quede en blanco. Por otro lado, encontramos una desventaja en relación con el SEO de la página. Al enviar solo un fragmento de la página y el resto ser descargado posteriormente, los buscadores tienen problema ya que no son capaces de indexar esa información final que nos proporciona la aplicación.

La solución al problema anterior pasa por la renderización en el lado del servidor o *server render*. Este elemento no se llegará a implementar por parte de la parte cliente de la aplicación.

4.3.2.2. Lógica de la API de búsqueda

La API de búsqueda surge como solución al manejo de una de las tareas más utilizadas en la aplicación, la búsqueda, combinado el uso de patrones de diseño dentro de *Javascript*. Al realizar esta división, aligeramos la carga del servidor que maneja a los usuario y listas de recursos.

Esta aplicación tiene varios objetivos a cumplir:

1. Disminuir el número de peticiones que recibe el servidor principal de la aplicación.
2. Ofrecer una interfaz única a la hora de realizar las peticiones a las diferentes *APIs*.
3. Cachear las respuestas de peticiones recientes, puesto que, con alta probabilidad los usuarios puedan volver a realizar una búsqueda igual.

4.3.2.3. Lógica del servicio de autenticación

Al estar en una arquitectura de microservicios, nos encontramos que el servicio de autenticación se encuentra separado del resto de componentes de la aplicación. En este apartado se explica las diferentes partes de la arquitectura necesarias para el correcto funcionamiento.

El primer punto que vamos a llevar a cabo es la manera en la que nuestros usuarios se van a autenticar. Para una primera fase de desarrollo, sólo se utilizará el correo electrónico y contraseña para llevar a cabo el proceso. Posteriormente, el servicio podría utilizar otros sistemas de autenticación mediante *Google* o *Facebook*, facilitando y agilizando el proceso a los usuarios.

En una aplicación monolítica, de las cuales ya hemos hablado anteriormente, es capaz de mantener un estado en las peticiones a través de las sesiones. Dicho estado, es visible por toda la aplicación.

Este hecho, es justo el comportamiento contrario al que tenemos en nuestra arquitectura de microservicios. Cada petición necesita ser autenticada, sin embargo, nuestro servicio de autenticación está separado del resto.

En una primera aproximación [Figura 13], podríamos realizar una petición al servicio de autenticación cada vez que una petición fuese realizada a un microservicio. Este mecanismo, aunque implementado por muchos sistemas, supone una gran carga para el

servidor de autenticación. No es la implementación óptima, puesto que cuando hablamos de microservicios la latencia y los fallos del sistema son un riesgo añadido.

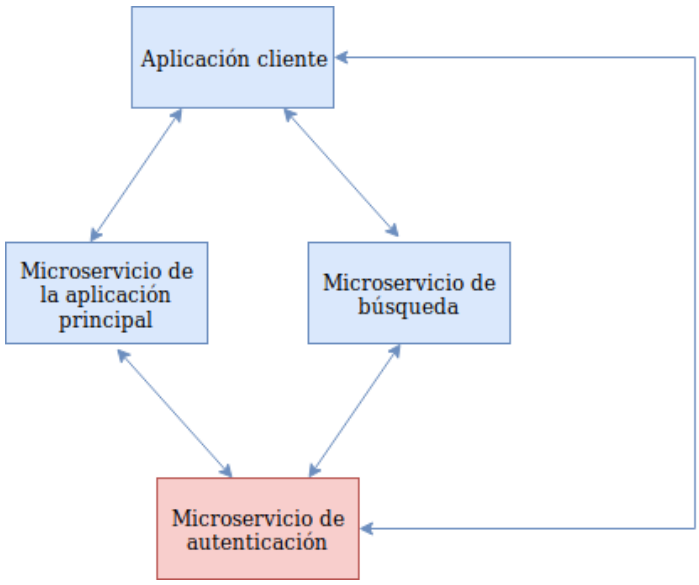

Figura 13 Arquitectura de microservicios con autenticación centralizada

La solución pasa por el uso de *JSON Web Tokens,* o *JWT* para acortar. Según el RFC 7519[26], *JWT* se define como una manera compacta y autocontenida de transmitir información de manera segura entre sus partes con el formato *JSON*. Esto es posible ya que los datos van firmados digitalmente, haciendo que la información pueda ser verificada y de confianza. Los *JWTs* pueden ir firmados usando una clave secreta (con el algoritmo HMAC) o con un par de claves pública/privada usando *RSA* o *ECDSA*.

Antes lo llamábamos compacto, esto se debe a que los JWTs se dividen en tres únicas partes (cabecera, cuerpo y firma) separadas por puntos. Cada una de sus partes está codificada en *Base64Url*. Un ejemplo de su formato podría ser el siguiente:

xxxxxxxxx.yyyyyyyyy.zzzzzzzzz

Hablando un poco más sobre su estructura, la cabecera normalmente consiste en dos partes: el tipo del token, que será *JWT* y el algoritmo utilizado para llevar a cabo la firma, puede ser HMAC, SHA256 o RSA.

Después nos encontramos con el cuerpo o *payload* del *token*. Aquí encontramos toda la información que queremos firmar, en nuestro caso será toda la información relacionada con el usuario. Un importante dato que destacar, es que la información en el

[26] https://tools.ietf.org/html/rfc7519

cuerpo del mensaje está protegida ante la manipulación, pero sigue siendo legible por cualquiera. Por ello, no se debe poner información importante, por ejemplo, contraseñas, en la cabecera o cuerpo del mensaje, a excepción que dicha información esté encriptada.

Por último, nos encontramos la firma del mensaje. Para crearla, cogeremos la cabecera y el cuerpo codificados, más una clave secreta y el algoritmo de cifrado especificado en la cabecera.

Como resultado de ejemplo a todo el proceso anterior, obtendríamos un *token* muy similar al siguiente:

eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.
EyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

El último paso, antes de pasar a hablar sobre la arquitectura de microservicios, es la manera de usar este *token* en nuestra aplicación. Para ello, debemos enviar nuestro *token* cada vez que queramos acceder a una ruta o recurso protegido.

Normalmente, utilizaremos la cabecera *Authorization* usando el esquema *Bearer*. Por lo tanto, el resultado final sería:

Authorization: Bearer <token>

Antes hemos hablado de dos tipos de cifrado, clave secreta o simétrico y clave pública/privada o asimétrico. Usando claves simétricas podría ser suficiente si confiásemos en todos los microservicios. Sin embargo, si alguno viese comprometida su seguridad y la clave fuese descubierta, un atacante malicioso podría hacerse pasar por cualquier usuario, ya que podría generar su *JWT* a partir de toda esta información.

Es por ello por lo que el uso de clave asimétrica es más adecuado para esta situación, pues si alguno de nuestros servicios se ve comprometido, el servidor de autenticación no se vería afectado.

Ahora que ya sabemos qué son, cómo utilizar y proteger los *JWT,* podemos explicar de manera más clara la siguiente parte de nuestra arquitectura de autenticación. Para ello nos apoyaremos en la siguiente imagen [Figura 14]. En ella observamos como cada microservicio es capaz de comprobar el token. Gracias a este mecanismo, ya no es necesario sobrecargar el servicio de autenticación cada vez que se quiera autorizar una petición.

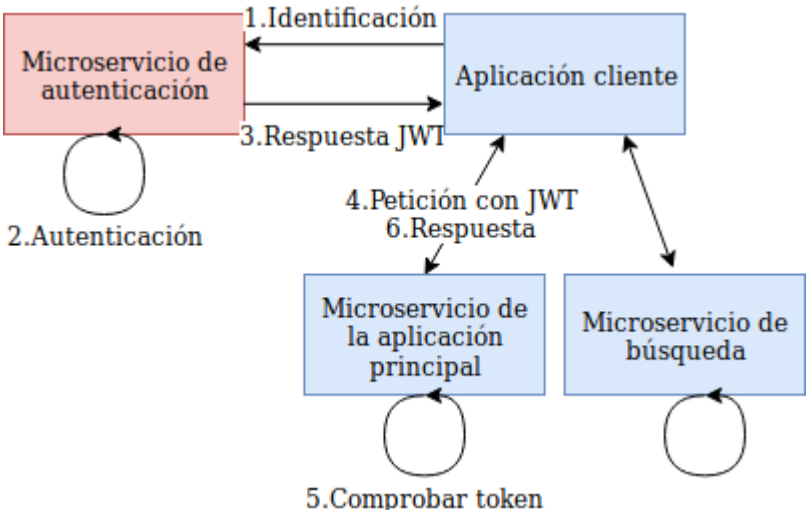

Figura 14 Arquitectura de microservicios con autenticación descentralizada

Además, si algún microservicio quisiese obtener información de otro podría realizar una petición adjuntando el token que ha recibido. En nuestro caso [Figura 15], el servicio principal necesita cierto datos de usuario antes de devolver la respuesta. Para ello, se los pide al servidor de autenticación, donde, además guardamos toda la información relacionada con los usuarios como foto de perfil, nombre, etc.

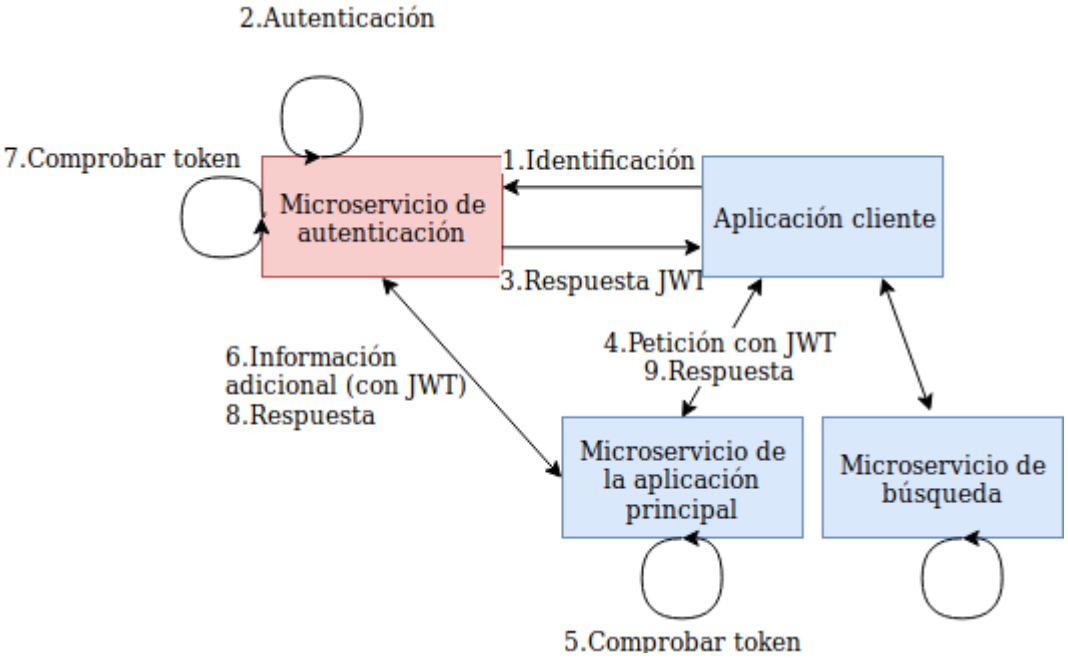

Figura 15 Petición con más de un microservicio involucrado

4.3.3. Diseño de la Interfaz de Usuario

La interfaz que se ha diseñado pretende cumplir todos los requisitos enumerados en la fase de análisis manteniendo una estética simple e intuitiva para el usuario.

Para desempeñar la tarea de diseño se ha utilizado *Balsamiq* como herramienta de maquetado. Todas las interfaces diseñadas muestran el resultado en una pantalla de ordenador, sin embargo, todas tienen la capacidad de adquirir un formato más adecuado para los navegadores de dispositivos móviles.

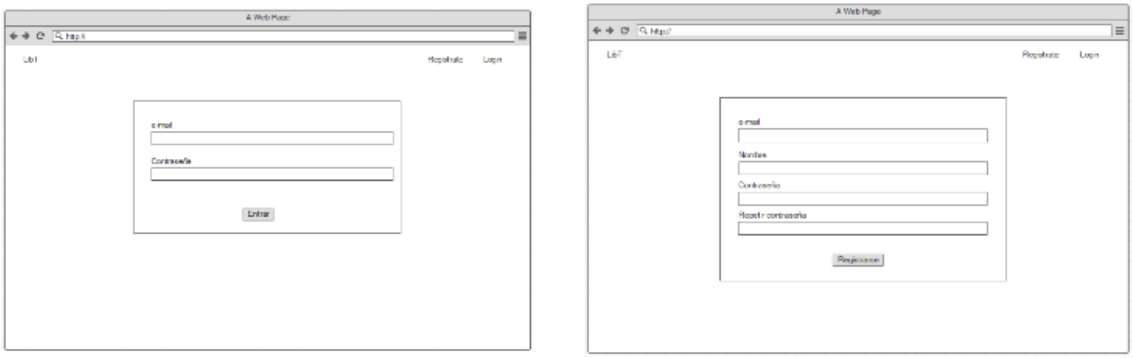

Figura 16 Mockup de la identificación y registro de usuario

Las primeras interfaces que se diseñaron fueron la de registro e inicio de sesión [Figura 16]. Ambos diseños son muy simples, pues son dos formularios embebidos en un rectángulo.

La primera pantalla que veremos una vez estemos registrados y con la sesión iniciada es la pantalla de inicio cuyo objetivo principal es actuar de centro de notificaciones [Figura 17] para nuestros usuarios, es decir, mostrar las actividades más recientes de los usuarios a los que siguen. Además, como objetivo secundario desde esta pantalla se proporciona acceso directo a ciertas funcionalidades como la creación de una nueva biblioteca, visionado de los seguidores y seguidos o actualización de los recursos que estamos consumiendo actualmente.

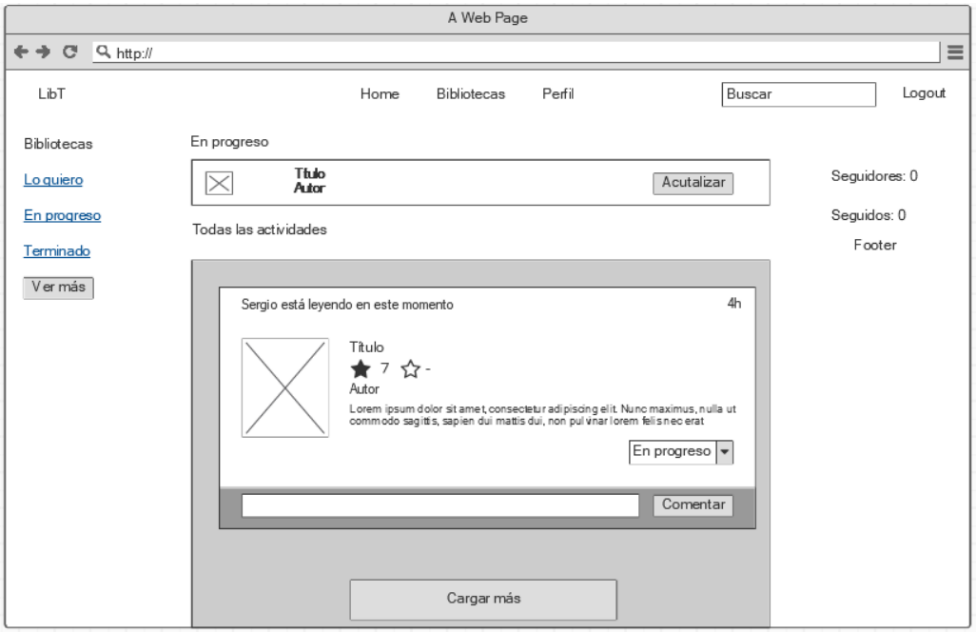

Figura 17 Mockup página principal de la aplicación

Una de las partes más importantes de la aplicación es el diseño del apartado de bibliotecas [Figura 18], puesto que es el objetivo principal de la aplicación.

En este apartado se ha decidido que el usuario pudiese ver sus librerías en todo momento, para ello se reutilizará el mismo menú que se muestra en la página de inicio. Los recursos están ordenados en una lista principal. Cada uno de ellos contará con al menos su título, el nombre de sus autores y una pequeña descripción.

