Universitat Oberta de Catalunya

uoc.edu

Toshokan. Tu biblioteca de anime

Javier Valor Martínez
Grau d'Enginyeria Informàtica
Desenvolupament Web

Gregorio Robles Martínez

12/06/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Toshokan. Tu biblioteca de anime*
Nombre del autor:	*Javier Valor Martínez*
Nombre del consultor/a:	*Gregorio Robles Martínez*
Nombre del PRA:	*Gregorio Robles Martínez*
Fecha de entrega (mm/aaaa):	*06/2020*
Titulación:	*Grau d'Enginyeria Informàtica*
Área del Trabajo Final:	*Desenvolupament Web*
Idioma del trabajo:	*Castellano*
Palabras clave	*Programación, TFG, React*

Resumen del Trabajo (máximo 250 palabras):

Aplicación para crear tu propia biblioteca de Anime. Permite hacer búsquedas, ver diferentes categorías según género de la obra, ver la biblioteca del usuario, etc.

Esta aplicación obtiene los datos a través de una API (Kitsu API) y registra las bibliotecas de los usuarios en nuestro sistema.

Para la gestión, se ha seguido una metodología Kanban, donde se desglosaban las historias de usuario en tareas, puestas en una pizarra con cuatro columnas: *backlog*, *in progress*, *test* y *done*.

Para el desarrollo quería haber utilizado una metodología de desarrollo guiado por pruebas (TDD – Test-driven development), donde involucra dos prácticas, escribir primero los tests y la refactorización. Pero por problemas de tiempo no he podido realizarla.

El proyecto une diversas capas y tecnologías, por un lado, una parte servidor que interactúa con la base de datos y por otro lado la parte cliente que hace peticiones al servidor para trabajar con esos datos. Además, he añadido una API externa que proporciona la información para abastecer al cliente.

Finalmente, el desarrollo de este proyecto ha sido muy interesante ya que he podido poner en práctica diversas tecnologías que de otra forma no las hubiera utilizado, como por ejemplo Docker para desplegar diferentes aplicaciones dentro de contenedores de software, o Node para montar un servidor, que tenía curiosidad de ver cómo funcionaba. Todo ello ha derivado en una aplicación muy ágil e interesante.

Abstract (in English, 250 words or less):

Application to create your own anime library. It allows you to search, see different categories according to the genre of the work, see the user's library, etc.

This application obtains the data through an API (Kitsu API) and registers the users' libraries in our system.

For management, a Kanban methodology has been followed, where user stories were broken down into tasks, placed on a board with four columns: backlog, in progress, test and done.

For development I wanted to have used a Test-Driven Development (TDD) methodology, which involves two practices, writing the tests first and refactoring. But due to time problems I have not been able to do it.

The project unites various layers and technologies, on the one hand, a server part that interacts with the database and on the other hand the client part that makes requests to the server to work with that data. Also, I have added an external API that provides the information to supply the client.

Finally, the development of this project has been very interesting since I have been able to put into practice various technologies that would not have otherwise been used, such as Docker to deploy different applications inside software containers, or Node to mount a server, I was curious to see how it worked. All this has resulted in a very agile and interesting application.

Índice

Lista de figuras

No se encuentran elementos de tabla de ilustraciones.

1. Introducción

1.1 Contexto y justificación del Trabajo

Inicialmente, la idea de este proyecto vino para poder organizarme el listado de series que tenía o había visto, ya que actualmente hay muchas plataformas de video bajo de manda y algunas compradas en formato físico. Así con este proyecto podemos aunar todas desde un único espacio. Seguramente ya habrá otras aplicaciones que lo hagan, pero no he encontrado ninguna y quería hacer mi propia aplicación obteniendo los datos de una fuente externa siempre actualizada.

1.2 Objetivos del Trabajo

- Conectar, configurar y utilizar una API externa.
- Listar y facilitar el uso de la aplicación, haciéndola lo más accesible posible.
- Crear una base de datos simple para registro de usuarios y sus bibliotecas.
- Crear un servidor en javascript, utilizando como tecnología Node.
- Utilizar y manejar Docker para desplegar las aplicaciones.
- Desarrollo por componentes, con la biblioteca de javascript React.

1.3 Enfoque y método seguido

El enfoque inicial era desarrollar una aplicación nueva, utilizando como *data* una API externa que alimente la aplicación de datos. Esta estrategia me ha permitido centrarme en el producto más que en rellenar

una base de datos con las diferentes series que hay, permitiendo así tener la información siempre actualizada.

Esto me ha permitido centrarme en el diseño y la usabilidad del producto, así como en la innovación, haciendo uso de tecnologías y herramientas nuevas que me permiten acercarme a un marco más actual del mercado laboral.

1.4 Planificación del Trabajo

Para realizar este trabajo disponemos de un solo recurso, para desarrollar las siguientes tareas:

- **Definición de objetivos**: Redactar el listado de objetivos a los que aspiramos a llegar.
- **Análisis de requerimientos y desglose de componentes**: Ver que necesitamos y como estructuramos el proyecto. Crear historias de usuarios y desglosarlas en tareas. Creación del tablero Kanban.
- **Diseño funcional y usabilidad**: Crear un prototipo de diseño y establecer pautas de usabilidad para el proyecto.
- **Configuración del entorno, de la app y del repositorio Git**: Valorar y configurar en que entorno se va a realizar el trabajo, crear el repositorio y establecer el *work-flow* del mismo, y crear la aplicación con sus ajustes de configuración, así como instalar los paquetes para realizar este trabajo.
- **Estudio y conexión con la API**: Investigar como funciona esta API y realizar tareas de conexión con la misma.
- **Desarrollo de la aplicación**: Configurar el servidor, conexión con la base de datos y con el servidor, crear componentes, maquetar el resultado, etc.
- **Evaluación de código**: Usar herramientas para obtener una evaluación del código del proyecto y ver en que puntos se puede mejorar.

- **Finalización de la memoria**: Acabar de realizar el trabajo de documentación de la memoria.

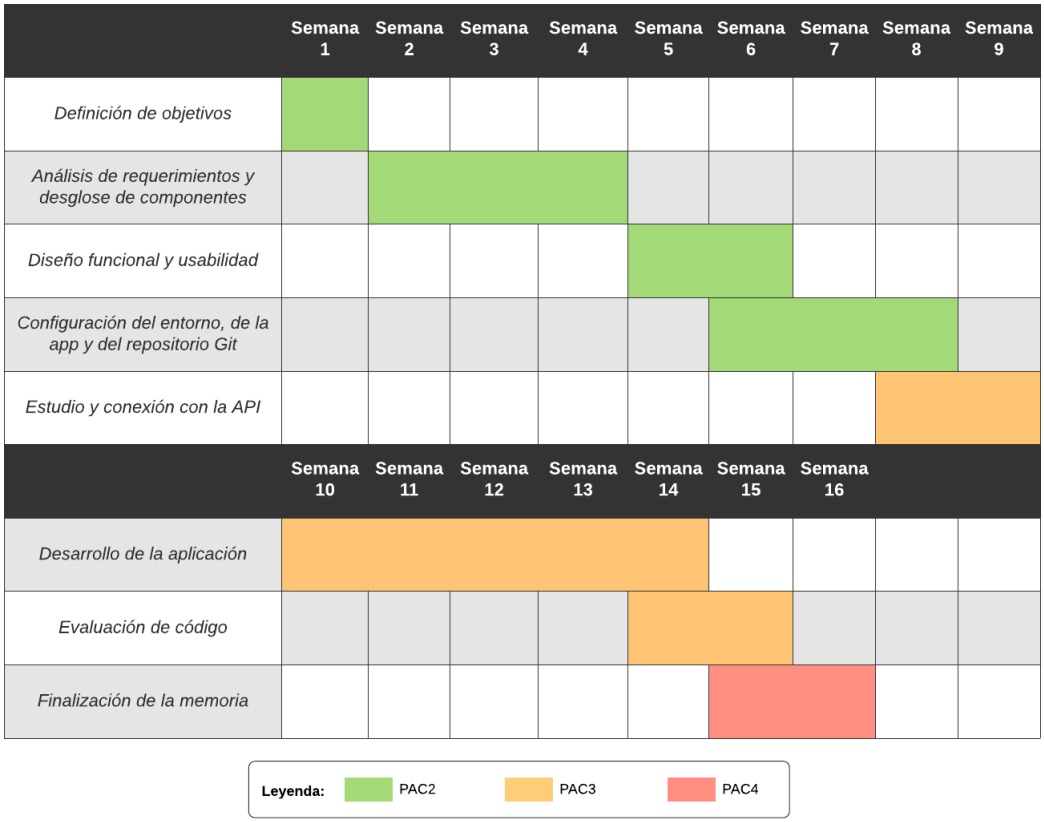

Figura 1. Diagrama de Gantt

1.5 Breve sumario de productos obtenidos

Servidor que hace de API con conexión a base de datos donde se almacenan los usuarios con sus bibliotecas.

Aplicación que permite a los usuarios registrarse para ver un gran volumen de series proporcionadas por una API, que les permite seleccionar las series que quieren añadir o eliminar de su biblioteca.

1.6 Breve descripción de los otros capítulos de la memoria

- **Tecnologías aplicadas**. Breve listado de tecnologías aplicadas en el proyecto.
- **Planificación y metodología Kanban.** Uso de la metodología Kanban para planificar y desglosar las historias de usuario en tareas. Permitiendo llevar un control de lo que se esta haciendo, lo que queda por hacer y lo que está terminado.

- **Diseño y prototipado de la aplicación.** Crear un prototipo con una idea de diseño de cómo debería quedar la aplicación.
- **Definición de entorno**. Hablar de la decisión de elegir Docker para desplegar las diferentes aplicaciones y la necesidad de tener un entorno de trabajo ágil, estable y fácilmente escalabre.
- **Configuración y workflow del repositorio Git.** Como se ha establecido el *workflow* del repositorio basado en GitFlow para mantener un orden dentro del proyecto.
- **Configuración de la aplicación cliente.** Como esta organizada y configurada la parte front-end del proyecto
- **Arquitectura de la aplicación.** Entramos más profundamente en los patrones de diseño y arquitectura de la aplicación.
- **Arquitectura de componentes**. Explicación del ciclo de vida de los componentes y como interactúan con el entorno.
- **Configuración del servidor y base de datos.** Breve explicación de como esta configurado el servidor y como está montada la base de datos y sus relaciones.
- **Resultados.**

1.7. Enlaces del proyecto

- Repositorio Git: https://github.com/icecap2k/anime-project
- Diseño en Figma: https://www.figma.com/file/ooRWy3xaoAhWi7laROJGYf/Untitled?node-id=0%3A1

2. Tecnologías y herramientas aplicadas

En este capítulo hablaremos de las tecnologías aplicadas en el proyecto, pero sin entrar en las diferentes alternativas de cada tecnología o herramienta aplicada para no extenderse en exceso, ya que existen muchas opciones distintas.

