Aplicaciones reactivas con React, NodeJS y MongoDB.

UN ENFOQUE PRÁCTICO

Oscar Javier Blancarte Iturralde

PRIMERA EDICIÓN

Oscar Blancarte Blog
software architect

Programación Reactiva con React, NodeJS & MongoDB

❝Se proactivo antes los cambios, pues una vez que llegan no tendrás tiempo para reaccionar**❞**

—Oscar Blancarte (2018)

Datos del autor:

Ciudad de México

e-mail: oscarblancarte3@gmail.com

Autor y Editor

Composición y redacción:

Oscar Javier Blancarte Iturralde

Edición:

Oscar Javier Blancarte Iturralde

Portada

Arq. Jaime Alberto Blancarte Iturralde

Primera edición

Se publicó por primera vez en Enero del 2018

Acerca del autor

Oscar Blancarte es originario de Sinaloa, México donde estudió la carrera de Ingeniería en Sistemas Computacionales y rápidamente se mudó a la Ciudad de México donde actualmente radica.

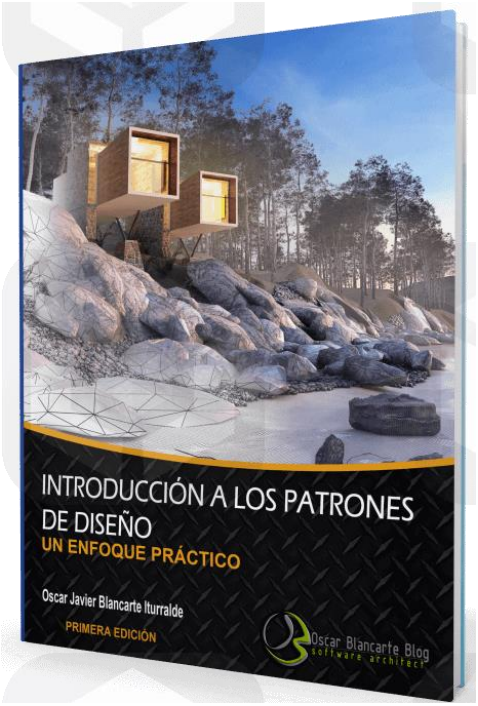
Oscar Blancarte es un Arquitecto de software con más de 12 años de experiencia en el desarrollo y arquitectura de software. Certificado como Java Programmer (Sun microsystems), Análisis y Diseño Orientado a Objetos (IBM) y Oracle IT Architect (Oracle).

A lo largo de su carrera ha trabajado para diversas empresas del sector de TI, entre las que destacan su participación en diseños de arquitectura de software y consultoría para clientes de los sectores de Retail, Telco y Healt Care.

Oscar Blancarte es además autor de su propio blog https://www.oscarblancarteblog.com desde el cual está activamente publicando temas interesantes sobre Arquitectura de software y temas relacionados con la Ingeniería de Software en general. Desde su blog ayuda a la comunidad a resolver dudas y es por este medio que se puede tener una interacción más directa con el autor.

Otras obras del autor

Introducción a los patrones de diseño – Un enfoque práctico



Hoy en día aprender patrones de diseño no es una cualidad más, **si no una obligación**. Y es que estudiar y comprender los patrones de diseño nos convierte en un mejor programador/arquitecto y es clave para conseguir una mejor posición en el mundo laboral.

Este libro fue creado con la intención de enseñar a sus lectores cómo utilizar los patrones de diseño de una forma **clara y simple** desde un enfoque práctico y con escenarios del mundo real.

Tengo que aceptar que este no es un libro convencional de patrones de diseño debido, principalmente, a que no sigue la misma estructura de las primordiales obras relacionadas con este tema. En su lugar, me quise enfocar en ofrecer una **perspectiva del mundo real**, en donde el lector pueda aprender a utilizar los patrones de diseño en entornos reales y que puedan ser aplicados a proyectos reales.

Cuando empecé a estudiar sobre patrones de diseño, me di cuenta que siempre se explicaban en escenarios irracionales que poco o ninguna vez podrías utilizar, como por ejemplo para aprender a crear figuras geométricas, hacer una pizza o crear una serie de clases de animales que ladren o maúllen; esos eran los ejemplos que siempre encontraba, que, si bien, explicaban el concepto, se complicaba entender cómo llevarlos a escenarios reales.

En este libro trato de ir un poco más allá de los **ejemplos típicos** para crear cosas **realmente increíbles**. Por ejemplo:

- Crear tu propia consola de línea de comandos.
- Crear tu propio lenguaje para realizar consultas SQL sobre un archivo de Excel.
- Crear aplicaciones que puedan cambiar entre más de una base de datos, por ejemplo, Oracle y MySQL según las necesidades del usuario.
- Administrar la configuración global de tu aplicación.
- Crear un Pool de ejecuciones para controlar el número de hilos ejecutándose simultáneamente, protegiendo nuestra aplicación para no agotar los recursos.
- Utilizar proxis para controlar la seguridad de tu aplicación.

- Utilizar estrategias para cambiar la forma en que los usuarios son autenticados en la aplicación; como podría ser por Base de datos, Webservices, etcétera.
- Crear tu propia máquina de estados para administrar el ciclo de vida de tu servidor.

Éstos son sólo algunos de los **25 ejemplos que abordaremos en este libro**, los cuales están acompañados, en su totalidad, con el **código fuente** para que seas capaz de descargarlos, ejecutarlos y analizarlos desde tu propia computadora.

Adquiérelo en https://patronesdediseño.com

Agradecimientos

Este libro tiene una especial dedicación a mi esposa Liliana y mi hijo Oscar, quienes son la principal motivación y fuerza para seguir adelante todos los días, por su cariño y comprensión, pero sobre todo por apoyarme y darme esperanzas para escribir este libro.

A mis padres, quien con esfuerzo lograron sacarnos adelante, darnos una educación y hacerme la persona que hoy soy.

A todos mis lectores anónimos de mi blog y todas aquellas personas que, de buena fe, compraron y recomendaron mi primer libro (Introducción a los patrones de diseño) y fueron en gran medida lo que me inspiro en escribir este segundo libro.

Finalmente, quiero agradecerte a ti, por confiar en mí y ser tu copiloto en esta gran aventura para aprender React, NodeJS, MongoDB y muchas cosas más. :)

Prefacio

Cada día, nacen nuevas tecnologías que ayudan a construir aplicaciones web más complejas y elaboras, con relativamente menos esfuerzo, ayudando a que casi cualquiera persona con conocimientos básicos de computación puede realizar una página web. Sin embargo, no todo es felicidad, pues realizar un trabajo profesional, requiere de combinar muchas tecnologías para poder entregar un producto terminado de calidad.

Puede que aprender React o NodeJS no sea un reto para las personas que ya tiene un tiempo en la industria, pues ya están familiarizados con HTML, JavaScript, CSS y JSON, por lo que solo deberá complementar sus conocimientos con una o dos tecnologías adicionales, sin embargo, para las nuevas generaciones de programadores o futuros programadores, aprender React o NodeJS puede implicar un reto a un mayor, pues se necesita aprender primero las bases, antes de poder programar en una capa más arriba.

Cuando yo empecé a estudiar el Stack completo de React + NodeJS + MongoDB fue necesario tener que leer 3 libros completos, pues cada uno enseñaba de forma separada una parte de la historia. Cada libro me enseñó a utilizar las tecnologías de forma individual, lo cual era fantástico, sin embargo, cuando llego el momento de trabajar con todas las tecnologías juntas empezó el problema, pues nadie te enseñaba como unir todo en un solo proyecto, optimizar y pasar a producción como una aplicación completa. Todo esto, sin mencionar los cientos o quizás miles de foros y blogs que tuve que analizar para aprender los trucos más avanzados.

Este libro pretende evitarte ese gran dolor de cabeza que yo tuve por mucho tiempo, pues a lo largo de este libro aprenderemos a utilizar React + NodeJS con Express + MongoDB y aderezaremos todo esto con Redux, uno de los módulos más populares y avanzados para el desarrollo de aplicaciones Web profesionales. Finalmente aprenderemos a crear un API REST completo con NodeJS y utilizando el Estándar de Autenticación JSON Web Tokens.

El objetivo final de este libro es que aprendas a crear aplicaciones Reactivas con React, apoyado de las mejores tecnologías disponibles. Es por este motivo que, durante la lectura de este libro, trabajaremos en un único proyecto que irá evolucionando hasta terminarlo por completo. Este proyecto será, una réplica de la red social Twitter, en la cual podremos crear usuarios, autenticarnos, publicar Tweets, seguir a otros usuarios y ver las publicaciones de los demás en nuestro feed.

Cómo utilizar este libro

Es te libro es en lo general fácil de leer y digerir, pues tratamos de enseñar todos los conceptos de la forma más simple y asumiendo que el lector tiene poco o nada de conocimiento de tema, así, sin importar quien lo lea, todos podamos aprender fácilmente.

Como parte de la dinámica de este libro, hemos agregado una serie de tipos de letras que hará más fácil distinguir entre los conceptos importantes, código, referencias a código y citas. También hemos agregado pequeñas secciones de tips, nuevos conceptos, advertencias y peligros, los cuales mostramos mediante una serie de íconos agradables para resaltar a la vista.

Texto normal:

Es el texto que utilizaremos durante todo el libro, el cual no enfatiza nada en particular.

Negritas:

El texto en negritas es utilizado para enfatizar un texto, de tal forma que buscamos atraer tu atención, que es algo importante.

Cursiva:

Es texto lo utilizamos para hacer referencia a fragmentos de código como una variable, método, objeto o instrucciones de líneas de comandos. Pero también es utilizada para resaltar ciertas palabras técnicas.

Código

Para el código utilizamos un formato especial, el cual permite colorear ciertas palabras especiales, crear el efecto entre líneas y agregar el número de línea que ayude a referencia el texto con el código.

```
1.  ReactDOM.render(
2.    <h1>Hello, world!</h1>,
3.    document.getElementById('root')
4.  );
```

El texto con fondo verdes, lo utilizaremos para indicar líneas que se agregan a un código existente.

~~Mientras que el rojo y tachado, es para indicar código que se elimina de un archivo existente.~~

Por otra parte, tenemos los íconos, que nos ayudan para resaltar algunas cosas:

Nuevo concepto: <concepto>
Cuando mencionamos un nuevo concepto o termino que vale la pena resaltar, es explicado mediante una caja como esta.

Tip
Esta caja la utilizamos para dar un tip o sugerencia que nos puede ser de gran utilidad.

Importante
Esta caja se utiliza para mencionar algo muy importante.

Error común
Esta caja se utiliza para mencionar errores muy comunes que pueden ser verdadero dolor de cabeza o para mencionar algo que puede prevenir futuros problemas.

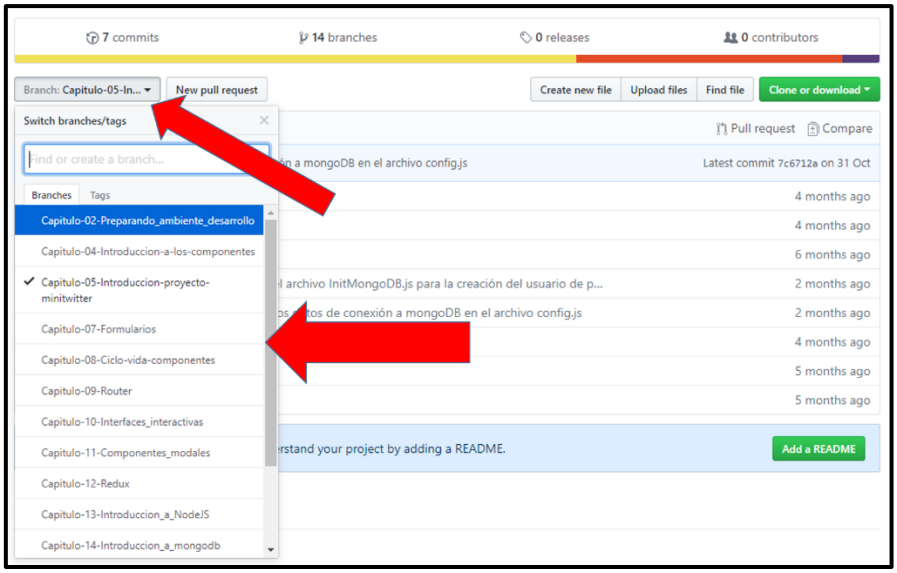
Documentación
Esta caja se muestra cada vez que utilizamos un nuevo servicio del API REST, la cual tiene la intención de indicar al lector donde puede encontrar la documentación del servicio, como es la URL, parámetros de entrada/salida y restricciones de seguridad.

Código fuente

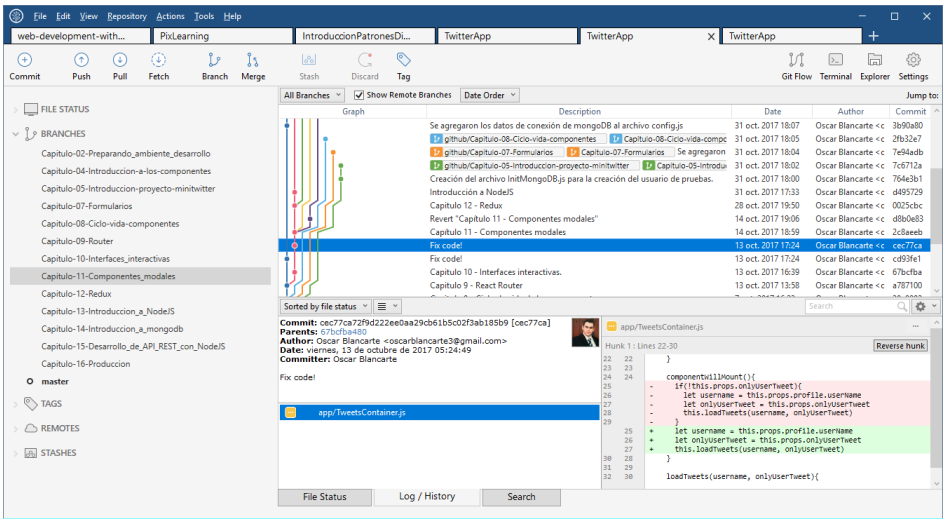
Todo el código fuente de este libro está disponible en GitHub y dividido de tal forma que, cada capítulo tenga un Branch independiente. El código fuente lo puedes encontrar en:

https://github.com/oscarjb1/books-reactiveprogramming.git

Para ver el código de cada capítulo, solo es necesario dirigirse al listado de branches y seleccionar el capitulo deseado:



La segunda y más recomendable opción es, utilizar un cliente de Git, como lo es Source Tree y clonar el repositorio, lo que te permitirá tener acceso a todos los branche desde tu disco duro.



Requisitos previos

Este libro está diseñado para que cualquier persona con conocimientos básicos de programación web, puedo entender la totalidad de este libro, sin embargo, debido a la naturaleza de React y NodeJS, es necesario conocer los fundamentos de JavaScript, pues será el lenguaje que utilizaremos a lo largo de todo este libro.