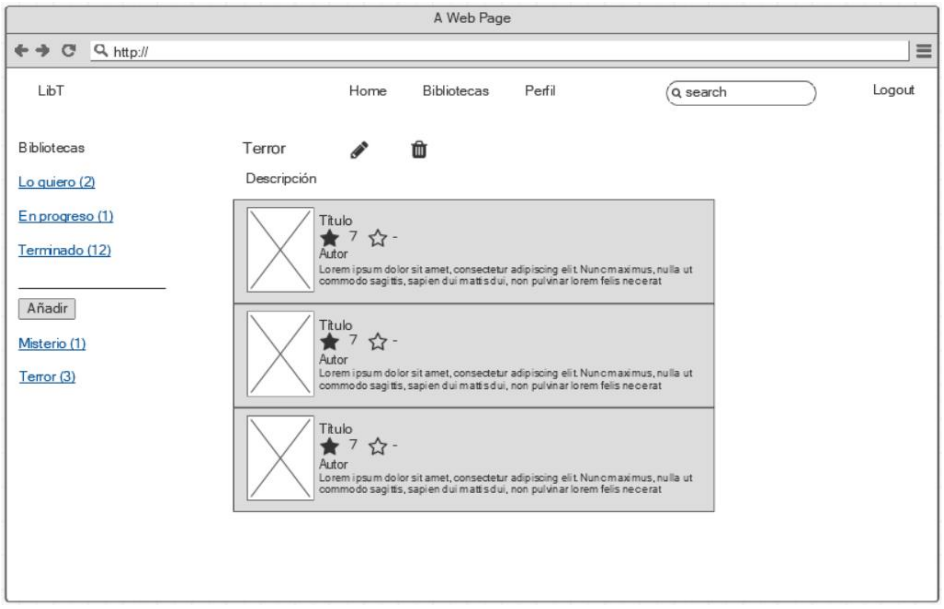

Figura 18 Mockup de la página bibliotecas

Para que los usuarios puedan añadir o modificar bibliotecas, se utilizará una ventana modal [Figura 19] donde introducirán la información de los campos requeridos (nombre y descripción) de la biblioteca.

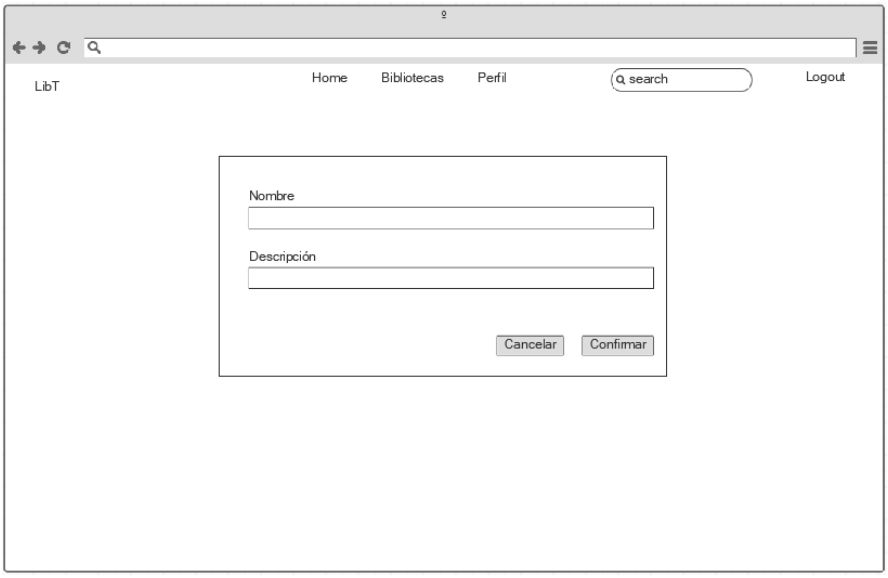

Figura 19 Mockup ventana modal para añadir bibliotecas

Para que los usuarios encuentren el recurso que buscan, se ha creado una página de búsqueda además de un atajo de búsqueda en la barra de navegación. En esta página [*Figura 20*] se podrán ver todos los resultados disponibles de la búsqueda junto a un pequeño filtro que nos permitirá buscar por un único tipo de recurso.

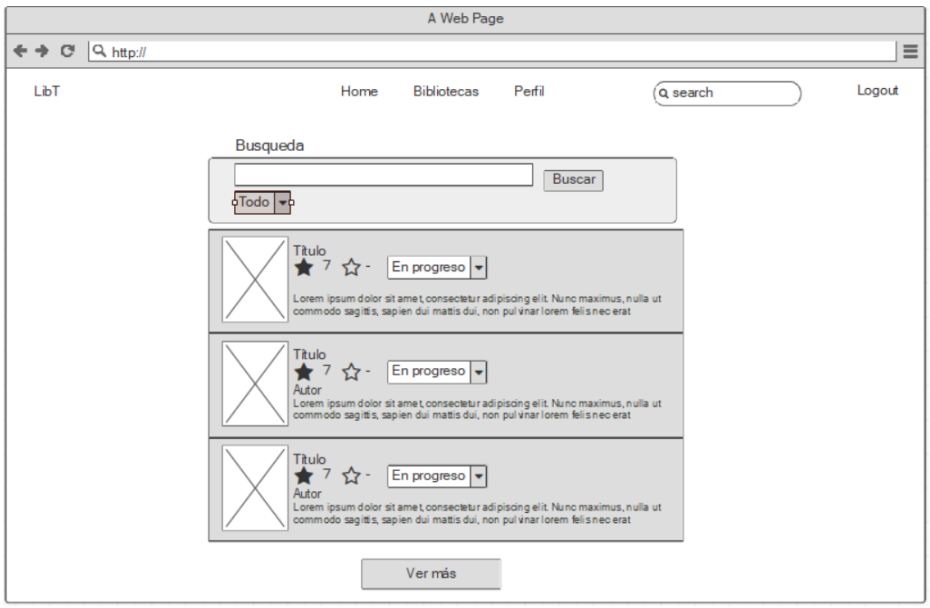

Figura 20 Mockup página de búsqueda

Para terminar de hablar sobre los recursos, nos encontramos con la página por recurso [Figura 21], cuya principal función es la de mostrar toda la información que se disponga del recurso. En el boceto que vemos a continuación se muestra la información básica.

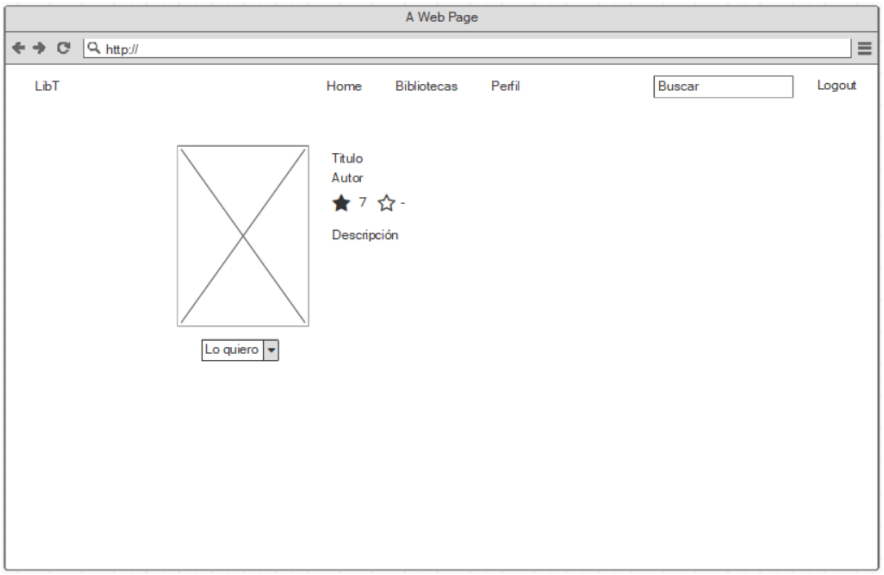

Figura 21 Mockup página de recurso

Por último, hablaremos de la página de perfil [Figura 22], en ella encontramos varios elementos. En primer lugar, la información básica del perfil de usuario, es decir, foto de perfil, nombre y biografía. En el caso de que el perfil visitado coincida con el usuario de la sesión actual, veremos la opción de editar esta información básica. En el caso contrario, esto es, cuando estamos visitando el perfil de otro usuario nos ofrecerá la opción de seguir o dejar de seguir a dicho usuario.

En segundo lugar, encontramos un menú de pestañas que nos permite gestionar toda la información relacionada con los usuarios, esto es, bibliotecas, personas a las que sigue el usuario y personas que le siguen a este usuario.

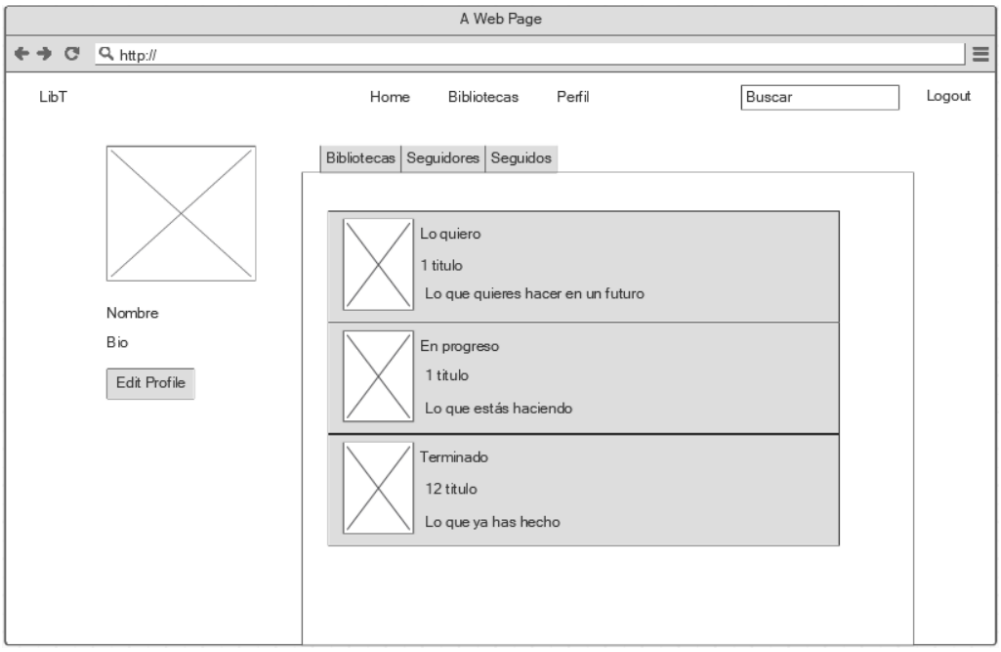

Figura 22 Mockup página perfil de usuario con bibliotecas

Tanto el apartado de seguidores y seguidos sigue un mismo patrón, mostrando la foto de perfil del usuario y su nombre junto a un botón que nos permite seguirlos o dejar de seguirlos sin tener que acceder a su perfil [Figura 23].

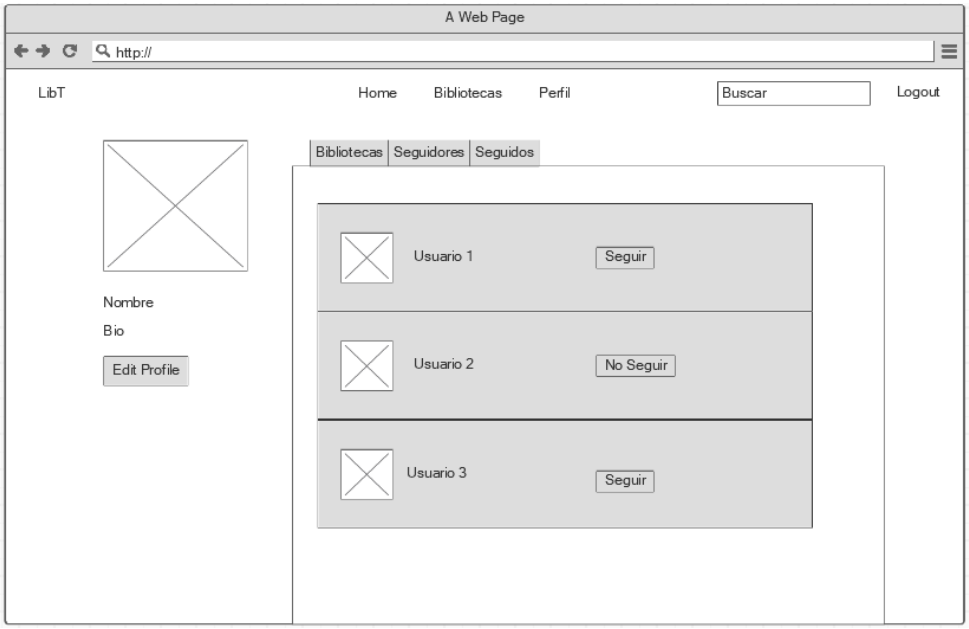

Figura 23 Mockup página de perfil con seguidores y seguidos

5. Desarrollo de la solución propuesta

Una vez concluida la parte de diseño, daremos comienzo a la implementación de dicho diseño. En este apartado hablaremos sobre los puntos más importantes de la implementación y distintos problemas que han ido surgiendo a lo largo de la misma.

5.1. Estructura de ficheros del repositorio

Uno de los retos a lo hora de empezar el desarrollo ha sido elegir la distribución de los distintos paquetes que encontramos en la aplicación. Durante la duración del proyecto, ha habido varios cambios en lo que respecta a la organización. Estos cambios se pueden dividir en tres etapas.

5.1.1. Etapa inicial

En la primera etapa del proyecto tuvimos que enfrentarnos a la decisión de elegir entre dos posibilidades.

- Multirepositorio: esta opción es la que menos configuración requiere. Se basa en mantener una estructura de directorios aislados y en cada una de ellas un repositorio diferente, es decir, el cliente en un repositorio y el servidor en otro.

- Monorepositorio: los diferentes paquetes de la aplicación se encuentran en un mismo repositorio separados en directorios diferentes.

Cada uno de ellos cuenta con una serie de ventajas y desventajas a la hora de llevar a cabo la organización, el desarrollo y el despliegue. Si bien al inicio, tener todo separado en directorios diferentes puede ayudarnos a tener una idea más clara del proyecto pronto veremos que es muy tedioso trabajar con dos instancias del editor de código que usemos.

Otra desventaja, es a la hora de gestionar las subidas al repositorio de *Github,* ya que, al haber un único desarrollador en el proyecto, tiene conocimiento de todos los cambios que se han hecho en las distintas partes del proyecto. Sin embargo, el hecho de tener que estar consultando varias fuentes, se ha descartado como opción óptima para este proyecto. La opción elegida, por tanto, fue monorepositorio aunque el trabajo con él no era el más adecuado.

En esta etapa se utilizaba *NPM* como gestor de paquetes, y para ejecutar el entorno de desarrollo debíamos tener dos instancias diferentes de terminal, una para el cliente y otra para el servidor.

5.1.2. Segunda etapa

Este segundo periodo fue el más breve, en el apareció un nuevo paquete *merge-api* el cual se ubicaba en un repositorio aparte, puesto que usaba otro gestor de paquetes, *Yarn*.

Como ya hemos dicho, esta etapa fue meramente de transición para llegar a la última y más productiva.

5.1.3. Etapa final

En esta última etapa se realizó un estudio más exhaustivo sobre las diferentes opciones a la hora de estructurar un proyecto. En ella se llevaron a cabo varios cambios.

- Utilización de *Yarn* como gestor de paquetes para todos los paquetes. Puesto que ofrecía una mayor velocidad a la hora de descargar los paquetes con respecto a *NPM*.

- Cambio de estructura del repositorio. Hasta ahora, los directorios de los diferentes archivos estaban ubicados en el directorio raíz del repositorio. En este momento se crea una nueva carpeta llamada *package* donde almacenar todos.