La base del proyecto es **JavaScript**, utilizando la versión **ES6** [1] (ECMAScript 6), que se utiliza tanto en la parte cliente como en la parte servidor, ciertamente, para la parte cliente se nos ocurren pocas que sean tan versátil como esta, pero para la parte servidor si, y mucho mejor en nuestra opinión. En este caso podríamos haber elegido PHP por comodidad y conocimiento, o Python por curiosidad y obligarnos a aprender a usarlo, pero en este caso hemos decidido usar JavaScript con **NodeJS** [2], que, pese a no haberlo utilizado nunca para el servidor, nos invadía la curiosidad de ver cómo funcionaba.

La aplicación cliente está desarrollada en **ReactJS** [3], que es una biblioteca de JavaScript para construir interfaces de usuario de forma sencilla, está basado en componentes encapsulados que manejan su propio estado, lo que facilita la reutilización de componentes y el dinamismo de los mismos. También hemos utilizado **Webpack**, un *module bundle* que nos permite gestionar todos los recursos necesarios para que nuestra aplicación funcione, tomando unos ficheros de entrada, después pasan por el transpilador de **Babel** (que hace que el código programado en ES6 sea compilado en JavaScript puro para que los navegadores puedan procesarlo) y saca unos ficheros de salida en forma de *bundle* para el uso de la aplicación.

Para proporcionar estilos a la aplicación he utilizado una librería llamada **styled-components** que permite crear componentes aplicándoles estilos, así como pasar propiedades de los componentes para poder hacerlos más

dinámicos, también permite extender los estilos a otros componentes con modificaciones.

La aplicación servidor está desarrollada en **NodeJs**, concretamente con el framework Express [4]. Node es un entorno que trabaja en tiempo de ejecución, de código abierto y multi-plataforma que permite crear herramientas de lado servidor. Express es el framework más popular de Node y proporciona:

- Escritura de manejadores de peticiones con diferentes verbos HTTP en diferentes rutas URL.
- Integración con motores de renderización de “vistas” para generar respuestas mediante la introducción de datos en plantillas.
- Establecer ajustes de aplicaciones web, como el puerto de conexión o la localización de las plantillas.

Lo bueno de **Express** es que es bastante minimalista, pero permite añadir diferentes paquetes *middleware* para poder abordar diferentes problemas, dando así libertad al desarrollador para añadir lo necesario para su proyecto.

Para el control de versiones he decidido utilizar **Git**, hay otros sistemas como CVS o SVN, pero para el tipo de proyecto a realizar el más recomendable es Git sin dudas, quizás si en el proyecto hubiera tenido que trabajar mucho con ficheros binarios, me hubiera decantado por SVN. Además, estoy acostumbrado a trabajar con Git por línea de comando y me parece muy fácil y útil su sistema de ramas para mantener ordenado el desarrollo del proyecto. Puede que su curva de aprendizaje sea más elevada y los flujos de trabajo entre los desarrolladores un poco más complejos, pero una vez dominado esto no quieres probar otra cosa. En un próximo capítulo hablaré más extendidamente de como configurar y establecer unas normas de uso en Git.

Añado una breve comparativa entre los diferentes sistemas de control de versiones

			
Actualizado recientemente	✗	☑	☑
Software libre	☑	☑	☑
Licencia	GPL	Apache/BSD	GPL
Tipo de sistema	CVCS Centralized Version Control System	CVCS Centralized Version Control System	DVCS Distributed Version Control System
Curva de aprendizaje	Baja	Baja	Alta
Manejo de Ramas	✗ Complejo	✗ Complejo	☑ Simple
Conexión a internet	✗ Necesaria	✗ Necesaria	☑ No necesaria
Fiabilidad	✗ Menor	☑ Mayor	✗ Menor

Figura 2. Cuadro comparativo sistema de control de versiones

3. Planificación y metodología Kanban

Para este proyecto he decido adoptar una metodología **Kanban** en la planificación, y es que, dentro de la metodología Agile nos encontramos con el método Kanban y el método Scrum, ambos dividen las tareas complejas en historias de usuarios y se visualizan en un flujo de trabajo, normalmente en un tablero Kanban.

El método Scrum permite a los equipos a generar un producto en cada finalización de un Sprint, y así hacer pequeños entregables al cliente en cada 2 o 3 semanas que puede durar un Sprint. En mi caso, he decidido utilizar el método Kanban ya que se basa en el desarrollo y entrega continuos, y así mantener un listado de tareas a realizar durante todo el periodo de la asignatura, añadiendo tareas nuevas que vayan surgiendo y organizado el tablero en tres columnas, Backlog (tareas pendientes), En curso (tareas en desarrollo) y Listo (Tareas finalizadas).

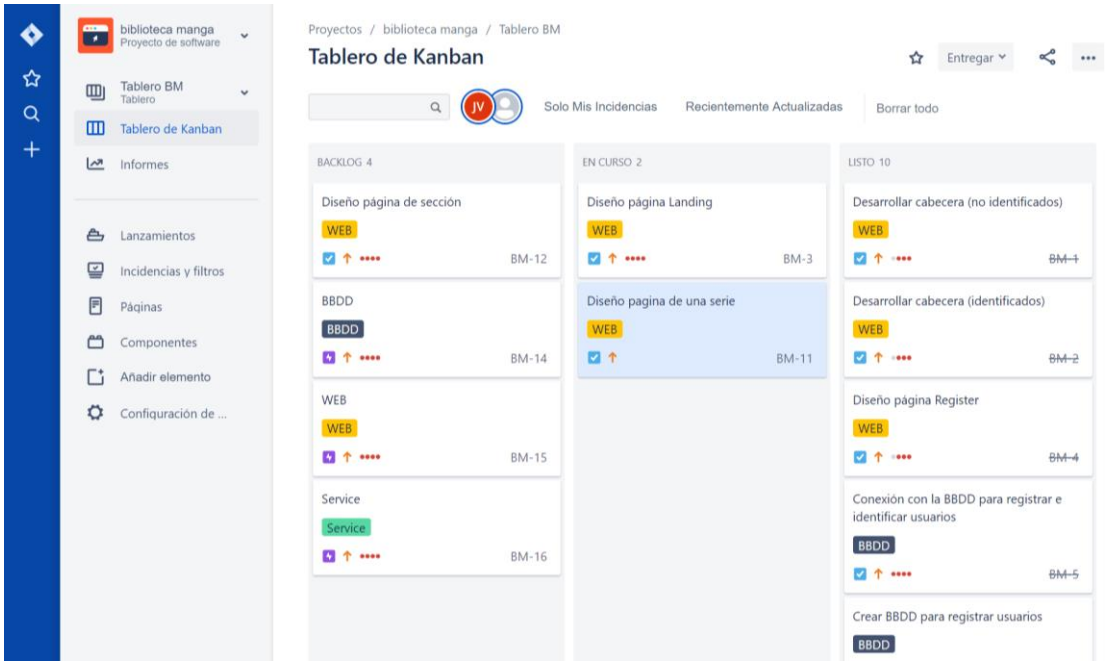

Figura 3. Tablero Kanban del proyecto

Gracias a esta metodología, puedes reducir las historias de usuario en pequeñas tareas, puedes visualizar fácilmente que queda por desarrollar y además te permite llevar un orden a la hora de desarrollar, ya que a veces

pecamos de estar haciendo una cosa, ver otra y cambiar sin darse cuenta, de esta forma te centras en una tarea hasta finalizarla.

Otro tema es enlazar el identificador de las tareas con las ramas del Git, esto es muy útil para ver saber porque has hecho algo en el código o al revés, para ver en una tarea como se ha desarrollado o que ficheros se han visto implicados.

4. Diseño y prototipado de la aplicación

Al inicio del proyecto, decidimos que no se empezaría a implementar código sin antes crear un diseño de la misma, porque muchas veces acabas haciendo el doble de trabajo por no ver y pensar con calma que es lo que quieres.

Por eso decidimos crear un diseño (dentro de nuestras limitaciones) del producto que queríamos desarrollar. Para ello estuve investigando como y donde realizar este diseño/prototipo y creo que la mejor herramienta para ello es Figma. **Figma** proporciona un sinfín de herramientas fáciles de usar y que, sin ser un especialista en la materia, te permite crear diseños y prototipos muy interesantes, además añade un pequeño tutorial a base de plantillas para conocer que puede llegar a hacer, y aunque me hubiera gustado profundizar más y realizar un diseño para una versión para móviles, estamos contentos con el trabajo realizado con la misma.

A continuación, expongo las imágenes del primer prototipo.