También será interesante contar con conocimientos básicos de CSS3, al menos para entender como declarar selectores y aplicarlos a los elementos de HTML.

INTRODUCCIÓN

Muy lejos han quedado los tiempos en los que Tim Berners Lee, conocido como el padre de la WEB; estableció la primera comunicación entre un cliente y un servidor, utilizando el protocolo HTTP (noviembre de 1989). Desde entonces, se ha iniciado una guerra entre las principales compañías tecnológicas por dominar el internet. El caso más claro, es la llamado **guerra de los navegadores**, protagonizado por Netscape Communicator y Microsoft, los cuales buscaban que sus respectivos navegadores (Internet Explorer y Netscape) fueran el principal software para navegar en internet.

Durante esta guerra, las dos compañías lucharon ferozmente, Microsoft luchaba por alcanzar a Netscape, quien entonces le llevaba la delantera y Netscape luchaba para no ser alcanzados por Microsoft. Como hemos de imaginar, Microsoft intento comprar a Netscape y terminar de una vez por todas con esta guerra, sin embargo, Netscape se negó reiteradamente a la venta, por lo que Microsoft inicio una de las más descaradas estrategias; amenazo con copiar absolutamente todo lo que hiciera Netscape si no accedían a la venta.

Económicamente hablando, Microsoft era un gigante luchando contra una hormiga, el cual pudo contratar y poner a su plantilla a cuento ingeniero fuera necesarios para derrotar a Netscape.

En esta carrera por alcanzar a Netscape, ambas empresas empezaron a lanzar releaces prácticamente cada semana, y semana con semana, los navegadores iban agregando más y más funcionalidad, lo cual podría resultar interesante, pero era todo lo contrario. Debido a la guerra sucia, ninguna empresa se apegaba a estándares, que además eran prácticamente inexistentes en aquel entonces. Lo que provocaba que los programadores tuvieran que escribir dos versiones de una misma página, una que funcionara en Netscape y otra que funcionara para Internet Explorer, pues las diferencia que existían entre ambos navegadores era tan grande, que era imposible hacer una misma página compatible para los dos navegadores.

Para no hacer muy larga esta historia, y como ya sabrás, Microsoft termino ganando la guerra de los navegadores, al proporcionar Internet Explorer totalmente gratis y preinstalado en el sistema operativo Windows.

Hasta este punto te preguntarás, ¿qué tiene que ver toda esta historia con React y NodeJS?, pues la verdad es que mucho, pues durante la guerra de los navegadores Netscape invento JavaScript. Aunque en aquel momento, no era un lenguaje de programación completo, sino más bien un lenguaje de utilidad, que permitía realizar cosas muy simples, como validar formularios, lanzar alertas y realizar algunos cálculos. Es por este motivo que debemos recordar a Netscape, pues fue el gran legado que nos dejó.

Finalmente, Netscape antes de terminar de morir, abre su código fuente y es cuando nace Mozilla Firefox, pero esa es otra historia…

Retornando a JavaScript, este lenguaje ha venido evolucionando de una manera impresionante, de tal forma que hoy en día es un lenguaje de programación completo, como Java o C#. Tan fuerte ha sido su evolución y aceptación que hoy en día podemos encontrar a JavaScript del lado del servidor, como es el caso de NodeJS, creado por Ryan Dahl en 2009. De la misma forma, Facebook desarrollo la librería React basada en JavaScript.

Tanto React como NodeJS funcionan en su totalidad con JavaScript, React se ejecuta desde el navegador, mientras que NodeJS se ejecuta desde un servidor remoto, el cual ejecuta código JavaScript.

JavaScript es un lenguaje de programación interpretado, lo que quieres decir que el navegador va ejecutando el código a medida que es analizado, evitando procesos de compilación. Además, JavaScript es un lenguaje regido por el Estándar ECMAScript el cual, al momento de escribir estas líneas, se encuentra en la versión 6 estable y se está desarrollando la versión 7.

Nuevo concepto: ECMAScript
ECMAScript es una especificación de lenguaje de programación propuesta como estándar por NetScape para el desarrollo de JavaScript.

Comprender la historia y la funcionalidad de JavaScript es fundamental para aprender a React y NodeJS, pues ambos trabajan 100% con este lenguaje. Aunque en el caso de React, existe un lenguaje propio llamado JSX, que al final se convierte en JavaScript. Pero eso lo veremos en su momento.

Índice

Por dónde empezar

Capítulo 1

Hoy en día existe una gran cantidad de propuestas para desarrollar aplicaciones web, y cada lenguaje ofrece sus propios frameworks que prometen ser los mejores, aunque la verdad es que nada de esto está cercas de la realidad, pues el mejor framework dependerá de lo que buscas construir y la habilidad que ya tengas sobre un lenguaje determinado.

Algunas de las propuestas más interesante para el desarrollo web son, Angular, Laravel, Vue.JS, Ember.js, Polymer, React.js entre un gran número de etcéteras. Lo que puede complicar la decisión sobre que lenguaje, librería o framework debemos utilizar.

Desde luego, en este libro no tratamos de convencerte de utilizar React, sino más bien, buscamos enseñarte su potencial para que seas tú mismo quien pueda tomar esa decisión.

Solo como referencia, me gustaría listarte algunas de las empresas que actualmente utilizan React, para que puedas ver que no se trata de una librería para hacer solo trabajos caseros o de baja carga:

- Udemy
- Bitbucket
- Anypoint (Mule Soft)
- Facebook
- Courcera
- Airbnb
- American Express
- Atlassian
- Docker
- Dropbox
- Instagram
- Reddit

Son solo una parte de una inmensa lista de empresas y páginas que utilizan React como parte medular de sus desarrollos. Puedes ver la lista completa de páginas que usan React aquí: https://github.com/facebook/react/wiki/sites-using-react. Esta lista debería ser una evidencia tangible de que React es sin duda una librería madura y probada.

Un dato que debemos de tener en cuenta, es que React es desarrollado por Facebook, pero no solo eso, sino que es utilizado realmente por ellos, algo muy diferente con Angular, que, a pesar de ser mantenido por Google, no es utilizando por ellos para ningún sitio crítico. Esto demuestra la fe que tiene Facebook sobre React, pero sobre todo garantiza la constante evolución de esta gran librería.

Introducción a React

React es sin duda una de las tecnologías web más revolucionarias de la actualidad, pues proporciona todo un mecanismo de desarrollo de aplicaciones totalmente desacoplado del backend y ejecutado 100% del lado del cliente.

React fue lanzado por primera vez en 2013 por Facebook y es actualmente mantenido por ellos mismo y la comunidad de código abierto, la cual se extiende alrededor del mundo. React, a diferencia de muchas tecnologías del desarrollo web, es una librería, lo que lo hace mucho más fácil de implementar en muchos desarrollos, ya que se encarga **exclusivamente de la interface gráfica del usuario** y consume los datos a través de API que por lo general son REST.

El nombre de React proviene de su capacidad de crear **interfaces de usuario reactivas**, la cual es la capacidad de una aplicación para actualizar toda la interface gráfica en cadena, como si se tratara de una formula en Excel, donde al cambiar el valor de una celda automáticamente actualiza todas las celdas que depende del valor actualizado y esto se repite con las celdas que a la vez dependía de estas últimas. De esta misma forma, React reacciona a los cambios y actualiza en cascada toda la interface gráfica.

Uno de los datos interesantes de **React es que es ejecutado del lado del cliente** (navegador), y no requiere de peticiones GET para cambiar de una página a otra, pues toda la aplicación es empaquetada en un solo archivo JavaScript (bundle.js) que es descargado por el cliente cuando entra por primera vez a la página. De esta forma, la aplicación solo requerirá del backend para recuperar y actualizar los datos.

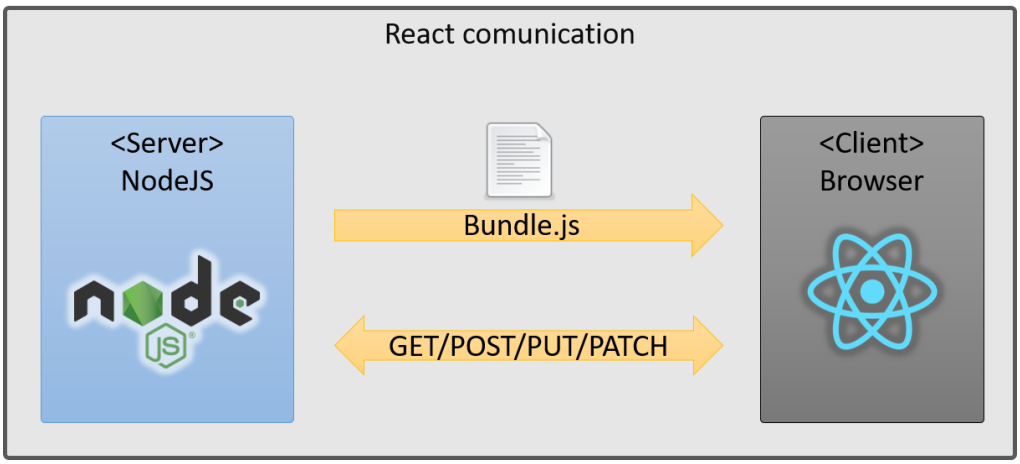


Fig. 1 - Comunicación cliente-servidor

React suele ser llamado React.js o ReactJS dado que es una **librería de JavaScript,** por lo tanto, el archivo descargable tiene la extensión `.js`, sin embargo, el nombre real es simplemente React.

Server Side Apps vs Single Page Apps

Para comprender como trabaja React es necesario entender dos conceptos claves, los cuales son Server side app (Aplicaciones del lado del servidor) y Single page app (Aplicaciones de una sola página)

Server side app

Las aplicaciones del lado del servidor, son aquellas en las que el código fuente de la aplicación está en un servidor y cuando un cliente accede a la aplicación, el servidor solo le manda el HTML de la página a mostrar en pantalla, de esta manera, cada vez que el usuario navega hacia una nueva sección de la página, el navegador lanza una petición GET al servidor y este le regresa la nueva página.

Esto implica que cada vez que el usuario de click en una sección se tendrá que comunicar con el servidor para que le regresa la nueva página, creado N solicitudes GET para N cambios de página. En una página del lado del servidor, cada petición retorna tanto el HTML para mostrar la página, como los datos que va a mostrar.

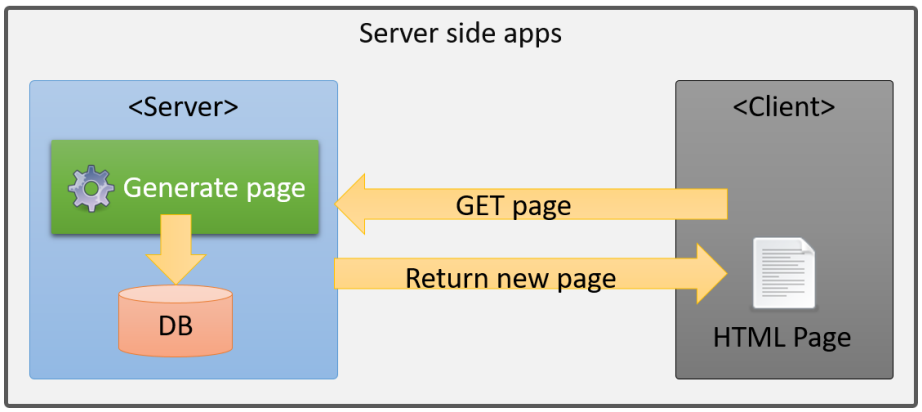


Fig. 2 - Server side apps Architecture

Como vemos en la imagen, el cliente lanza un GET para obtener la nueva página, el servidor tiene que hacer un procesamiento para generar la nueva página y tiene

que ir a la base de datos para obtener la información asociada a la página de respuesta. La nueva página es enviada al cliente y este solo la muestra en pantalla. En esta arquitectura **todo el trabajo lo hace el servidor** y el cliente solo se limita a mostrar las páginas que el server le envía.

Single page app

Las aplicaciones de una sola página se diferencian de las aplicaciones del lado del servidor debido a que gran parte del procesamiento y **la generación de las vistas las realiza directamente el cliente** (navegador). Por otro lado, el servidor solo expone un API mediante el cual, la aplicación puede consumir datos y realizar operaciones transaccionales.

En este tipo de arquitectura, se libera al servidor de una gran carga, pues no requiere tener que estar generando vistas para todos los usuarios conectados.

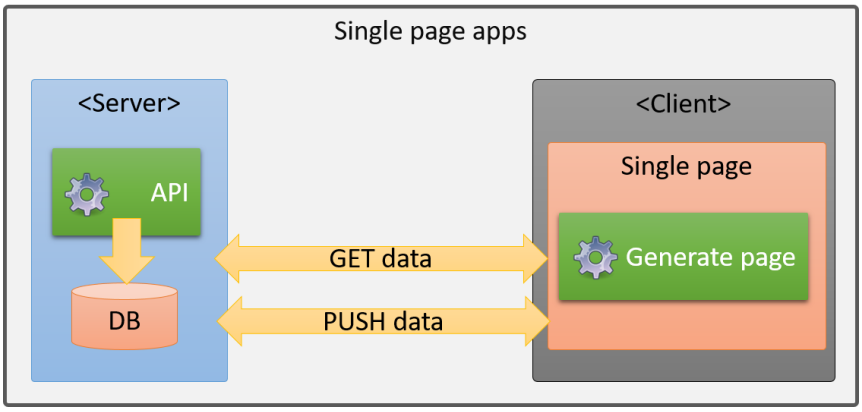


Fig. 3 - Single page apps Architecture.

Como vemos en esta nueva imagen, el cliente es el encargado de realizar las vistas y realizar algunos procesamientos, mientras que el servidor solo expone un API para acceder a los datos.

React es Single page app
Es muy importante resalta que React sigue la arquitectura de Single page app, por lo que las vistas son generadas del lado del cliente y solo se comunica al backend para obtener y guardar datos.

Introducción a NodeJS

NodeJS es sin duda una de las tecnologías que más rápido está creciendo, y que ya hoy en día es indispensable para cubrir posiciones de trabajo. NodeJS ha sido revolucionario en todos los aspectos, desde la forma de trabajar hasta que ejecuta JavaScript del lado del servidor.

NodeJS es básicamente un **entorno de ejecución JavaScript del lado del servidor**. Puede sonar extraño, ¿JavaScript del lado del servidor? ¿Pero que no JavaScript se ejecuta en el navegador del lado del cliente? Como ya lo platicamos, JavaScript nace inicialmente en la década de los noventas por los desarrolladores de Netscape, el cual fue creado para resolver problemas simples como validación de formularios, alertas y alguna que otra pequeña lógica de programación, nada complicado, sin embargo, JavaScript ha venido evolucionando hasta convertirse en un lenguaje de programación completo. NodeJS es creado por Ryan Lienhart Dahl, quien tuvo la gran idea de tomar el motor de JavaScript de Google Chrome llamado V8, y montarlo como el Core de NodeJS.