- Creación de *workspaces* o espacios de trabajo [Figura 24]. La inclusión de *Yarn* fue principalmente para llevar a cabo una gestión de bibliotecas más ligera y ordenada.

```
"workspaces": [
  "package/client",
  "package/server",
  "package/merge-api",
  "package/auth"
]
```

Figura 24 Configuración de los espacios de trabajo

- Utilización de la biblioteca *concurrently*[27]. Esta biblioteca permite ejecutar en un único comando diferentes aplicaciones, lo que permitió, junto con la creación de nuevos *scripts* en el fichero *package.json* ubicado en el directorio raíz del repositorio [Figura 25].

[27] https://www.npmjs.com/package/concurrently

```
"scripts": {
  "client": "cd package/client && yarn start",
  "server": "cd package/server && yarn run dev",
  "auth": "cd package/auth-api && yarn run dev",
  "merge-api": "cd package/merge-api && yarn run dev",
  "dev": "concurrently \"yarn run server\" \"yarn run client\" \"yarn run auth\" \"yarn run merge-api\"",
  "build": "cd package/client && yarn run build"
}
```

Figura 25 Scripts utilizados durante el desarrollo

5.2. Cliente

En este apartado se tratará toda la implementación realizada en la aplicación cliente. Como ya se ha adelantado anteriormente, las tecnologías utilizadas serán *React*, *Redux* y *React-Router*. En este apartado, hablaremos de la organización del paquete y de los patrones más utilizados a la hora de llevar a cabo la implementación.

5.2.1. Organización de ficheros

A la hora de llevar a cabo un proyecto la estructura es muy clara [Figura 26]. Contamos con dos carpetas principales dentro del proyecto *src,* donde ubicaremos nuestro código, y *build,* la carpeta donde encontraremos el código final de nuestra aplicación una vez se ha procesado para reducir su peso.

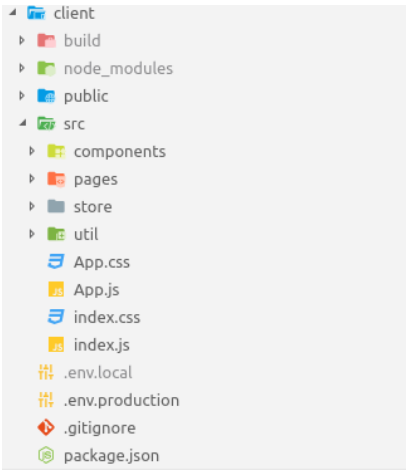

Figura 26 Estructura de ficheros en el paquete cliente

Dentro de la carpeta fuente (*src*) encontramos el punto de entrada, *index.html* y la aplicación principal *App.js* y donde encontramos las diferentes rutas [Figura 27] de

nuestra aplicación. *En ellas diferenciamos a un usuario que está autenticado de uno que no lo esté.*

```
let routes = (
  <Switch>
    <Route path="/login" exact component={LoginPage} />
    <Route path="/signup" exact component={SignupPage} />
    <Route path="/" component={Home} />
  </Switch>
);

if (this.props.isAuth) {
  routes = (
    <Switch>
      <Route path="/" exact component={FeedPage} />
      <Route path="/library/:username?/:list?" component={LibraryPage} />
      <Route path="/profile/:username?" component={ProfilePage} />
      <Route path="/resource/:type/:id" component={ResourcePage} />
      <Route path="/search/:query?" component={SearchPage} />
      <Redirect to="/" />
    </Switch>
  );
}
```

Figura 27 Rutas de la aplicación con react-router

Cada una de las páginas que encontramos en el apartado de diseño tienen su representación dentro del directorio *pages [*Figura 28*]*. Cada página está formada por un documento *Javascirpt* y una hoja de estilo.

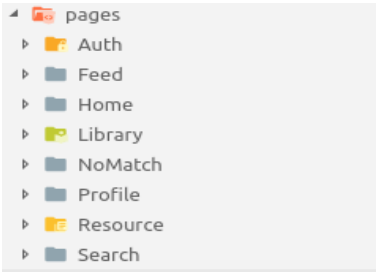

Figura 28 Estructura de las páginas de la aplicación

El siguiente recurso importante que encontramos en *React* son los componentes, ubicados, de manera lógica, en el directorio *components*. Estos componentes son la subdivisión en apartados más simples de las distintas páginas. La idea principal de estos componentes es la reutilización de código en diferentes páginas. Un ejemplo de ellos es el componente *ResourceEntry,* el cual es usado hasta en tres sitios distintos de la aplicación.

La última parte con más relevancia de la implementación es el directorio *store,* si lo traducimos al español, almacén. En él, está ubicado todo el código que controla el estado de la aplicación, es decir, almacena la información que el usuario tiene en un

determinado momento en la aplicación. La idea de su uso es ser capaz de utilizar el estado en cualquier componente de la aplicación. La principal ventaja que nos ofrece es no tener que pasar una propiedad en todos los componentes de la aplicación para compartir datos entre estos dos componentes [Figura 29].

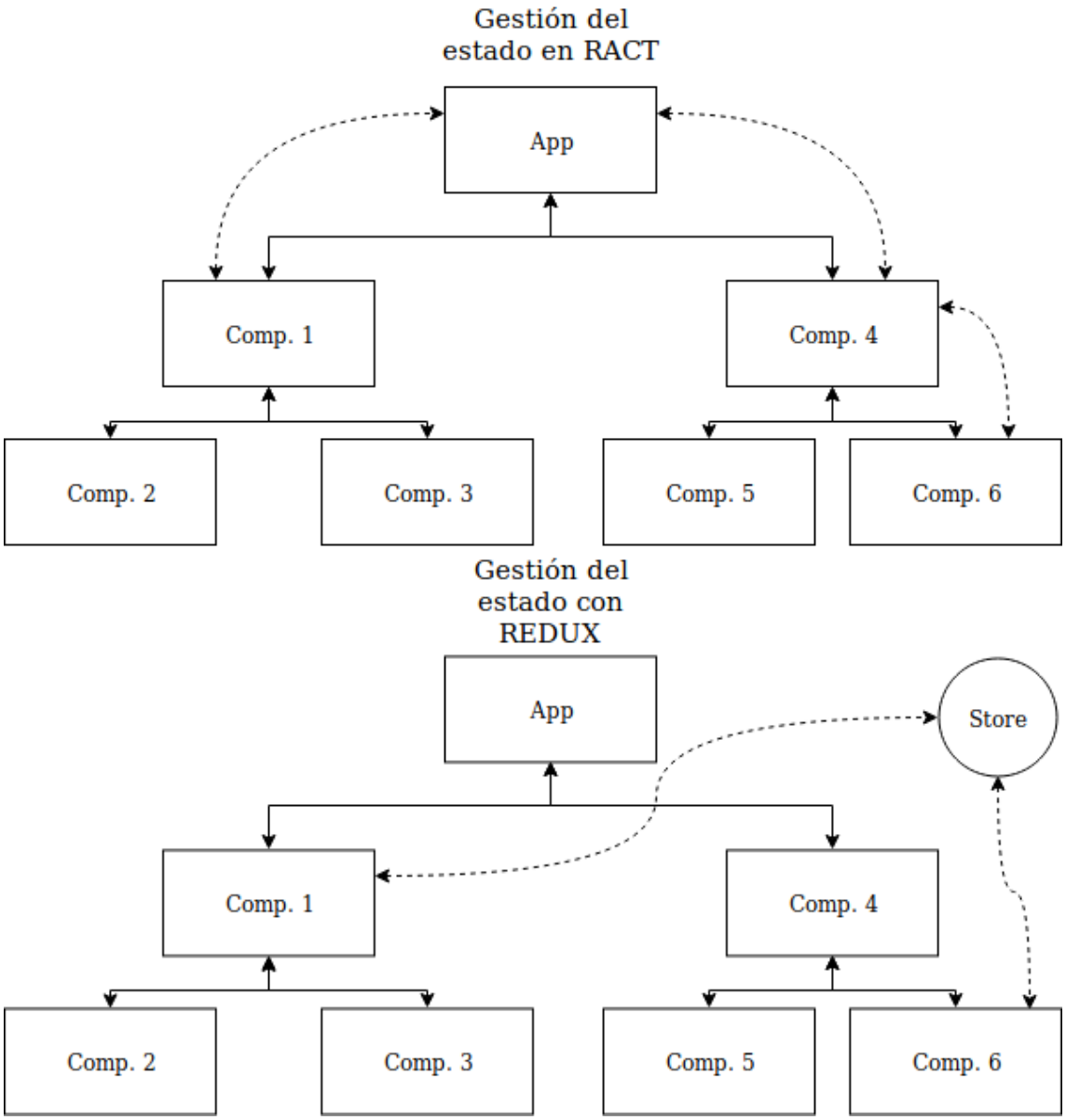

Figura 29 Diferencia entre la gestión del estado de React y React-Redux

Ahora que tenemos acceso al estado en toda la aplicación debemos aprender a utilizarlo y alterarlo. Para ello contamos con los *reducers,* quienes mantienen el estado y las acciones o *actions,* quienes son las encargadas de alterar el estado.

Para terminar de explicar el uso de este patrón contaremos con el ejemplo de la autenticación. Para ello contamos con un *reducer* donde almacenamos el estado y registramos los cambios que llevarán a cabo las acciones [Figura 30].

```
const initialState = {
  userId: null,
  token: null,
  authLoading: false,
  authRedirectPath: '/'
};
const authStart = (state, action) => {
  return updateObject(state, { authLoading: true });
};

const authSuccess = (state, action) => {
  return updateObject(state, {
    token: action.idToken,
    userId: action.userId,
    authLoading: false
  });
};

const authFail = (state, action) => {
  return updateObject(state, {
    authLoading: false
  });
};

const authLogout = (state, action) => {
  return updateObject(state, { token: null, userId: null });
};

const setAuthRedirectPath = (state, action) => {
  return updateObject(state, { authRedirectPath: action.path });
};

const updateUser = (state, action) => {
  return updateObject(state, { userId: action.userId });
};

export default (state = initialState, action) => {
  switch (action.type) {
    case actionTypes.AUTH_START:
      return authStart(state, action);
    case actionTypes.AUTH_SUCCESS:
      return authSuccess(state, action);
    case actionTypes.AUTH_FAIL:
      return authFail(state, action);
    case actionTypes.AUTH_LOGOUT:
      return authLogout(state, action);
    case actionTypes.SET_AUTH_REDIRECT_PATH:
      return setAuthRedirectPath(state, action);
    case actionTypes.UPDATE_USER:
      return updateUser(state, action);
    default:
      return state;
  }
};
```

Figura 30 Ejemplo de reducer para la autenticación

La siguiente clase que debemos comprender son la de las acciones, mucho más extensa que la anterior la dividirlos en dos partes. Una primera donde se declaran las acciones que alterarán el estado de manera asíncrona. Es decir, de qué manera cambiará el estado al iniciar, finalizar o abortar cierta acción [Figura 31].

```
export const authStart = () => {
  return {
    authLoading: true,
    type: actionTypes.AUTH_START
  };
};

export const authSuccess = (token, userId) => {
  return {
    type: actionTypes.AUTH_SUCCESS,
    idToken: token,
    userId: userId,
    authLoading: false
  };
};

export const authFail = () => {
  return {
    type: actionTypes.AUTH_FAIL,
    authLoading: false
  };
};

export const logout = () => {
  history.push('/');
  localStorage.removeItem('token');
  localStorage.removeItem('expiryDate');
  localStorage.removeItem('userId');
  return {
    type: actionTypes.AUTH_LOGOUT,
    token: null
  };
};
export const checkAuthTimeout = milliseconds => {
  return dispatch => {
    setTimeout(() => {
      dispatch(logout());
    }, milliseconds);
  };
};
```

Figura 31 Posibles acciones en la autenticación

La segunda parte del fichero es donde se encuentran los métodos encargados de realizar la lógica que alterará el estado. En ellos es donde se realizar, por ejemplo, las distintas llamadas a las *APIs* [Figura 32].

```
export const auth = params => {
  return dispatch => {
    dispatch(authStart());
    let reqData;
    if (params.isSignUp) {
      reqData = {
        url: '/auth/login',
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        data: JSON.stringify({
          email: params.email,
          password: params.password
        })
      };
    } else {
      reqData = {
        url: '/auth/signup',
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json'
        },
        data: JSON.stringify({
          email: params.email,
          password: params.password,
          username: params.username,
          name: params.name
        })
      };
    }

    AuthApi.request(reqData)
      .then(res => {
        if (res.status === 422) { throw new Error('Validation failed.'); }
        if (res.status !== 200 && res.status !== 201) { throw new Error('Could not authenticate you!'); }
        return res.data;
      })
      .then(resData => {
        localStorage.setItem('token', resData.user.token);
        localStorage.setItem('userId', resData.user.username);

        const remainingMilliseconds = 60 * 60 * 1000;
        const expiryDate = new Date(new Date().getTime() + remainingMilliseconds);
        localStorage.setItem('expiryDate', expiryDate.toISOString());
        dispatch(authSuccess(resData.user.token, resData.user.username));
        dispatch(checkAuthTimeout(remainingMilliseconds));
        history.push('/');
      })
      .catch(err => {
        dispatch(authFail());
        console.log(err);
        dispatch(addError(err));
      });
  };
```

Figura 32 Ejemplo de acción dentro de Redux

Por último, nos quedaría utilizar toda esta información dentro de nuestro paquete de la aplicación cliente. Para ello nos dirigiremos al fichero principal *App.js* y en la parte inferior del archivo haremos uso del estado. Las variables de estado se almacenarán en forma de mapa en la variable *props* de nuestro componente. Las acciones, por otra parte, podrán ser tratadas como cualquier otro método en *React*. Por tanto, podremos pasársela a otros componentes o utilizarlas en el mismo donde la hemos referenciado [Figura 33 Uso del estado y de sus acciones en un componente].

```
const mapStateToProps = state => {
  return {
    error: state.common.error,
    isAuth: state.auth.token != null
  };
};

const mapDispatchToProps = dispatch => {
  return {
    onTryAutoSignup: () => dispatch(actions.authCheckState()),
    onErrorHandler: () => dispatch(actions.errorHandler())
  };
};

export default withRouter(
  connect(
    mapStateToProps,
    mapDispatchToProps
  )(App)
);
```

Figura 33 Uso del estado y de sus acciones en un componente

5.3. Servicio de autenticación

A lo largo del siguiente apartado explicaremos la implementación que hemos llevado a cabo de la arquitectura planteada en el apartado 4.3.2.3. Lógica del servicio de autenticación.

5.3.1. Organización de ficheros

La organización de ficheros que se muestra a continuación [Figura 34] es la misma que se ha seguido a lo largo de todo el proyecto, separando la aplicación por modelos, controladores y rutas (donde la información es validada).

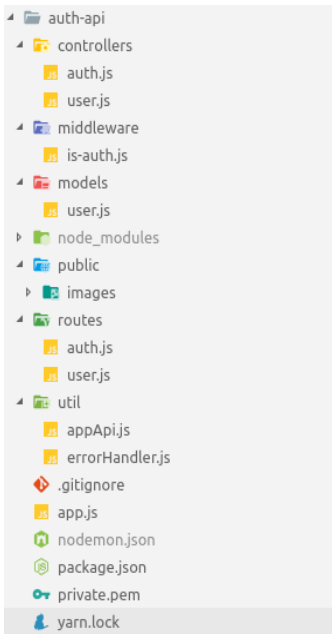

Figura 34 Estructura de ficheros paquete de autenticación

5.3.2. Módulos

En esta aplicación contamos con dos módulos diferentes. El primero de ellos destinado a la autenticación, donde se lleva la labor de identificación y registro de usuario.