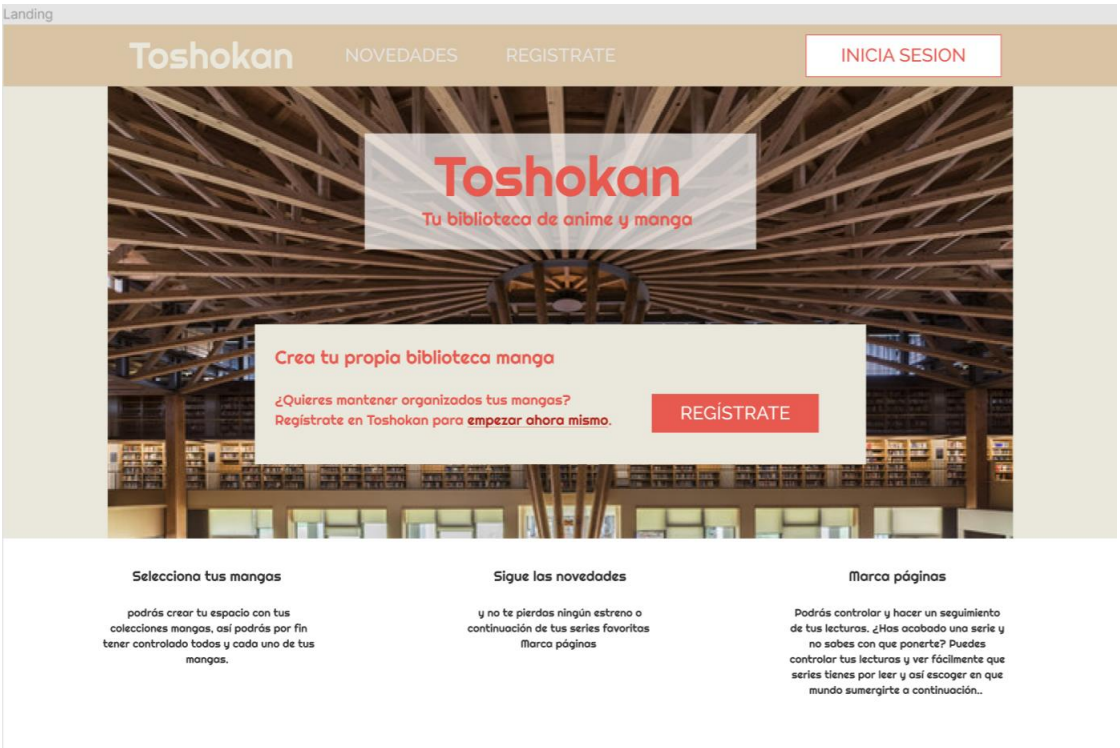

Figura 4. Prototipo página Landing

Figura 5. Prototipo modal de registro

Figura 6. Prototipo modal de acceso

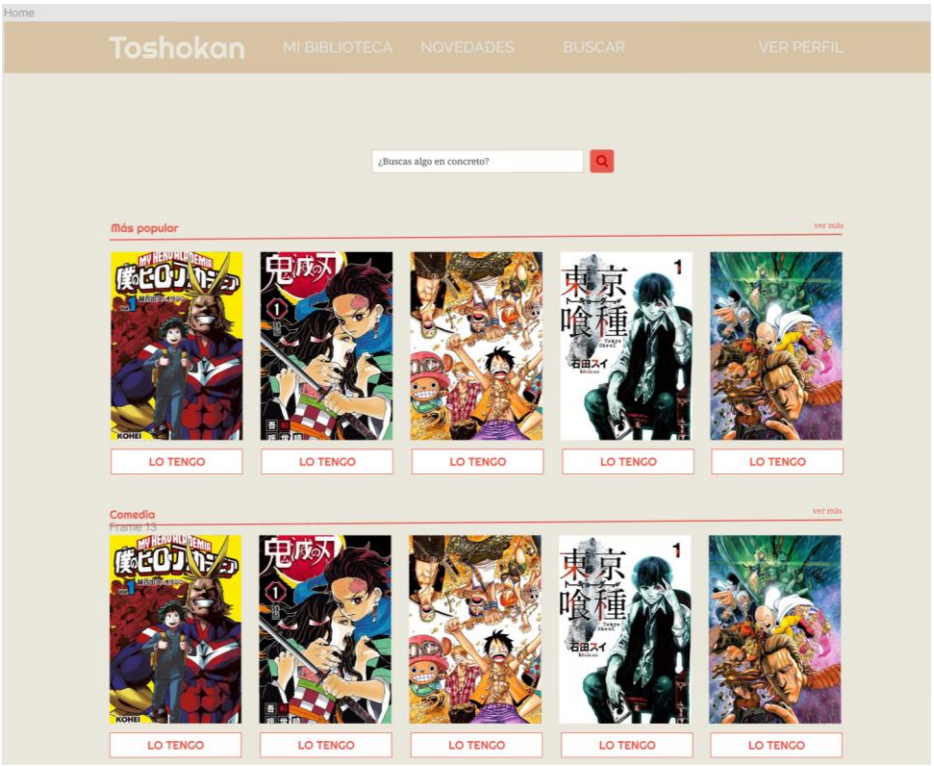

Figura 7. Prototipo página Home
Fuente. Projecto en Figma:
https://www.figma.com/file/ooRWy3xaoAhWi7laROJGYf/Untitled?node-id=0%3A1

Investigando más sobre esta herramienta, es muy util para grupos de trabajo sobre un mismo diseño, ya que permite añadir comentarios, edición en grupo online, etc.

Como se puede comprobar, en el producto final ha habido cambios como las pantallas de registrar usuarios y la de sign up, colores y disposición de los elementos.

5. Definición del entorno

En este capítulo queremos hablar de la decisión de elegir Docker para desplegar las diferentes aplicaciones. La necesidad de tener un entorno de trabajo ágil, estable y fácilmente escalabre era una de las principales prioridades.
He sopesado diferentes opciones como montar maquinas *vagrants* o directamente alojar la base de datos y el servidor en un *hosting*.

Vagrant
Es una herramienta para la creación y configuración de entornos de desarrollo virtualizados. Me hubiera gustado aprender a utilizar esta herramienta junto con Chef, pero me parecía una curva de aprendizaje demasiado elevada para el beneficio que podría obtener, aún así le veo mucho potencial a esta herramienta.

Hosting
El problema del hosting, es encontrar alguno que sea gratuito y permita utilizar Node, después está el tema del *deploy* y que lo permita, no me he decantado por esta opción porque quería tener un entorno funcional en local, que permitiera trabajar con las diferentes partes de la aplicación sabiendo que, en una fase final, se podría subir este código a un servidor sin perder tiempo y con la tranquilidad de que todo funcionaría igual que en producción.

Docker
Docker es un proyecto de código abierto, permite crear contenedores, que son algo así como pequeñas máquinas virtuales. Lo importante es que sirve para probar sistemas o aplicaciones en un entorno seguro e igual al de producción, permitiendo reducir tiempos de pruebas y adaptaciones a cambios de hardware desde un entorno de test o desarrollo a otro de producción.
Sus principales características son:

- **Portabilidad**: el contenedor Docker permite desplegarse en cualquier otro sistema que soporte esta tecnología.

- **Ligereza**: En comparación con otros sistemas de virtualización, el peso de este es ridículo, por ejemplo, con VirtualBox, cualquier imagen de Ubuntu pesa entorno de 1Gb, mientras con Docker, un Ubuntu, con Apache y una aplicación, no llega a los 200Mb.
- **Autosuficiencia**: Docker se encarga de la gestión del contenedor y de las aplicaciones, mientras que un contenedor tiene únicamente la configuración, las librerías y los archivos necesarios para desplegar las funcionalidades del mismo.

Configuración de Docker en el proyecto

En este proyecto, tenemos un cliente en ReactJS, un servidor en NodeJS y una base de datos en MySql, además he añadido también en Docker un PhpMyAdmin para visualizar el estado de las tablas.

Para la configuración del Docker, se puede ver el fichero Docker-compose.yml. En este fichero, hemos colocado un grupo de variables que añadiremos a los tres principales contenedores, con datos de conexión de la base de datos (user, pass y database), además, tenemos la configuración de los cuatro contenedores descritos anteriormente:

- **MySQL**: utiliza el puerto 3306 que es el usado por defecto en mysql, *restard* con *unless-stopped*, para evitar perder datos de la base de datos cuando se paren los servicios. Y, por último, toma inicialmente el fichero db.sql con la creación de las tablas y algunos datos de prueba.
- **PhpMyAdmin**: como he dicho antes, no es necesario, pero si útil para ver la información alojada en la base de datos, en este caso la url es http://localhost:8080/
- **Server**: aquí se hace un deploy de la aplicación en NodeJS que hace de API entre cliente y base de datos. Como dato importante en la configuración, MYSQL_HOST_IP nos permite resolver la dirección PI del servicio mysql para asignarla a la conexión mysql.
- **Client**: aquí se hace el deploy de la aplicación en ReactJS, el puerto por defecto de esta es el 3000. La variable de entorno NODE_PATH

permite que la aplicación reconozca su carpeta *src* como punto de partida para que funcionen los alias.

Como vemos, para los dos primeros contenedores utilizamos las imágenes de mysql y de PhpMyAdmin del repositorio de Docker, mientras que para el server como para el client utilizamos nuestro propio código, para ello utilizamos un fichero Dockerfile tanto en la carpeta server como en la de client con instrucciones para crear las imágenes de cada una.

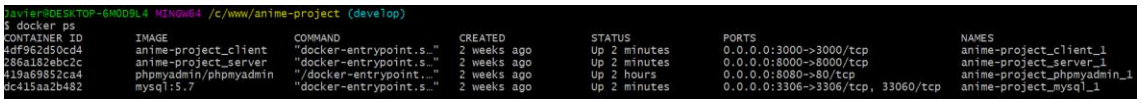

Figura 8. Listado de contenedores Docker

6. Configuración y *workflow* del repositorio Git

Ya hemos hablado antes de Git, ahora quiero explicar como he establecido el *workflow* del mismo basado en GitFlow [5]. Para mantener un orden dentro del proyecto, inicialmente cree dos ramas importantes, develop y master. La rama master es la que tiene la última versión o *release* estable, mientras que en develop se suben las ramas de las tareas en desarrollo, cuando ya hay varias tareas realizadas y el funcionamiento de la aplicación es estable, se hace un merge hacia master.

Una tarea, una rama, con este principio, cada vez que hay que realizar una tarea nueva, hemos de ir a la rama *develop* y hacer *pull* para bajar los cambios que pueda haber, a partir de aquí creamos una rama nueva con el nombre del identificador de la tarea del Jira (Nuestro tablero Kanban), por ejemplo, BM-1. A continuación, podemos trabajar en este rama, y cada vez que hagamos un *commit*, primero se pone el nombre identificador de la tarea (BM-1), después el estado de la tarea o el *commit*, como por ejemplo, WIP (Work In Progress), IMP (Improvement), US (Update Submodule) o FIX (Fixes), y después una breve descripción de lo que se ha hecho, siempre en inglés. Una vez has finalizada la tarea (*commit* y *push* en la rama de la tarea), volvemos a la rama *develop* por si hay algún cambio, bajarlo, volvemos a la rama de nuestra tarea y hacemos *merge* de *develop*, aquí resolvemos los posibles conflictos de código que nos encontremos, comprobamos que todo funciona con normalidad y ya podemos pasar la tarea a la rama de *develop*.

Puede que escrito suene un poco complicado, pero de esta forma te aseguras que todo lo que haces en tu tarea lo tienes en tu rama, y nunca se pasa nada a la rama de *develop* que pueda crear conflictos, ya que los conflictos de código se resuelven en la rama de la tarea. Esto en grupos a partir de 3 personas es muy útil y necesario para tener un control de versiones, y un repositorio ordenado, limpio y fácilmente escalable. También se pueden añadir mas ramas troncales como una de *testing* (para un entorno de testeo) o una rama para ir realizando una versión nueva de la aplicación. También es útil tener las ramas

de *develop* y *master* separadas, por si hubiera algún fallo critico en producción, pero no se quiere subir lo que hay aun en desarrollo, para estos casos, se puede sacar una rama de master para hacer un *hotfix* que posteriormente se subiría a *master* y, para evitar conflictos, también se pasaría a *develop*.

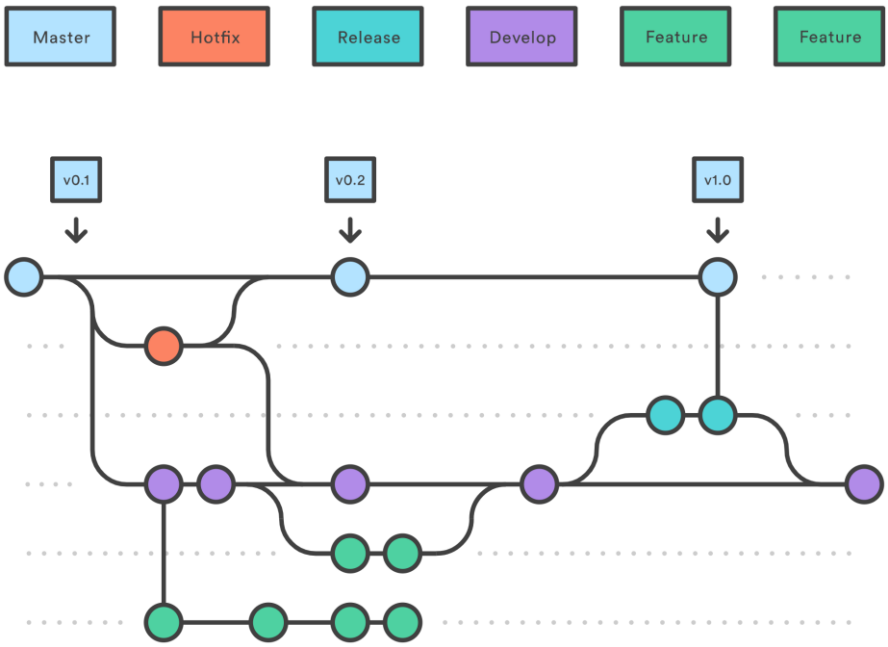

Figura 9. Ejemplo de ramas en un proyecto.