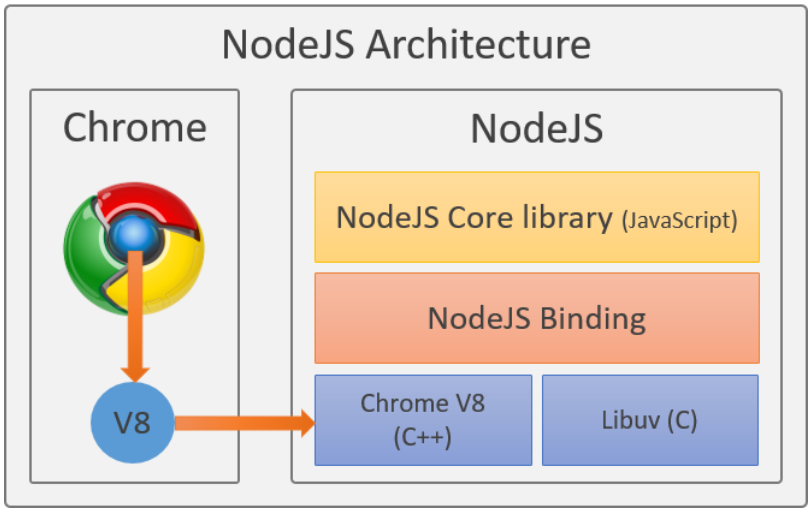


Fig. 4 - Arquitectura de NodeJS

Como puedes ver en la imagen, NodeJS es en realidad el motor de JavaScript V8 con una capa superior de librerías de NodeJS, las cuales se encargan de la comunicación entre el API de NodeJS y el Motor V8. Adicionalmente, se apoya de la librería Libuv la cual es utilizada para el procesamiento de entradas y salidas asíncronas.

Cabe mencionar que NodeJS es Open Source y multiplataforma, lo que le ha ayudado a ganar terrenos rápidamente.

Que es NodeJS (Según su creador):

Node.js® es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome. Node.js usa un modelo de operaciones E/S sin bloqueo y orientado a eventos, que lo hace liviano y eficiente. El ecosistema de paquetes de Node.js, npm, es el ecosistema más grande de librerías de código abierto en el mundo.

NodeJS y la arquitectura de Micro Servicios

Cuando hablamos de NodeJS es imposible no hablar de la arquitectura de Microservicios, ya que es una de las arquitecturas más utilizadas y que están de moda en la actualidad. La cual consiste en separar una aplicación en pequeños módulos que corren en contenedores totalmente aislados, en lugar de tener aplicaciones monolíticas que conglomeran toda la funcionalidad un mismo proceso.

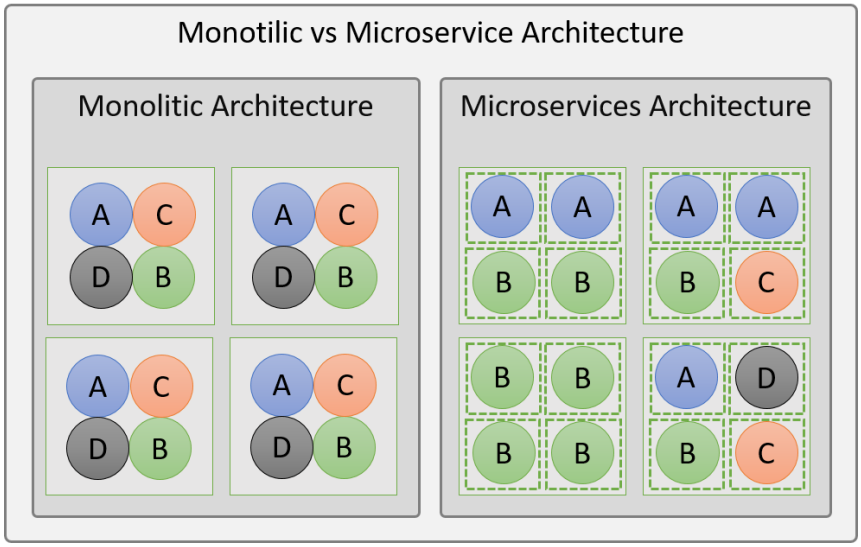


Fig. 5 - Arquitectura monolítica vs Microservices

En una arquitectura monolítica todas las aplicaciones escalan de forma monolítica, es decir, todos los módulos son replicados en cada nodo, aunque no se requiera de todo ese poder de procesamiento para un determinado módulo, en cambio, en una arquitectura de microservicios, los módulos se escalan a como sea requerido.

Todo esto viene al caso, debido a que NodeJS se ha convertido en unos de los servidores por excelencia para implementar microservicios, ya que es muy ligero y puede ser montado en servidores virtuales con muy pocos recursos, algo que es imposible con servidores de aplicaciones tradiciones como Wildfy, Websphere, Glashfish, IIS, etc.

Hoy en día es posible rentar un servidor virtual por 5 USD al mes con 512mb de RAM y montar una aplicación con NodeJS, algo realmente increíble y es por eso mi insistencia en que NodeJS es una de las tecnologías más prometedoras actualmente. Por ejemplo, yo suelo utilizar Digital Ocean, pues me permite rentar servidores desde 5 usd al mes.

Introducción a MongoDB

MongoDB es la base de datos NoSQL líder del marcado, ya que ha demostrado ser lo bastante madura para dar vida a aplicaciones completas. MongoDB nace con la idea de crear aplicaciones agiles, que permita realizar cambios a los modelos de datos si realizar grandes cambios en la aplicación. Esto es gracias a que permite almacenar su información en documentos en formato JSON.

Además, MongoDB es escalable, tiene buen rendimiento y permite alta disponibilidad, escalando de un servidor único a arquitecturas complejas de centros de datos. MongoDB es mucho más rápido de lo que podrías imaginas, pues potencia la computación en memoria.

Uno de los principales retos al trabajar con MongoDB es entender cómo funciona el paradigma NoSQL y abrir la mente para dejar a un lado las tablas y las columnas para pasar un nuevo modelo de datos de Colecciones y documentos, los cuales no son más que estructuras de datos en formato JSON.

Actualmente existe una satanización contra MongoDB y de todas las bases de datos NoSQL en general, ya que los programadores habituales temen salir de su zona de confort y experimentar con nuevos paradigmas. Esta satanización se debe en parte a dos grandes causas: el desconocimiento y la incorrecta implementación. Me explico, el desconocimiento se debe a que los programadores no logran salir del concepto de tablas y columnas, por lo que no encuentra la forma de realizar las operaciones que normalmente hacen con una base de datos tradicional, como es hacer un Join, crear procedimientos almacenados y crear transacciones. Y Finalmente la incorrecta implementación se debe a que los programadores inexpertos o ignorantes utilizan MongoDB para solucionar problemas para los que no fue diseñado, llevando el proyecto directo al fracaso.

Es probable que hallas notado que dije que MongoDB no soporta transacciones, y es verdad, pero eso no significa que MongoDB no sirva, si no que no fue diseñado para aplicaciones que requieren de transacciones y bloque de registro, un que es posible simularlos con un poco de programación. Más adelante analizaremos más a detalle estos puntos.

Bases de datos Relacionales VS MongoDB

Probablemente lo que más nos cueste entender cuando trabajamos con MongoDB, es que no utiliza tablas ni columnas, y en su lugar, la información se **almacena en objetos completos en formato JSON**, a los que llamamos **documentos**. Un documento se utiliza para representar mucha información contenida en un solo objeto, que, en una base de datos relacional, probablemente guardaríamos en más de una tabla. Un documento MongoDB suele ser un objeto muy grande, que se asemejan a un árbol, y dicho árbol suele tener varios niveles de profundidad, debido a esto, MongoDB requiere de realizar *joins* para armar toda la información, pues un documento por sí solo, contiene toda la información requerida

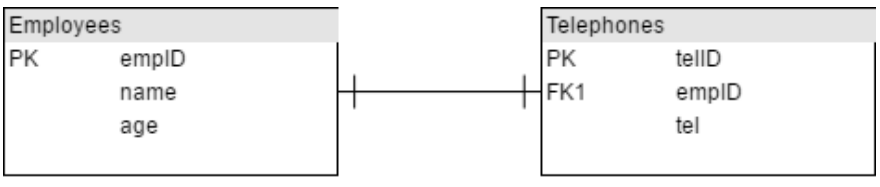
Otros de los aspectos importantes de utilizar documentos, es que no existe una estructura rígida, que nos obligue a crear objetos de una determinada estructura, a diferencia una base de datos SQL, en la cual, las columnas determinan la estructura de la información. En MongoDB no existe el concepto de columnas, por lo que los dos siguientes objetos podrían ser guardados sin restricción alguna:

```
1.  {
2.      "name": "Juan Perez",
3.      "age":  20,
4.      "tel":  "1234567890"
5.  }
```

```
6.  {
7.      "name": "Juan Perez",
8.      "age":  20,
9.      "tels": [
10.         "1234567890",
11.         "0987654321"
12.     ]
13. }
```

Observemos que los dos objetos son relativamente similares, y tiene los mismos campos, sin embargo, uno tiene un campo para el teléfono, mientras que el segundo objeto, tiene una lista de teléfonos. Es evidente que aquí tenemos dos incongruencias con un modelo de bases de datos relacional, el primero, es que tenemos campos diferentes para el teléfono, ya que uno se llama *tel* y el otro *tels*. La segunda incongruencia, es que la propiedad *tels*, del segundo objeto, es en realidad un arreglo, lo cual en una DB relacional, sería una tabla secundaria unida con un Foreign Key.

El segundo objeto se vería de la siguiente manera modelado en una DB relacional:



Como ya nos podemos dar una idea, en un DB relacionar, es necesario tener estructuras definidas y relacionadas entre sí, mientras que en MongoDB, se guardan documentos completos, los cuales contiene en sí mismo todas las relaciones requeridas. Puede sonar algo loco todo esto, pero cuando entremos a desarrollar nuestra API REST veremos cómo utilizar correctamente MongoDB.

La relación entre React, NodeJs & MongoDB

Hasta este momento, ya hemos hablado de estas tres tecnologías de forma individual, sin embargo, no hemos analizado como que estas se combinan para crear proyectos profesionales.

Resumiendo un poco lo ya platicado, **React se encargará de la interface gráfica de usuario** (UI) y será la encargada de solicitar información al Backend a medida que el usuario navega por la página. Por otra parte, tenemos NodeJS, al cual utilizaremos para programar la **lógica de negocio** y la comunicación con la base de datos, mediante NodeJS expondremos un API REST que luego React consumirá. Finalmente, MongoDB es la base de datos en donde **almacenamos la información**.

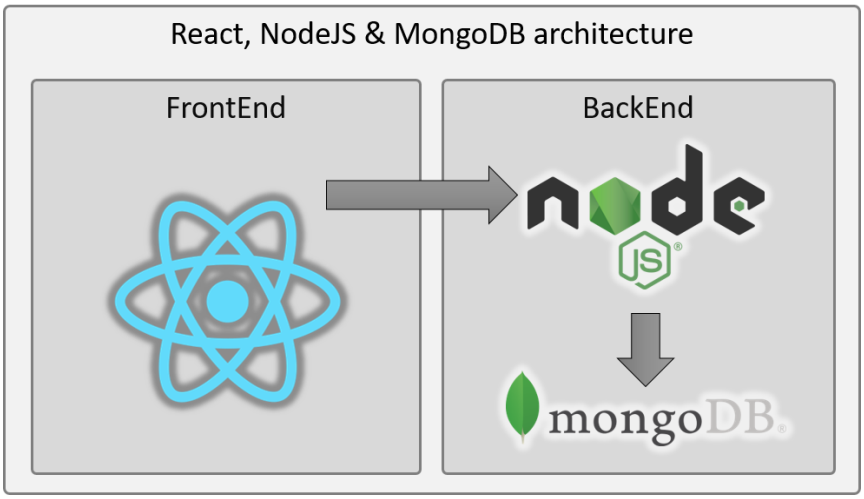


Fig. 6 - MERN Stack

Como vemos en la imagen, React está del lado del FrontEnd, lo que significa que su único rol es la representación de los datos y la apariencia gráfica. En el BackEnd tenemos a NodeJS, quien es el intermediario entre React y MongoDB. MongoDB también está en el Backend, pero este no suele ser accedido de forma directa por temas de seguridad.

Cuando una serie de tecnologías es utilizada en conjunto, se le llama **Stack**, el cual hace referencia a todas las tecnologías implementadas. A lo largo de este libro, utilizaremos el Stack **MERN**, el cual es el acrónimo de MongoDB, Express, React & NodeJS, estas

tecnologías suelen acompañarse con Webpack y Redux. Puedes encontrar más información de MERN en la página web: http://mern.io/

Resumen

En este capítulo hemos dado un vistazo rápido a la historia de JavaScript y como es que ha evolucionado hasta convertirse en un lenguaje de programación completo. De la misma forma, hemos analizado como es que JavaScript es la base para tecnologías tan importantes como React y NodeJS.

También hemos analizado de forma rápida a React, NodeJS y MongoDB para dejar claro los conceptos y como es que estas 3 tecnologías se acoplan para dar soluciones de software robustas.

Por ahora no te preocupes si algún concepto no quedo claro o no entiendes como es que estas 3 tecnologías se combinan, ya que más adelante entraremos mucho más a detalle.

Introducción al proyecto Mini Twitter

Capítulo 5

Hasta este punto, ya hemos visto muchas de las características de React y ya estamos listo para empezar a desarrollar el proyecto Mini Twitter, el cual es una réplica de la famosa red social Twitter. Esta app no busca ser una copia de la aplicación, si no ejemplo educativo que nos lleve de la mano para construir una aplicación Totalmente funcional utilizando las tecnologías de React, NodeJS y MongoDB.

Debido a que la aplicación Mini Twitter abarca el desarrollo del FrontEnd y el BankEnd, organizaremos el libro de tal forma que, primero veremos toda la parte del FrontEnd, en donde aprenderemos React y utilizaremos Bootstrap como framework para hacer nuestra aplicación Responsive. Una vez terminada la aplicación, tendremos un capítulo especial para estudiar Redux y cómo implementarlo en nuestro proyecto. En segundo lugar, veremos el desarrollo de un API REST, utilizando NodeJS + Express, el popular framework de NodeJS para desarrollo web. Adicional, veremos todo lo referente a MongoDB, como crear los modelos, conectarnos, realizar consultar y actualizar los datos.

Un vistazo rápido al proyecto

Antes de iniciar con el desarrollo del proyecto Mini Twitter, es importante entender lo que vamos a estar desarrollando, y es por eso que, este capítulo está especialmente dedicado a ello. Lo primero que haremos será dar un tour por la aplicación terminada, luego regresaremos para analizar cómo está compuesta, y finalmente iniciar con el desarrollo.

Página de inicio

La siguiente página corresponde a la página de inicio de Mini Twitter, en la cual podemos ver el menú bar en la parte superior, los datos del usuario del lado izquierdo, en el centro tenemos los tweets y del lado derecho, tenemos una lista de usuarios sugeridos.