En este servicio se generará el token de autenticación [Figura 35]. Para ello se utilizará la librería *jsonwebtoken,* en su interior incluiremos el identificador de usuario que, en nuestro caso, será el nombre de usuario. Como información adicional, adjuntaremos el nombre completo del usuario y su dirección de correo electrónico.

```
userSchema.methods.generateJWT = function() {
  const today = new Date();
  let exp = new Date(today);
  exp.setDate(today.getDate() + 60);

  const pathName = path.join(__dirname, '..', 'private.pem');
  const secret =
    pathName !== '/' && fs.existsSync(pathName)
      ? fs.readFileSync(pathName, 'utf8')
      : undefined;

  return jwt.sign(
    {
      userId: this.username,
      name: this.name,
      email: this.email
    },
    secret,
    { expiresIn: '1h' , algorithm: 'RS256' }
  );
};
```

Figura 35 Código con la generación del JWT

Tanto en este paquete como en los demás contamos con un apartado para el decodificado del token [Figura 36]. En ella seguimos los pasos descritos en el apartado de la lógica:

1. Comprobamos si existe la cabecera *Authorization*.
2. Si existe, debemos comprobar si sigue el formato *Bearer* dividiendo el token en dos partes.
3. Antes de decodificar el token, debemos verificar si su contenido el válido, evitando así posibles vulnerabilidades. Un ejemplo de ello es no especificar el

algoritmo del *JWT,* es decir, declararlo para que sea de tipo *none*, haciendo que no sea necesario la firma del mensaje.

4. El último paso, será la decodificación del token e insertarlo en la petición para tener acceso a esta información a lo largo del resto de peticiones. Gracias a la biblioteca utilizada, realizaremos este paso y el anterior en una única línea de código.

```
module.exports = (req, res, next) => {
  const authHeader = req.get('Authorization');
  if (!authHeader) {
    const error = new Error('Not authenticated.');
    error.status = 401;
    throw error;
  }
  const token = authHeader.split(' ')[1];
  let decodedToken;
  try {
    const pathName = path.join(__dirname, "..", "public.pem")
    const secret = pathName !== '/' && fs.existsSync(pathName) ? fs.readFileSync(pathName, 'utf8') : undefined;
    decodedToken = jwt.verify(token, secret, { algorithm: 'RS256' });
  } catch (err) {
    err.statusCode = 500;
    throw err;
  }
  if (!decodedToken) {
    const error = new Error('not authonticated.');
    error.statusCode = 401;
    throw error;
  }
  req.userId = decodedToken.userId;
  req.userToken = token;
  next();
};
```

Figura 36 Método de autenticación

Cuando queramos proteger una ruta, tendremos que realizar una llamada al método anterior antes de entrar a las siguientes llamadas o acciones [Figura 37].

```
router.put('/', isAuth, siguiente_llamada);
```

Figura 37 Ruta protegida mediante autenticación

El segundo es el encargado de llevar el resto de las acciones relacionadas con el usuario, es decir, las acciones de seguir usuarios, obtener usuarios o editar la información principal del perfil de usuario. Este último de mayor complejidad, puesto que requiere guardar imágenes de los usuarios.

El método implementado es muy extenso, dado el gran número de validaciones realizadas, dichas validaciones están relacionadas con la existencia de la imagen en la petición, la existencia de usuarios (a la hora de realizar un cambio de nombre). A la hora de llevar a cabo el tratamiento de la imagen se utilizará la biblioteca *multer,* la cual

introduce en el resto de la llamada metadatos relacionados con la imagen proporcionada por el usuario. Es importante, sobre todo en lo referido a las capacidades del espacio en memoria del servidor, que cada vez que el usuario suba una nueva imagen, la imagen anterior será eliminada de nuestro directorio de archivos.

En un futuro se puede cambiar el controlador de las imágenes para utilizar un servicio en la nube que almacenen las imágenes en otro espacio externo al servidor.

5.4. Servicio de la aplicación principal

El servicio de la aplicación principal es un *API REST* escrito en *Node JS* y usando el marco de trabajo o *framework Express*. En este apartado hablaremos sobre las características principales de la aplicación.

5.4.1. Organización de ficheros

La estructura utilizada en el paquete anterior es muy similar a la que utilizaremos en el servidor principal [Figura 38]. El paquete está divido en tres directorios principales: controladores (*controllers*), modelos (*models*) y rutas (*routes*) y tres secundarios (*public*, *util* y *middleware*).

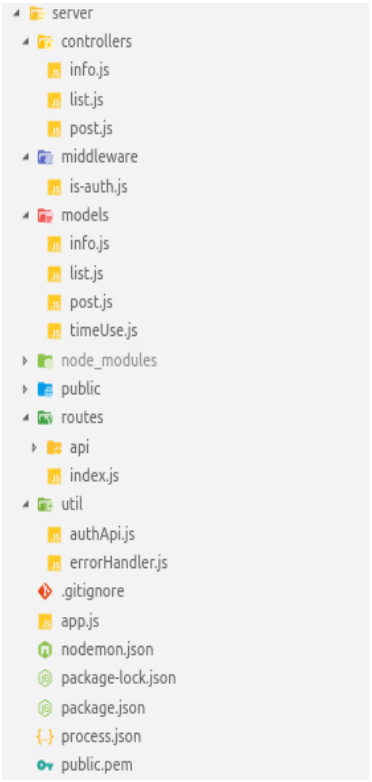

Figura 38 Directorios y ficheros de la API principal

5.4.2. Módulos

Fichero principal

En este caso estamos hablando del fichero *app.js,* es el fichero que contiene el servidor HTTP y toda la configuración utilizada o, por el propio servidor, o por las distintas bibliotecas de la aplicación [Figura 39].

```
const path = require('path');

const express = require('express');
const bodyParser = require('body-parser');
const mongoose = require('mongoose');
const logger = require('morgan');
const multer = require('multer');

const indexRouter = require('./routes/index');

const app = express();

app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use('/api/public', express.static(path.join(__dirname, 'public')));

app.use((req, res, next) => {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader(
    'Access-Control-Allow-Methods',
    'OPTIONS, GET, POST, PUT, PATCH, DELETE'
  );
  res.setHeader('Access-Control-Allow-Headers', 'Content-Type, Authorization');
  next();
});

app.use('/', indexRouter);

// error handler
app.use(function(err, req, res, next) {
  // set locals, only providing error in development
  console.log(err);
  const message = err.message;
  const data = req.app.get('env') === 'development' ? err.data : {};

  // render the error page
  res.status(err.statusCode || 500).json({ message: message, data: data });
});

mongoose.connect(
    process.env.MONGO_URL,
    { useNewUrlParser: true }
  )
  .then(result => {
    app.listen(process.env.PORT || 3001);
    console.log('Server on!');
  })
  .catch(err => console.log(err));
```

Figura 39 Configuración de la aplicación principal

Directorio de rutas de la aplicación *(routes)*

En los ficheros de este directorio realizaremos dos posibles acciones. La primera redirigir las diferentes peticiones que nos llegan al servidor a un método distinto [Figura 40]. En *Express* indicamos el método de la petición como una función de *Javascript,* el primer parámetro de esta función corresponde a la ruta y los siguientes son los correspondientes a todos lo *middlewares* por los que pasa la petición. En nuestro caso encontraremos hasta dos *middlewares* (el de autenticación, opcional, y la función correspondiente a dicha ruta).

```
const router = require('express').Router();

const infoController = require('../../controllers/info');
const isAuth = require('../../middleware/is-auth');

// /info
router.post('/', isAuth, infoController.createOrModifyResource);

// /info/status/:resourceId
router.get('/status/:resourceId', isAuth, infoController.getStatus);

module.exports = router;
```

Figura 40 Ejemplo de rutas en Express

La segunda acción será la validación del contenido del cuerpo del mensaje, en caso de que ésta sea necesaria [Figura 41]. Con esto comprobamos que los valores que recibidos son los deseados, minimizando así, posibles fallos de seguridad.

```
router.post(
  '/',
  isAuth,
  [
    body('title')
      .trim()
      .isLength({ min: 3 }),
    body('description').trim(),
    body('public').isBoolean(),
    body('creator').trim()
  ],
  listController.createList
);
```

Figura 41 Validación del cuerpo del mensaje

Directorio de los controladores *(controllers)*

En este directorio de carpetas almacenamos las diferentes funciones utilizadas por las rutas. En estos ficheros es donde se realiza toda la lógica de negocio, realizándose peticiones sobre la base de datos para, o bien obtener el estado de algún recurso [Figura

42] o bien para alterarlo, ya sea creando, modificando o eliminando objetos de la base de datos.

```
exports.getStatus = async (req, res, next) => {
  const resourceId = req.params.resourceId;

  try {
    const info = await Info.findOne({
      searchId: resourceId,
      creator: req.userId
    }).populate('lists');
    let status = 0;
    let lists = [];
    if (info && info !== null) {
      lists = info.lists.map(x => x._id);
      status = info.actualState;
    }

    res.status(200).json({ status: status, lists: lists });
  } catch (err) {
    if (!err.statusCode) err.statusCode = 500;
    next(err);
  }
};
```

Figura 42 Función para obtener el estado de un recurso

Directorio de los modelos *(models)*

Junto a la biblioteca *mongoose, la* cual nos proporciona un entorno de trabajo utilizando esquemas (*Schemes) [*Figura 43*], mantendremos* una estructura más organizada dentro de nuestra base de datos no relacional.

Además de la organización, esta biblioteca nos proporciona funciones de búsqueda, creación, modificación y borrado, de una manera más simple y potente que las peticiones realizadas sobre *Mongo DB*.

Gracias a los esquemas, estas funciones comprueban que las peticiones realizadas son válidas y siguen la estructura definida en estos esquemas. También podremos implementar nuestros propios métodos para que sean llamados a partir de un objeto de la base de datos.

```
const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const infoSchema = new Schema(
  {
    type: { type: String, required: true },
    searchId: { type: String, required: true },
    lists: [{ type: Schema.Types.ObjectId, ref: 'List' }],
    actualState: { default: 0, type: Number },
    review: { type: String },
    stars: { type: Number, enum: [0, 1, 2, 3, 4, 5] },
    creator: { type: String, required: true },
    times: [{ type: Schema.Types.ObjectId, ref: 'TimeUse' }]
  },
  { timestamps: true }
);

infoSchema.methods.addList = function(listId) {
  if (this.lists.indexOf(listId) === -1) {
    this.lists.push(listId);
  }

  return this.save();
};

infoSchema.methods.removeList = function(listId) {
  if (this.lists.indexOf(listId) !== -1) {
    this.lists.remove(listId);
  }
  return this.save();
};

infoSchema.methods.setActualState = function(actualState) {
  this.actualState = actualState;
  return this.save();
};

module.exports = mongoose.model('Info', infoSchema);
```

Figura 43 Esquema del objeto Info

Directorios secundarios

- Directorio *public*: los ficheros estáticos de la aplicación, como imágenes, hojas de estilo, ficheros de texto, etc. serán ubicados aquí.
- Directorio *middleware:* componentes de *software* que se pueden utilizar en cualquier petición de la aplicación. En este directorio encontramos como ejemplo el método de autenticación.
- Directorio *util:* contiene ficheros con funciones de *Javascript* más genéricas. En ella escribiremos funciones de ayuda, como validadores o funciones de configuración para bibliotecas.

5.5. Servicio de búsqueda

En este apartado vamos a profundizar en la implementación necesaria para conseguir los objetivos propuestos en el apartado de diseño de la lógica. De nuevo contamos con

un API REST como interfaz de petición de datos. Como veremos a continuación, la estructura de este módulo es ligeramente diferente a la de los dos anteriores.

5.5.1. Organización de ficheros

El paquete donde está ubicada toda la lógica de la aplicación se llama *merge-api* [*Figura* 44], la estructura que sigue es muy simple. Tenemos el fichero *app.js* que es la aplicación principal. Dos directorios, uno donde almacenamos la fábrica de estrategias y el otro donde ubicaremos todo el set de estrategias implementadas.

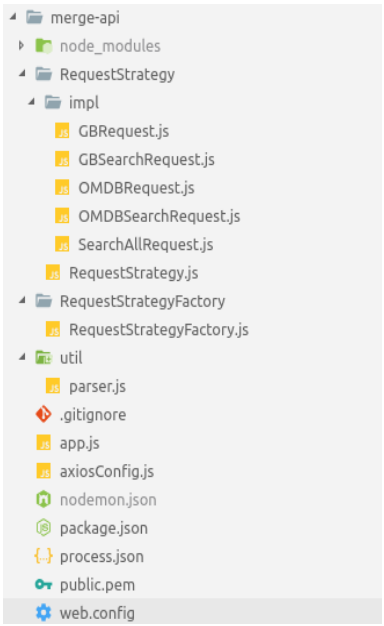

Figura 44 Estructura de ficheros servicio de búsqueda

5.5.2. Módulos

En este apartado cubriremos la implementación de los puntos más importantes de este paquete.

Caché del servidor

Para realizar este almacenamiento en caché se hizo uso de *axios-cache-adapter*[28] una biblioteca que almacena las peticiones en un *store* (almacenamiento en memoria) para prevenir peticiones innecesarias a través de la red. En nuestro caso lo configuraremos para que las peticiones se almacenen durante quince minutos [Figura 45], cache distintos parámetros de petición (si no lo hiciésemos, sobrescribiríamos las peticiones) y que

[28] https://github.com/RasCarlito/axios-cache-adapter

ignore las cabeceras del resto de peticiones, puesto que siempre deseamos utilizar la caché.

```
const axios = require('axios');

const { setupCache } = require('axios-cache-adapter');

const cache = setupCache({
  maxAge: 15 * 60 * 1000,
  exclude: {
    query: false
  },
  readHeaders: false
});

let api;

const init = () => {
  api = axios.create({
    adapter: cache.adapter
  });
};

module.exports = params => {
  if (!api) {
    init();
  }

  return api(params);
};
```

Figura 45 Configuración de la caché

Un buen ejemplo es su uso en la búsqueda global, es decir, en todas las *APIs* [Figura 46]. En este fragmento de código vemos como se utiliza la caché en cada una de las peticiones. El resto de código se utiliza para mezclar todas las respuestas en una única.

Como última anotación, observamos que hay dos llamadas a la mismo *API* de OMDb. Esta llamada se podría convertir en una única si quisiésemos disminuir el número de llamadas a dicha *API*. Por el contrario, se decidió llevar a cabo esta separación para mostrar un máximo de diez recursos de cada tipo y así siempre tener una lista más heterogénea de recursos.

```
doRequest(data) {
  const { name, page } = data;
  const OMDBPageSearch = page && page > 1 ? `&page=${page}` : '';
  const GBPageSearch = page && page > 1 ? `&startIndex=${page * 10 - 1}` : '';
  return axios
    .all([
      axiosCache({
        url: `http://www.omdbapi.com/?s=${name}&type=game&apikey=${process.env.OMDB_API_KEY}${OMDBPageSearch}`,
        method: 'get'
      }).catch(err => console.log('GET GAMES FAIL')),
      axiosCache({
        url: `http://www.omdbapi.com/?s=${name}&type=movie&apikey=${process.env.OMDB_API_KEY}${OMDBPageSearch}`,
        method: 'get'
      }).catch(err => console.log('GET MOVIES FAIL')),
      axiosCache({
        url: `https://www.googleapis.com/books/v1/volumes?q=${name}${GBPageSearch}`,
        method: 'get'
      }).catch(err => console.log('GET BOOKS FAIL'))
    ])
    .then(([games, movies, books]) => {
      const cleanGames = games ? omdbParser(games.data) : emptyParserResponse();
      const cleanMovies = movies ? omdbParser(movies.data) : emptyParserResponse();
      const cleanBooks = books ? bookParser(books.data) : emptyParserResponse();

      const itemsOutput = [...cleanBooks.search, ...cleanMovies.search, ...cleanGames.search]
        .sort((a, b) => a.title > b.title);
      const lengthsOutput = [cleanGames.totalResults, cleanBooks.totalResults, cleanMovies.totalResults]
        .reduce((acc, curr) => (curr && curr > acc ? curr : acc), 0);

      const responseOutput = [cleanGames.response, cleanBooks.response, cleanMovies.response].find(x => x);

      return { search: [...itemsOutput], totalResults: lengthsOutput, response: responseOutput ? true : false };
    })
    .catch(err => {
      console.error(err);
    });
}
```

Figura 46 Método de búsqueda en todas las APIs

En la siguiente tabla [Table 2] podemos observar que los tiempos en la primera búsqueda son superiores al resto. Estos datos han sido obtenidos con la ayuda de las herramientas de desarrollador proporcionadas por el navegador Firefox.