Fuente: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

7. Configuración de la aplicación cliente

Como hemos dicho con anterioridad, la parte cliente está desarrollada en ReactJS, utilizando el transpilador de Babel y module bundle de Webpack. Para instalar todo esto en la aplicación he utilizado NPM como sistema de gestión de paquetes de Node. Es por eso que el fichero más importante para que esto funcione es el package.json donde se almacena toda la información del proyecto, con los paquetes para desarrollo diferenciados de las dependencias para el producto y con su versión.

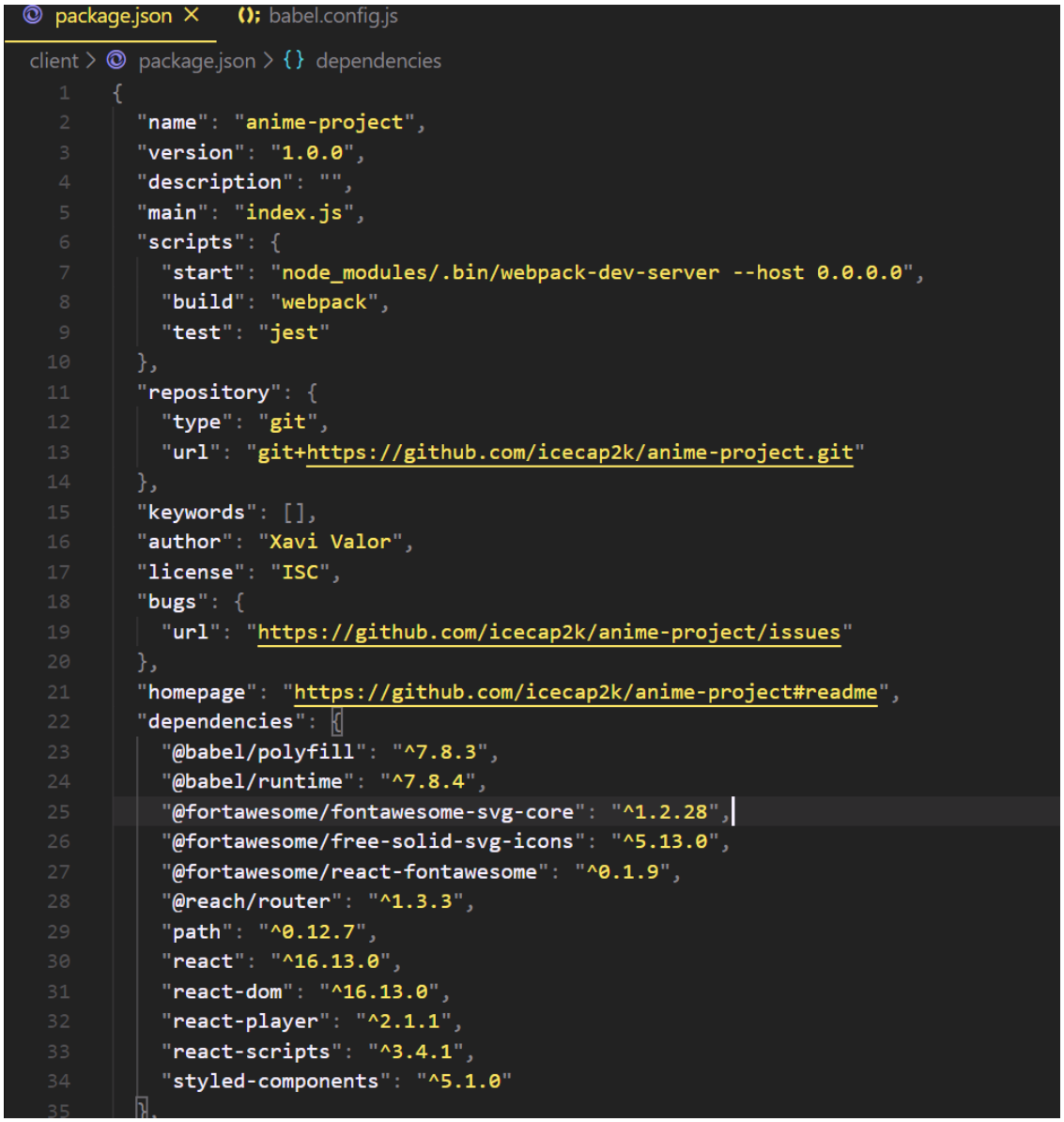

Figura 10. Fichero package.json de la parte cliente

Una de las partes a tener en cuenta son los scripts:

- **Start**: lanza la aplicación en local y la mantiene actualizada mientras desarrollas.
- **Build**: Llama a webpack para que con los datos de la aplicación (input), se genere un par de ficheros como bundle de la aplicación (output), esta pensado para subir la aplicación a producción, excluyendo en el proceso los paquetes o ficheros que no son importados en la aplicación, haciendo que el peso de la misma se reduzca, tiene muchas opciones para crear estos paquetes, como por ejemplo un fichero bundle con la aplicación y otro vendor con los paquetes de terceros que se utilizan, se puede configurar para reducir bastante estos ficheros y que su peso sea más liviano.
- **Test**: Sirve para pasar los test con Jest. Nuestra intención inicial era hacer test y después programar, pero con el escaso tiempo dispuesto no lo hemos podido aplicar, pese a que en la configuración inicial se preparó para ello.

La configuración de webpack del proyecto es la siguiente:

Figura 11. Fichero configuración de webpack

- Entry: toma el fichero index.js del proyecto, a partir de aquí, webpack va añadiendo todos los ficheros que se importan de este y así recursivamente hasta tener tods los ficheros que utiliza la aplicación.
- Output: estableces la ruta y el nombre del fichero de salida con el bundle de la aplicación.
- devServer [3]: El servidor emite información sobre el estado de compilación al cliente, que reacciona a estos eventos. Toma como base la carpeta especificada, en este caso dist como la del output de webpack, y le especifico el puerto 3000 (el mismo que hemos puesto en el docker inicialmente).
- Module/Rules: Aquí se le informa a webpack que ficheros a de tomar (extension js o jsx), excluir la carpeta node modules, donde npm se descarga infinidad de paquetes que no es necesario, ya que le bundle ya ha tomado lo que realmente necesita la aplicación, y que utilice babel para transpilar los ficheros en EcmaScript 6 para que los navegadores actuales interpreten el código.

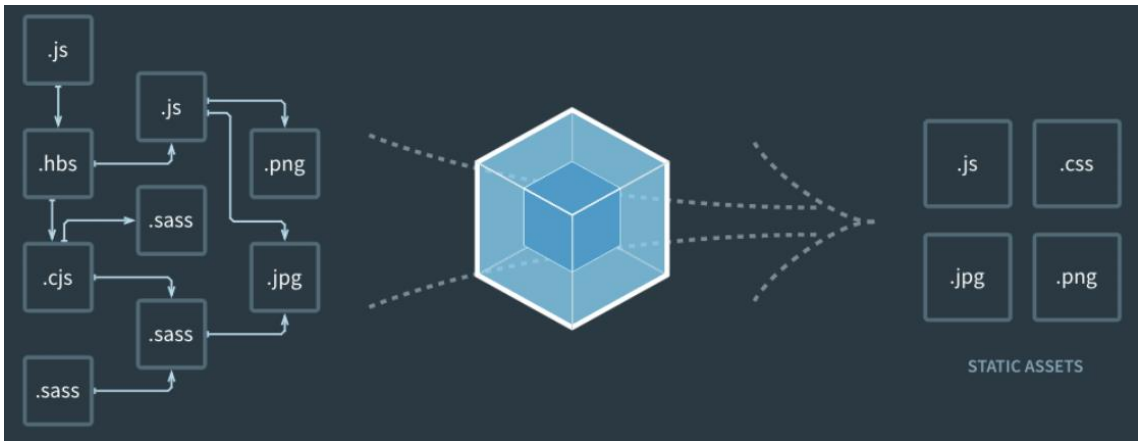

Figura 12. Diagrama informativo de Webpack.

Fuente. https://miro.medium.com/max/2000/1*FtZtDdZgv58Q8oojAvluHw.png

8. Arquitectura de la aplicación

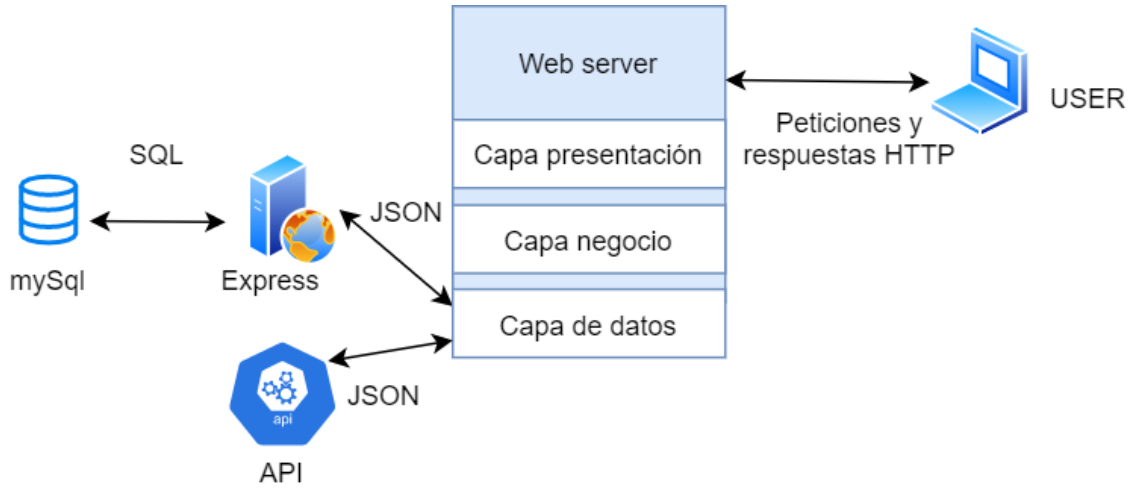

Figura 13. Diagrama arquitectura de la aplicación

La aplicación esta planteada para recibir los datos de forma externa, la información de las series a través de fetch de una API, y la información de los usuarios y sus bibliotecas a través de un servidor a una base de datos.