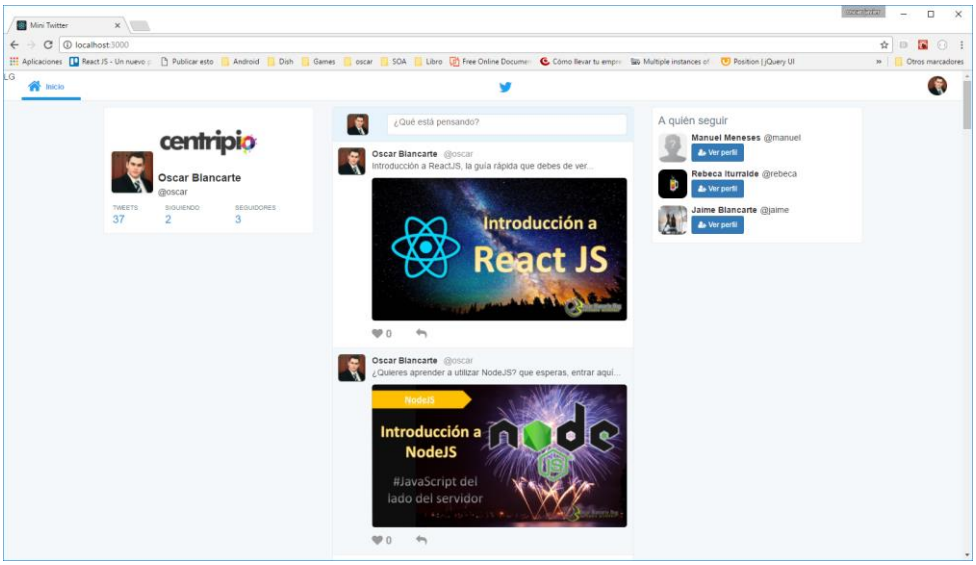


Fig. 52 - Página de inicio de Mini Twitter

Perfil de usuario

La siguiente imagen corresponde a la página de perfil de los usuarios, en la cual puede visualizar un Banner, la foto de perfil, su nombre de usuario y una descripción. En el centro podemos visualizar los tweets del usuario, seguidores o los que lo siguen, del lado derecho, tenemos nuevamente usuario sugeridos.

Fig. 53 - Página de perfil de Mini Twitter

Editar perfil de usuario

La página de perfil puede cambiar de estado a editable, la cual permite cambiar la foto de perfil, banner, nombre y la descripción:

Fig. 54 - Perfil de usuario en modo edición.

Observemos que el banner, cómo la foto, cambian, agregando un ícono de una cámara, la cual, al poner el mouse enzima se subrayada de color naranja, habitando cargar una foto nueva con tan solo hacer clic. También, donde aparece el nombre de usuario y descripción, pasan a ser editables.

Página de seguidores

La siguiente foto corresponde a la sección de seguidores, la cual es igual a la de las personas que lo siguen:

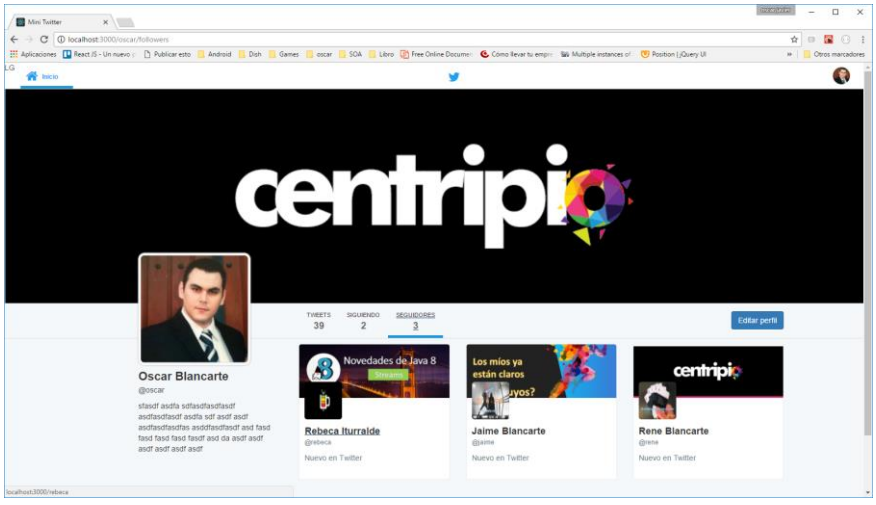


Fig. 55 - Sección de seguidores de Mini Twitter

Detalle del Tweet

También es posible ver el detalle de cada Tweet, para ver los comentarios que tiene y agregar nuevos.

Fig. 56 - Detalle de un Tweet en Mini Twitter

Inicio de sesión (Login)

Otra de las páginas que cuenta la aplicación son las clásicas pantallas de iniciar sección (login), la cual autenticarse ante la aplicación mediante usuario y password:

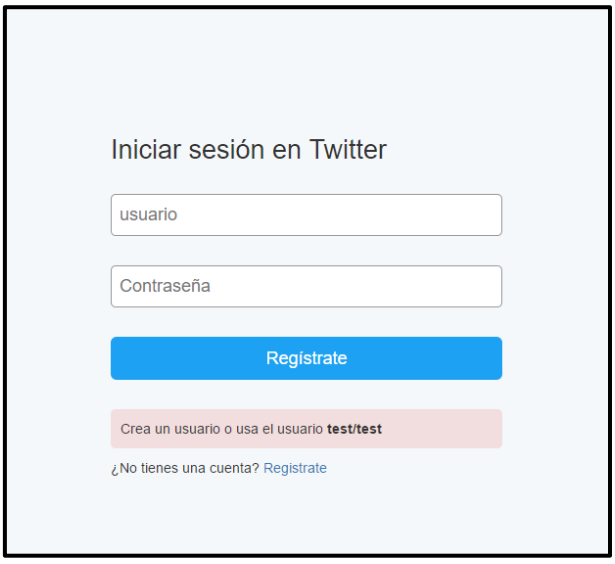


Fig. 57 - Inicio de sesión de Mini Twitter

Registro de usuarios (Signup)

Mediante esta página, es posible crear una nueva cuenta para poder acceder a la aplicación.

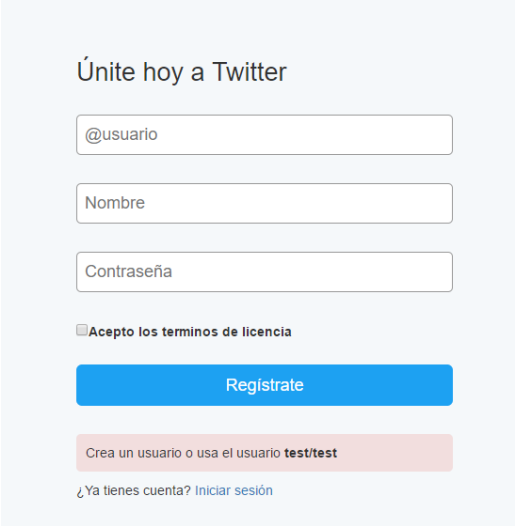


Fig. 58 - Página de registro de Mini Twitter

Hemos visto un recorrido rápido a lo que será la aplicación de Mini Twitter, pero la aplicación es engañosa, porque tiene muchas más cosas de las podemos ver a simple vista, las cuales tenemos que analizar mucho más a detalle. Es por ese motivo, que una vez que dimos un tour rápido de la aplicación, es hora de verla con rayos X y ver cómo es que la aplicación se compone y todos los Components que vamos a requerir para termina la aplicación.

Análisis al prototipo del proyecto

En esta sección analizaremos con mucho detalle todos los componentes que conforman la aplicación Mini Twitter.

Componente TwitterDashboard

TwitterDashboard es la página de inicio para un usuario autenticado, en la cual es posible ver los últimos Tweets de los usuarios, también es posible ver un pequeño resumen de tu perfil (lazo izquierdo) y una lista de usuario sugeridos para seguir (lado derecho).

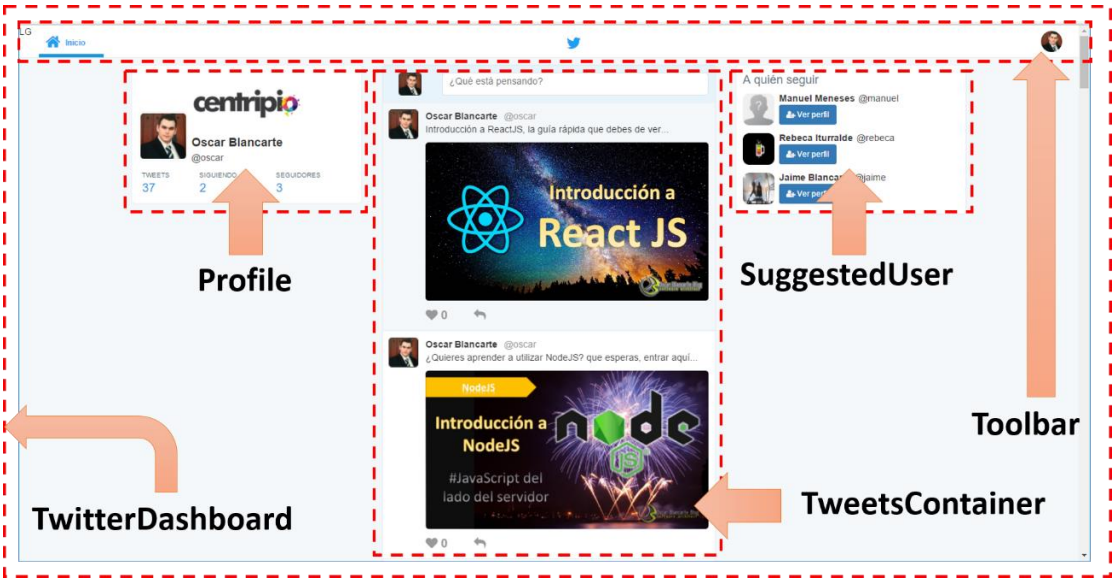


Fig. 59 - Topología de la página de inicio

En la imagen anterior, podemos ver con más detalle cómo está compuesta la página de inicio. A simple vista, es posible ver 5 componentes, los cuales son:

- **TwitterDashboard:** Es un componente contenedor, pues alberga al resto componentes que podemos ver en pantalla.
- **Toolbar:** Componente que muestra al usuario autenticado.
- **Profile:** Componente que muestra los datos del usuario autenticado, como foto, número de Tweets, número de suscriptores y personas que lo sigue.
- **TweetsContainer:** Es un componente contendedor, pues en realidad solo muestra una serie de componentes Tweet, los cuales veremos con más detalle más adelante.
- **SuggestedUser:** Muestra una lista de usuario sugeridos para seguir.

Adicional a los componentes que podemos ver en pantalla, existe uno más llamado *`TwitterApp`*, el cual envuelve toda la aplicación e incluyendo las demás componentes, como las páginas de login, signup, y el perfil del usuario.

Componente TweetsContainer

Hora daremos un zoom al componente *`TweetsContainer`* para ver cómo está compuesto:

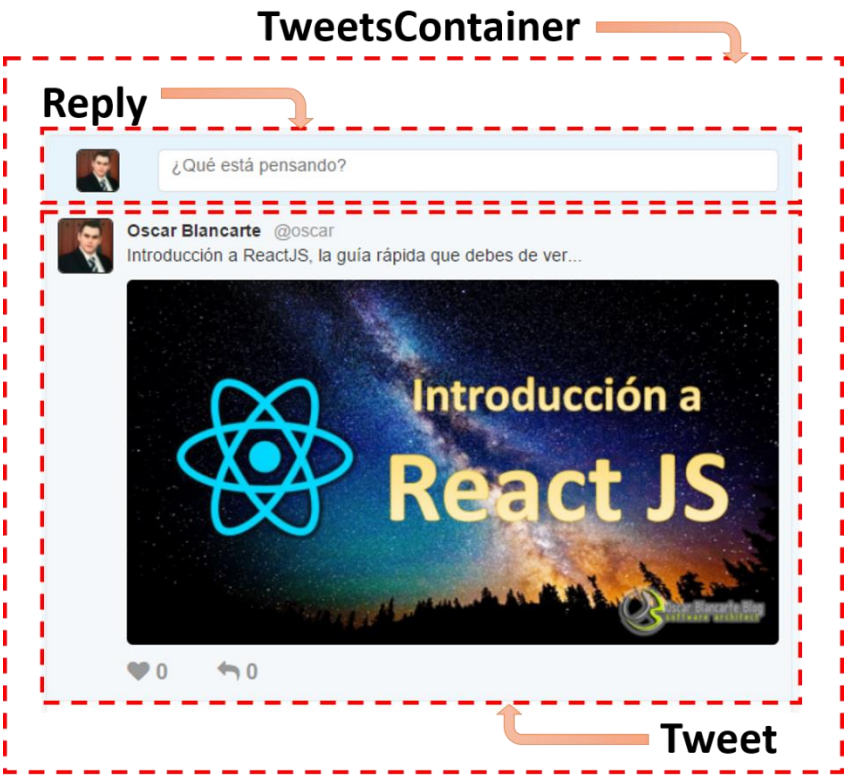


Fig. 60 - Topología del componente TweetsContainer

Como podemos apreciar en la imagen, el componente está conformado del componente *`Reply`*, el cual sirve para crear un nuevo Tweet. Adicional, es posible ver una lista de componentes *`Tweet`*, que corresponde a cada tweet de los usuarios.

Componente UserPage

La página de perfil, permite a los usuarios ver su perfil y ver el perfil de los demás usuarios. Este componente se muestra de dos formas posibles, ya que si estás en tu propio los datos siempre y cuando estés en tu propio perfil. Por otra parte, si estas en el perfil de otro usuario, te dará la opción de seguirlo.

Fig. 61 - Topología de la página UserPage

En esta página es posible ver varios componentes que se reúnen para formar la página:

- **UserPage:** Es un componente contenedor, pues alberga al resto de componentes, como son SuggestedUsers, Followers, Followings, TweetsContainer.
- **TweetsContainer:** Este componente ya lo analizamos y aquí solo lo reutilizamos.
- **SuggestedUsers:** Nuevamente, este componente ya lo analizamos y solamente lo reutilizado.
- **Followings:** Este componente se muestra solo cuando presionamos el tab "siguiendo", el cual muestra un listado de todas las personas que seguimos.
- **Followers:** Igual que el anterior, solo que esta muestra nuestros seguidores.

Componente Signup

Este es el formulario para crear un nuevo usuario, el cual solo solicita datos mínimos para crear un nuevo perfil.