Table 2 Tiempos por petición

Parámetro de búsqueda	Tiempo en la primera petición	Tiempo en la segunda petición	Tiempo en la tercera petición
Harry	780ms	4ms	5ms
You	688ms	5ms	6ms
Guardians of nothing	559ms	4ms	7ms

Patrones de diseño

Con el objetivo de tener una mayor flexibilidad y posterior mantenimiento del código, se decidió buscar la mejor solución a la hora de implementar toda la comunicación con

las *APIs*. El primer patrón que utilizaremos es el de *Factory Method,* también conocido como constructor virtual. Estamos hablando de un patrón de tipo creacional.

"Define una interfaz para crear un objeto, pero las subclases deciden que clase instanciar. Factory Method permite a las clases aplazar la instanciación de subclases." [2]

El objetivo principal es la extensibilidad. Este patrón de diseño es usado con frecuencia cuando hay que manejar, mantener y manipular colecciones de objetos que difieren al mismo tiempo en muchas características en común. En nuestro caso estamos hablando en las estrategias a la hora de realizar la obtención de los recursos [Figura 47].

```
class RequestStrategyFactory {
  createRequestStrategy(type) {
    const requestStrategy = new RequestStrategy();
    switch (type) {
      case "book":
        requestStrategy.request = new GBRequest();
        break;
      case "movie":
      case "game":
        requestStrategy.request = new OMDBRequest();
        break;
      case "search-all":
        requestStrategy.request = new SearchAllRequest();
        break;
      case "search-book":
        requestStrategy.request = new GBSearchRequest();
        break;
      case "search-game":
      case "search-movie":
        requestStrategy.request = new OMDBSearchRequest();
        break;
      default:
        const error = new Error("Not supported strategy");
        error.statusCode = 422;
        throw error;
    }
    return requestStrategy;
  }
}
```

Figura 47 Implementación fábrica abstracta en Javascript

El siguiente patrón es de tipo comportamiento, el cual va de la mano con el expuesto anteriormente, es el patrón *Strategy* o estrategia.

"Definir una familia de algoritmos, encapsulando cada uno, y haciéndolos intercambiables. El patrón estrategia permite a los algoritmos variar independientemente del cliente que lo use" [2]

Y como ya hemos adelantado en el patrón anterior, este patrón nos ayuda en la obtención de diferentes tipos de recursos. La interfaz utilizada en su implementación es la siguiente [Figura 48], donde únicamente almacenamos en un objeto la estrategia (realizando la encapsulación de la que se habla en su definición) y tenemos un método común que implementaremos en todas las estrategias *doRequest(data)*.

```
class RequestStrategy {
  constructor() {
    this._request = null;
  }
  set request(request) {
    this._request = request;
  }
  get request() {
    return this._request;
  }
  doRequest(data) {
    return this._request.doRequest(data);
  }
}

module.exports = RequestStrategy;
```

Figura 48 Interfaz de la estrategia usada en las búsquedas en Javascript

6. Implantación

Una vez terminada la implementación de nuestro código, debemos encontrar un sitio donde publicarla. Sin embargo, antes hemos llevado a cabo un estudio del mercado entre las principales opciones de despliegue, para después implantar nuestra aplicación en dos de ellas y así poder decidir el entorno más adecuado para nuestra aplicación.

6.1. Estudio de entornos para el despliegue

A la hora de llevar a cabo un despliegue con microservicios debemos tener en cuenta que la latencia y la disponibilidad son uno de los factores más importantes. Es por ello por lo que los servicios que escojamos deben ofrecernos ambas funcionalidades.

Como añadido, es muy importante poder llevar a cabo, por separado, despliegues y réplicas de los distintos servicios, dependiendo de la demanda por parte los usuarios.

***Heroku*[29]**

Heroku es una plataforma en la nube como servicio [Figura 49]. Nace en 2007 con el objetivo de facilitar al desarrollador la etapa de despliegue, donde, normalmente, tiene que empaquetar su aplicación, ejecutarla y si tiene éxito escalarla.

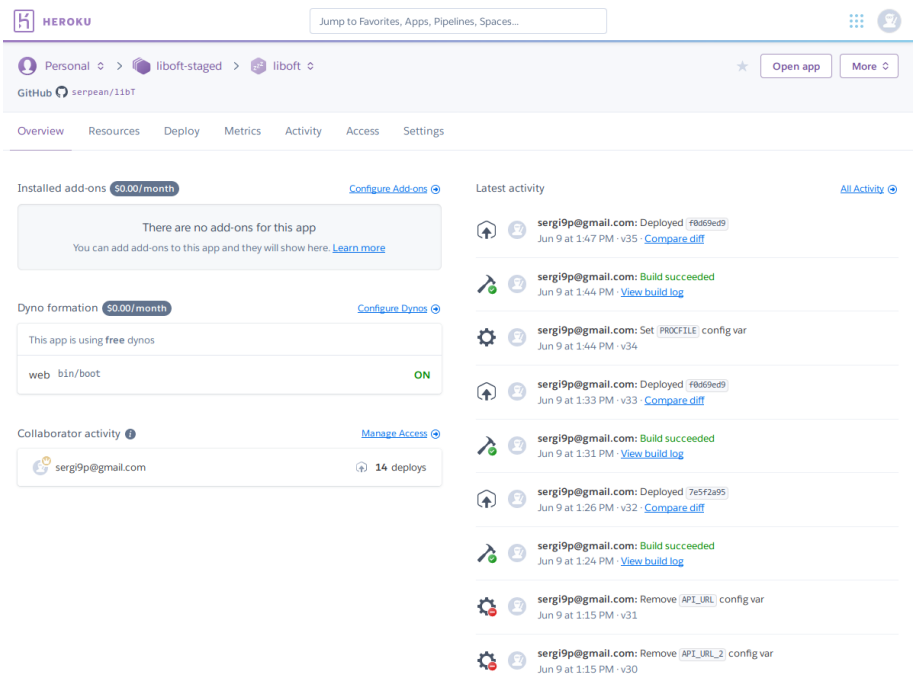

Figura 49 Página principal del panel de control en Heroku

[29] https://heroku.com

Su forma de trabajo es simple y clara. La aplicación debe ser enviada a través de un repositorio, *Dropbox,* o vía API. En nuestro caso contamos con *Github*. El despliegue debe poder llevarse a cabo de manera automática a través de *scripts* incluidos en el código fuente o con su fichero de configuración *Procfile*.

Por detrás de nuestro despliegue encontraremos un contenedor *Unix* aislado que sigue las instrucciones indicadas a través de lo que *Heroku* denomina *Buildpacks*. También contaremos con la ayuda de variables de entorno y *add-ons* (en nuestro entorno utilizaremos *mLab,* que es como se llama el servicio de *MongoDB* que ofrece *Heroku).* Se profundizará con un mayor detalle del despliegue en el Anexo 1.

Dynos es el nombre que *Heroku* otorga a la unidad principal que provee el entorno de ejecución. En nuestro caso ubicaremos un servicio por *Dyno*. En su formato gratuito, el *Dyno* terminará su ejecución tras treinta minutos de inactividad.

Azure[30]

La siguiente plataforma estudiada fue *Azure,* de nuevo nos encontramos con una plataforma en la nube como servicio [Figura 50]. En el caso de *Azure,* contamos con un mayor número de opciones y una mayor libertad para realizar el despliegue ya que podemos acceder a los contenedores directamente a través de *SSH*.

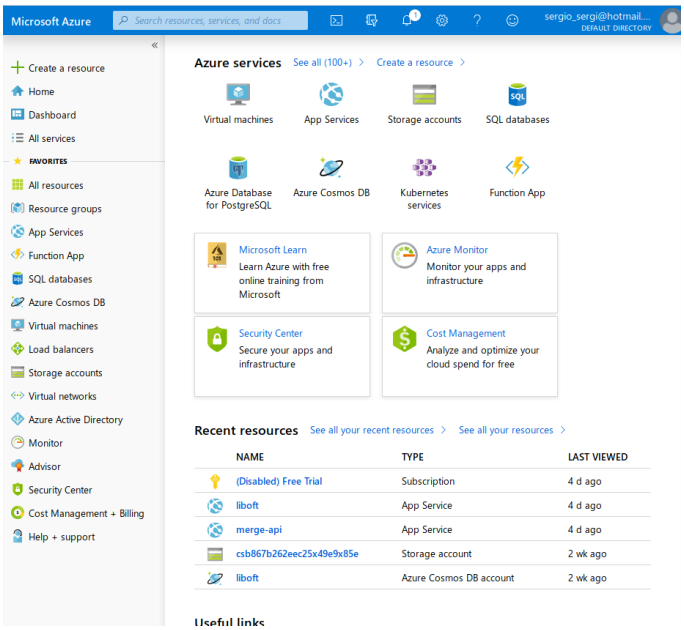

Figura 50 Panel de control Azure

[30] www.portal.azure.com

Desde *Azure* tenemos que utilizar distintos tipos de servicios para llevar a cabo el despliegue. Entre ellos contamos con:

- Cuenta de almacenamiento (*Storage accounts*): utilizado para el almacenamiento de páginas estáticas. Es el candidato perfecto para colocar nuestra aplicación de *React*.
- Servicio de aplicaciones (*App Services*): su nombre nos va dando una pista, nos permite tener servicios en la red. Está basado en contenedores *Docker* y es donde ejecutaremos las distintas instancias de nuestros servicios.
- *CosmosDB:* para poder tener una base de datos basada en *MongoDB,* debemos utilizar el propio servicio que nos ofrece *Azure* en la nube. No tendremos que hacer ningún cambio en el código para hacer funcionar esta base de datos.

Para poder enviar el código fuente de nuestra aplicación dispondremos de distintos medios de comunicación. Entre ellos están repositorios de Git, almacenamiento en la nube de *Azure* o directamente por *SFTP*.

En cada servicio podemos declarar variables de entorno las cuales nos ayudan con la configuración de la aplicación. Esta ha sido la solución empleada para controlar la comunicación entre APIs.

6.2. Conclusiones de la implantación

Ambos despliegues se han llevado de manera exitosa a producción. Cada uno de ellos cuenta con ventajas y desventajas.

Características comunes

- Desplegar a través de repositorios de *Github*.
- Capacidad de volver a desplegar pulsando un único botón.
- Sistema de registro en tiempo real. Lo que permite la detección de errores tanto en el despliegue como en el tiempo de ejecución.

Características de *Heroku*

- Creación más directa: sólo tenemos una posible opción de creación y con ella podemos llevar a cabo todo tipo de despliegues.

- Entorno más minimalista: toda la configuración se realiza en el código fuente por lo que el panel de control no tiene muchos elementos.
- Cuentan con un nivel gratuito de horas de cómputo, el cual se reinicia mensualmente. Es una gran ventaja para mantener aplicaciones sencillas de manera gratuita. Por el contrario, nuestro caso no es válido puesto que el número de tiempo de cómputo debe dividirse entre cuatro.
- Los *Dynos* entran en suspensión a los treinta minutos de inactividad en el nivel gratuito, sin embargo, hay aplicaciones online que permite mantener nuestra aplicación despierta continuamente.

Características de *Azure*

- El número de opciones ofrecidas por *Azure* es mayor.
- Cuenta con un sistema de registro con gráficas para ver el consumo mensual.
- Cuenta con una consola *SSH,* la cual nos permite realizar comandos. Esto ha sido de gran ayuda a la hora de llevar a cabo un primer despliegue, puesto que el sistema no ejecutó correctamente un comando de terminal que resolvía las dependencias del proyecto y las instalaba. Con la ayuda de esta interfaz pudimos volver a ejecutar el comando fallido.
- No contamos con un nivel gratuito mensual. Sin embargo, contamos con un depósito inicial de dinero el cual nos permite llevar a cabo todas nuestras pruebas. La experiencia en este proyecto no fue la deseada, pues consumía más del presupuesto esperado bloqueando la aplicación.

7. Pruebas

Aunque este capítulo está ubicado al final de la memoria del proyecto su uso fue recurrente durante todo el proceso de desarrollo. En este apartado nos centraremos en particular en las pruebas de la interfaz de usuario. El objetivo es ofrecer a los usuarios una experiencia óptima y lo más accesible posible.

Para agilizar el proceso vamos a hacer uso de las herramientas de desarrollo del navegador Google Chrome. En ellas tenemos una sección de auditoría. En esta sección podemos configurar una serie de pruebas que posteriormente se ejecutaran automáticamente en nuestro navegador.

En nuestro caso realizaremos pruebas tanto para navegador móvil como de escritorio, desactivando la opción de *Progressive Web App, ya que no es un* objetivo de nuestro proyecto [Figura 51].

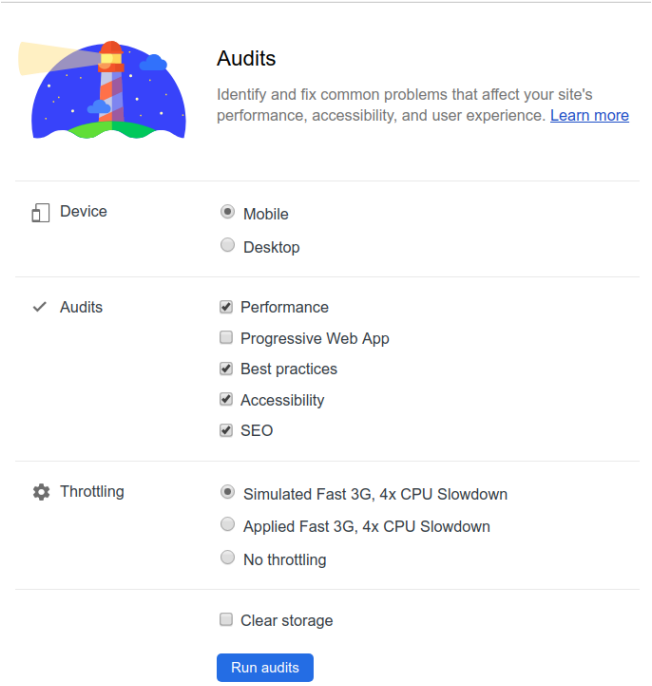

Figura 51 Configuración herramienta de auditoría

Tras lanzar la aplicación por las distintas páginas, obteniendo un resultado entre cero y cien puntos, observamos que nuestra accesibilidad no es la deseada [Table 2]. A partir del resultado, y con la ayuda de las descripciones proporcionadas por la herramienta, podremos conseguir una mejor puntuación, pues estos nos dan información más detallada de los errores cometidos.

URL de la página	Rendimiento		Accesibilidad		Buenas prácticas		SEO	
	PC	Móvil	PC	Móvil	PC	Móvil	PC	Móvil
/*	100	100	100	100	93	93	100	100
/	100	96	52	78	93	93	100	100
/login	93	91	92	100	93	93	100	100
/register	94	89	92	100	93	93	100	100
/library	90	85	57	87	93	93	100	100
/profile	75	70	57	70	93	93	100	100
/search	80	84	71	76	93	93	100	100

Tabla 1: Puntuación en la auditoria de accesibilidad, valores entre 0 y 100

*: usuario no autenticado

Los errores encontrados tenían que ver, principalmente, con el incorrecto etiquetado de las imágenes y enlaces en ciertas partes de la aplicación. Una vez arreglado todos los errores de accesibilidad, la puntuación obtenida es de 100 puntos en todos los casos de accesibilidad, sin afectar el contenido del resto de la tabla. En futuras versiones del código sería interesante mejorar el rendimiento a la hora cargar las páginas.

8. Conclusiones

En este trabajo se ha llevado a cabo la tarea compleja de llevar a cabo el desarrollo de una aplicación en todas sus etapas. El trabajo comienza con la fase de análisis, pasando a la de diseño. Una vez estas dos fases se encontraron listas, fue el turno de llevar a cabo la implementación, la cual llevó el mayor grueso del trabajo. A continuación, se llevó a cabo la implantación o despliegue del software desarrollado. Por último, se realizaron pruebas de rendimiento y accesibilidad a la aplicación.

Cada una de las etapas han supuesto una serie de retos muy diversos, los cuales han requerido un estudio previo a las decisiones a tomar.