Para reducir su complejidad se ha planteado con el patrón de diseño Facade y así mantener y estructurar mejor el entorno de programación. Se ha separado tanto las llamadas a la API como las llamadas al server que consulta en la base de datos, así la parte de presentación solo ha de mostrar los datos recibidos, proporcionando así una interfaz simple y desacoplando los subsistemas.

```
const signIn = async (email, password) => {
  const response = await login(email, password)
  if (response) {
    const { id, name, series } = response
    globalState.dispatch(
      { type: 'login', id, name, series },
      ToggleSignInModal(),
      navigate('/home')
    )
  } else {
    setError(true)
  }
}

const registerUser = async (name, email, password) => {
  const response = await register(name, email, password)
  if (response.request) {
    const { id, name } = response
    globalState.dispatch({ type: 'register', id, name }, navigate('/home'))
  }
  setRegisterError(!response.request)
  setRegisterMessage(response.message)
  ToggleRegisterModal()
  ToggleRegisterMessage()
}
```

Figura 14. Llamadas a los servicios de login y register

Figura 15. Fichero services.js llamadas tanto al server como a la API

La arquitectura del cliente ha dejado de ser siguiendo el patrón MVC (Modelo Vista Controlador) para dar paso a Flux, que propone una arquitectura en la que el flujo de datos es unidireccional. Los datos van desde la vista a través de acciones al store, desde la cual se vuelve a actualizar la vista. El beneficio de tener un único camino, y un lugar donde se almacenan el estado de la app, hace que depurar errores y ver que pasa en cada momento sea mucho más sencillo.

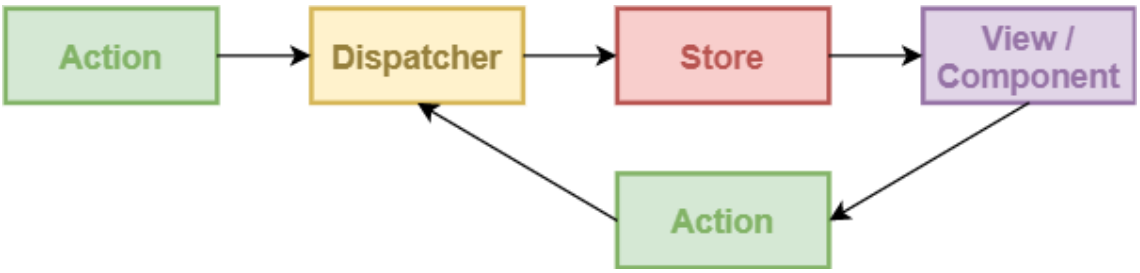

Figura 16. Esquema del patrón Flux

El recorrido de este patrón seria como el visto en el esquema:

- La vista, mediante un evento envía una acción con la intención de cambiar el estado.
- La acción, contiene el tipo y los datos si son necesarios y lo envia al dispatcher.
- El dispatcher, propaga la acción i se procesa en orden de llegada.
- El Store, recibe la acción, qué dependiendo del tipo recibido, actualiza el estado y notifica a las vistas de ese cambio.
- Las vistas, reciben la notificación y se actualiza con los cambios.

En nuestra aplicación, Action y Dispatcher van unidos, ya que no utilizamos Redux si no que hemos trabajado con los nuevos Hooks de ReactJS que simplifican el trabajo, haciéndolo más mantenible.

El sistema de carpetas del proyecto está separado en tres partes:

DB

Aquí está el fichero db.sql con el deproy de la base de datos cuando se monta de cero con el Docker.

Server

La parte del servidor es sencilla y tiene los ficheros de configuración (package.json y dockerfile) y el index.js. De haber tenido más lógica lo hubiera separado las rutas de los servicios, pero esta parte la comentaré más adelante.

Cliente

Como en la parte servidor, los ficheros de configuración están en la raíz, dejando la lógica en la carpeta src. En esta, están los ficheros genéricos de services y store, además del punto de entrada de la aplicación, que es el index.js, dentro también esta la carpeta components, donde están separados

por componentes y dentro de cada carpeta contiene un fichero styles.js que proporciona estilos CSS a los componentes, por lo tanto, cuando son importados en diferentes partes de la aplicación, estos vienen con su lógica y sus estilos. Hemos decidido este sistema por lo beneficios que aportan, ya que como acabo de decir, se puede aplicar estilos para cada componente independientemente, también, como cada componente es como un pequeño paquete de componentes, es más sencillo eliminarlo y puedes crear componentes genéricos para utilizarlos en otras aplicaciones o en diferentes partes de la misma.

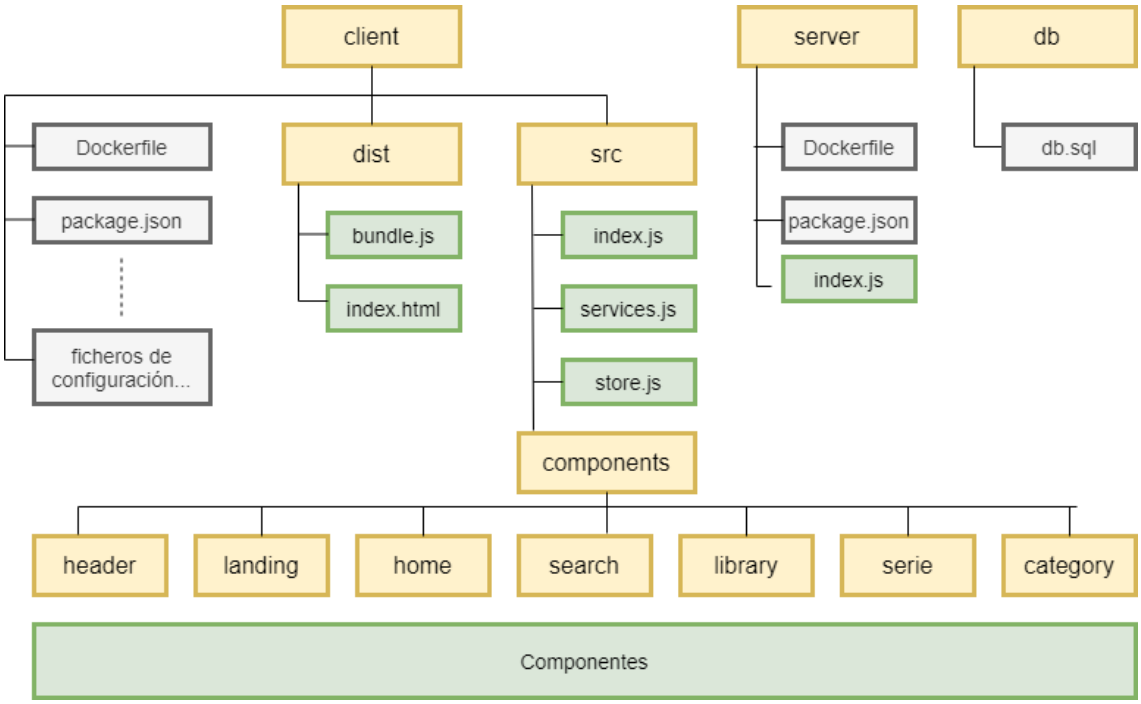

Figura 17. Estructura de carpeta de la aplicación

9. Arquitectura de componentes

Para hablar de la arquitectura de componentes, primero hemos de hablar del ciclo de vida de los componentes en ReactJS, y de los Hooks de React con los que hemos montado toda la aplicación. Un Hook es una función especial que permite “conectarse” a características de React

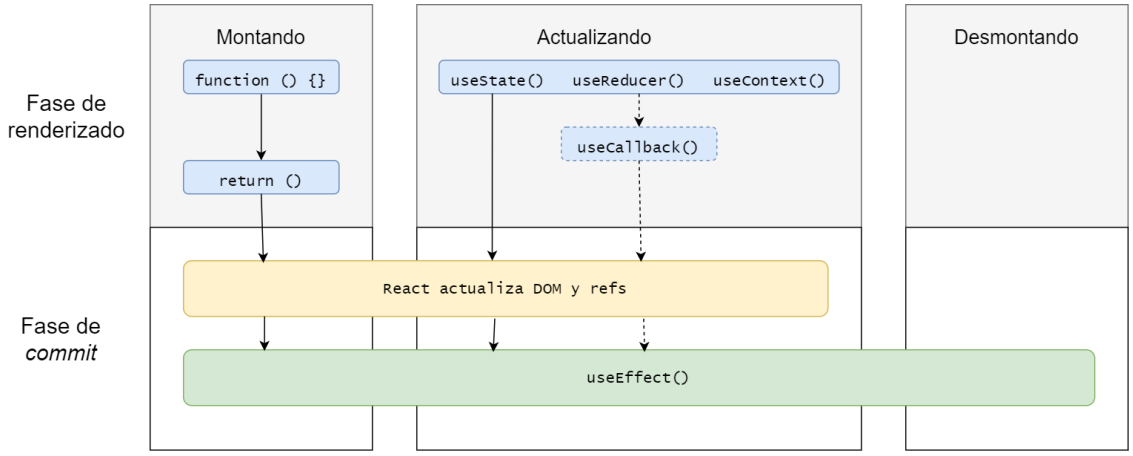

Figura 18. Ciclo de vida de un componente con React Hooks

Un componente funcional, se monta siendo una función que retorna un renderizado html, para añadirle estado al componente si es necesario, utilizamos el Hook useState [7], para llevar acabo efectos secundarios en los componentes utilizamos useEffect [8] que se aplican al montar el componente, cuando hay una actualización y al desmontarlo, dependiendo claro de como este programado, por ejemplo, puedes aplicar el efecto cuando haya algún cambio en alguna variable del estado. useContext [9] lo utilizamos para “conectar” con el estado global de la aplicación. En este estado global también utilizamos useReducer [10] que es una variable más compleja que el useState y es preferible cuando se tiene una lógica compleja que involucra múltiples subvalores o cuando el próximo estado depende del anterior, es por eso que la utilizamos en el estado global de la aplicación. Con cada actualización que afecta al DOM, React lo actualiza, así como las referencias.

Hablemos ahora de la arquitectura para la parte cliente o *front-end*, como hemos dicho inicialmente, el *entry point* de la aplicación es el fichero index.js de la carpeta *src*, este invocara el *Router* que decidirá que vista mostrar, aquí las

vistas no dejan de ser componentes que están importados en este fichero, el Router lo que hace es comprobar que ruta nos llega y selecciona que componente esta asociado con esta vista, en nuestro caso tenemos dos tipos de componentes los públicos y los privados, cuando intenta entrar en una ruta privada, primero comprueba en el estado de la aplicación, que el usuario ya haya accedido con sus credenciales, si no es así, redireccionará a la vista pública. En esta parte también hay espacio para poner los componentes genéricos para todas las vistas, como puede ser la cabecera o el *footer* de la aplicación, en nuestro caso, hemos añadido la cabecera, que hace de sistema de navegación del proyecto y se repite en todas las vistas.