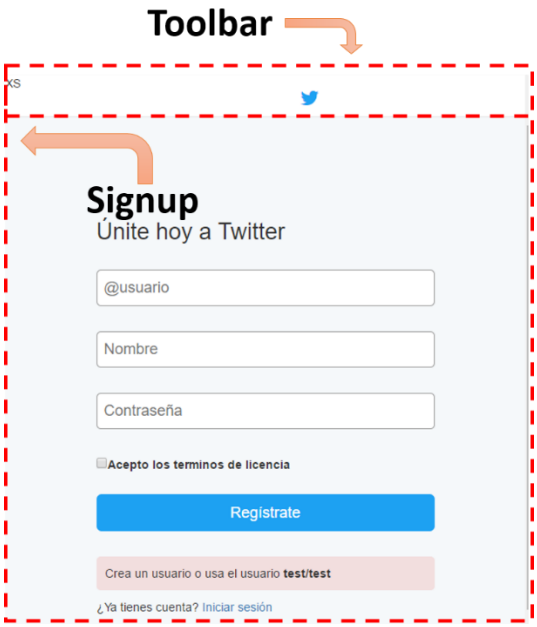


Fig. 62 - Topología de la página Signup

La página de login está compuesta únicamente por el componente *`Login`* y *`Toolbar`*.

Componente Login

La página de Login es bastante parecida a la de Signup, es solo un formulario donde el usuario captura su usuario y password para autenticarse.

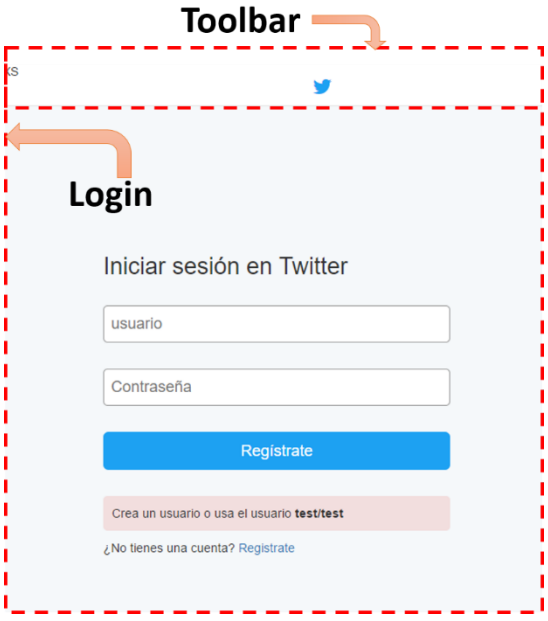


Fig. 63 - Topología de la página Login

Hasta este momento, hemos visto los componentes principales de la aplicación, lo que falta son algunas componentes de popup y compontes secundarios en los cuales no me gustaría nombrar aquí, pues no quisiera entrar en mucho detalle para no perdernos. Por ahora, con que tengamos una idea básica de cómo está formada la aplicación será más que suficiente y a medida que entremos en los detalles, explicaremos los componentes restantes.

Jerarquía de los componentes del proyecto

Tal vez recuerdes que en el capítulo pasado hablamos acerca de la jerarquía de componentes, pues en este capítulo mostraremos la jerarquía completa del proyecto. La idea es que puedas imprimir esta imagen o guardarla en un lugar accesible, ya que nos servirá muchísimo para entender cómo vamos a ir armando el proyecto, así de cómo vamos a reutilizar los componentes.

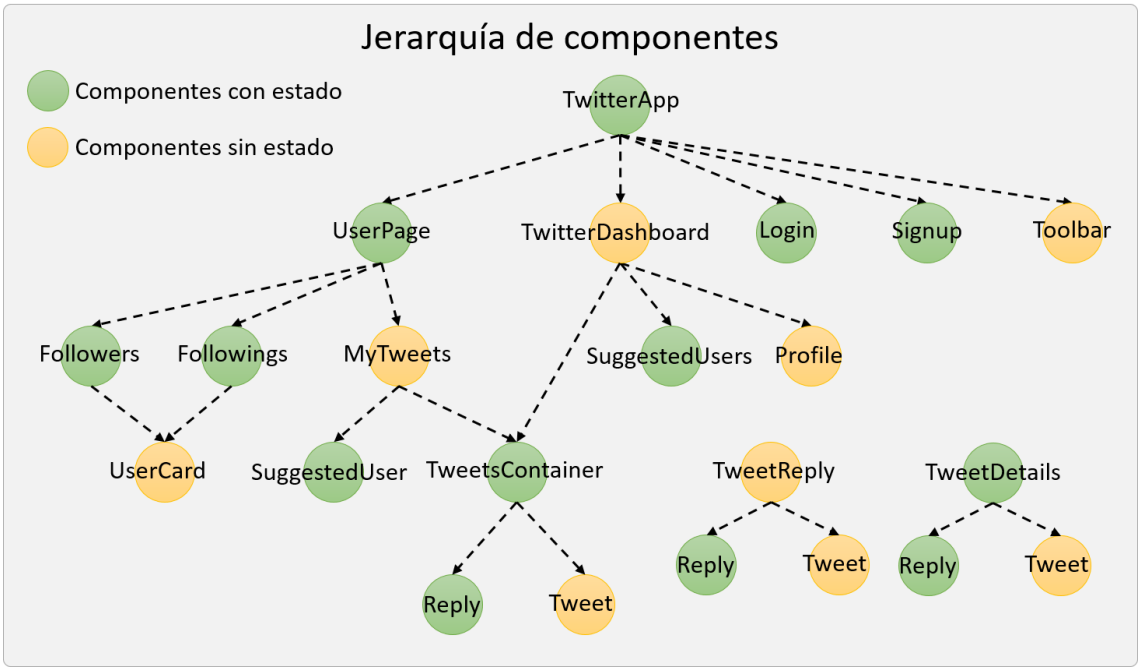


Fig. 64 - Jerarquía de componentes de Mini Twitter

La imagen anterior, nos da una fotografía general de toda la aplicación Mini Twitter, en la cual podemos ver cómo está compuesto cada componente, así como también, podemos aprecias donde reutilizamos los componentes.

El enfoque Top-down & Bottom-up

Uno de los aspectos más importantes cuando vamos a desarrollar una nueva aplicación, es determina el orden en que vamos a construir los componentes, pues la estrategia que tomemos, repercutirá en la forma que vamos a trabajar. Es por este motivo que vamos a presentar el en enfoque Top-down y Bottom-up para analizar sus diferencias y las ventajas que traen cada una.

Top-down

Este enfoque consiste en empezar a construir los componentes de más arriba en la jerarquía y continuar desarrollando los componentes hacia abajo. Este este es el enfoque más simple y que es utilizado con más frecuencia por desarrolladores inexpertos o proyectos donde no hubo una fase de análisis que ayudara a identificar los componentes necesarios y su jerarquía.

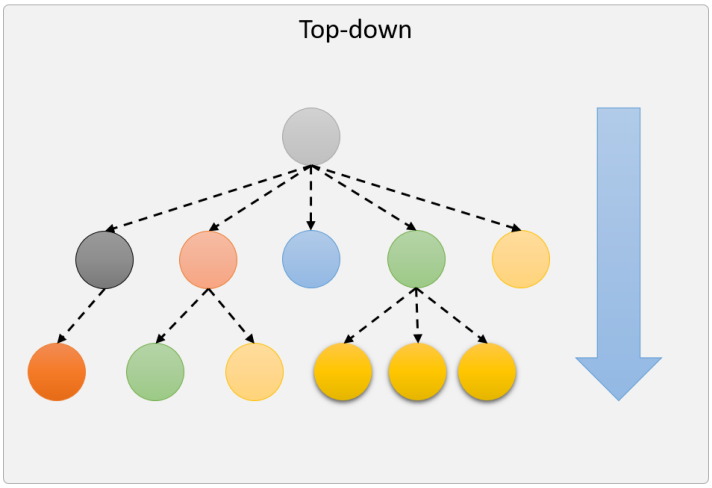


Fig. 65 - El enfoque Top-down

En este enfoque se requiere poco o nula planeación, pues se empieza a construir de lo menos especifico o lo más específico, de esta manera, vamos construyendo los componentes a como los vamos requiriendo en el componente padre. El inconveniente de este enfoque, es que es muy propenso a la refactorización, pues a medida que vamos descendiendo en la jerarquía, vamos descubriendo datos que requeríamos arrastrar desde la parte superior, también descubrimos que algunos componentes que ya desarrollamos pudieron ser reutilizados, por lo que se requiere refactorizar para reutilizarlos o en el peor de los casos, hacemos un componente casi idéntico para no

modificar el trabajo que ya tenemos echo. Otra de las desventajas, es que casi cualquier dependencia a otros componentes que requiera algún componente, no existirá y tendremos que irlos creando al vuelo.

Bottom-up

Este otro enfoque es todo lo contrario que Top-down, pues propone empezar con los componentes más abajo en la jerarquía, de esta forma, iniciamos con los componentes que no tienen dependencias y vamos subiendo en la jerarquía hasta llegar al primer componente en la jerarquía.

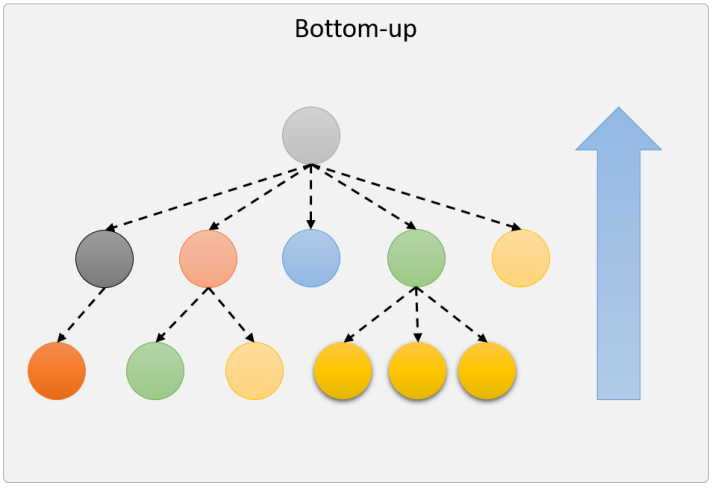


Fig. 66 - El enfoque Bottom-up

El enfoque Bottom-up requiere de una planeación mucho mejor, en la cual salgan a relucir el mayor número de componentes requeridos para el desarrollo, también se identifica la información que requiere cada componente y se va contemplando a medida que subimos en la jerarquía.

Este enfoque es utilizado por los desarrolladores más experimentados, que son capases de analizar con buen detalle la aplicación a desarrollar. Un mal análisis puede hacer que replanteemos gran parte de la estructura y con ellos, se genera un gran impacto en el desarrollo.

Bien ejecutado, reduce drásticamente la necesidad de realizar refactor, también ayuda a identificar todos los datos y servicios que será necesarios para la aplicación en general.

El enfoque utilizado y porque

Aunque lo mejor sería utilizar el enfoque Botton-up, la realidad es que apenas estamos aprendiendo a utilizar React, por lo que aventurarnos a utilizar este enfoque puede ser un poco riesgoso y nos puede complicar más el aprendizaje. Es por este motivo que en este libro utilizaremos el enfoque Top-down he iremos construyendo los componentes a como sea requeridos.

Preparando el entorno del proyecto

Dado que la aplicación Mini Twitter cuenta con una serie de servicios para funcionar, deberemos instalar y ejecutar nuestra API Rest antes de empezar a desarrollar, ya que toda la información que consultemos o actualicemos, será por medio del API REST.

Por el momento no entraremos en detalles acerca del API REST, pues más adelante hablaremos de cómo desarrollar desde cero, todo el API, publicarlos y prepararlo para producción. Por ahora, solo instalaremos el API y lo utilizaremos.

Instalar API REST

Para instalar el API, deberemos bajar la última versión del repo, es decir el branch "**Capitulo-16-Produccion**" directamente desde el repo del libro:

https://github.com/oscarjb1/books-reactiveprogramming.git

El siguiente paso será cambiar la configuración de conexión a la base de datos. Para esto, será necesario regresar al Mongo Atlas para recuperar el String de conexión. Una vez que ya estemos allí, deberemos presionar en "**CONNECT**" y en la pantalla emergente presionaremos "**Connect Your Application**" y en la nueva pantalla presionamos el botón "**COPY**" como ya lo habíamos visto.

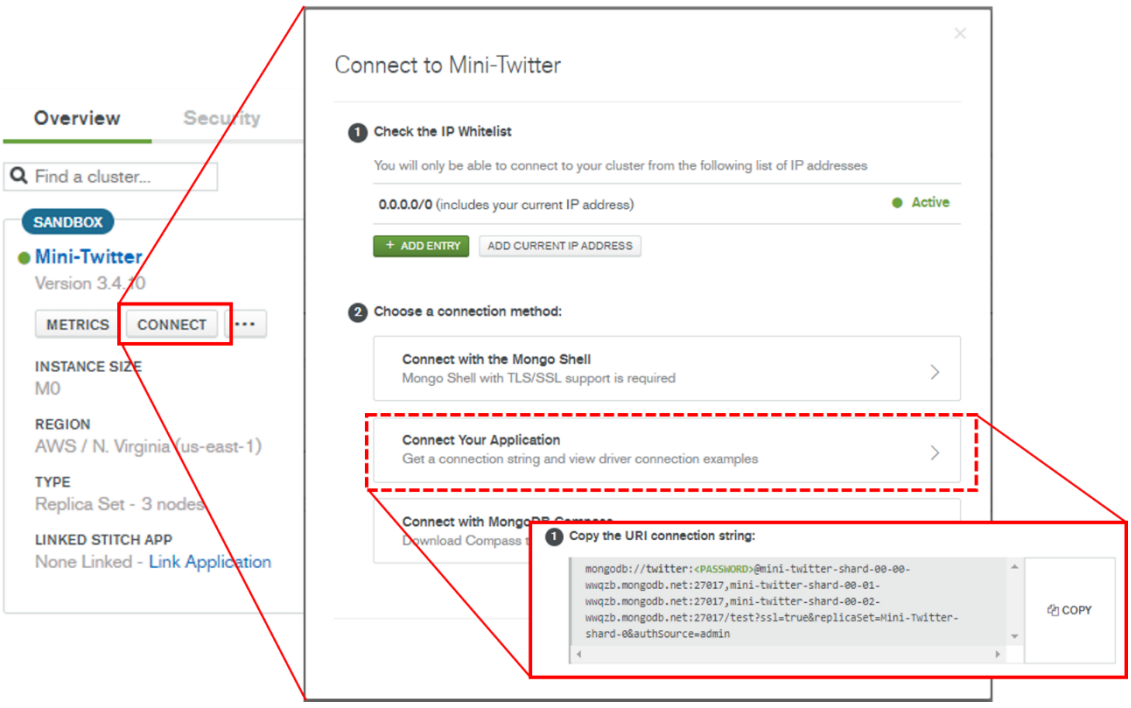


Fig. 67 - Obtener el String de conexión.

Este String de conexión lo deberemos de agregar en las secciones *connectionString* del archivo *config.js*, por ahora dejaremos el mismo el mismo valor para *development* y *production*. No olvides remplazar la sección **<PASSWORD>** del String de conexión por el password real.

```
1.  module.exports = {
2.    server: {
3.      port: 3000
4.    },
5.    tweets: {
6.      maxTweetSize: 140
7.    },
8.    mongodb: {
9.      development: {
10.       connectionString: "<Copy connection String from Mongo Atlas>"
11.     },
12.     production: {
13.       connectionString: "<Copy connection String from Mongo Atlas>"
14.     }
15.   },
16.   jwt: {
17.     secret: "#$%EGt2eT##$EG%Y$Y&U&/IETRH45W$%whth$Y$%YGRT"
18.   }
19. }
```

Aprovecharemos para **cambiar el puerto**, de tal forma que cambiaremos del *80* al *3000*, como lo podemos ver en la línea 3.