En la primera y segunda etapa, el proyecto giró en torno a la observación de otras plataformas y aplicaciones similares. El objetivo era mejorar y pulir las ideas que teníamos sobre el proyecto. La planificación y diseño obtenidos como resultado de esta fase resultaron ser acertados, sin embargo, requirió de ciertos ajustes durante la etapa de implementación por el volumen masivo de información a tratar. Sin embargo, estos cambios no afectaron ni a la planificación ni al resultado final.

En la etapa de implementación, se siguió la planificación prevista a lo largo del desarrollo centrado en *sprints*. Como esta etapa fue la más larga del desarrollo del proyecto, se adquirieron conocimientos más avanzados a los propuestos inicialmente en la etapa de diseño. Esto supuso distintos cambios de diseño e implementación a lo largo del desarrollo los cuales tuvieron un impacto positivo al finalizar esta etapa.

La última etapa, implantación, nos permitió explorar nuevas herramientas y tecnologías con la que no habíamos trabajado hasta el momento. El hecho de tener que desplegar distintas aplicaciones y distintos contenedores requirió de un estudio más exhaustivo de las plataformas utilizadas. En esta etapa, el código sufrió cambios, puesto que el alumno no había tenido en cuenta posibles conflictos entre tecnologías, este fue el caso de *MongoDB Atlas,* así como errores con el direccionamiento de los microservicios.

Para finalizar el proyecto, y a título de resumen, se han alcanzado todos los objetivos propuesto al inicio de este, creando una base sólida donde seguir implementando futuras funcionalidades.

9. Trabajos futuros

En este proyecto se quedan fuera diversas funcionalidades que se escapan de la planificación inicial, así como mejoras propuestas por el alumno debido al aumento de conocimientos técnicos y de diseño adquiridos a lo largo del proyecto.

En cuanto a las funcionalidades, la más próximas y necesarias para el éxito del proyecto serían:

- Mejora del diseño a la hora de cargar el contenido.
- La implementación de renderización en lado del servidor. Sería de gran utilidad, pues solo cargaríamos las páginas necesarias en cada momento, en lugar de cargar todo el *Javascript* de la aplicación al inicio de la descarga.
- Creación de una página de búsqueda de amigos.
- Desarrollo e integración de un microservicio que permita el uso de chat en la aplicación.
- Desarrollo de una sección de valoración para cada recurso de la aplicación.
- Mejora de la configuración de búsqueda, muchas veces el usuario no es capaz de encontrar el recurso que esta buscado si no introduce exactamente el título o autor del recurso.

Las siguientes propuestas para el proyecto estarían relacionadas con su despliegue o implantación. El estudio de otras tecnologías, destacando *Kubernetes,* junto con la creación y configuración de un dominio serían los últimos pasos antes de dar el salto a un entorno productivo más fiable.

10. Referencias

[1] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems 1st Edition. O'Reilly Media.

[2] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). DesignPatterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc.

[3] Freeman, E., Robson, E., Bates & B. Head First Design Patterns: A Brain-Friendly Guide (2004). O'Reilly Media.

[4] Hunt, A. & Thomas, D. (1999). The Pragmatic Programmer: From Journeyman to Master. Addison Wesley.

[5] Martin, R.C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Pearson Education, Inc.

[6] Rad, N. K. & Turley, F. (s.f). The scrum Master Trainin Manual. A Guide to Passin the PSM Exam.

[7] Maximilian Schwarzmüller. (2019). Academind: Dynamic Vs. Static Vs. SPA. Recuperado de http://www.academind.com

[8] Ruth Reinicke (s.f). Authorization Authentication with microservices. Recuperado de https://blog.leanix.net/en/authorization-authentication-with-microservices

[9] *Ajax – Web developer guides. MDN Web Docs. (2018). https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX*

[10] JSON Web Tokens. https://jwt.io/

[11] Microsoft. Azure. https://docs.microsoft.com/en-us/azure/

[12] JSONWEBTOKEN. *Rfc7519*. https://tools.ietf.org/html/rfc7519

[13] Dofactory (s.f). JavaScript + jQuery Design Pattern Framework. Recuperado de https://www.dofactory.com/

[14] Brian Pontarelli (2018). Authetntication as Microservice. Recuperado de https://www.youtube.com/watch?v=SLc3cTlypwM&t=844s

[15] GITHUB. About project boards
https://help.github.com/en/articles/about-project-boards

[16] Patricio Letelier. Organización ágil del trabajo. Recuperado de agilismoatwork.blogspot.com

11. Anexo 1: Despliegue en *Heroku*

Heroku fue la primera plataforma que se planteó a la hora de llevar a cabo el despliegue. Las ventajas se han descrito durante el trabajo, pero el hecho más determinante a la hora de escoger a esta plataforma fue su simplicidad a la hora de comenzar.

El primer paso que tendremos que realizar, como en la mayoría de los servicios de internet es crearnos una cuenta [Figura 52 Registro en Heroku]. *Heroku* nos ofrece la posibilidad de crearnos una cuenta, sin coste alguno, para realizar las pruebas iniciales. También nos ofrece aumentar el número de horas gratuitas mensuales estableciendo una tarjeta de crédito en la cuenta de usuario.

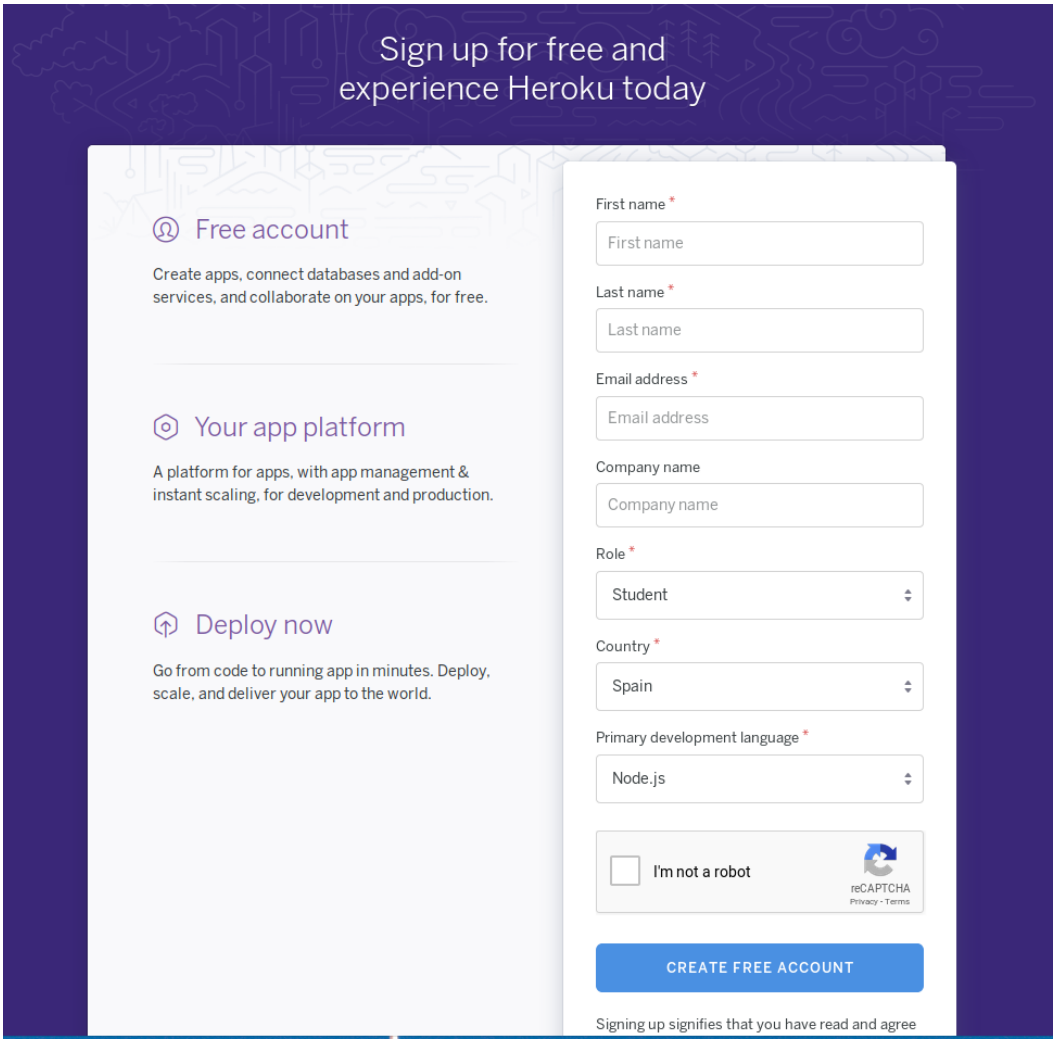

Figura 52 Registro en Heroku

Una vez registrados y en la página de inicio, crearemos nuestra primera aplicación [Figura 53 Crear aplicación en Heroku]. Las opciones de configuración son mínimas, estableceremos el nombre de la aplicación *liboft* y la ubicación del contenedor *Europe [*Figura 54*]*.

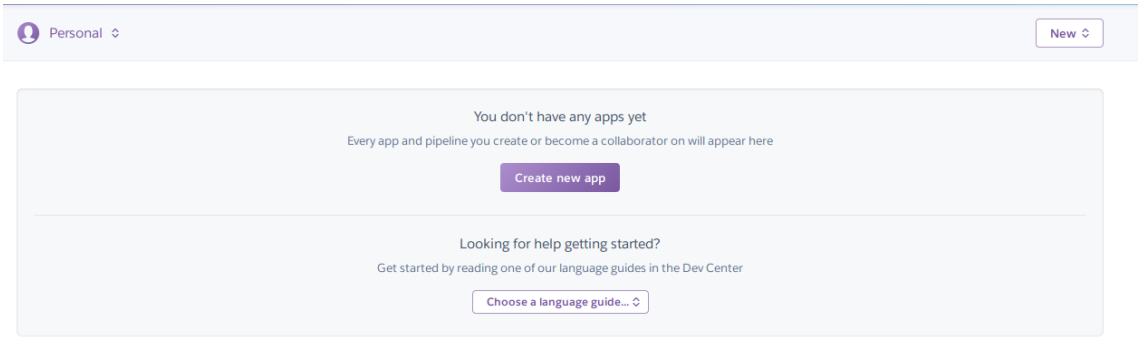

Figura 53 Crear aplicación en Heroku

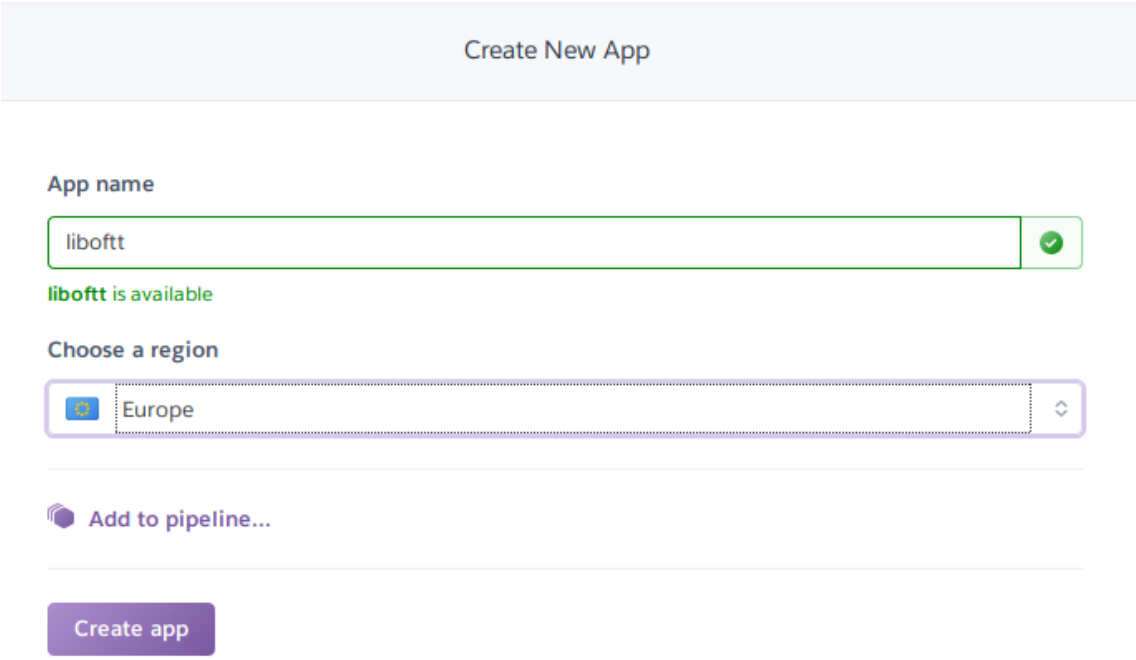

Figura 54 Crear configuración de Heroku

Para poder llevar a cabo la subida del código al servidor se utilizará *Github*, para ello *Heroku* nos pedirá que nos identifiquemos con nuestras credenciales en la página de *Github* y le demos permiso para usar nuestros repositorios. Una vez finalice el proceso, elegiremos el repositorio que queremos desplegar. A continuación, nos aparecerán las diferentes ramas del repositorio [Figura 55]. En nuestro caso de estudio realizaremos un despliegue de la rama de desarrollo de manera manual, antes tendremos que configurar las aplicaciones correctamente.

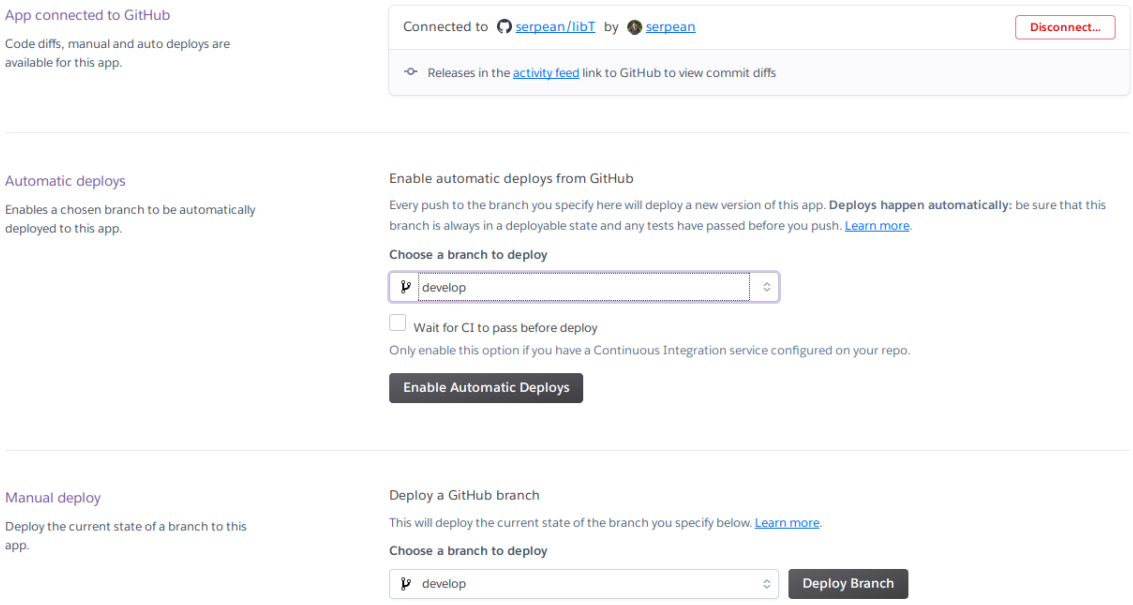

Figura 55 Configuración de la rama a desplegar en Heroku

Las configuraciones que tenemos que llevar a cabo serán diferentes en función del servicio. Esta configuración dependerá de:

- *Procfile*
- *Buildpacks*
- *Variables de entorno*

La única similitud que encontraremos entre ellos será el uso de *multiprocfile*[31], un *buildpack* que nos permitirá establecer más de un fichero *Procfile* en nuestro repositorio. En nuestro caso lo utilizaremos para establecer la ruta del fichro *Procfile* en cada servicio.

Despliegue de React

A la hora de desplegar nuestra aplicación de *React,* en primer lugar, tendremos que establecer los *buildpacks* utilizados. Heroku nos brinda la oportunidad de crear uno propio con toda la configuración o, como en nuestro caso, utilizar los que nos proporciona la comunidad.