Este fichero también invoca al stateProvider, para mantener los componentes conectados con el store de la aplicación, donde se almacenan los estados de la misma y que utilizan los componentes para leer y escribir en ella.

Por último, se enlaza el renderizado de la aplicación con un el identificador de un elemento del index.html.

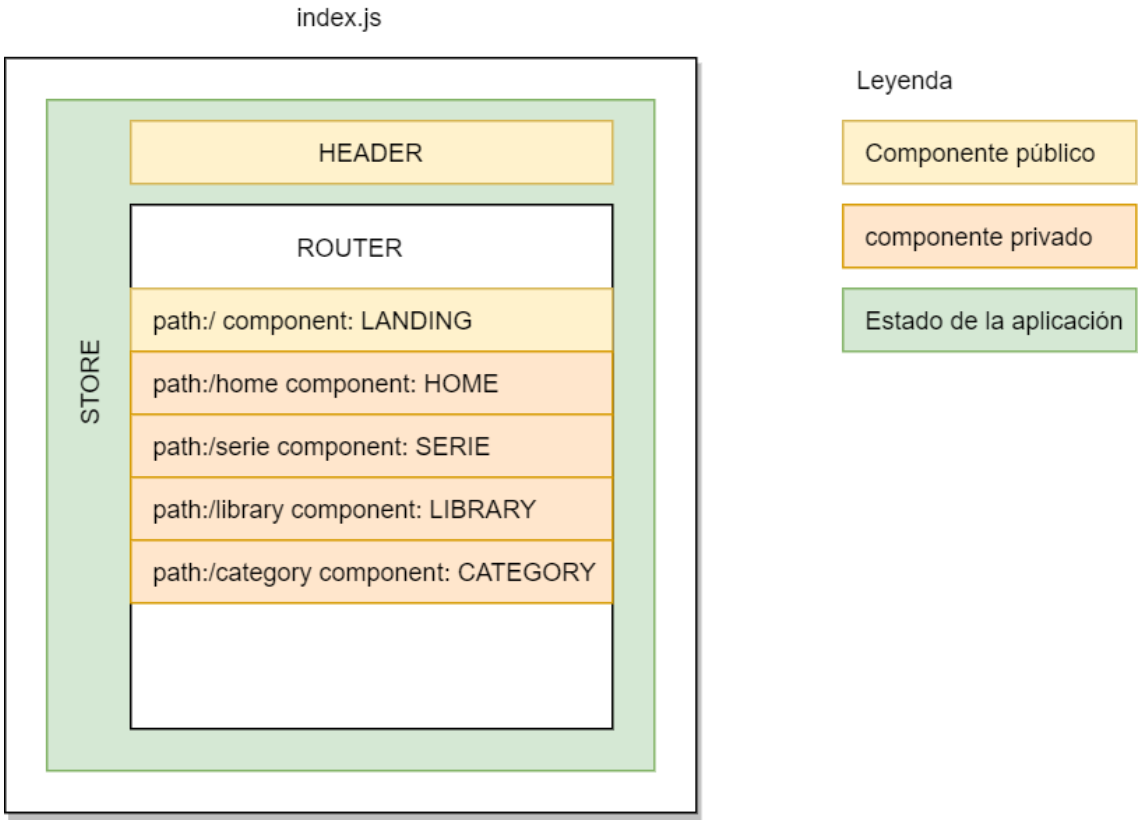

Figura 19. Estructura del fichero index.js

Ya hemos visto como empieza todo, vamos a continuar con los componentes que se enlazan desde el enrutador. El más sencillo es la *landing page*, que hace de puerta de entrada al usuario, en ella básicamente se muestra texto informativo y como destacable tiene la imagen del banner que es dinámica, solicita a la API los 20 primeros mangas más populares y toma uno al azar, dando así un ápice de dinamismo al usuario cada vez que entra.

Otro de los componentes, este con más complejidad, es el *Header*, aquí obtenemos los datos de la Store para saber si el usuario ya ha accedido y así mostrar un sistema de navegación diferente. Cuando el usuario ya ha entrado sus credenciales, es un simple sistema de navegación, más el componente de búsqueda. Si aún no ha accedido, da opción a registrarse o a acceder, en cada uno de los casos muestra un modal al pulsar los respectivos botones. Dentro de cada modal (son componentes separados con propiedades que le pase el componente padre), cuando el usuario rellene el formulario y pulse el botón de *submit*, este llama a la función que le pasa el padre (header.js) y desde él, se conecta con los servicios que a su vez conectan con el server y este con la base de datos, que devuelve una respuesta al servidor, este la analiza y devuelve una estructura de datos al servicio y que a su vez, devuelve un json al componente *Header*, con esta información hace un *dispatch* al Store para almacenar los datos del estado de la aplicación. En el caso del *modalSigin*, cuando finaliza con éxito, continua la navegación hacia la *home*, mientras que, en el *modalRegister*, cuando finaliza con éxito muestra otro modal con un mensaje.

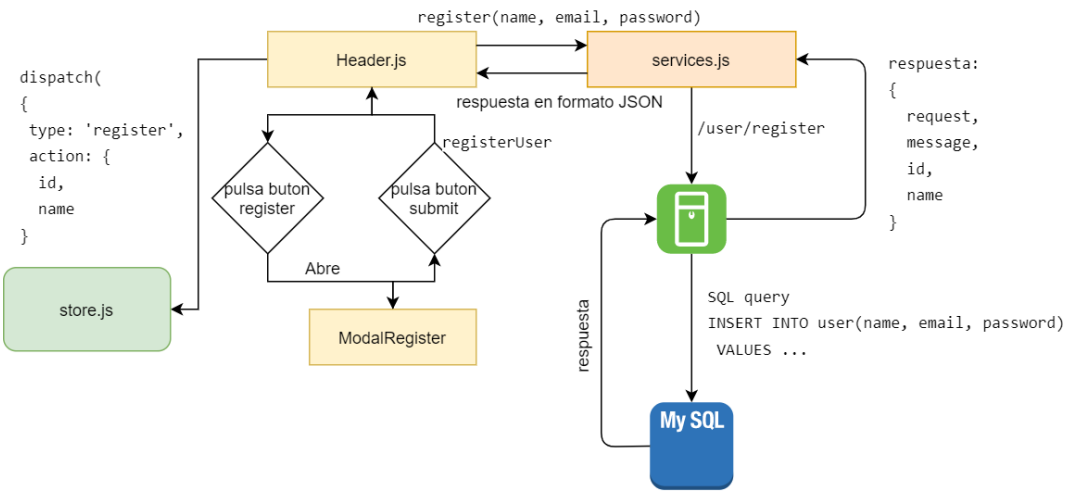

Figura 20. Diagrama de registrar usuario

Otro de los componentes más completos es *serieInfo*. Este componente obtiene los datos de tipo de serie (manga o anime) y el id de la serie, a través de la navegación, se introducen los datos en el estado de la navegación y al cargar este componente, en el *useEffect*, observa si ha habido cambio en estos valores y hace una petición a la API para obtener los datos de la serie e introducirlos con el *useState* en el estado del componente, al introducir estos datos, el componente se renderiza con esta información.

En la descripción de la serie se muestra un botón de añadir o eliminar de la biblioteca esta serie, según los datos que tenemos en el estado de la aplicación, si encuentra la serie en el array de series del estado global, muestra el botón eliminar, si no, mostrará el botón de añadir. Cada botón llama a su función (añadir o eliminar) que, como en el caso del *header*, llama a los servicios que estos conectan con el server y a su vez este con la base de datos, con el fin de añadir o eliminar una serie de la biblioteca, cuando devuelven la petición al componente, este actualiza el estado global, y al haber un cambio en el estado, hace que se renderice el botón contrario (eliminar o añadir).

Por último, hablaremos de los estilos de la aplicación, utilizamos un paquete npm llamado styled-component, esta herramienta se utiliza para crear un componente nuevo, al que solamente aplicamos estilos, esto hace que se puedan reutilizar componentes con los mismos estilos en otros componentes, un caso concreto seria SerieBanner, que lo utilizamos tanto en la información de una serie, como en la página *landing*, simplemente importandolo y pasándole la propiedad *image* que la aplica como fondo. Estos *styled components*, se exportan como un componente normal y se le dice que es un *styled* y a continuación que tipo de etiqueta html es, en el ejemplo anterior, le decimos que sea una etiqueta HTML .

```
export const SerieBanner = styled.div`
  background-image: url(${props => props.image});
  height: 400px;
  width: 100%;
  background-position: center;
  background-repeat: no-repeat;
```

Figura 21. Ejemplo de uso del styled-components

10. Configuración del servidor y base de datos

Como el cliente, el servidor también utiliza la tecnología NodeJS, por lo tanto el fichero de configuración es el package.json.

```
{
  "name": "server",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "nodemon index.js",
    "test": "jest"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "jest": {
    "testEnvironment": "node",
    "coveragePathIgnorePatterns": [
      "/node_modules/"
    ]
  },
  "dependencies": {
    "body-parser": "^1.19.0",
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "mysql": "^2.18.1",
    "nodemon": "^2.0.3"
  },
  "devDependencies": {
    "jest": "^25.4.0",
    "supertest": "^4.0.2"
  }
}
```

Figura 22. Fichero package.json parte servidor

En el tenemos en las dependencias lo siguientes paquetes:

- Body-parser: sirve para manejar la solicitud HTTP POST en ExpressJS, extrae todo la parte del body de una secuencia de solicitud y la expone en el req.body
- Cors: Utilizado para evitar errores de cross-domain
- Express: es un framework de desarrollo de aplicaciones web minimalista y flexible.
- Mysql: se utiliza para conectar con la base de datos.
- Nodemon: se encarga de reiniciar automáticamente el servidor de aplicaciones Node en modo desarrollo.

Como vemos, los scripts son muy sencillos:

- Start: llama a nodemon y le dice el fichero de inicio, como hemos comentado antes, nodemon se encarga de reiniciar el servidor cuando hay un cambio en los ficheros.
- Test: llama a jest para pasar los test de la aplicación.

El servidor esta estructurado en un único fichero, para evitar añadir más complejidad, ya que tan solo necesitamos cuatro funciones: *login* y registro de usuarios, y añadir y eliminar series de la biblioteca del usuario.

Podríamos haber añadido un enrutador con las llamadas a cada una de las funciones y separar el fichero en dos, uno para las llamadas *user* y otro para las llamadas *serie*, esto debería de estudiarse en próximas versiones y cuando vaya creciendo las llamadas y la base de datos.

Base de datos

Hemos optado por un sistema de gestión de base de datos relacional MySQL, en nuestro caso utilizamos únicamente dos tablas *user* y *user_series*.