Lo siguiente será configurar el sistema operativo para encontrar nuestra API como un subdominio del *localhost*, para ello, realizaremos la siguiente configuración (siempre como administrador):

En Windows:

No dirigimos a la carpeta *C:\Windows\System32\drivers\etc* y abrimos el archivo *hosts* como administrador, y agregamos la siguiente línea:

```
1.  127.0.0.1    api.localhost
```

En Linux

En el caso de Linux, el procedimiento es exactamente el mismo, solamente que el archivo que tendremos que editar es */etc/hosts/*. De tal forma que agregaremos solamente la siguiente línea:

```
1.  127.0.0.1    api.localhost
```

En Mac

En Mac, tendremos que hacer exactamente lo mismo que en los anteriores, sin embargo, el archivo se encuentra en */private/etc/hosts*, allí agregaremos la línea:

```
1.  127.0.0.1    api.localhost
```

NOTA: Esto hará que cualquier llamada al subdominio *api.** se redirija al *localhost*.

Iniciar el API

Seguido, abrimos la terminal y nos dirigimos a la raíz del API (donde descargamos el repo), una vez allí ejecutamos el siguiente comando:

npm start.

Con esto, si todo sale bien, habremos iniciado correctamente el API REST.

Probando nuestra API

Una vez iniciado el servidor, entramos al navegador y navegamos a la URL: http://api.localhost:3000/ la cual nos mostrará la siguiente pantalla:

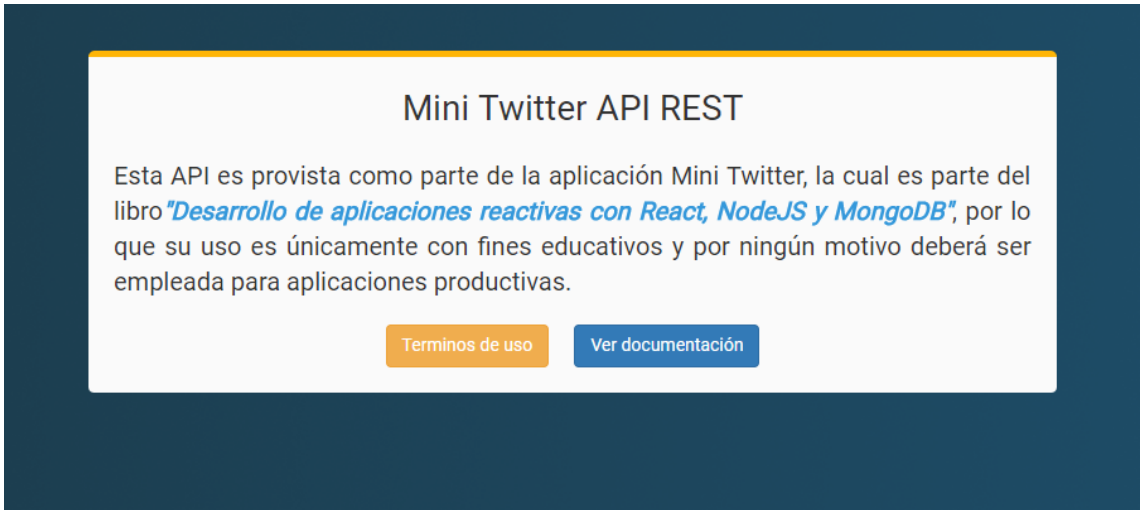


Fig. 68 - Probando el API REST

Si en el navegador podemos la pantalla anterior, quiere decir que hemos instalado el API Correctamente. También podrá revisar la documentación de todos los servicios que utilizaremos al presionar el botón "**Ver documentación**".

Fig. 69 - Documentación del API.

En cada servicio podemos ver el nombre, la descripción y el método en el que acepta peticiones, el path para ejecutarlo y una leyenda que indica si el servicio tiene seguridad.

Creando un usuario de prueba

Antes de poder empezar a utilizar nuestra API, será necesario crear un usuario, por lo cual, utilizaremos un archivo de utilidad llamado *InitMongoDB.js*, que se encargará de esto. El archivo se ve así:

```
1.  var mongoose = require('mongoose')
2.  var configuration = require('./config')
3.  var Profile = require('./api/models/Profile')
4.  var bcrypt = require('bcrypt')
5.
6.  var opts = {
7.      server: {
8.          socketOptions: {keepAlive: 1}
9.      }
10. }
11. mongoose.connect(configuration.mongodb.development.connectionString, opts)
12.
13. const newProfile = new Profile({
14.   name: "Usuario de prueba",
15.   userName: "test",
16.   password: bcrypt.hashSync('1234', 10)
17. })
18.
19. Profile.findOne({username: 'test'}, function(err, queryUser){
20.   if(queryUser !== null){
21.     console.log("====> El usuario de prueba ya ha sido registrado")
22.     process.exit()
23.   }
24. })
25.
26. newProfile.save(function(err){
27.   if(err){
28.     console.log("====> Error al crear el usuario de prueba",err)
29.     process.exit()
30.     return
31.   }
32.   console.log("====> Usuario de prueba creado correctamente")
33.   process.exit()
34. })
```

No vamos entrar en los detalles, pues este archivo tiene cosas avanzadas que no hemos visto aún. Sin embargo, lo muestro por si desean analizarlo.

Por ahora, tendremos que dirigirnos a la terminal y dirigirnos a la carpeta del API (donde descargamos el código) y ejecutar el comando

node InitMongoDB.js

El resultado será el siguiente:

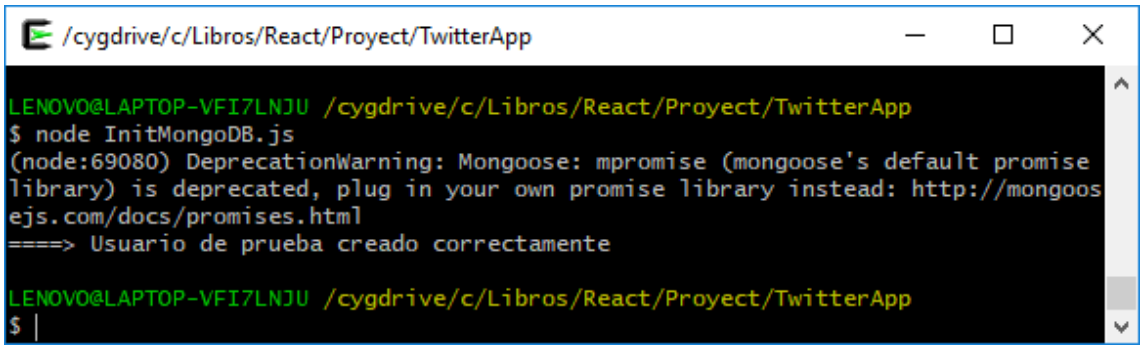


Fig. 70 - Creación del usuario de prueba.

Si sale algún error, deberemos revisar los datos de conexión a la base de datos.

El usuario creado por la ejecución pasada será:

Username: test
Password: 1234

Para comprobar que todo salió bien, podemos regresar a Compass y ver la colección *profiles*, en ella deberíamos ver el usuario:

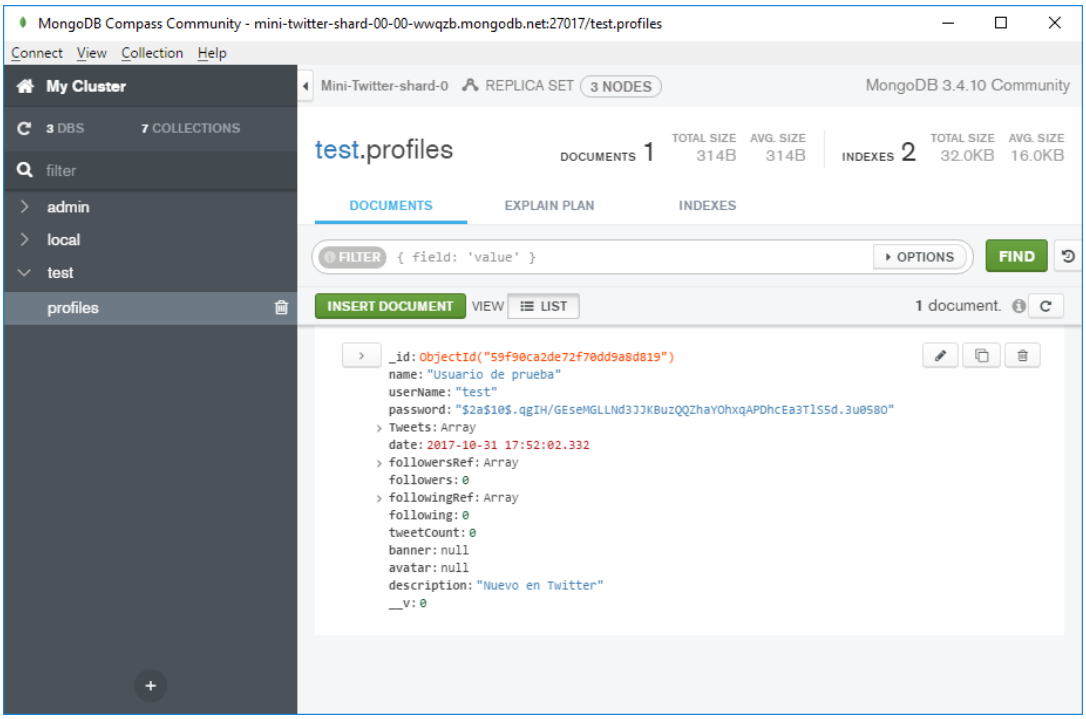


Fig. 71 - Usuario Test creado en MongoDB.

Del lado izquierdo seleccionamos **test ➔ profiles** y veremos podremos ver los usuarios. Si no puedes ver la sección "**profiles**" tendrás que actualizar la vista.

Invocando el API REST desde React

En este punto ya tenemos el API funcionando y listo para ser utilizando, pero debemos aprender cómo es que un servicio REST es consumido desde una aplicación React.

React proporciona la función *fetch* la cual se utiliza para consumir cualquier recurso den la WEB mediante *HTTP*, por lo que es posible consumir servicios REST mediante el método *POST*, *GET*, *PUT*, *DELETE* y *PATCH*. Esta función recibe dos parámetros para funcionar, el primero corresponde a la URL en la que se encuentre el servicio y el segundo parámetro corresponde a los parámetros de invocación, como el Header y el Body de la petición.

Veamos un ejemplo simple de cómo consumir los Tweets de un usuario. Para esto deberemos crear una nueva clase en llamada *APIInvoker.js* en el directorio */app/utils/*. El archivo se ve la siguiente manera:

```
1.  class APIInvoker {
2.
3.    invoke(url, okCallback, failCallback, params){
4.        fetch(`http://api.localhost:3000${url}`, params)
5.        .then((response) => {
6.          return response.json()
7.        })
8.        .then((responseData) => {
9.            if(responseData.ok){
10.             okCallback(responseData)
11.           }else{
12.             failCallback(responseData)
13.           }
14.       })
15.     }
16. }
17. export default new APIInvoker();
```

Lo primero a tomar en cuenta es la función *invoke*, la cual servirá para consumir servicios del API de una forma reutilizable. Esta función recibe 4 parámetros obligatorios:

1. **url:** String que representa el recurso que se quiere consumir, sin contener el host y el puerto, ejemplo *"/tweets"*.
2. **okCallback:** deberá ser una función, la cual se llamará solo en caso de que el servicio responda correctamente. La función deberá admitir un parámetro que representa la respuesta del servicio.
3. **failCallback:** funciona igual al anterior, solo que este se ejecuta cuando el servicio responde con algún error.
4. **params:** Representa los parámetros de invocación HTTP, como son los header y body.

De estos cuatro parámetros solo dos son requeridos por la función *fetch* en la línea 4, la *url* y *params*. Cuando el servicio responde, la respuesta es procesada mediante una promesa (*Promise*), es decir, una serie de *then*, los cuales procesan la respuesta por partes. El primer *then* (línea 5) convierte la respuesta en un objeto *json* (línea 6) y lo retorna para ser procesado por el siguiente *then* (línea 8), el cual, valida la propiedad *ok* de la respuesta, si el valor de esta propiedad es *true*, indica que el servicio termino correctamente y llamada la función *okCallback*, por otra parte, si el valor es *false*, indica que algo salió mal y se ejecuta la función *failCallback*.

Finalmente exportamos una nueva instancia de la clase *APIInvoker* en la línea 17, con la finalidad de poder utilizar el objeto desde los componentes de React.

El siguiente paso será probar nuestra clase con un pequeño ejemplo que consulte los Tweets de nuestro usuario de pruebas, para esto modificaremos la clase App para dejarla de la siguiente manera:

```
1.  import React from 'react'
2.  import { render } from 'react-dom'
3.  import APIInvoker from "./utils/APIInvoker"
4.
5.  class App extends React.Component{
6.
7.    constructor(props){
8.      super(props)
9.      this.state = {
10.       tweets: []
11.     }
12.   }
13.
14.   componentWillMount(){
15.     let params = {
16.       method: 'put',
17.       headers: {
18.         'Content-Type': 'application/json',
19.       },
20.       body: null
21.     };
22.
23.     APIInvoker.invoke('/tweets/test', response => {
24.       this.setState({
25.         tweets: response.body
26.       })
27.     },error => {
28.       console.log("Error al cargar los Tweets", error);
29.     })
30.   }
31.
32.   render(){
33.     console.log(this.state.tweets);
34.     return (
35.       <ul>
36.         <For each="tweet" of={this.state.tweets}>
37.           <li key={tweet._id}>{tweet._creator.name}: {tweet.message}</li>
38.         </For>
39.       </ul>
40.     )
```

```
41.   }
42. }
43.
44. render(<App/>, document.getElementById('root'));
```

No entraremos en los detalles del todo el componente, pues nos es el objetivo de esta sección, pero lo que solo nos centraremos en la función *componentWillMount* (línea 14). Esta función no tiene un nombre aleatorio, si no que corresponde a una de las funciones del ciclo de vida de los componentes de React, el cual se ejecuta automáticamente justo antes del que el componente sea montado, es decir, que sea visible en el navegador. Lo primero que hacemos será preparar los parámetros de la invocación (línea 15). Dentro de la variable definimos un objeto con las propiedades:

- **method:** representa el método HTTP que vamos a utilizar para la llamada al API, recordemos que los métodos utilizados son: *GET*, *POST*, *PUT*, *DELETE*, *PATCH*.
- **headers:** dentro de header podemos enviar cualquier propiedad de cabecera que necesitemos, pueden ser propios de HTTP o custom. En este caso, indicamos el header *Content-Type* para indicarle que esperamos un JSON como respuesta
- **body:** se utiliza para enviar información al servidor, como podría ser un JSON. En este caso no es necesario, pues el método GET no soporta un body.

El siguiente paso será invocar el método *invoke* de la clase *APIInvoker* que acabamos de crear, para esto, le estamos enviando la URL del servicio para consultar los Tweets del usuario test (*/tweets/test*), el segundo parámetros será la función que se ejecutará en caso de éxito, la cual solamente actualiza el estado del componente con la respuesta del servicio, el tercer parámetro es la función de error, la cual solo imprimirá en la consola los errores retornados.

Finalmente, los Tweets retornados son mostrados en el método *render* con ayuda de un *<For>* para iterar los resultados.

Mejorando la clase APIInvoker

Hasta este punto ya sabemos cómo invocar el API REST desde React, sin embargo, necesitamos mejorar aún más nuestra clase *APIInvoker* para reutilizarla en todo el proyecto.

Lo primero que aremos será quitar todas las secciones Hardcode, como lo son el host y el puerto y enviarlas a un archivo de configuración externo llamado *config.js*, el cual deberemos crear justo en la raíz del proyecto, es decir a la misma altura que los archivos *package.json* y *webpack.config.js*:

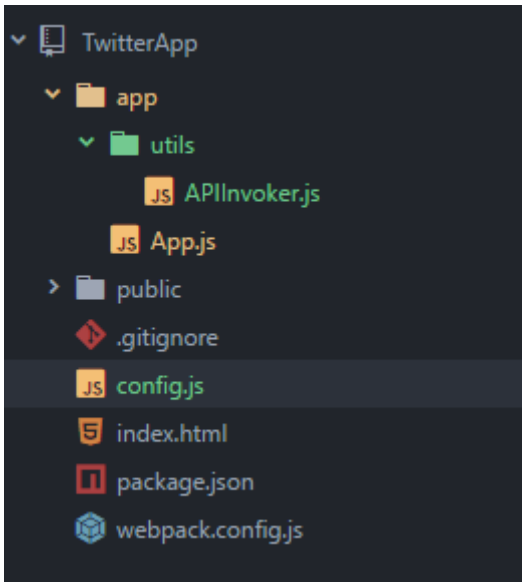


Fig. 72 - Archivo config.js

El archivo deberá quedar de la siguiente manera:

```
module.exports = {
  debugMode: true,
  server: {
    port: 3000,
    host: "http://api.localhost"
  }
}
```

Una vez que tenemos el archivo de configuración, deberemos de modificar el archivo *APIInvoker* para que utilice estas configuraciones:

```
var configuration = require('../../config')
const debug = configuration.debugMode

class APIInvoker {

  getAPIHeader(){
    return {
        'Content-Type': 'application/json',
        authorization: window.localStorage.getItem("token"),
    }
  }

  invokeGET(url, okCallback, failCallback){
    let params = {
      method: 'get',
      headers: this.getAPIHeader()
    }
    this.invoke(url, okCallback, failCallback,params);
  }

  invokePUT(url, body, okCallback, failCallback){
    let params = {
      method: 'put',
      headers: this.getAPIHeader(),
      body: JSON.stringify(body)
```

```
26.     };
27.
28.     this.invoke(url, okCallback, failCallback,params);
29.   }
30.
31.   invokePOST(url, body, okCallback, failCallback){
32.     let params = {
33.       method: 'post',
34.       headers: this.getAPIHeader(),
35.       body: JSON.stringify(body)
36.     };
37.
38.     this.invoke(url, okCallback, failCallback,params);
39.   }
40.
41.   invoke(url, okCallback, failCallback,params){
42.     if(debug){
43.       console.log("Invoke => " + params.method + ":" + url );
44.       console.log(params.body);
45.     }
46.
47.     fetch(`${configuration.server.host}:${configuration.server.port}${url}`,
48.       params)
49.     .then((response) => {
50.       if(debug){
51.         console.log("Invoke Response => " );
52.         console.log(response);
53.       }
54.       return response.json()
55.     })
56.     .then((responseData) => {
57.         if(responseData.ok){
58.           okCallback(responseData)
59.         }else{
60.           failCallback(responseData)
61.         }
62.
63.     })
64.   }
65. }
66. export default new APIInvoker();
```

Como vemos, la clase creció bastante a como la teníamos originalmente, lo que podría resultar intimidador, pero en realidad es más simple de lo que parece. Analicemos los cambios. Se han agregado una serie de métodos adicionales al método *invoke*, pero observemos que todos se llaman invoke + un método de HTTP, los cuales son:

- **invokeGET:** Permite invocar un servicio con el método GET
- **InvokePUT:** Permite invocar un servicio con el método PUT
- **InvokePOST:** Permite invocar un servicio con el método POST

Si nos vamos al detalle de cada uno de estos métodos, veremos que en realidad contienen lo mismo, ya que lo único que hacen es crear los parámetros HTTP, como son los headers y el body, para finalmente llamar al método *invoke* (sin postfijo). La única diferencia que tienen estas funciones es que el método *invokeGET* no requiere un body.

Otro punto interesante a notar es que cuando definimos los *header*, lo hacemos mediante la función *getAPIHeader*, la cual retorna el *Content-Type* y una propiedad llamada *authorization*. No entraremos en detalle acerca de esta, pues lo analizaremos más adelante cuando veamos la parte de seguridad.

Finalmente, hemos realizado algunos cambios en la función *invoke*, lo primero que podemos apreciar, son dos bloques de debug (líneas 42 y 50) las cuales nos permite imprimir en el log la petición y la respuesta de cada invocación. Lo segundo interesante es que en la línea 47 hemos remplazado la URL del API REST por los valores configurados en el archivo *config.js*.

En este punto tenemos lista la clase *APIInvoker* para ser utilizada a lo largo de toda la implementación del proyecto Mini Twitter, Y ya solo nos restaría ajustar la clase *App.js* para reflejar estos últimos cambios, por lo que vamos a modificar únicamente la función *componentWillMount* para que se vea de la siguiente manera:

```
1.  componentWillMount(){
2.    APIInvoker.invokeGET('/tweets/test', response => {
3.      this.setState({
4.        tweets: response.body
5.      })
6.    },error => {
7.      console.log("Error al cargar los Tweets", error);
8.    })
9.  }
```

Apreciemos que ya no es necesario definir los parámetros de HTTP, logrando que sea mucho más simple invocar un servicio.

El componente TweetsContainer

Finalmente ha llegado el momento de iniciar con la construcción del proyecto Mini Twitter, por lo que iniciaremos con la clase *TweetsContainer* la cual se encarga de cargar todos los Tweets. Veamos nuevamente la imagen de la estructura del proyecto para entender dónde va este componente y saber qué es lo que estamos programando.

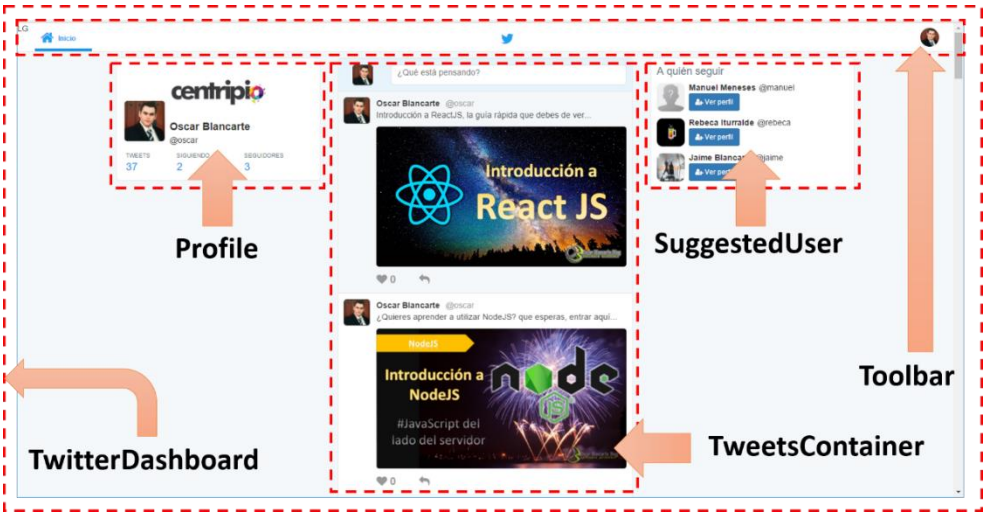


Fig. 73 - Ubicando el componente TweetsContainer

En la siguiente imagen podemos observar una fotografía más detallada del componente *`TweetsContainer`*, en el cual se puedan apreciar los componentes secundarios que conforman a este.

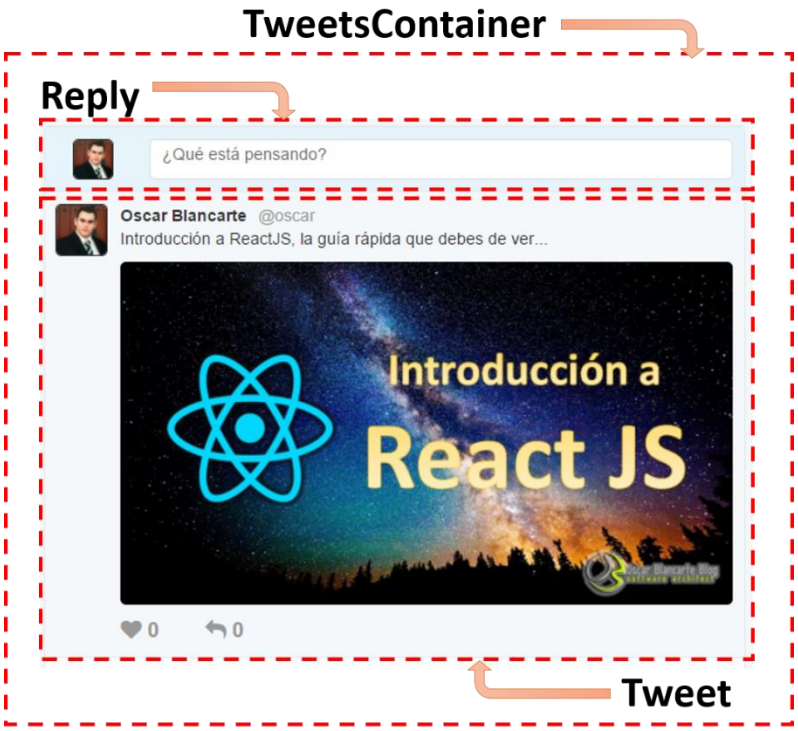


Fig. 741 - Repaso a la estructura del proyecto.

Este componente tiene dos responsabilidades, la primer, es cargar los Tweets desde el API y la segunda, es funcionar como un contenedor para ver todos los Tweets que retorne el API, así como al componente Reply, el cual analizaremos más adelante.

Lo primero que haremos será crear el archivo *TweetsContainer.js* en el path */app*, es decir, a la misma altura que el archivo *App.js* y lo dejaremos de la siguiente manera:

```
1. import React from 'react'
2. import APIInvoker from "./utils/APIInvoker"
3. import PropTypes from 'prop-types'
4.
5. class TweetsContainer extends React.Component{
6.   constructor(props){
7.     super(props)
8.     this.state = {
9.       tweets: []
10.    }
11.  }
12.
13.  componentWillMount(){
14.    let username = this.props.profile.userName
15.    let onlyUserTweet = this.props.onlyUserTweet
16.    this.loadTweets(username, onlyUserTweet)
17.  }
18.
19.  loadTweets(username, onlyUserTweet){
20.    let url = '/tweets' + (onlyUserTweet  ? "/" + username : "")
21.    APIInvoker.invokeGET(url , response => {
22.      this.setState({
23.        tweets: response.body
24.      })
25.    },error => {
26.      console.log("Error al cargar los Tweets", error);
27.    })
28.  }
29.
30.  render(){
31.
32.    return (
33.      <main className="twitter-panel">
34.        <If condition={this.state.tweets != null}>
35.          <For each="tweet" of={this.state.tweets}>
36.            <p key={tweet._id}>{tweet._creator.userName}
37.              : {tweet._id}-{tweet.message}</p>
38.          </For>
39.        </If>
40.      </main>
41.    )
42.  }
43. }
44.
45. TweetsContainer.propTypes = {
46.   onlyUserTweet: PropTypes.bool,
47.   profile: PropTypes.object
48. }
49.
50. TweetsContainer.defaultProps = {
51.   onlyUserTweet: false,
52.   profile: {
53.     userName: ""
54.   }
55. }
56.
57. export default TweetsContainer;
```

Lo primero interesante a resaltar es la función *componentWillMount*, la cual recupera dos propiedades, *username* y *onlyUserTweet*, las cuales necesitará pasar a la función *loadTweets* para cargar los Tweet iniciales. Cabe mencionar que este componente puede mostrar Tweet de dos formas, es decir, puede mostrar los Tweet de todos los usuarios de forma cronológica o solo los Tweet del usuario autenticado, y ese es el motivo por el cual son necesarios estas dos propiedades. Si la propiedad *onlyUserTweet* es true, le indicamos al componente que muestre solo los Tweet del usuario autenticado, de lo contrario, mostrara los Tweet de todos los usuarios. Más adelante veremos la importancia de hacerlo así para reutilizar el componente.

Antes de ser montado el componente, el método *componentWillMount* es ejecutado, lo que dispara la carga de Tweets mediante el método *loadTweets*, que a su vez, utilizará la clase APIInvoker para cargar los Tweets desde el API REST, pero antes, la URL deberá ser generada (línea 20), Si requerimos los Tweets de todos los usuarios, entonces invocamos el API con la URL *"/tweets",* pero si requerimos los Tweets de un usuario en específico, entonces invocamos la URL *"/tweets/{username}"*. Prestemos atención en la parte *{username}*, pues esta parte de la URL en realidad es un parámetro, y podría recibir el nombre de usuario de cualquier usuario registrado en la aplicación.

Usuarios registrados
De momento solo tendremos el usuario **test**, por lo que de momento solo podremos consultar los Tweets de este usuario. Más adelante implementaremos el registro de usuarios para poder hacer pruebas con más usuarios.

Documentación del servicio de Tweets
Los servicios de consulta de Tweets se utilizan para recuperar todos los Tweets de forma cronológica (/tweets) y todos los de un determinado usuario (/tweets/:username).

En las líneas 35 a 38 podemos ver como se Iteran todos los Tweets consultados para mostrar el nombre de usuario del Tweet, el ID y el texto del Tweet.

Al final del archivo también podemos apreciar que hemos definidos los *PropTypes* y los *DefaultProps*, correspondientes a las propiedades *onlyUserTweet* y *profile*. El primero ya lo hemos mencionado, pero el segundo corresponde al usuario autenticado en la aplicación. De momento no nos preocupemos por esta propiedad, más adelante regresaremos a analizarla.

Una vez terminado de editar el archivo *TweetsContainer*, regresaremos al componente App.js y actualizaremos la función *render* para que se vea de la siguiente manera:

```
1.  render(){
2.    return (
3.      <TweetsContainer />
4.    )
5.  }
```

También será necesario agregar el *import* del componente *TweetsContainer al inicio de la clase*.

```
1.  import TweetsContainer from './TweetsContainer'
```

Ya con estos últimos pasos actualizamos el navegador y veremos los tweets

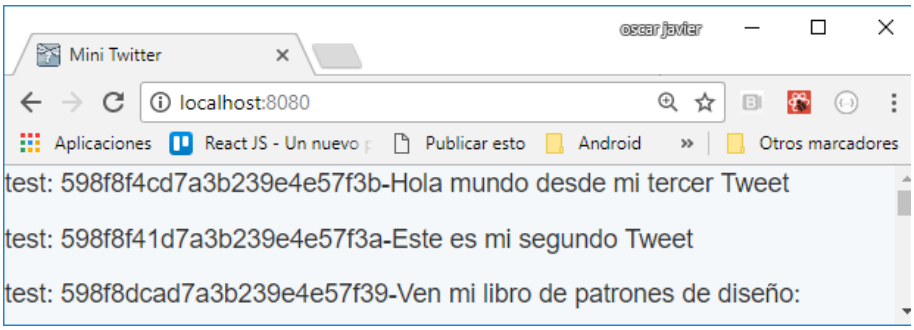


Fig. 75 - Resultado del componente TweetsContainer.

El componente Tweet

Hasta este momento solo representamos algunos campos del Tweet para poder comprobar que el componente *TweetsContainer* está consultando realmente los datos desde el API REST, por lo que ahora nos concentraremos en el componente *Tweet*, el cual utilizaremos pare representar los Tweets en pantalla.

Lo primero que haremos será crear un nuevo archivo llamado *Tweet.js* en el path */app* y lo dejaremos de la siguiente manera:

```
1.  import React from 'react'
2.  import APIInvoker from './utils/APIInvoker'
3.  import { render } from 'react-dom';
4.  import PropTypes from 'prop-types'
5.
6.  class Tweet extends React.Component{
7.
8.    constructor(props){
9.      super(props)
10.     this.state = props.tweet
11.   }
12.
```

```
13.   render(){
14.     let tweetClass = null
15.     if(this.props.detail){
16.       tweetClass = 'tweet detail'
17.     }else{
18.       tweetClass = this.state.isNew ? 'tweet fadeIn animated' : 'tweet'
19.     }
20.
21.     return (
22.         <article  className={tweetClass} id={"tweet-" + this.state._id}>
23.           <img src={this.state._creator.avatar} className="tweet-avatar" />
24.           <div className="tweet-body">
25.             <div className="tweet-user">
26.               <a href="#">
27.                 <span  className="tweet-name" data-ignore-onclick>
28.                   {this.state._creator.name}</span>
29.               </a>
30.               <span className="tweet-username">
31.                 @{this.state._creator.userName}</span>
32.             </div>
33.             <p className="tweet-message">{this.state.message}</p>
34.             <If condition={this.state.image != null}>
35.               <img className="tweet-img" src={this.state.image}/>
36.             </If>
37.             <div className="tweet-footer">
38.               <a className={this.state.liked ? 'like-icon liked' :
39.                   'like-icon'} data-ignore-onclick>
40.                 <i className="fa fa-heart " aria-hidden="true"
41.                   data-ignore-onclick></i> {this.state.likeCounter}
42.               </a>
43.               <If condition={!this.props.detail} >
44.                 <a className="reply-icon" data-ignore-onclick>
45.                   <i className="fa fa-reply " aria-hidden="true"
46.                     data-ignore-onclick></i> {this.state.replys}
47.                 </a>
48.               </If>
49.             </div>
50.           </div>
51.           <div id={"tweet-detail-" + this.state._id}/>
52.         </article>
53.     )
54.   }
55. }
56.
57. Tweet.propTypes = {
58.   tweet: PropTypes.object.isRequired,
59.   detail: PropTypes.bool
60. }
61.
62. Tweet.defaultProps = {
63.   detail: false
64. }
65.
66. export default Tweet;
```

Este no es un libro HTML ni de CSS, por lo que no nos detendremos en explicar para que es cada clase de estilo utilizada ni la estructura del HTML generado, salvo en ocasiones donde tiene una importancia relacionada con el tema en cuestión.

Lo primero que debemos de resaltar es que este componente recibe dos propiedades, la primera representa el objeto *Tweet* como tal y un *boolean*, que indica si el Tweet debe mostrar el detalle, es decir, los comentarios relacionados al Tweet.

En la línea 10 podemos ver como la propiedad *this.tweet* es establecida como el Estado del componente (más adelante revisaremos los estados).

En la función *render* podremos ver que este ya es un componente más complejo, pues retorna un HTML con varios elementos. Por lo pronto te pido que ignores las líneas 14 a 19, pues las explicaremos más adelante y nos centremos en la parte del *return*. Quiero que veas que cada Tweet es englobado como un *<article>* para darle más semántica al HTML generado. Cada *article* tendrá un ID generado a partir del ID del Tweet (línea 22), de esta forma podremos identificar el Tweet más adelante.

En la línea 23 definimos una imagen, la cual corresponde al Avatar del usuario que publico el Tweet. Fijémonos como tomamos la imagen del estado, mediante *this.state._creator.avatar*. De esta misma forma definimos nombre (línea 28), nombre de usuario (línea 31), el mensaje de tweet (línea 33) y la imagen asociada al Tweet, siempre y cuando tenga imagen (línea 35).

Las líneas 38 a 48 son las que muestran los iconos de like y compartir, por el momento no tendrá funcionalidad, pero más adelante regresaremos para implementarla.

En la línea 51 tenemos un div con solo un ID, este lo utilizaremos más adelante para mostrar el detalle del Tweet, por lo pronto no lo prestemos atención.

En este punto el componente Tweet ya debería estar funcionando correctamente, sin embargo, hace falta mandarlo llamar desde el componente *TweetsContainer*, para esto regresaremos al archivo *TweetsContainer.js* y editaremos solamente el método *render* para dejarlo de la siguiente manera:

```
1.  render(){
2.    return (
3.      <main className="twitter-panel">
4.        <If condition={this.state.tweets != null}>
5.          <For each="tweet" of={this.state.tweets}>
6.            <Tweet key={tweet._id} tweet={tweet}/>
7.          </For>
8.        </If>
9.      </main>
10.   )
11. }
```

Adicional tendremos que agregar el *import* al componente al inicio del archivo:

```
1.  import Tweet from './Tweet'
```

Como último paso tendremos que agregar las clases de estilo CSS al archivo *styles.css* que se encuentra en el path */public/resources/css/styles.css*. Solo agreguemos lo siguiente al final del archivo:

```
/** TWEET COMPONENT **/

.tweet{
  padding: 10px;
  border-top: 1px solid #e6ecf0;
}


.tweet .tweet-link{
  position: absolute;
  display: block;
  left: 0px;
  right: 0px;
  top: 0px;
  bottom: 0px;
}

.tweet:hover{
  background-color: #F5F8FA;
  cursor: pointer;
}

.tweet.detail{
  border-top: none;
}

.tweet.detail:hover{
  background-color: #FFF;
  cursor: default;
}


.tweet .tweet-img{
  max-width: 100%;
  border-radius: 5px;
}

.tweet .tweet-avatar{
  border: 1px solid #333;
  display: inline-block;
  width: 45px;
  height: 45px;
  position: absolute;
  border-radius: 5px;
  text-align: center;
}

.tweet .tweet-body{
  margin-left: 55px;
}

.tweet .tweet-body .tweet-name{
  color: #333;
}

.tweet .tweet-body .tweet-name{
  font-weight: bold;
  text-transform: capitalize;
  margin-right: 10px;
  z-index: 10000;
```

```
61. }
62.
63. .tweet .tweet-body .tweet-name:hover{
64.   text-decoration: underline;
65. }
66.
67. .tweet .tweet-body .tweet-username{
68.   text-transform: lowercase;
69.   color: #999;
70. }
71.
72. .tweet.detail .tweet-body .tweet-user{
73.   margin-left: 70px;
74. }
75.
76. .tweet.detail .tweet-body{
77.   margin-left: 0px;
78. }
79.
80. .tweet.detail .tweet-body .tweet-name{
81.   font-size: 18px;
82. }
83.
84. .tweet-detail-responses .tweet.detail .tweet-body .tweet-message{
85.   font-size: 16px;
86. }
87.
88. .tweet.detail .tweet-body .tweet-username{
89.   display: block;
90.   font-size: 16px;
91. }
92.
93. .tweet.detail .tweet-message{
94.   position: relative;
95.   display: block;
96.   margin-top: 25px;
97.   font-size: 26px;
98.   left: 0px;
99. }
100.
101. .reply-icon,
102. .like-icon{
103.   color: #999;
104.   transition: 0.5s;
105.   padding-right: 40px;
106.   font-weight: bold;
107.   font-size: 18px;
108.   z-index: 99999;
109. }
110.
111. .like-icon:hover{
112.   color: #E2264D;
113. }
114.
115. .like-icon.liked{
116.   color: #E2264D;
117. }
118.
119. .reply-icon:hover{
120.   color: #1DA1F2;
121. }
122.
123. .like-icon i{
124.   color: inherit;
125. }
126.
```

```
127. .reply-icon i{
128.   color: inherit;
129. }
```

Guardamos todos los cambios y refrescar el navegador para apreciar cómo va tomando forma el proyecto.

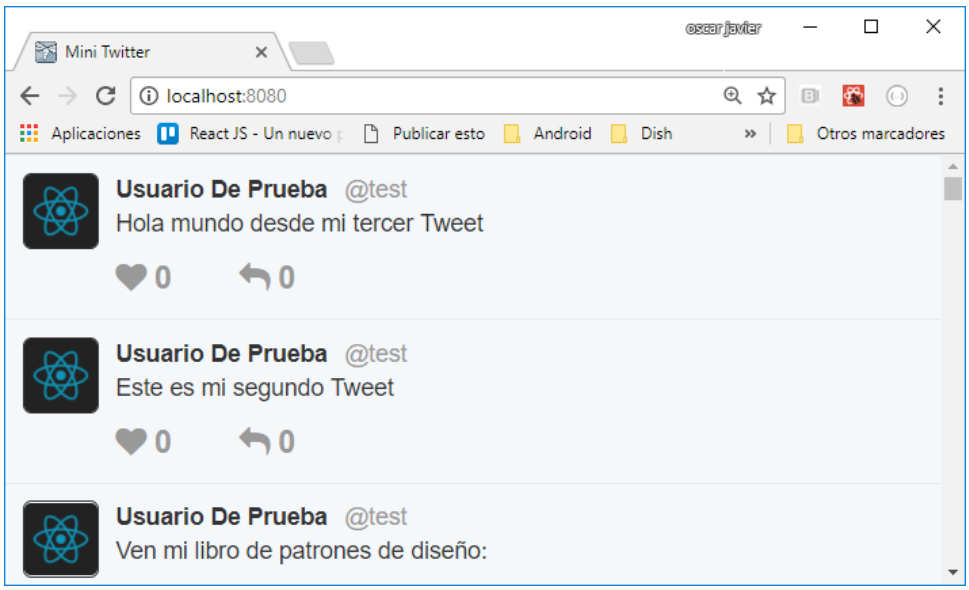


Fig. 76 - Componente Tweet

Finalmente, nuestro proyecto se debe ver de la siguiente manera:

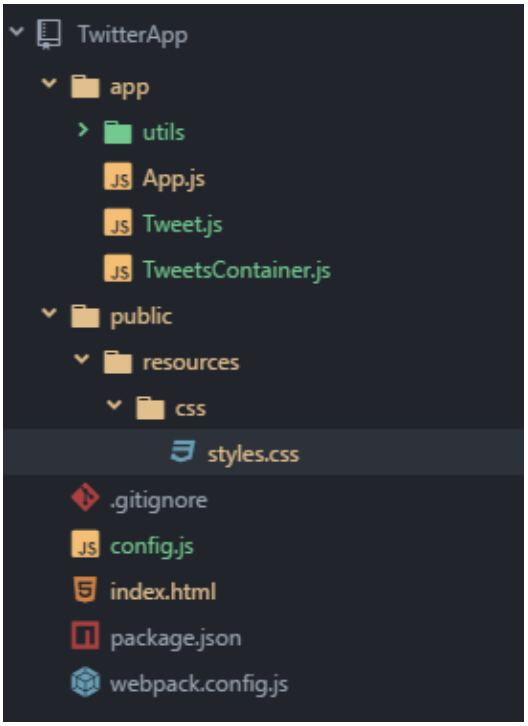


Fig. 77 - Estructura actual del proyecto

Una cosa más antes de concluir este capítulo. Hasta el momento hemos trabajado con la estructura de los Tweets retornados por el API REST, pero no los hemos analizado, es por ello que dejo a continuación un ejemplo del JSON.

```
{
    "ok":true,
    "body":[
       {
          "_id":"598f8f4cd7a3b239e4e57f3b",
          "_creator":{
             "_id":"598f8c4ad7a3b239e4e57f38",
             "name":"Usuario de prueba",
             "userName":"test",
             "avatar":""
          },
          "date":"2017-08-12T23:29:16.078Z",
          "message":"Hola mundo desde mi tercer Tweet",
          "liked":false,
          "likeCounter":0,
          "replys":0,
          "image":null
       }
    ]
}
```

Lo primero que vamos a observar de aquí en adelante es que todos los servicios retornados por el API tienen la misma estructura base, es decir, regresan un campo llamado ok y un *body*, el *ok* nos indica de forma booleana si el resultado es correcto y el *body* encapsula todo el mensaje que nos retorna el API. Si regresamos al componente *`TweetContainer`* en la línea 22 veremos lo siguiente:

```
    this.setState({
       tweets: response.body
    })
```

Observemos que el estado lo crea a partir del *body* y no de todo el retorno del API.

Ya que conocemos la estructura general del mensaje, podemos analizar los campos del Tweet, los cuales son:

- **Id:** Identificador único en la base de datos.
- **_creator:** Objeto que representa el usuario que creo el Tweet
  - **_id:** Identificador único del usuario en la base de datos.
  - **Name:** Nombre del usuario
  - **userName:** Nombre de usuario
  - **avatar:** Representación en Base 64 de la imagen del Avatar.
- **Date:** fecha de creación del Tweet.
- **Message:** El mensaje que escribió el usuario en el Tweet.
- **Liked:** Indica si el usuario autenticado le dio like al Tweet.
- **likeCounter:** número de likes totales recibidos en este Tweet.
- **Reply:** número de respuestas o comentarios recibidos en el Tweet.
- **Image:** Imagen asociada al Tweet (opcional)

Resumen

Este capítulo ha sido bastante emocionante pues hemos iniciado con el proyecto Mini Twitter y hemos aplicados varios de los conceptos que hemos venido aprendiendo a lo largo del libro. Si bien, solo hemos empezado, ya pudimos apreciar un pequeño avance en el proyecto.

Por otra parte, hemos visto como instalar el API REST y hemos aprendido como consumirlo desde React, también hemos analizado la estructura de un Tweet retornado por el API.