[31] https://github.com/heroku/heroku-buildpack-multi-procfile

En el caso de React utilizaremos [Figura 56]: *multiprocfile,* del que ya hemos hablado, y *create-react-app-buildpack* [32] , para más información recomendamos visitar su repositorio. Un pequeño resumen de su funcionalidad es: construir la versión *minificada* de nuestro proyecto de *React* y generar un servidor *Nginx*.

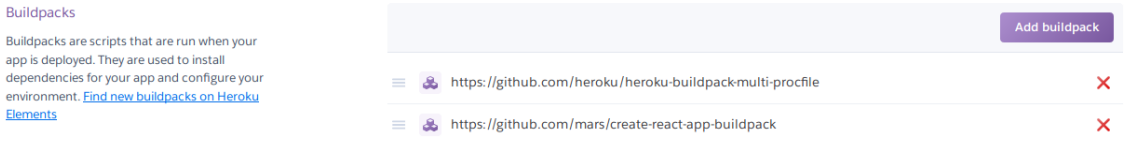

Figura 56 Buildpacks para React

Una vez establecidos los *buildpacks,* tendremos que aplicar su configuración. Dicha configuración se realiza a través de variables de entorno [Figura 57] y un fichero en la raíz del proyecto llamado *static.js [*Figura 58*].*

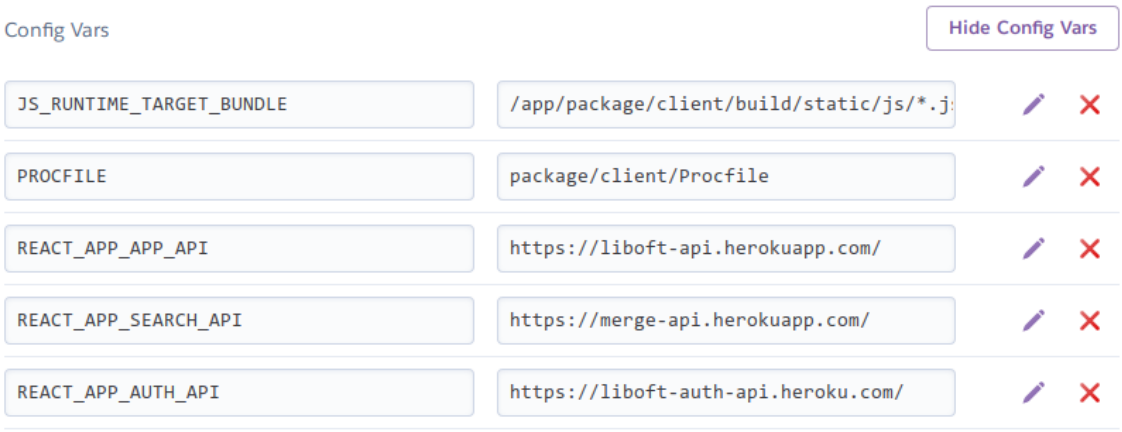

Figura 57 Variables de entorno de React en Heroku

- JS_RUNTIME_TARGET_BUNDLE: entorno de ejecución que *Nginx* utilizará.
- PROCFILE: ubicación del fichero *Procfile* de *React* en el proyecto.
- RACT_APP_<nombre de la API>: dirección de los distintos microservicios.

[32] https://github.com/mars/create-react-app-buildpack

```
{
  "root": "package/client/build/",
  "routes": {
    "/**": "index.html"
  }
}
```

Figura 58 Configuración del fichero static.js

Antes de poder iniciar el despliegue de este servicio debemos establecer el punto de entrada de la aplicación a través del fichero *Procfile*. En este caso iniciaremos el servidor de *Nginx* instalado.

web: bin/boot

Desplegando la API principal

El despliegue de las diferentes *APIs* de escritas en *NodeJS* tienen una implantación muy similar, pues todas se basan en los mismo *buildpacks,* una de ellas no utilizará la configuración de la base de datos y las variables de entorno en cada caso serán distintas.

Entre *buildpacks* escogidos, encontramos de nuevo el *multiprocfile*. A éste se le suma el paquete de *NodeJS* que encontramos por defecto en *Heroku [*Figura 59*]*.

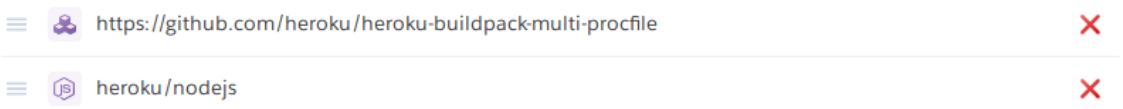

Figura 59 Buildpacks NodeJs

El siguiente paso será añadir el *add-on* correspondiente a una base de datos *MongoDB*. Para este ejemplo utilizaremos *mLab*. Para añadirla, nos dirigimos a la pestaña de recursos y elegimos el *add-on* correspondiente [Figura 60].

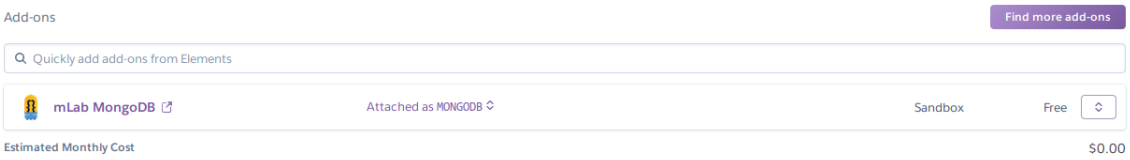

Figura 60 Incluir MongoDB en el proyecto

El último paso será configurar las variables de entorno y el fichero *Procfile* de manera adecuada. En nuestro caso utilizaremos las siguientes variables de entorno:

- MONGO_URL, donde estará ubicada la dirección de la base de datos
- AUTH_API, dirección de la API de autenticación
- PROCFILE (con la ruta: /package/server/Procfile). En el *Procfile,* encontraremos la siguiente orden:

web: node package/server/app.js

Desplegando la API de autenticación

El despliegue de esta API seguirá el mismo procedimiento que el anterior, con los siguientes parámetros de configuración:

- El nombre de la aplicación será liboft-auth-api
- La idea es utilizar el mismo clúster que la principal, en una base de datos diferente. También podríamos utilizar clústers diferentes.
- La variable de entorno PROCFILE será: /package/auth-api/Procfile
- En el fichero *Procfile,* encontraremos la siguiente orden: web: node package/auth-api/app.js

Desplegando la API de búsqueda

La configuración sigue los pasos de las anteriores, salvo las siguientes diferencias:

- El nombre de la aplicación será: merge-api
- No contamos con base de datos, y por lo tanto, variable de entorno relacionada.
- La variable de entorno PROCFILE será: /package/merge-api/Procfile
- Añadiremos una nueva variable de entorno con la clave de la API de búsqueda de OMDB: OMDB_API_KEY
- En el fichero *Procfile,* encontraremos la siguiente orden: web: node package/merge-api/app.js

Una vez terminada la configuración de todas las aplicaciones, encenderemos los todos los *Dynos,* ubicados en la pestaña de recursos [Figura 61].

Si el despliegue se ha realizado correctamente, tendríamos acceso a nuestra aplicación y a los distintos servicios a través de: <nombre de nuestra aplicación>.herokuapp.es.

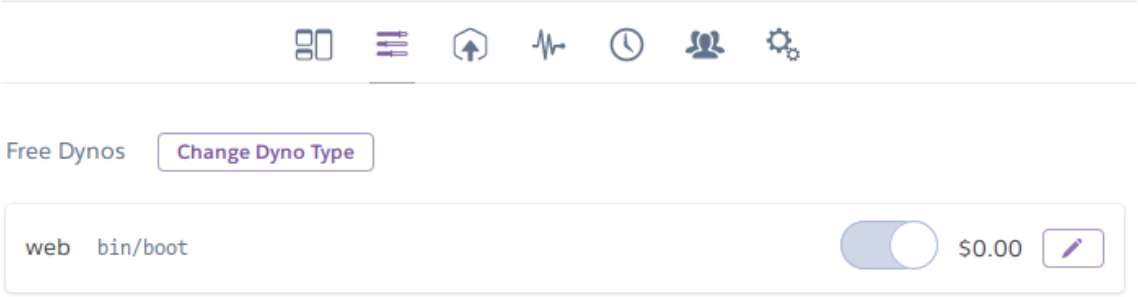

Figura 61 Dynos activos

12. Anexo 2: Despliegue en *Azure*

En este anexo se lleva el despliegue en *Azure,* la plataforma de despliegue como servicio en la nube de *Microsoft*.

El primer paso que tendremos que será el registro de una cuenta en la página de *Azure*. Una vez finalizado el registro, la plataforma nos proporcionará alrededor de doscientos dólares de crédito para realizar las pruebas y mantener nuestra aplicación activa. La pantalla que deberíamos encontrarnos al finalizar el proceso tendría que ser la siguiente [Figura 62]:

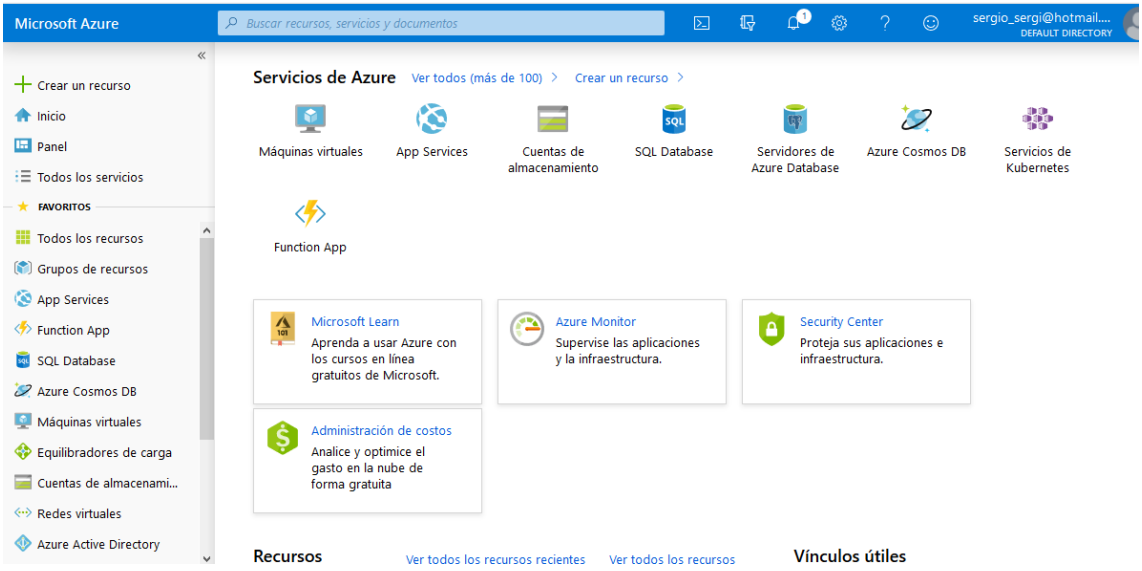

Figura 62 Página inicial del panel de control de Azure

Creando la base de datos

La base de datos que utilizaremos es la que acompaña a la plataforma. Su nombre es *CosmosDB,* no tenemos que entrar en confusión, pues funciona exactamente igual que una base de datos *MongoDB*.

Nos dirigimos al apartado *Azure Cosmos DB* del menú lateral. Para crear una nueva base de datos, pulsaremos el botón "Agregar".

Esto lanzará una secuencia de pestañas con las que configuraremos nuestra base de datos. En este paso configuramos un nuevo grupo de recursos [*Figura 63*].

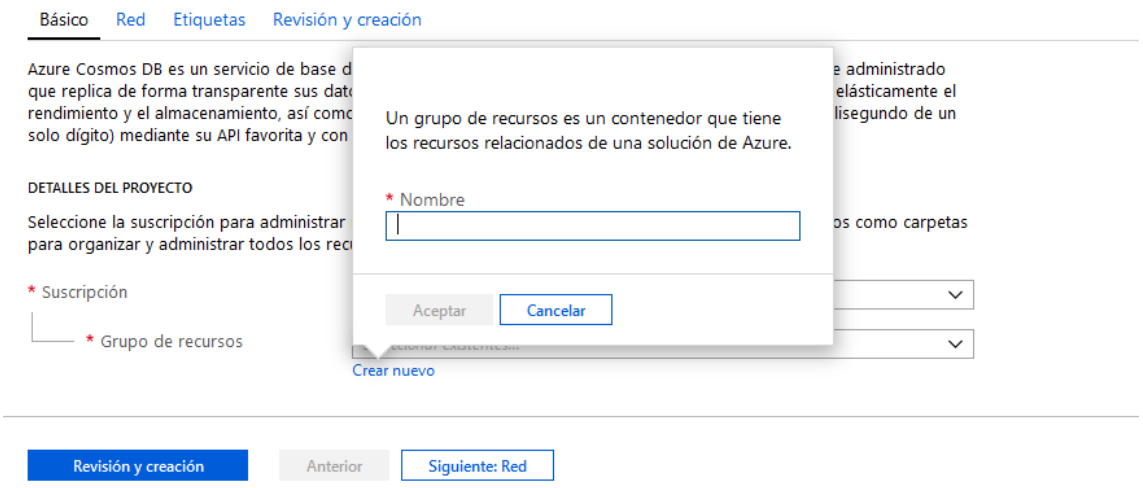

Figura 63 Creación de un grupo de recursos en Azure

El siguiente paso será la configuración básica [*Figura 64*]. Elegiremos la API de "Azure Cosmos DB para la API de MongoDB" y elegiremos la que más nos convenga por proximidad. En cuanto al resto, lo dejaremos todo por defecto para esta primera versión, posteriormente, podremos modificar todas las opciones proporcionadas. Una vez rellenada, podemos saltar directamente a revisión y

creación para finalizar el proceso. Revisaremos que todo es correcto y crearemos la base de datos.

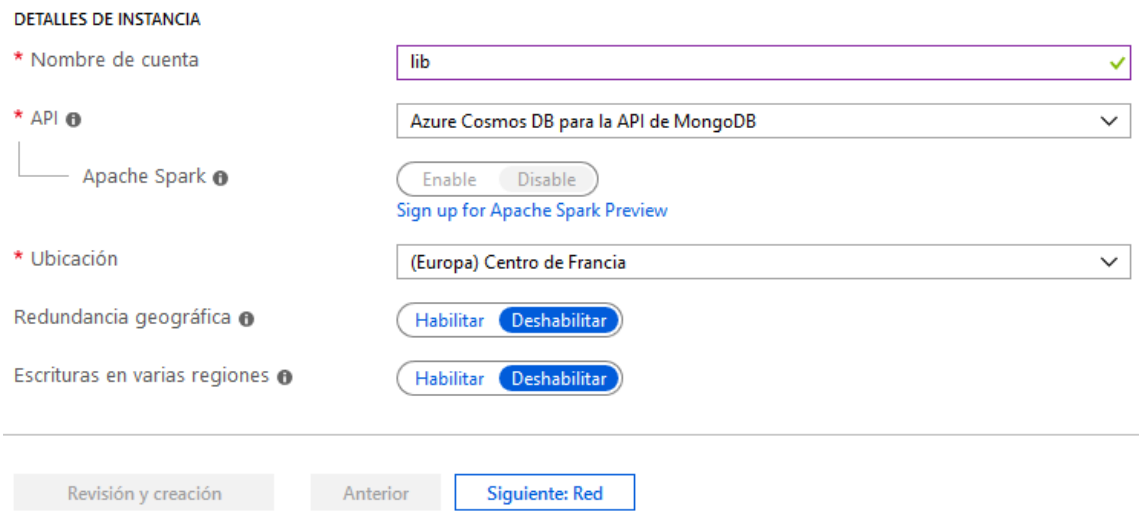

Figura 64 Configuración básica de la base de datos en Azure

Para concluir la configuración, nos dirigimos a "Características de vista previa" y activamos el protocolo de transferencia para poder conectar correctamente con los drivers de las distintas API [*Figura 65*].