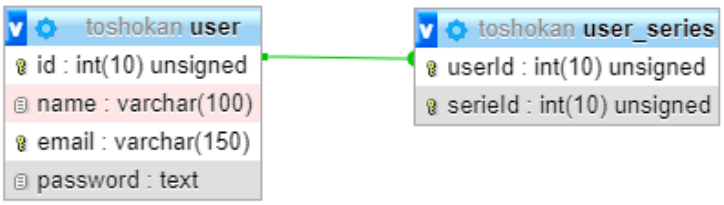

Figura 23. Diagrama entidad relación de la base de datos

La creación de las tablas se realiza en el fichero /db/db.sql, cuando lanzamos inicialmente el docker (docker-compose up --build), lee este fichero y lanza la sequencia SQL. La tabla user tiene como clave primaria el id que tiene la propiedad de auto incremento y es un entero sin signo, para optimizar el campo, ya que no pueden haber ids negativos. También tiene como clave única el email, para evitar que un mismo usuario se registre varias veces utilizando el mismo correo electrónico.

Por otra parte, la tabla user_series tiene dos campos, el id de usuario y el id de serie, ambos son clave primaria, ya que un mismo usuario no puede tener dos veces la misma serie. El id de usuario a su vez es clave foranea del id de la tabla user.

Añado sequencia SQL de la creación de lal tablas.

```sql
--
-- Estructura de tabla para la tabla `user`
--

CREATE TABLE `user` (
  `id` int UNSIGNED NOT NULL,
  `name` varchar(100) COLLATE utf8_unicode_ci NOT NULL,
  `email` varchar(150) COLLATE utf8_unicode_ci NOT NULL,
  `password` text COLLATE utf8_unicode_ci NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;


--
-- Indices de la tabla `user`
--
ALTER TABLE `user`
  ADD PRIMARY KEY (`id`),
  ADD UNIQUE KEY `EMAIL` (`email`);
--
-- Estructura de tabla para la tabla `user_series`
--

CREATE TABLE `user_series` (
  `userId` int UNSIGNED NOT NULL,
  `serieId` int UNSIGNED NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;

--
-- Indices de la tabla `user_series`
--
ALTER TABLE `user_series`
  ADD PRIMARY KEY (`userId`, `serieId`);

--
-- Enlazar clave foranea de user userId
ALTER TABLE `user_series` ADD constraint `FK_id_user` FOREIGN KEY (`userId`) REFERENCES `user` (`id`);
```

Figura 24. Fichero db.sql. Creación de las tablas en la base de datos

11. Resultados

El producto obtenido tras el desarrollo del proyecto, es una modesta aplicación con la que poder llevar un orden en nuestra biblioteca de mangas y animes.

Parte de una página que hace de bienvenida a los usuarios, explicando brevemente que pueden hacer con esta aplicación.

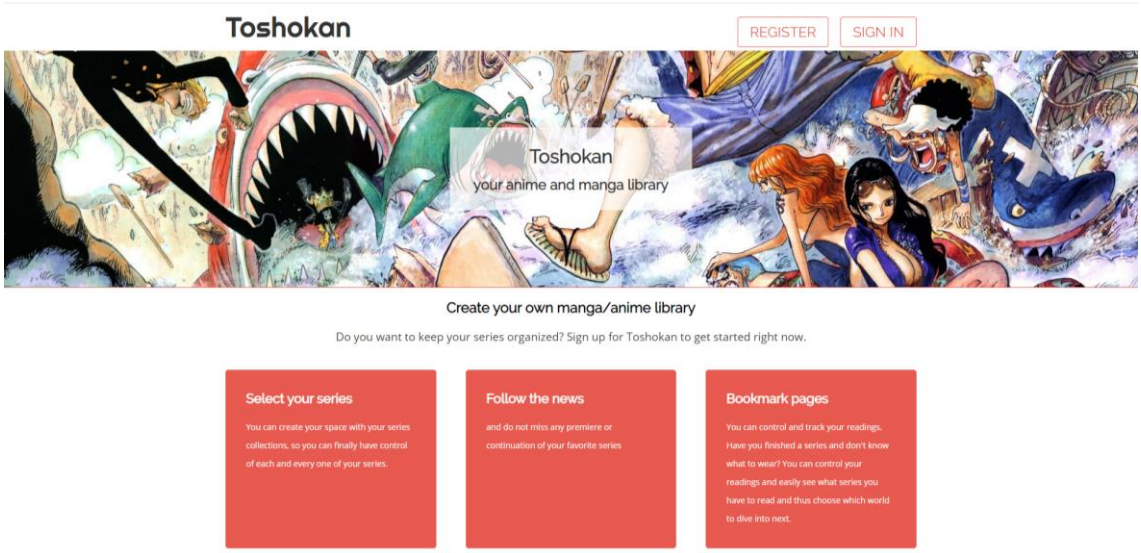

Figura 25. Página *Landing* de la aplicación

Un usuario recién llegado puede registrarse, o si ya está registrado, puede acceder con su correo electrónico y contraseña.

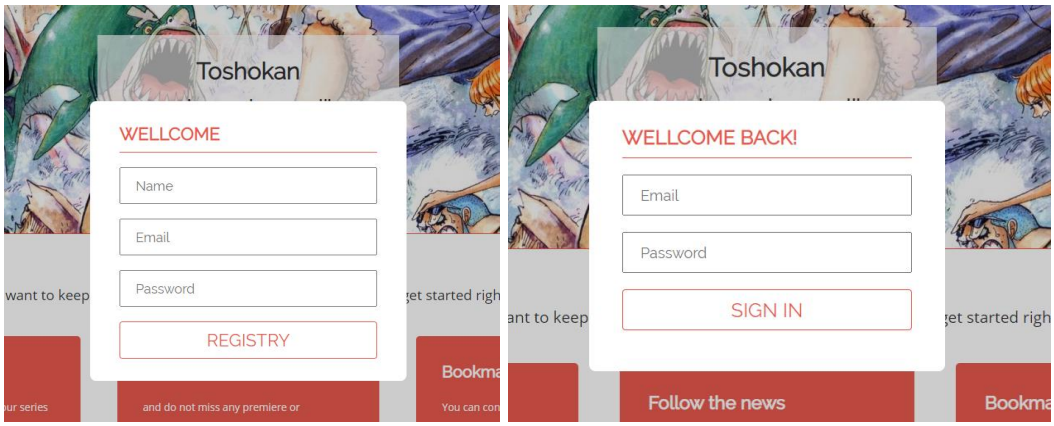

Figura 26. Modales de registro (izquierda) y de acceso a la aplicación (derecha)

Una vez dentro de la aplicación, se muestra en la parte central, cuatro listados de series como propuesta para ir introduciendo en su biblioteca:

- Series anime trending
- Series anime
- Series manga trending
- Series manga

Al lateral se ha añadido un listado de categorias para ver las diferentes series de esas categorias. Listadas en orden de más contenido a menos, es decir, las primeras categorias incluyen un número más elevado de series.

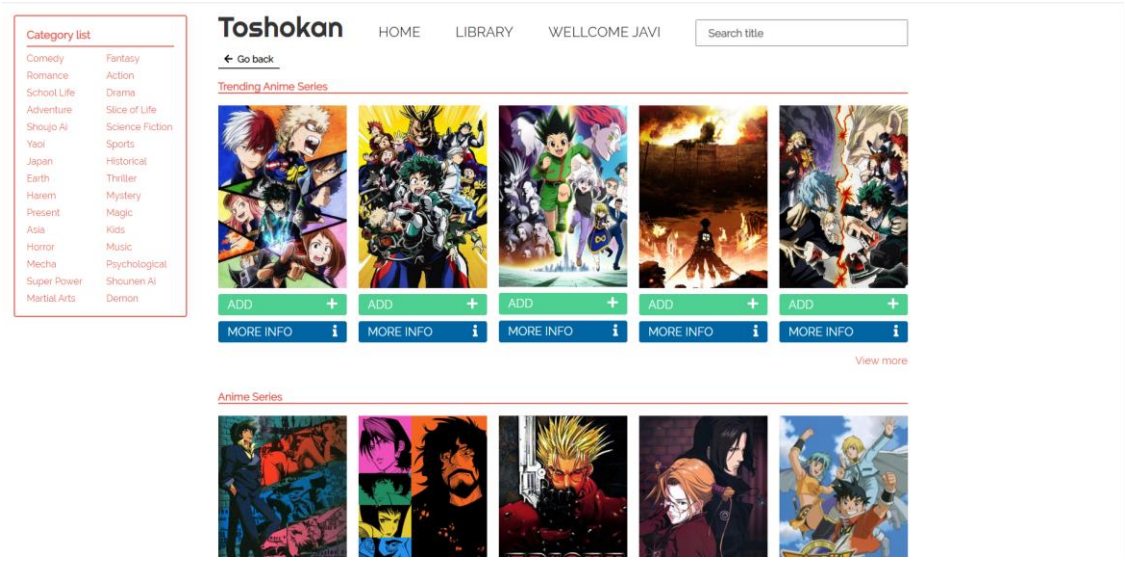

Figura 27. Página *Home* de la aplicación

Desde este mismo instante, el usuario ya puede ir añadiendo las series propuestas en su biblioteca. Pero, si decide ir navegando, encontrará más series e información de las mismas.

Cuando accedemos a mostrar mas información de una serie, tenemos dos recursos gráficos, uno como portada de la misma, y otro como banner ilustrativo, aunque, en algunas series este último recurso no esta incluido, hemos decidido incorporar un trozo de la caratula, más adelante, podríamos cambiar esta por una imagen default descriptiva y genérica. También incluye un resumen de la serie, y, en algunos casos, un trailer de la misma. Hemos incorporado también el botón de añadir o eliminar de nuestra biblioteca.

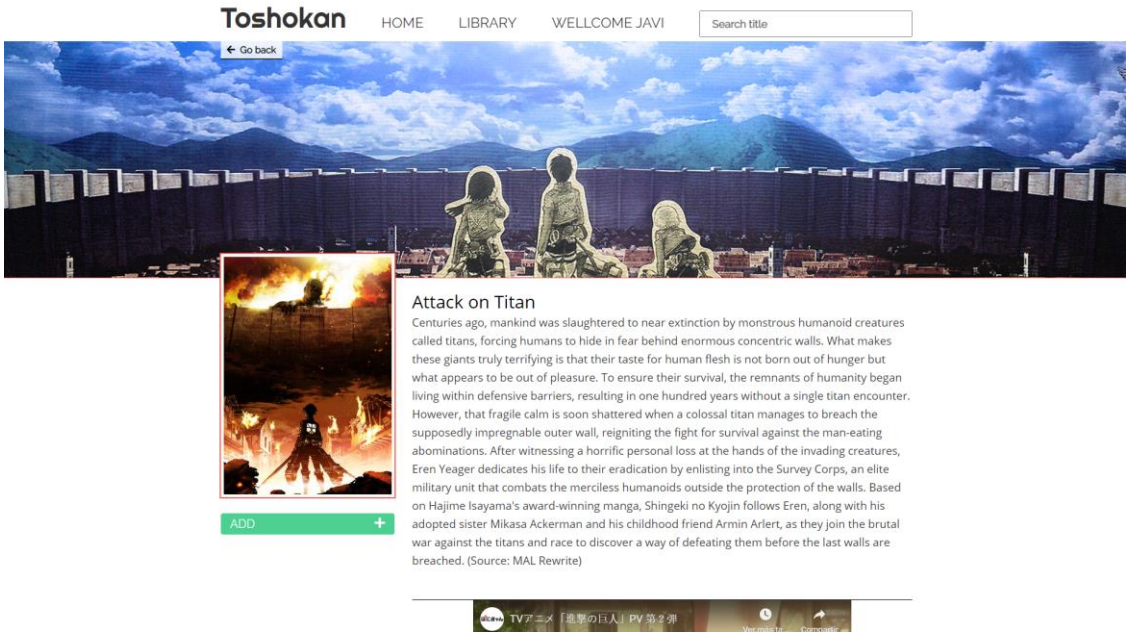

Figura 28. Página de información de una serie

Cuando entramos por categorias, incluye una descripción de la categoría y el número total de series que están etiquetadas en esta. A continuación, podemos ver las 15 seies más “trending” y las primeras 15 series más actuales.

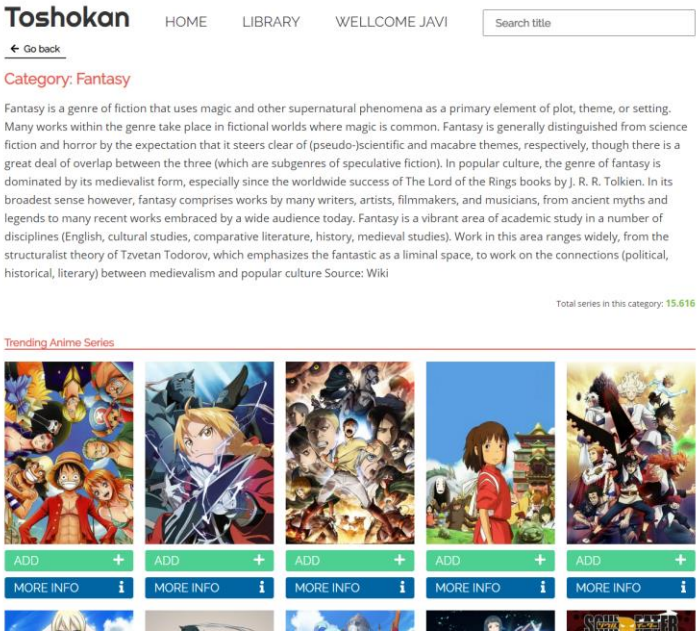

Figura 29. Página al seleccionar categoría

Otro de los accesos que tienen los usuario para encontrar sus series favoritas, es el buscador, esta desarrollado de tal forma que al empezar a escribir, va

haciendo consultas de la búsqueda, y va mostrando resultados en tiempo real, tal que el usuario puede encontrar series sin acordarse o coincidir completamente con el nombre real. Se muestran cinco resultado de anime y otros cinco de manga.

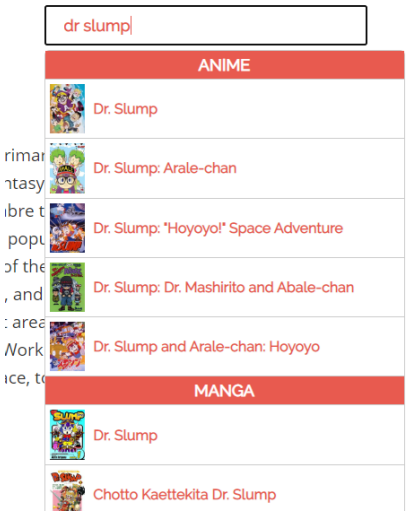

Figura 30. Sección del buscador

Dejando lo más importante para el final, la página donde ver su propia biblioteca. Donde pueden acceder a más información de las series, o eliminarlas de la biblioteca.

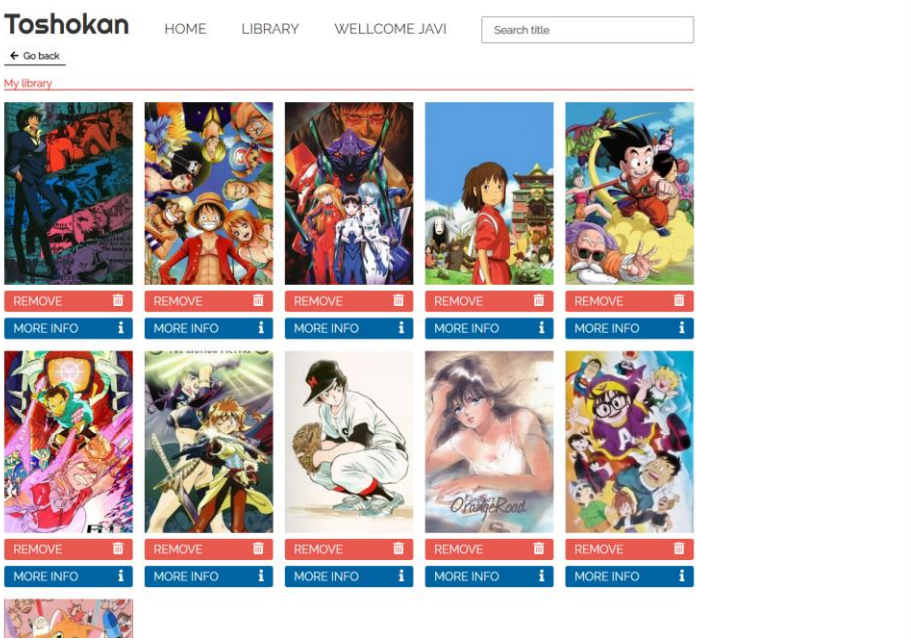

Figura 31. Página de la biblioteca de un usuario

12. Conclusiones

Introducción y evaluación de riesgos

Ha sido un camino largo y difícil, pero se ha llevado con gusto al estar haciendo algo que realmente quieres hacer. Al inicio de la asignatura exponíamos una evaluación de riesgos que podría sufrir el proyecto, pero nunca nos imaginamos la difícil situación que hemos vivido, el principal riesgo, decía, era la falta de tiempo, ya que se trata de montar todo un proyecto desde cero con una única persona.

Ahora veo que me he quedado corto cuando decía falta de tiempo, seguramente todos hemos tenido nuestras circunstancias y no las expongo como excusas, ni pretendo dar lástima, únicamente mostrar una instantánea del estado en que hemos tenido que llevar el proyecto: somos cuatro en casa, mi pareja, trabaja en el ámbito sanitario y ha teniendo que ir presencialmente cada día (incluido festivos), por mi parte yo he podido continuar teletrabajando cuando mi pareja volvía a casa, mientras en casa estábamos los tres, mis dos hijos, un bebe de 9 meses, mi hijo de 3 años y yo.

Aprendiendo

He aprendido mucho de este proyecto, crear una idea, hacer una planificación y desglose de tareas, configurar y montar los contenedores en Docker, montar una aplicación desde cero, etc. Ha sido complicado, pero a la vez gratificante cuando ibas cumpliendo plazos y viendo como poco a poco se conectaban todas las piezas. Sobre todo, he aprendido mucho.

Objetivos, planificación y cambios

No se han podido cumplir con todos los objetivos, tenia ganas de añadir muchas más cosas, esencialmente los test tanto unitarios como de integración, ya que los veo imprescindibles, pero valorando el tiempo de configuración de los entornos de test y el peso que aporta a la aplicación, he tenido que descartarlos, y eso que el peso que aportan es enorme, en estabilidad, detección de errores, etc. Pero tenia que elegir entre un prototipo testeado y con apenas funcionalidades o una aplicación funcional sin tests.

Por otra parte, quería añadir Storybooks para crear componentes atomizados y más reutilizables. También era interesante haber podido instalar Sonarqube para evaluar y mejorar la calidad del código.

Futuro

Como he dicho anteriormente, introduciré los test, tanto unitario como de integración, y a continuación Sonarcube para hacer una valoración y según sus métricas, ver el estado de la aplicación y que partes son mejorables y optimizables.

También he de añadir más secciones a la navegación, así como el enlace de ver más que hay en cada listado de series que se muestra, en ella iba hacer peticiones paginadas de la API para poder mostrar todos los resultados con un sistema de páginas.

Otras mejoras ha añadir, es que, al hacer la búsqueda, tengas la opción de ir a una página de resultados, también paginada, así como una sección de búsqueda avanzada, donde poder buscar por autor, nombre, dentro de una categoría, etc.

Por último, creo que estaría bien poder ordenar la biblioteca, o permitir diferentes tipos de ordenación, por año, por autor, por categoría, etc. Ya que esta parte, es la más importante de cara al usuario.

13. Glosario

- **ES6 o ES2015**. ECMAScript 6. Es el estándar que sigue JavaScript desde junio de 2015
- **NodeJS**. Es un entorno de ejecución de Javascript orientado a eventos asíncronos, está diseñado para crear aplicaciones network escalabres.
- **ReactJS**. Es una librería JavaScript de código abierto para crear interfaces de usuario permitiendo construir aplicaciones que usan datos que se actualizan todo el tiempo.
- **Framework**. Es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.
- **Middleware**. Es un software que se sitúa entre un sistema y las aplicaciones que se jecutan en él, funciona como una capa de traducción oculta para permitir la comunicación y la administración de datos en aplicaciones distribuidas.
- **MVC (Modelo Vista Controlador)**. Es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.
- **Flux.** Es la arquitectura de aplicaciones que Facebook usa para construir aplicaciones web del lado del cliente. Utilizando un flujo de datos unidireccional. Los datos de una aplicación con la arquitectura flux, fluyen en una sola dirección.
- **API (Application Programming Interface).** Es un conjunto de subrutinas, funciones y procedimientos o métodos, que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
- **NPM.** Es el sistema de gestión de paquetes por defecto para Node.js, un entorno de ejecución para JavaScript, bajo Artistic License 2.0.

14. Bibliografía

1. ECMAScript 6, el nuevo estándar de JavaScript
   Autor: Carlos Azaustre
   Web: https://carlosazaustre.es/ecmascript6 (28/05/20)
2. Node.js. Acerca de Node.js
   Web: https://nodejs.org/es/about/ (28/05/20)
3. ReactJS
   Web: https://es.reactjs.org/ (28/05/20)
4. Introducción a Express/Node.
   Web: https://developer.mozilla.org/es/docs/Learn/Server-side/Express_Nodejs/Introduction (29/05/20)
5. Flujo de trabajo de Gitflow.
   Web: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow (29/05/20)
6. Webpack devServer.
   Web: https://webpack.js.org/configuration/dev-server/ (06/06/20)
7. React Hooks. UseState.
   Web: https://es.reactjs.org/docs/hooks-state.html (07/06/20)
8. React Hooks. UseEffect.
   Web: https://es.reactjs.org/docs/hooks-effect.html (07/06/20)
9. React Hooks. UseContext.
   Web: https://es.reactjs.org/docs/hooks-reference.html#usecontext (07/06/20)
10. React Hooks. UseReducer.
    Web: https://es.reactjs.org/docs/hooks-reference.html#usereducer (07/06/20)