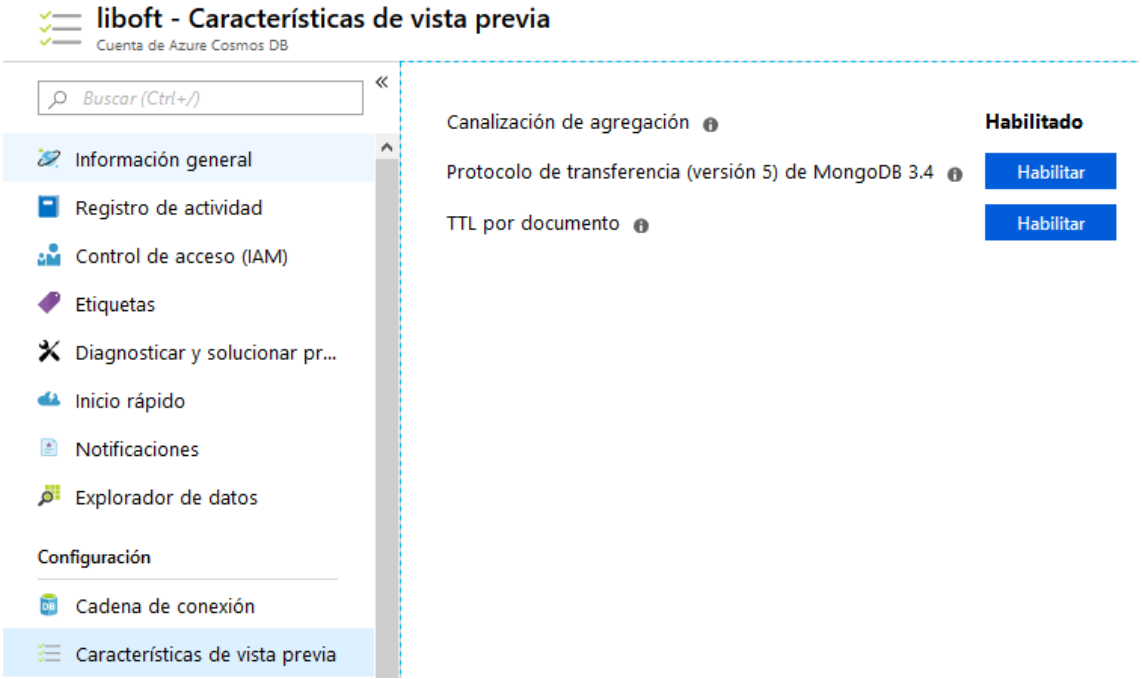

Figura 65 Activar protocolo de transferencia de CosmosDB

Desplegando React

Para llevar el despliegue de *React* usaremos la herramienta de "Cuentas de almacenamiento". De igual forma que en el paso anterior, agregaremos un nuevo recurso en este apartado.

En la configuración solo tendremos que agregar el nombre del recurso y el grupo al que pertenece, dejando el resto por defecto [*Figura 66*].

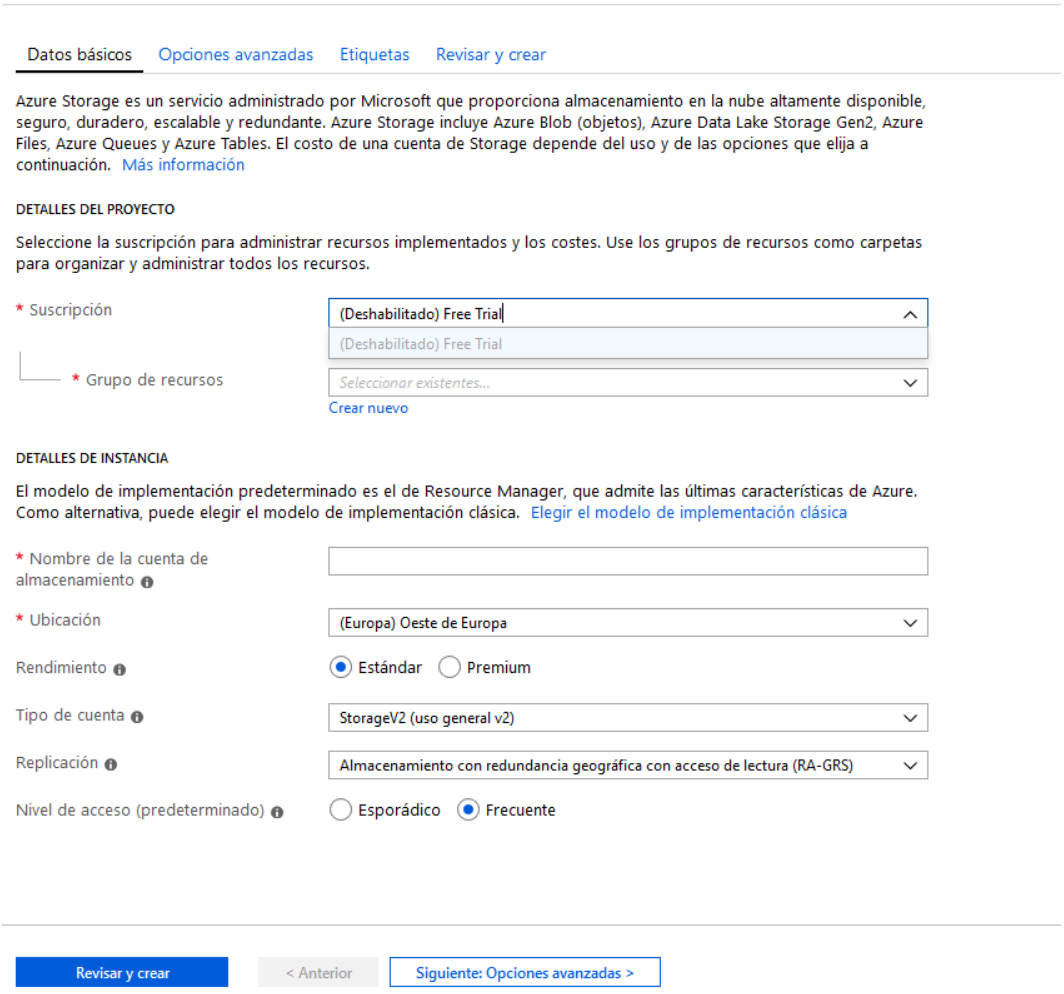

Figura 66 Configuración de React en Azure

Una vez tenemos configurado el entorno, nos vamos a nuestro proyecto en *Visual Studio Code*. Deberemos instalar la extensión para Azure Storage[33] y identificarnos con nuestra cuenta de Azure.

Una vez dentro, configuraremos el sitio web estático de la cuenta. Para ello:

33 https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-azurestorage

1. Insertaremos index.html como punto de entrada de la aplicación.
2. Insertaremos index.html como punto de error de la aplicación.
3. Generaremos la carpeta *build* de nuestro proyecto de *React* con el comando: *yarn run build*.
4. Desplegaremos la carpeta *build* tras generarla [*Figura 67*].

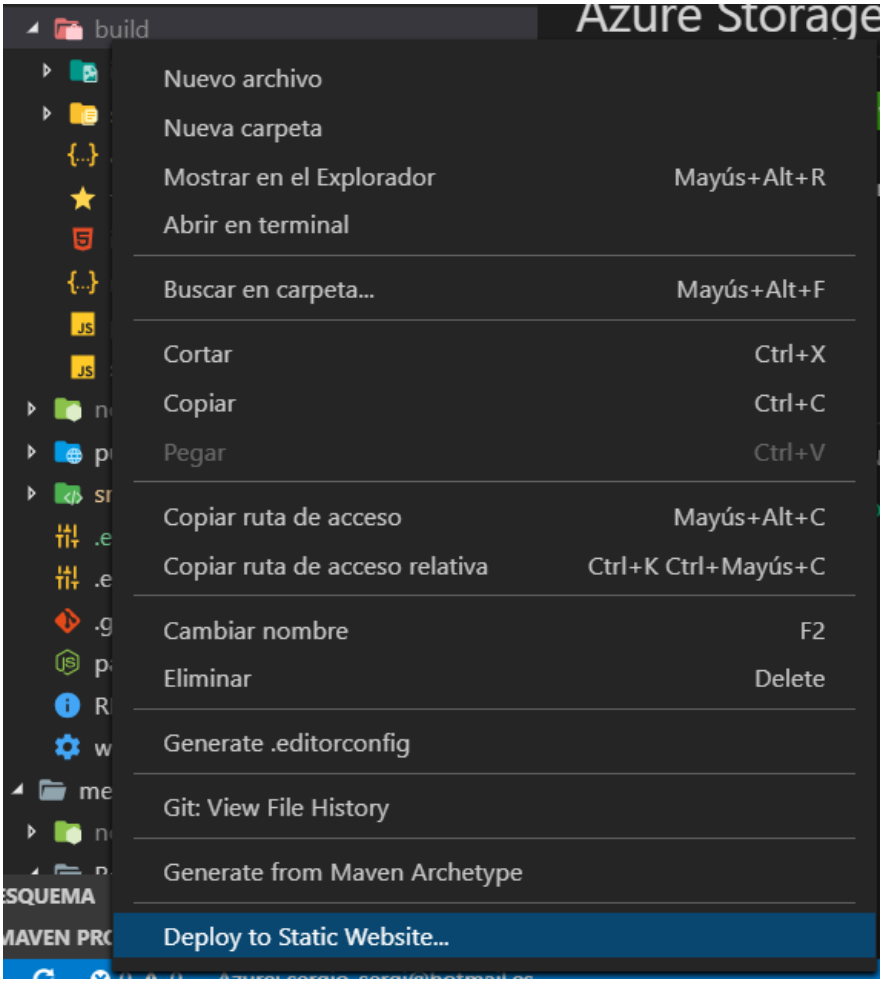

Figura 67 Desplegar React en Azure

Puntos comunes del despliegue de las APIs

Antes de comenzar cualquier despliegue, debemos agregar dos ficheros de configuración a cada uno de nuestros paquetes de API [*Figura 68*] [*Figura 69*].

```
{
  "name"        : "worker",
  "script"      : "./app.js",
  "instances"   : 1,
  "merge_logs"  : true,
  "log_date_format" : "YYYY-MM-DD HH:mm Z",
  "watch": true,
  "watch_options": {
    "followSymlinks": true,
    "usePolling"   : true,
    "interval"     : 5
  }
}
```

Figura 68 Fichero process.json

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer><webSocket enabled="false" />
    <handlers>
      <add name="iisnode" path="app.js" verb="*" modules="iisnode"/>
    </handlers>
    <rewrite>
      <rules>
        <rule name="NodeInspector" patternSyntax="ECMAScript" stopProcessing="true">
          <match url="^app.js\/debug[\/]?" />
        </rule>

        <rule name="StaticContent">
          <action type="Rewrite" url="public{REQUEST_URI}"/>
        </rule>
        <rule name="DynamicContent">
          <conditions>
            <add input="{REQUEST_FILENAME}" matchType="IsFile" negate="True"/>
          </conditions>
          <action type="Rewrite" url="app.js"/>
        </rule>
      </rules>
    </rewrite>

    <security>
      <requestFiltering>
        <hiddenSegments>
          <remove segment="bin"/>
        </hiddenSegments>
      </requestFiltering>
    </security>
    <httpErrors existingResponse="PassThrough" />
    <iisnode watchedFiles="web.config;*.js"/>
  </system.webServer>
</configuration>
```

Figura 69 Configuración extra de las APIs en Azure

El primer paso del despliegue será crear un *App Service,* ubicado en el panel lateral. Agregaremos un servicio por cada una de nuestras aplicaciones. Para ello, completaremos la información básica [*Figura 70*]. Los nombres de cada una de ellas será el siguiente:

- liboft
- merge-api
- liboft-auth-api

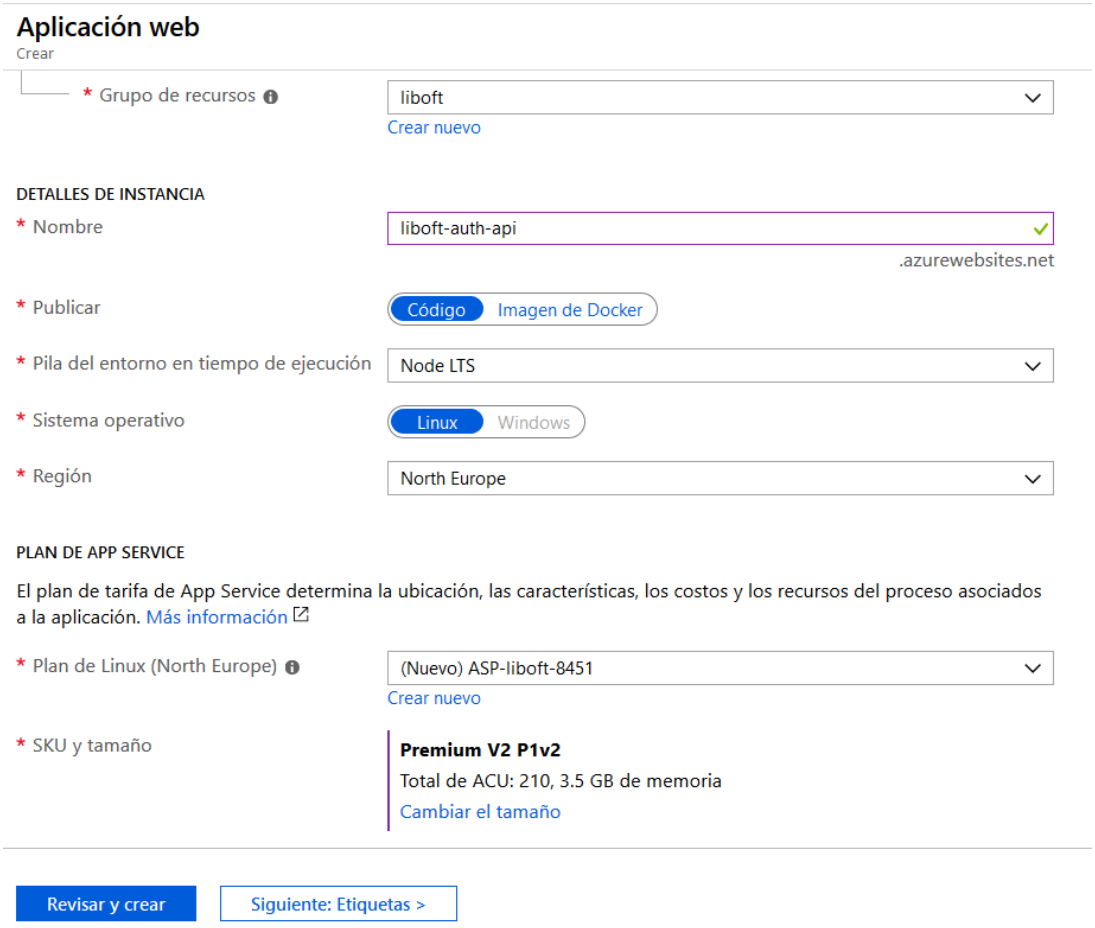

Figura 70 Configuración básica de un App Service

Al finalizar este paso, nos dirigiremos al centro de implementación. La manera de desplegar nuestro código será a través de *Github*. Para ello vincularemos nuestra cuenta con Azure. Al finalizar la autenticación, elegiremos el repositorio y la rama a desplegar.

Una vez finalizado este proceso en cada una de las API podemos pasar a configurar los parámetros en cada una de ellas. La única configuración extra que

tendremos que hacer serán las variables de entorno y políticas de CORS[34] (Agregando el servidor de pruebas y el nombre de la página web estática [*Figura 71*]).

Figura 71 Configuración de CORS en Azure

Para poder definir variables de entorno nos dirigimos al apartado de configuración de cada uno de nuestros servicios.

Configuración la API de la aplicación principal

- MONGO_URL: dirección de la base de datos proporcionadas por CosmosDB
- AUTH_API, dirección de la *API* de autenticación

Configuración la API de autenticación

- MONGO_URL: dirección de la base de datos proporcionadas por *CosmosDB*
- APP_API, dirección de la *API* de principal

Configuración la API de búsqueda

- OMDB_API_KEY: credenciales necesarias para utilizar la *API* de *OMDB*.

[34] https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS