SURVIVEJS

React

DE APRENDIZ A MAESTRO

Juho Vepsäläinen

Traducido por Raúl Expósito

SurviveJS - React

De aprendiz a maestro

Juho Vepsäläinen y Raúl Expósito

Este libro está a la venta en http://leanpub.com/survivejs-react-es

Esta versión se publicó en 2017-06-06

Índice general

Introducción

El desarrollo del frontend se mueve muy deprisa. Una buena señal de ello es el ritmo en el que están surgiendo nuevas tecnologías. React[1] es uno de los recién llegados. Incluso cuando la tecnología en sí es sencilla, hay mucho movimiento en torno a ella.

El objetivo de este libro es ayudarte a comenzar con React y darte una perspectiva del ecosistema que hay en torno a él para que sepas por dónde mirar.

Nuestro desarrollo va a utilizar Webpack. Hay un libro aparte[2] que bucea en él, pero no espero que conozcas Webpack para poder utiliza este libro.

¿Qué es React?

React es una librería JavaScript, creada por Facebook, que basa su funcionamiento en la abstración de las vistas mediante el uso de componentes. Un componente puede ser un formulario de entrada, un botón, o cualquier otra cosa del interfaz de usuario. Esto nos proporciona un interesante contraste con respecto a enfoques anteriores ya que, por diseño, React no está vinculado al árbol DOM. Puedes utilizarlo por ejemplo para desarrollar aplicaciones móviles.

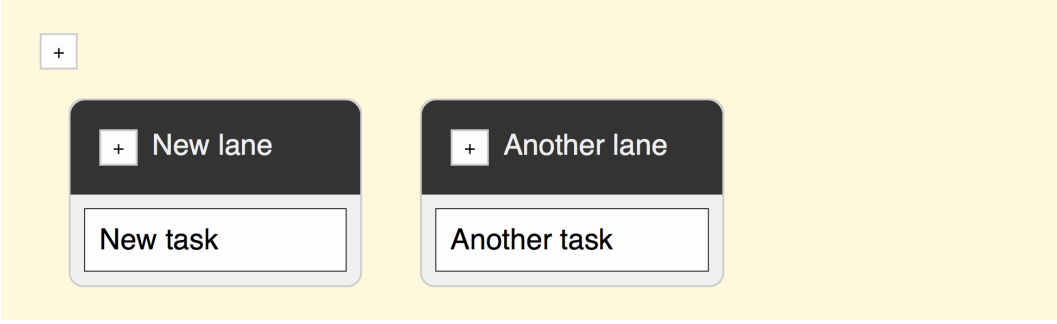
React es sólo una Parte del Todo

Para usarlo tendrás que complementarlo con otras librerías que te den aquello que te falte, ya que React se centra únicamente en la vista. Esto brinda un contraste con respecto a utilizar frameworks que traen mucho más de serie. Ambos enfoques tienen sus méritos. En este libro nos centraremos en el uso de librerías.

Las ideas presentadas por React han tenido su influencia el desarrollo de frameworks, aunque más importante es que nos ha ayudado a entender cómo de bien encaja el pensar en componentes en el desarrollo de aplicaciones web.

[1] https://facebook.github.io/react/
[2] http://survivejs.com/webpack/introduction/

¿Qué Vas a Aprender?



Aplicación Kanban

Este libro te enseña a crear una aplicación de tipo Kanban[3]. Más allá de esto, debatiremos acerca de aspectos de desarrollo web más teóricos. Completar el proyecto te dará una buena idea de cómo implementar algo por tí mismo. Durante el proceso aprenderás por qué ciertas librerías son útiles y serás capaz de justificar mejor la elección de tus tecnologías.

¿Cómo está Organizado este Libro?

Para comenzar, desarrollaremos un pequeño clon de una famosa aplicación de TODO[4]. Esto nos llevará a tener problemas de escalado. A menudo es necesario hacer cosas de la forma sencilla para entender por qué al final es necesario usar mejores soluciones.

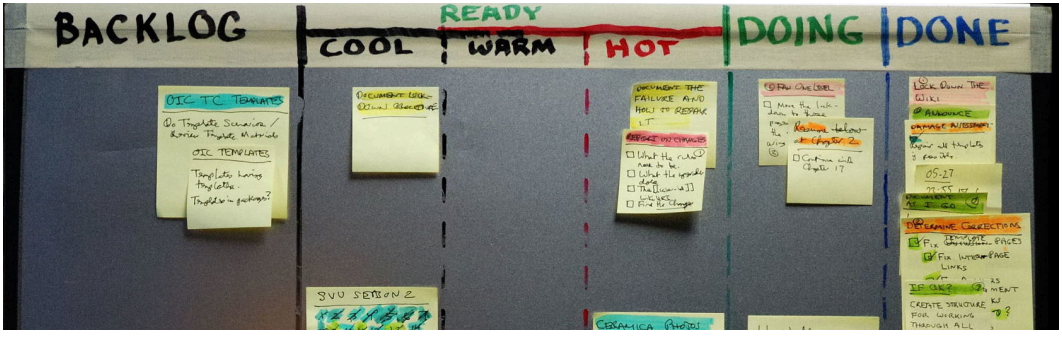
Empezaremos a generalizar en este punto y comenzaremos a utilizar la arquitectura Flux[5]. Usaremos la magia del Drag and Drop (DnD)[6] (arrastrar y soltar) para comenzar a arrastrar cosas por ahí. Al terminar tendremos hecho algo que podremos poner en producción.

[3] https://en.wikipedia.org/wiki/Kanban
[4] http://todomvc.com/
[5] https://facebook.github.io/flux/docs/overview.html
[6] https://gaearon.github.io/react-dnd/

La parte final y teórica del libro cubre aspectos más avanzados. Si estás leyendo la versión comercial del libro encontrarás algo extra para tí. Te mostraré cómo mejorar al programar con React para que generes código de mayor calidad. También aprenderás a probar tus componentes y tu lógica mediante tests. Aprenderás a dar estilo a tu aplicación hecha con React de varias formas y tendrás una mejor idea de cómo estructurar tu proyecto.

Los apéndices del final sirven para darte cosas en las que pensar y explicar conceptos, tales como características del lenguaje, en un mayor detalle. Si hay algo de sintaxis en el libro que te resulte extraño seguramente encuentres más información allí.

¿Qué es Kanban?



Kanban por Dennis Hamilton (CC BY)

Kanban, desarrollado originalmente por Toyota, te permite seguir el estado de las tareas. Puede ser modelado en conceptos como `Carriles` y `Notas`. Las `Notas` se mueven entre `Carriles` que representan etapas que van de izquierda a derecha hasta que se completan. Las `Notas` pueden contener información sobre ellas mismas, su prioridad y toda aquello que sea necesario.

Este sistema puede ser extendido de varias formas. Una manera sencilla consiste en aplicar un límite de Trabajo En Proceso (WIP) por carril. El objetivo es obligarte a centrarte en tener tareas terminadas, lo cual es una de las consecuencias positivas de utilizar Kanban. Mover estas notas entre carriles es satisfactorio, ya que puedes ver cómo van las tareas y qué tareas hay que hacer todavía.

¿Dónde Podemos Usar Kanban?

Este sistema puede utilizarse en varios escenarios, incluyendo el desarrollo del software y la gestión eficaz del tiempo. Puedes utilizarlo para hacer un seguimiento sobre tus proyectos personales o tus metas en la vida, por ejemplo. Aunque sea una herramienta muy sencilla es muy poderosa y puedes encontrarle un uso práctico en muchos escenarios.

¿Cómo Construir un Kanban?

La forma más sencilla de construir un Kanban es conseguir un paquete de Post-its y encontrar una pared. Tras esto, la divides en columnas. Estos `Carriles` pueden ser las siguientes etapas: Por Hacer, Haciendo, Hecho. Todas las `Notas`, inicialmente, irán en Por Hacer. A medida que trabajes en ellas las irás moviendo a Haciendo y, finalmente, a Hecho una vez se hayan terminado. Esta es la forma más sencilla de comenzar.

Pero este es símplemente un ejemplo de cómo configurar los carriles, ya que éstos pueden definirse para encajar con el proceso que quieras seguir. Por ejemplo, puedes incluir pasos que requieran aprobación. Si estás modelando un proceso de desarrollo de software puedes tener, por ejemplo, carriles separados para la realizacion de pruebas y el despliegue.

Implementaciones de Kanban Disponibles

Quizá Trello[7] sea la implementación online más conocida de Kanban. Sprintly ha liberado su implementación de React de Kanban[8]. Otro buen ejemplo es wekan[9], basado en Meteor. El nuestro no será tan sotisficado como aquellos, pero será lo suficientemente bueno para comenzar.

[7] https://trello.com/
[8] https://github.com/sprintly/sprintly-kanban
[9] https://github.com/wekan/wekan

¿Para Quién es Este Libro?

Espero que tengas ciertos conocimientos básicos sobre JavaScript y Node.js. Deberías ser capaz de utilizar npm en un nivel básico. Sería genial que supieras algo sobre React o ES6, leyendo este libro obtendrás un mayor entendimiento de estas tecnologías.

Una de las cosas más difíciles a la hora de escribir un libro es hacerlo al nivel adecuado. Dado que el libro en sí cubre mucho terreno hay apéndices que cubren aspectos básicos, tales como detalles del lenguaje, con un mayor detalle que el incluido en el contenido principal, así que si te sientes inseguro échale un vistazo.

Además, hay un chat para la comunidad[10] disponible. Estamos allí para ayudar si quieres preguntar algo directamente. Los comentarios que puedas tener ayudarán a mejorar el contenido del libro. Lo último que quiero tener es a alguien que se encuentra en apuros mientras está siguiendo el libro.

¿Cómo Abordar el Libro?

Aunque la manera natural de leer un libro es comenzar por el primer capítulo y leer el resto de capítulos de forma secuencial, esta no es la única manera de abordar este libro. El orden de los capítulos es únicamente una sugerencia sobre cómo leerlos. Dependiendo de tus conocimientos previos, puede que ojees la primera parte y profundices en los conceptos avanzados.

Este libro no cubre todo lo que necesitas saber para desarrollar aplicaciones frontend, ya que es demasiado para un único libro. Creo, sin embargo, que puede ser capaz de empujarte en la dirección correcta. El ecosistema en torno a React es enorme y lo he hecho lo mejor que he podido para cubrir una parte de él.

Dado que el libro se basa en varias características nuevas del lenguaje, he reunido las más importantes en un apéndice aparte llamado *Características del Lenguaje* que te ofrece un rápido resumen a todas ellas. Es un buen lugar que revisar si quieres entender algunas de estas características por separado o no estás seguro de algo.

[10] https://gitter.im/survivejs/react

Versionado del Libro

Se ha definido un esquema de versionado severo dado que este libro recibe mucho mantenimiento y mejoras debido al ritmo de innovación. Mantengo las notas de cada liberación en el blog del libro[11] para indicar qué ha cambiado entre cada versión. Otra cosa beneficiosa es examinar el repositorio de GitHub, recomiendo utilizar la comparación de GitHub con este fin, por ejemplo:

```
https://github.com/survivejs-translations/react-book-es/compare/v2.1\
.0...v2.5.7
```

Esta página te mostrará los commits individuales que hay en el rango de versiones indicado. También puedes ver qué lineas han cambiado en el libro. Esto excluye los capítulos privados, pero será suficiente para que te hagas una idea de qué cambios se han hecho en el libro.

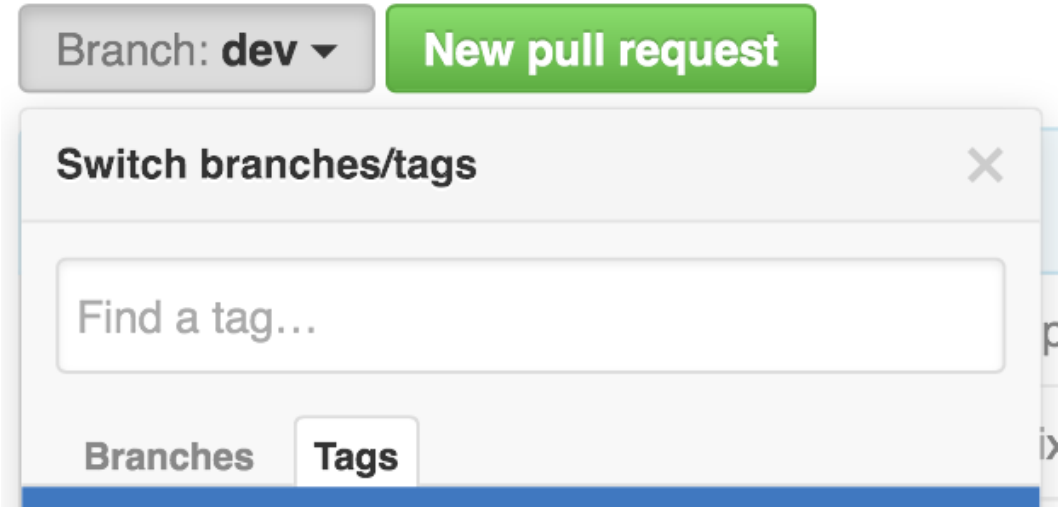
La versión actual del libro es la **2.5.7**.

Material Extra

El contenido del libro y el código fuente están disponibles en el repositorio del libro de GitHub[12]. Por favor, sé consciente de que la rama por defecto es `dev` y que esto facilita que sea posible contribuir. Puedes utilizar el selector de etiquetas de GitHub para encontrar el fuente que encaje con la versión del libro que estás leyendo, tal y como se muestra a continuación:

[11] http://survivejs.com/blog/

[12] https://github.com/survivejs/react



selector de etiquetas de GitHub

El repositorio del libro contiene el código de cada capítulo. Esto significa que puedes comenzar desde cualquier punto sin tener que escribirlo todo por tí mismo. Si no estás seguro de algo, siempre puedes usarlo como referencia.

Puedes encontrar mucho más material complementario en el repositorio de survivejs[13]. Hay implementaciones alternativas disponibles de la aplicación escritas en MobX[14], Redux[15], y Cerebral/Baobab[16]. Estudiarlas puede darte una buena idea sobre cómo funcionan las distintas arquitecturas utilizando el mismo ejemplo.

Cómo Obtener Soporte

Dado de ningún libro es perfecto, puede que quieras presentarme cuestiones y problemas que te hayan surgido relacionadas con el contenido. Hay un par de formas de conseguirlo:

- Contacta conmigo a través de el sistema de incidencias de GitHub[17]

[13] https://github.com/survivejs/
[14] https://github.com/survivejs/mobx-demo
[15] https://github.com/survivejs/redux-demo
[16] https://github.com/survivejs/cerebral-demo
[17] https://github.com/survivejs/react/issues

- Súmate al chat de Gitter[18]
- Sigue @survivejs[19] en Twitter para recibir novedades oficiales o dame un codazo directamente a través de @bebraw[20]
- Mándame un correo a info@survivejs.com[21]
- Pregúntame cualquier cosa sobre Webpack o React en SurviveJS AmA[22]

Si haces preguntas en Stack Overflow[23], etiquétalas utilizando **survivejs**[24] para que pueda recibir notificaciones. También puedes utiliza el hashtag **#survivejs** en Twitter.

He tratado de cubrir algunos problemas comunes en el apéndice *Solución de Problemas*, el cual crecerá a medida que se vayan encontrando más problemas comunes.

Notificaciones

Notifico noticias relacionadas con SurviveJS en un par de canales:

- Lista de Correo[25]
- Twitter[26]
- RSS del Blog[27]

Siéntete libre de subscribirte.

[18] https://gitter.im/survivejs/react
[19] https://twitter.com/survivejs
[20] https://twitter.com/bebraw
[21] mailto:info@survivejs.com
[22] https://github.com/survivejs/ama/issues
[23] http://stackoverflow.com/search?q=survivejs
[24] https://stackoverflow.com/questions/tagged/survivejs
[25] http://eepurl.com/bth1v5
[26] https://twitter.com/survivejs
[27] http://survivejs.com/atom.xml

Agradecimientos

No habría sido posible realizar un esfuerzo como este sin el apoyo de la comunidad. Como resultado ¡hay mucha gente a la que dar las gracias!

Un enorme agradecimiento a Christian Alfoni[28] por comenzar el libro react-webpack-cookbook[29] conmigo. Aquel trabajo finalmente lideró este libro y finalmente se convirtió en un libro por sí mismo[30].

Este libro no podría ser la mitad de bueno sin la paciencia y el feedback de mi editor Jesús Rodríguez Rodríguez[31]. Gracias.

Un agradecimiento especial para Steve Piercy por sus numerosas contribuciones. Gracias a Prospect One[32] y Dixon & Moe[33] por ayudarme con el logo y el aspecto gráfico. Gracias por las revisiones a Ava Mallory y EditorNancy de fiverr.com.

Muchas personas a título individual han dado soporte y feedback a lo largo del camino. Gracias en un order sin nada en particular a Vitaliy Kotov, @af7, Dan Abramov, @dnmd, James Cavanaugh, Josh Perez, Nicholas C. Zakas, Ilya Volodin, Jan Nicklas, Daniel de la Cruz, Robert Smith, Andreas Eldh, Brandon Tilley, Braden Evans, Daniele Zannotti, Partick Forringer, Rafael Xavier de Souza, Dennis Bunskoek, Ross Mackay, Jimmy Jia, Michael Bodnarchuk, Ronald Borman, Guy Ellis, Mark Penner, Cory House, Sander Wapstra, Nick Ostrovsky, Oleg Chiruhin, Matt Brookes, Devin Pastoor, Yoni Weisbrod, Guyon Moree, Wilson Mock, Herryanto Siatono, Héctor Cascos, Erick Bazán, Fabio Bedini, Gunnari Auvinen, Aaron McLeod, John Nguyen, Hasitha Liyanage, Mark Holmes, Brandon Dail, Ahmed Kamal, Jordan Harband, Michel Weststrate, Ives van Hoorne, Luca DeCaprio, @dev4Fun, Fernando Montoya, Hu Ming, @mpr0xy, David "@davegomez" Gómez, Aleksey Guryanov, Elio D'antoni, Yosi Taguri, Ed McPadden, Wayne Maurer, Adam Beck, Omid Hezaveh, Connor Lay, Nathan Grey, Avishay Orpaz, Jax Cavalera, Juan Diego Hernández, Peter Poulsen, Harro van der Klauw, Tyler Anton, Michael Kelley, @xuyuanme, @RogerSep, Jonathan Davis, @snowyplover, Tobias Koppers, Diego Toro, George

[28] http://www.christianalfoni.com/
[29] https://github.com/christianalfoni/react-webpack-cookbook
[30] http://survivejs.com/webpack/introduction
[31] https://github.com/Foxandxss
[32] http://prospectone.pl/
[33] http://dixonandmoe.com/

Hilios, Jim Alateras, @atleb, Andy Klimczak, James Anaipakos, Christian Hettlage, Sergey Lukin, Matthew Toledo, Talha Mansoor, Pawel Chojnacki, @eMerzh, Gary Robinson, Omar van Galen, Jan Van Bruggen, Savio van Hoi, Alex Shepard, Derek Smith, Tetsushi Omi, Maria Fisher, Rory Hunter, Dario Carella, Toni Laukka, Blake Dietz, Felipe Almeida, Greg Kedge, Deepak Kannan, Jake Peyser, Alfred Lau, Tom Byrer, Stefanos Grammenos, Lionel Ringenbach, Hamilton Greene, Daniel Robinson, @karloxyz, Nicolò Ribaudo, Andrew Wooldridge, Francois Constant, Wes Price, Dawid Karabin, @alavkx, Aitor Gómez-Goiri, P.E. Butler III, @TomV, John Korzhuk, @markfox1, Jaime Liz, Richard C. Davis, y muchos otros. Si no he puesto tu nombre puede que haya olvidado incluirlo.

I Comenzando

React ha tenido un impacto significativo en la comunidad de desarrolladores de frontend a pesar de ser una librería reciente. Introduce conceptos, como el DOM virtual, y ha hecho que la comunidad entienda el poder de los componentes. Su dieño orientado a componentes funciona bien con la web, aunque React no está limitado a la web, puedes utilizarlo para desarrollar aplicaciones móviles e incluso interfaces de usuario que funcionen por una terminal.

En este apartado comenzaremos a bucear dentro de React e implementaremos una pequeña aplicación de notas. Esto es algo que en algún momento se convertirá en un Kanban. Aprenderás lo más basico de React y te acostumbrarás a trabajar con él.

1. Introducción a React

La librería React[1] de Facebook ha cambiado nuestra manera de pensar en cuanto al desarrollo de interfaces de usuario en aplicaciones web se refiere. Gracias a su diseño puede ser utilizado más allá de la web. Una característica llamada **DOM Virtual** es lo que lo permite.

En este capítulo veremos algunas de las ideas principales de la librería para que podamos comprender React un poco mejor antes de ponernos con él.

1.1 ¿Qué es React?

React

React es una librería JavaScript que nos obliga a pensar en componentes. Esta forma de pensar encaja bien con el desarrollo de interfaces de usuario. Dependiendo de tus conocimientos previos puede que te parezca un poco extraño al principio. Tendrás que pensar con detenimiento en el concepto de `estado` y dónde ubicarlo.

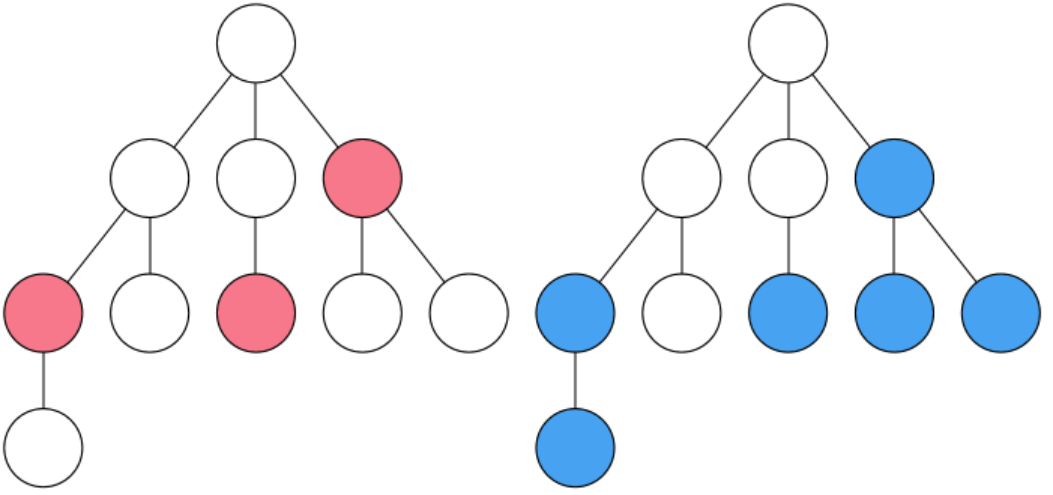
Han aparecido muchas soluciones ya que la mera **gestión del estado**, en sí, supone un problema complicado. En este libro comenzaremos gestionando el estado nosotros mismos y después nos moveremos a una implementación de Flux conocida como Alt.

[1] https://facebook.github.io/react/

Hay más implementaciones disponibles para otras alternativas, como Redux, MobX y Cerebral.

React es pragmático en el sentido de que tiene varias salidas de emergencia. Siempre es posible hacer algo a bajo nivel si el modelo de React no encaja contigo. Por ejemplo, hay puntos de enganche que pueden ser utilizados para envolver lógica antigua que dependa del DOM. Esto rompe con la abstracción y ata tu código a un entorno específico, pero a veces es la única cosa que puedes hacer.

1.2 DOM Virtual



DOM Virtual

Uno de los problemas principales de la programación es cómo lidiar con el estado. Supón que estás desarrollando una interfaz de usuario y quieres mostrar los mismos datos en varios sitios. ¿Cómo puedes estar seguro de que los datos que estás mostrando son consistentes?

Tradicionalmente se han mezclado tanto la gestión del DOM como la gestión del estado. React soluciona este problema de una manera diferente, ya que introduce el concepto de **DOM Virtual** para el público en general.

El DOM Virtual se encuentra por encima del DOM del navegador, o de cualquier

otro elemento que deba ser renderizado. Resuelve el problema de cómo manipular el estado a su propia manera. Sea cual sea el cambio que se ha hecho sobre él, se las apaña para encontrar y aplicar los cambios sobre el DOM subyacente. Es capaz de propagar los cambios sobre el árbol virtual tal y como se muestra en la imagen anterior.

El Rendimiento del DOM Virtual

Manipular el DOM de esta manera puede llevar a mejoras en el rendimiento. En cambio, manipular el DOM a mano suele ser ineficiente y difícil de optimizar. Puedes ahorrar un montón de tiempo y de esfuerzo si delegas el problema de la manipulación del DOM en una buena implementación que lo lleve a cabo.

React te permite realizar ciertos ajustes de rendimiento implementando unos puntos de enganche con los que ajustar la forma en la que se actualiza el árbol virtual. Por lo general esto es, a menudo, algo opcional.

El mayor coste de tener DOM Virtual es que hace que la implementación de React sea muy grande. Es de esperar que el tamaño de aplicaciones pequeñas gire en torno a los 150-200 kB minimizadas, con React incluido. La compresión ayuda, pero aún así sigue siendo grande.

Soluciones como preact[2] y react-lite[3] te permiten obtener tamaños más pequeños a costa de perder algunas funcionalidades. Si estás preocupado por el tamaño quizá podrías considerar estas soluciones.

Hay librerías como Matt-Esch/virtual-dom[4] o paldepind/snabbdom[5] que se centran totalmente en el DOM Virtual. Échales un vistazo si estás interesado en la teoría y quieres conocer más.

[2] https://developit.github.io/preact/
[3] https://github.com/Lucifier129/react-lite
[4] https://github.com/Matt-Esch/virtual-dom
[5] https://github.com/paldepind/snabbdom

1.3 Renderizadores de React

Como mencionamos anteriormente, el enfoque de React hace que éste esté desacoplado de la web. Puedes utilizar React para implementar interfaces para varias plataformas. En nuestro caso utilizaremos un renderizador conocido como react-dom[6], que permite renderizar tanto en el lado del cliente como en el lado del servidor.

Renderizado Universal

Podemos utilizar react-dom para implementar lo que se conoce como el renderizado *universal*. La idea es conseguir que el servidor renderice una página inicial y envie los datos al cliente. Esto mejora el rendimiento evitando tener que hacer llamadas innecesarias entre el cliente y el servidor que puedan provocar sobrecarga. Además, es útil para SEO.

Aunque el funcionamiento de esta técnica parezca sencillo, puede ser dificil de implementar en el caso de aplicaciones grandes. Aún así es algo que merece la pena conocer.

A menudo utilizar el renderizado del lado del servidor de react-dom es suficiente. Puedes utilizarlo, por ejemplo, para generar facturas[7], lo cual demuestra que React puede usarse de una forma flexible. La generación de informes es una funcionalidad común después de todo.

Renderizadores de React Disponibles

Aunque react-dom sea el renderizador más comúnmente utilizado, existen otros de los que deberías ser consciente. A continuación muestro una lista con algunas de las alternativas más populares:

- React Native[8] - React Native es un framework y un renderizador para plataformas móviles, incluido iOS y Android. También puedes ejecutar aplicaciones hechas con React Native en la web[9].

[6] https://www.npmjs.com/package/react-dom
[7] https://github.com/bebraw/generate-invoice
[8] https://facebook.github.io/react-native/
[9] https://github.com/necolas/react-native-web

- react-blessed[10] - react-blessed te permite escribir aplicaciones de terminal utilizando React. Incluso es posible animarlas.
- gl-react[11] - gl-react provee a React de soporte para WebGL. Con ello puedes, por ejemplo, hacer shaders.
- react-canvas[12] - react-canvas provee a React de soporte para gestionar el elemento Canvas.

1.4 `React.createElement` y JSX

Ya que vamos a trabajar con un DOM Virtual, hay un API de alto nivel[13] que nos permitirá gestionarlo. Este es el aspecto que tiene un componente nativo de React que utilice el API JavaScript:

```
const Names = () => {
  const names = ['John', 'Jill', 'Jack'];

  return React.createElement(
    'div',
    null,
    React.createElement('h2', null, 'Names'),
    React.createElement(
      'ul',
      { className: 'names' },
      names.map(name => {
        return React.createElement(
          'li',
          { className: 'name' },
          name
        );
      })
```

[10] https://github.com/Yomguithereal/react-blessed
[11] https://projectseptemberinc.gitbooks.io/gl-react/content/
[12] https://github.com/Flipboard/react-canvas
[13] https://facebook.github.io/react/docs/top-level-api.html

```
    )
  );
};
```

Puesto que es muy largo escribir componentes de esta manera y que son muy difíciles de leer, la gente por lo general suele preferir utilizar un lenguaje conocido como JSX[14] en su lugar. Observa el mismo componente escrito con JSX a continuación:

```
const Names = () => {
  const names = ['John', 'Jill', 'Jack'];

  return (
    <div>
      <h2>Names</h2>

      {/* Esto es una lista de nombres */}
      <ul className="names">{
        names.map(name =>
          <li className="name">{name}</li>
        )
      }</ul>
    </div>
  );
};
```

Ahora podemos ver que el componente renderiza un conjunto de nombres dentro de una lista HTML. Puede que no sea el componente más útil del mundo, pero es suficiente para ilustrar el concepto básico de qué es JSX. Nos facilita una sintaxis que parece HTML. También permite escribir JavaScript utilizando las llaves (`{}`).

Comparado con HTML plano, estamos usando `className` en lugar de `class`. Esto se debe a que el API ha sido inspirado en el nombrado de DOM. Lleva algo de tiempo acostumbrarse y puede que sufras un shock con JSX[15] hasta que comiences a apreciar esta aproximación. Nos da un nivel extra de validación.

[14] https://facebook.github.io/jsx/

[15] https://medium.com/@housecor/react-s-jsx-the-other-side-of-the-coin-2ace7ab62b98

HyperScript[16] es una alternativa interesante a JSX que brinda un API JavaScript más cercano al metal. Puedes utilizar la sintaxis con React a través de hyperscript-helpers[17].

Existe una diferencia semántica entre los componentes de React y los elementos de React. En el ejemplo anterior cada uno de los nodos JSX puede ser convertido en un elemento. Simplificando, los componentes pueden tener estado mientras que los elementos son, por naturaleza, más sencillos. Son objetos puros. Dan Abramov entra en más detalle en en la siguiente entrada[18].

1.5 Conclusión

Ahora que entendemos aproximadamente qué es React podemos ir a algo más técnico. Es la hora de tener un proyecto pequeño configurado y funcionando.

[16] https://github.com/dominictarr/hyperscript
[17] https://www.npmjs.com/package/hyperscript-helpers
[18] https://facebook.github.io/react/blog/2015/12/18/react-components-elements-and-instances.html

2. Configurando el Proyecto

Para que comenzar sea más sencillo he configurado una esqueleto basado en Webpack que nos permitirá adentrarnos en React de forma directa. Este esqueleto incluye un modo de desarrollo con una característica conocida como *recarga en caliente*.

La recarga en caliente le permite a Webpack a cambiar el código que se está ejecutando en el navegador sin tener que recargarlo todo. Funciona genial, especialmente a la hora de trabajar con estilos, aunque el soporte de React es también bastante bueno.

Por desgracia no es una tecnología a prueba de fallos y no siempre es capaz de detectar todos los cambios que se hayan hecho en el código, lo que significa que habrá veces que tendrás que recargar el navegador a mano para que éste tenga los últimos cambios.

Los editores de texto más comunes (Sublime Text, Visual Studio Code, vim, emacs, Atom, etc) tienen un buen soporte para React. Incluso los IDEs, como WebStorm[1], tienen soporte hasta cierto punto. Nuclide[2], un IDE basado en Atom, ha sido desarrollado con React en mente. Asegúrate de que tienes los plugins de React instalados y funcionando.

Si utilizas un IDE, deshabilita una característica conocida como **escritura segura**. Se sabe que causa problemas con la configuración que vamos a utilizar en este libro.

[1] https://www.jetbrains.com/webstorm/
[2] http://nuclide.io/

2.1 Configuración de Node.js y Git

Antes de comenzar, asegúrate de que tienes instaladas las últimas versiones tanto de Node.js[3] como de Git[4]. Te recomiento que utilices, al menos, la última versión LTS de Node.js. Puede que tengas errores difíciles de depurar con versiones anteriores, y lo mismo puede ocurrir con versiones posteriores a la LTS.

Una opción interesante es gestionar tu entorno a través de Vagrant[5] o mediante una herramienta como nvm[6].

Descargando el Esqueleto

Para poder descargar el esqueleto que nuestro proyecto necesita, clónalo con Git de la siguiente manera desde una terminal:

```
git clone https://github.com/survivejs/react-boilerplate.git kanban-\
app
```

Esto creará un nuevo directorio llamado *kanban-app*. Dentro encontrarás todo lo que necesitas para poder avanzar. Ya que el esqueleto puede cambiar dependiendo de la versión del libro, te recomiendo que cambies a la versión específica del mismo:

```
cd kanban-app
git checkout v2.5.6
```

El repositorio contiene una pequeña aplicación a modo de semilla que muestra un `Hello World!` y una configuración de Webpack básica. Para instalar las dependencias de la semilla simplemente ejecuta:

[3] https://nodejs.org
[4] https://git-scm.com/
[5] https://www.vagrantup.com/
[6] https://www.npmjs.com/package/nvm

```
npm install
```

Una vez termine deberás ver un directorio llamado `node_modules/` con todas las dependencias del proyecto.

Creando un Repositorio Nuevo de Git para tu Proyecto

No sólo te has descargado el proyecto `react-boilerplate`, sino que además te has descargado el historial del mismo. Este historial no es realmente importante para tu nuevo proyecto, así que es un buen momento para borrar el historial de git y comenzar con un repositorio limpio. Este nuevo repositorio reflejará la evolución de tu proyecto. Es una buena idea que, en el commit inicial, menciones la versión del esqueleto de la que partes:

```
rm -rf .git
git init
git add .
git commit -am "New project based on react-boilerplate (v2.5.6)"
```

Tras esto tendrás un repositorio limpio en el que trabajar.

2.2 Ejecutando el Proyecto

Ejecuta `npm start` para arrancar el proyecto. Deberías tener una salida como la siguiente si todo ha ido bien:

```
> webpack-dev-server

http://localhost:8080/
webpack result is served from /
content is served from .../kanban-app
404s will fallback to /index.html
Child html-webpack-plugin for "index.html":

webpack: bundle is now VALID.
```

En caso de que recibas un error asegúrate de que no tengas ningún otro proceso escuchando en el mismo puerto. Puedes hacer que la aplicación escuche en otro puerto distinto usando un comando similar a `PORT=3000 npm start` (sólo para Unix). La configuración utilizará el puerto que hayas indicado en la variable de entorno. Si quieres fijar el puerto con un valor específico configúralo en el fichero *webpack.config.js*.

Si todo ha ido bien deberás ver algo como esto en el navegador:

Salida del típico 'Hello world'

Puedes probar a modificar el código fuente para ver cómo funciona la recarga en caliente.

Hablaré del esqueleto con más detalle más adelante para que sepas cómo funciona. También mostraré brevemente qué características del lenguaje vamos a utilizar.

Las técnicas utilizadas en el esqueleto están cubiertas con más detalle en SurviveJS - Webpack[7].

[7] http://survivejs.com/webpack/introduction/

2.3 `scripts` de npm Presentes en el Esqueleto

Nuestro esqueleto es capaz de generar una aplicación que puede ser desplegada en producción. Hay una meta relacionada con el proceso de despliegue con la que podrás mostrar tu proyecto a otras personas mediante GitHub Pages[8]. A continuación tienes una lista con todos los `scripts`:

- `npm run start` (o `npm start`) - Ejecuta el proyecto en modo desarrollo. Navega hacia `localhost:8080` desde tu navegador para verlo funcionando.
- `npm run build` - Produce una compilación lista para producción en `build/`. Puedes abrir el fichero *index.html* en el navegador para ver el resultado.
- `npm run deploy` - Despliega el contenido de `build/` en la rama *gh-pages* de tu proyecto y lo sube a GitHub. Podrás acceder al proyecto a través de la URL `<usuario>.github.io/<proyecto>`. Para que funcione correctamente deberás configurar la variable `publicPath` del fichero *webpack.config.js* para que encaje con el nombre de tu proyecto en GitHub.
- `npm run stats` - Genera estadísticas (*stats.json*) sobre el proyecto. Puedes analizar los resultados[9] más adelante.
- `npm run test` (o `npm test`) - Ejecuta los tests del proyecto. El capítulo *Probando React* entra más adelante en este asunto. De hecho, una buena manera de aprender mejor cómo funciona React es escribir tests que prueben tus componentes.
- `npm run test:tdd` - Ejecuta los tests del proyecto en modo TDD, lo que significa que se quedará a la espera de cambios en los ficheros y lanzará los tests cuando se detecten cambios, lo que te permititá ir más deprisa ya que te evitará tener que lanzar los tests manualmente.
- `npm run test:lint` - Ejecuta ESLint[10] contra el código. ESLint es capaz de capturar pequeños problemas. Puedes configurar tu entorno de desarrollo para que lo utilice y te permitirá capturar errores potenciales a medida que los cometas.

[8] https://pages.github.com/

[9] http://survivejs.com/webpack/building-with-webpack/analyzing-build-statistics/

[10] http://eslint.org/

Revisa la sección `"scripts"` del fichero *package.json* para entender mejor cómo funciona cada uno de ellos. Es casi como configuración. Echa un vistazo a SurviveJS - Webpack[11] para saber más sobre este tema.

2.4 Características del Lenguaje Presentes en el Esqueleto

Babel

El esqueleto depende de un transpilador llamado Babel[12] que nos permite utilizar características del JavaScript del futuro. Se encarga de transformar tu código en un formato que los navegadores puedan entender. Puedes incluso desarrollar tus propias características del lenguaje. Permite el uso de JSX mediante un plugin.

Babel da soporte a algunas características experimentales[13] de ES7 que van más allá de ES6. Algunas de ellas podrían llegar formar parte del lenguaje, mientras que otras podrían ser eliminadas por completo. Las propuestas del lenguaje han sido categorizadas en etapas:

- **Etapa 0** - Hombre de paja
- **Etapa 1** - Propuesta
- **Etapa 2** - Borrador
- **Etapa 3** - Candidata
- **Etapa 4** - Finalizada

[11] http://survivejs.com/webpack/introduction/
[12] https://babeljs.io/
[13] https://babeljs.io/docs/plugins/#stage-x-experimental-presets-

Yo tendría mucho cuidado con las características de la **etapa 0**. El problema es que acabará rompiendo código que habrá que reescribir en caso de ésta cambie o sea borrada. Quizá en pequeños problemas experimentales merezca la pena correr el riesgo.

Aparte de ES2015 estándar y de JSX, vamos a utilizar algunas características extra en este proyecto. Las he listado a continuación. Echa un vistazo al apéndice *Características del Lenguaje* para saber más sobre ellas.

- Inicializadores de propiedades[14] - Ejemplo: `addNote = (e) => {`. Esto relaciona al método `addNote` automáticamente a una instancia. Esta característica tendrá más sentido a medida que la vayamos utilizando.
- Decoradores[15] - Ejemplo: `@DragDropContext(HTML5Backend)`. Estas anotaciones nos permitirán incluir funcionalidad a clases y a sus métodos.
- rest/spread de Objetos[16] - Ejemplo: `const {a, b, ...props} = this.props`. Esta sintáxis nos permite recuperar fácilmente propiedades específicas de un objeto.

He creado un preset[17] para que sea más sencillo configurar estas características. Contiene los plugins babel-plugin-transform-object-assign[18] y babel-plugin-array-includes[19]. El primero nos permite usar `Object.assign` mientras que el último incluye `Array.includes` sin que tengamos que preocuparnos de compatibilidades con entornos antiguos.

Un preset es simplemente un módulo de npm que exporta configuración de Babel. Mantener presets como éste puede ser útil si quieres mantener el mismo conjunto de funcionalidades entre varios proyectos.

Puedes probar Babel online[20] para ver el tipo de código que genera.

[14] https://github.com/jeffmo/es-class-static-properties-and-fields
[15] https://github.com/wycats/javascript-decorators
[16] https://github.com/sebmarkbage/ecmascript-rest-spread
[17] https://github.com/survivejs/babel-preset-survivejs-kanban
[18] https://www.npmjs.com/package/babel-plugin-transform-object-assign
[19] https://www.npmjs.com/package/babel-plugin-array-includes
[20] https://babeljs.io/repl/

Si estás interesado en una alternativa más ligera, echa un vistazo a Bublé[21].

2.5 Conclusión

Ahora que tenemos el clásico “Hello World!” funcionando podemos centrarnos en el desarrollo. Desarrollar y tener problemas es una buena forma de aprender después de todo.

[21] https://gitlab.com/Rich-Harris/buble

3. Implementando una Aplicación de Notas

Ahora que tenemos un buen entorno de desarrollo podemos hacer algo que funcione. Nuestra meta en este momento es acabar teniendo una aplicación primitiva que permita tomar notas. Tendremos las operaciones de manipulación básicas. Haremos que nuestra aplicación crezca desde la nada y nos meteremos en problemas, lo que nos permitirá entender por qué son necesarias arquitecturas como Flux.

3.1 Modelo de Datos Inicial

A menudo, una buena forma de comenzar con el desarrollo de una aplicación es empezar con los datos. Podemos modelar una lista de notas tal y como sigue:

```
[
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    task: 'Do laundry'
  }
];
```

Cada nota es un objeto que contiene los datos que necesitamos, incluyendo un identificador (`id`) y el nombre de la tarea (`task`) que queremos llevar a cabo. Más adelante podremos extender esta definición para incluir cosas como el color de las notas o su propietario.

Podríamos haber ignorado los identificadores en nuestra definición, pero esto puede volverse un problema a medida que la aplicación crece si tratamos de referenciarlas. Al fin y al cabo, cada columna de Kanban necesita ser capaz de referenciar algunas notas. Adoptando los índices desde el principio ahorraremos algo de esfuerzo más adelante.

Otra forma interesante de aproximarse a los datos puede ser normalizarlos. En este caso podríamos acabar con una estructura del tipo `[<id> -> { id: '...', task: '...' }]`. Incluso aunque quizá tenga algo de redundante, es conveniente utilizar la estructura de esta forma ya que nos facilita poder acceder mediante a los elementos mediante el índice. La estructura se volverá más útil todavía una vez empecemos a tener referencias entre entidades.

3.2 Renderizado de los Datos Iniciales

Ahora que tenemos un modelo de datos inicial, podemos tratar de renderizarlo utilizando React. Vamos a necesitar un componente que retenga los datos, lo llamaremos `Notas` de momento y lo haremos crecer si queremos que tenga más funcionalidad. Crea un fichero con un componente sencillo como el siguiente:

app/components/Notes.jsx

```
import React from 'react';

const notes = [
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    task: 'Do laundry'
  }
```

```
];

export default () => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
  )}</ul>
)
```

Estamos utilizando algunas características importantes de JSX en el trozo de código anterior. He destacado las partes más difíciles:

- `<ul>{notes.map(note => ...)}</ul>` - `{}` nos permite mexclar sintaxis de JavaScript con JSX. `map` devuelve una lista de elementos `li` para que React los renderice.
- `<li key={note.id}>{note.task}</li>` - Usamos la propiedad `key` para poder decirte a React qué items han sido cambiados, modificados o borrados. Es importante que sea único ya que, sino, React no será capaz de saber el orden correcto en el que debe renderizarlos. React mostrará una advertencia si no se indican. Puedes leer el enlace al artículo Renderizando Varios Componentes[1] para obtener más información.

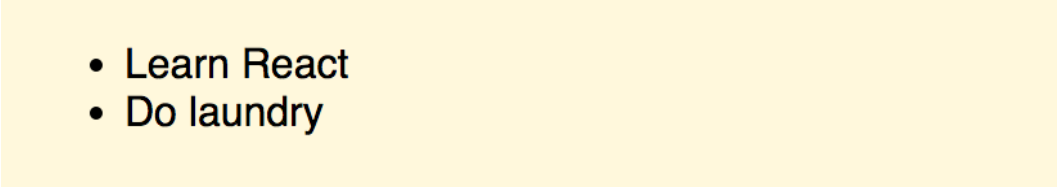
Necesitamos hacer referencia al componente desde el punto de entrada de nuestra aplicación:

app/index.jsx

[1] https://facebook.github.io/react/docs/lists-and-keys.html#rendering-multiple-components

```
import React from 'react';
import ReactDOM from 'react-dom';
import Notes from './components/Notes';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
  <div>Hello world</div>,
  <Notes />,
  document.getElementById('app')
);
```

Si ejecutas la aplicación verás una lista de notas. No es especialmente bonito ni útil todavía pero es un comienzo:



Una lista de notas

Necesitamos usar el `import` de React en *Notes.jsx* ya que hay transformaciones que hacer de JSX a JavaScript. Sin él el código resultante fallará.

3.3 Generando los Ids

Habitualmente el problema de generar los ids se resuelve por un backend. Ya que no tenemos ninguno todavía, en su lugar, vamos a utilizar un estándar conocido como RFC4122[2] que nos permitirá generar identificadores únicos. Utilizaremos una

[2] https://www.ietf.org/rfc/rfc4122.txt

implementación de Node.js conocida como *uuid* y su variante `uuid.v4` que nos dará ids tales como `1c8e7a12-0b4c-4f23-938c-00d7161f94fc` que, casi con toda seguridad, serán únicos.

Para utilizar el generador en nuestra aplicación modifícala como sigue:

app/components/Notes.jsx

```
import React from 'react';
import uuid from 'uuid';

const notes = [
  {
    id: '4e81fc6e-bfb6-419b-93e5-0242fb6f3f6a',
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: '11bbffc8-5891-4b45-b9ea-5c99aadf870f',
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

...
```

Nuestra configuración de desarrollo instalará la dependencia `uuid` automáticamente. Una vez que esto haya ocurrido y que la aplicación se haya recargado, todo debería tener el mismo aspecto. Sin embargo, si pruebas a depurar la aplicación, verás que los ids cambiarán cada vez que refresques la página. Puedes comprobarlo fácilmente o bien insertando la línea `console.log(notes);` o bien usando el comando debugger;[3] dentro del componente.

El comando `debugger;` es especialmente útil ya que le indica al navegador que debe parar la ejecución. De este modo es posible ver la pila de llamadas y examinar las

[3] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Sentencias/debugger

variables que estén disponibles. Es una buena forma de depurar las aplicaciones y suponer qué está ocurriendo si no estás seguro de cómo funciona algo.

`console.log` es una alternativa más ligera. Puedes incluso diseñar un sistema de gestión de logs en torno a él y usar ambas técnicas juntas. Echa un vistazo a MDN[4] y a la documentación de Chrome[5] para ver el API completo.

Si estás interesado en conocer las matemáticas que se esconden tras la generación de los id, puedes ver más detalles sobre cómo se hacen estos cálculos en la Wikipedia[6]. Verás que la posibilidad de que haya una colisión es realmente pequeña y que no debemos preocuparnos por ello.

3.4 Añadiendo Nuevas Notas a la Lista

Aunque de momento podemos mostrar notas de forma individual, todavía nos falta mucha lógica que hará que nuestra aplicación sea útil. Una forma lógica de comenzar puede ser implementando la inclusión de nuevas notas a la lista. Para conseguirlo necesitamos hacer que la aplicación crezca un poco.

Definiendo un Borrador para `App`

Para poder añadir nuevas notas necesitaremos tener un botón que nos lo permita en algún sitio. Actualmente nuestro componente `Notas` sólo hace una cosa: mostrar notas. Esto es totalmente correcto. Para hacer espacio a más funcionalidad podemos incluir un concepto conocido como `App` en lo más alto. Este componente orquestará la ejecución de nuestra aplicación. Podemos añadir el botón que queramos allí, podremos gestionar el estado y también podremos añadir notas. Inicialmente `App` puede tener el siguiente aspecto:

app/components/App.jsx

[4] https://developer.mozilla.org/es/docs/Web/API/Console

[5] https://developers.google.com/web/tools/chrome-devtools/debug/console/console-reference

[6] https://en.wikipedia.org/wiki/Universally_unique_identifier#Random_UUID_probability_of_duplicates

```
import React from 'react';
import Notes from './Notes';

export default () => <Notes />;
```

Todo lo que hace es renderizar `Notas`, así que debe hacer más cosas para conseguir que sea útil. Tenemos que alterar el punto de entrada tal y como sigue para incrustar `App` en nuestra aplicación:

app/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
~~import Notes from './components/Notes';~~
import App from './components/App';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
  ~~<Notes />,~~
  <App />,
  document.getElementById('app')
);
```

Si ejecutas la aplicación ahora verás que tiene exactamente el mismo aspecto que antes, pero ahora tenemos espacio para crecer.

Añadiendo un Borrador para el Botón *Añadir*

Un buen paso para llegar a algo más funcionar es añadir un borrador para el botón *añadir*. Para conseguirlo, `App` necesita evolucionar:

app/components/App.jsx

```
import React from 'react';
import Notes from './Notes';

export default () => <Notes />;
export default () => (
  <div>
    <button onClick={() => console.log('añadir nota')}>+</button>
    <Notes />
  </div>
);
```

Si pulsas sobre el botón que hemos añadido verás un mensaje con el texto "añadir nota" en la consola del navegador. Todavía necesitamos conectar de alguna forma el botón con nuestros datos. De momento los datos están atrapados dentro del componente `Notas` así que, antes de seguir, necesitamos sacarlos y dejarlos a nivel de `App`.

Tenemos que envolver nuestra aplicación dentro de un `div` porque todos los componentes de React deben devolver un único elemento.

Llevando los Datos a `App`

Para llevar los datos a `App` necesitamos hacer un par de cambios. Lo primero que necesitamos es moverlos literalmente allí y mandar los datos mediante una propiedad (prop en adelante) a `Notas`. Tras esto necesitamos realizar cambios en `Notas` para realizar operaciones en base a la nueva lógica. Una vez hayamos conseguido esto podremos empezar a pensar en añadir notas nuevas.

En la parte de `App` el cambio es sencillo:

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

export default () => (
  <div>
    <button onClick={() => console.log('añadir nota')}>+</button>
    <Notes />
    <Notes notes={notes} />
  </div>
);
```

Esto no hará mucho hasta que también cambiemos `Notas`:

app/components/Notes.jsx

```
import React from 'react';
import uuid from 'uuid';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
```

```
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

export default () => {
export default ({notes}) => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
  )}</ul>
);
```

Nuestra aplicación tendrá el mismo aspecto que antes de que hiciéramos los cambios, pero ahora estamos listos para añadir algo de lógica.

La forma de extraer `notes` de `props` (el primer parámetro) es un truco estándar que verás con React. Si quieres acceder al resto de `props` puedes usar una sintaxis como `{notes, ...props}`. Más adelante lo utilizaremos de nuevo para que te hagas una idea más clara de cómo funciona y para qué puedes utilizarlo.

Llevando el Estado a `App`

Ahora que tenemos todo colocado y en el lugar correcto podemos comenzar a preocuparnos sobre cómo modificar los datos. Si has utilizado JavaScript con anterioridad verás que la forma más intuitiva de hacer esto es configurar un evento de este estilo: `() => notes.push({id: uuid.v4(), task: 'New task'})`. Si lo intentas verás que no ocurre nada.

El motivo es sencillo. React no se ha enterado de que la estructura ha cambiado y, por tanto, no reacciona (esto es, no invoca a `render()`). Para solucionar este problema podemos implementar nuestra modificación haciendo que utilice el propio API de React, lo que hará que sí se entere de que la estructura ha cambiado y, como resultado, ejecutará `render()` tal y como esperamos.

En el momento de escribir esto la definición de componentes basada en funciones no soporta el concepto de estado. El problema aparece porque estos componentes no tienen una instancia por detrás que les respalde. Podríamos ver cómo arreglar esto utilizando únicamente funciones, pero por ahora tendremos que utilizar la alternativa de hacer el trabajo duro por nosotros mismos.

Además de funciones, también puedes crear componentes de React utilizando `React.createClass` o una definición de componentes basada en clases. En este libro utilizaremos componentes basados en funciones tanto como sea posible, y sólo si hay una buena razón por la cual estos componentes no pueden funcionar, entonces utilizaremos definiciones basadas en clases.

Con el objetivo de transformar nuestra `App` en un componente basado en clases, cámbialo como sigue para meter el estado dentro.

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

const notes = [
  {
    id: uuid.v4(),
    task: 'Learn React'
  },
  {
    id: uuid.v4(),
    task: 'Do laundry'
  }
];

export default () => (
  <div>
    <button onClick={() => console.log('añadir nota')}>+</button>
    <Notes notes={notes} />
  </div>
```

```
);

export default class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      notes: [
        {
          id: uuid.v4(),
          task: 'Learn React'
        },
        {
          id: uuid.v4(),
          task: 'Do laundry'
        }
      ]
    };
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={() => console.log('add note')}>+</button>
        <Notes notes={notes} />
      </div>
    );
  }
}
```

Tras este cambio `App` contiene el estado aunque la aplicación siga pareciendo la misma de antes. Es ahora cuando podemos comenzar a usar el API de React para modificar el estado.

Las soluciones de gestión de datos, tales como MobX[7], arreglan el problema a su manera. Utilizándolos sólo necesitas poner anotaciones en tus estructuras de datos, en los componentes de React, y ellos se encargan de resolver del problema de actualizarse. Volveremos más adelante a tratar este tema de la gestión de datos en detalle.

Estamos pasando `props` a `super` por convención. Si no lo haces, ¡`this.props` no cambiará!. Llamar a `super` provoca que se invoque al mismo método de la clase padre del mismo modo a como se hace en la programación orientada a objetos.

Implementando la Lógica de Añadir `Nota`

Todos nuestros esfuerzos pronto se verán recompensados. Sólo nos queda un paso, tan sólo necesitamos utilizar el API de React para manipular el estado. React facilita un método conocido como `setState` con este objetivo. En este caso lo invocaremos como sigue: `this.setState({... el nuevo estado viene aquí ...}, () => ...)`.

El callback es opcional. React lo invocará una vez haya establecido el estado y, por lo general, no tienes que preocuparte de ello para nada. React invocará a `render` una vez que `setState` haya terminado. El API asíncrono te puede parecer un poco extraño al principio pero le permite a React ser capaz de optimizar su rendimiento utilizando técnicas como las actualizaciones en bloque. Todo esto recae en el concepto de DOM Virtual.

Una forma de invocar a `setState` puede ser dejar toda la lógica relacionada en un método para llamarlo una vez se crea una nueva nota. La definición de componentes basada en clases no permite enlazar métodos personalizados como éste por defecto así que necesitaremos gestionar esta asociación en algún sitio. Podría ser posible hacer esto en el `constructor`, `render()`, o utilizando una sintaxis específica. Voy a optar por la solución de la sintaxis en este libro. Lee el apéndice *Características del Lenguaje* para aprender más.

Para atar la lógica al botón, `App` debe cambiar como sigue:

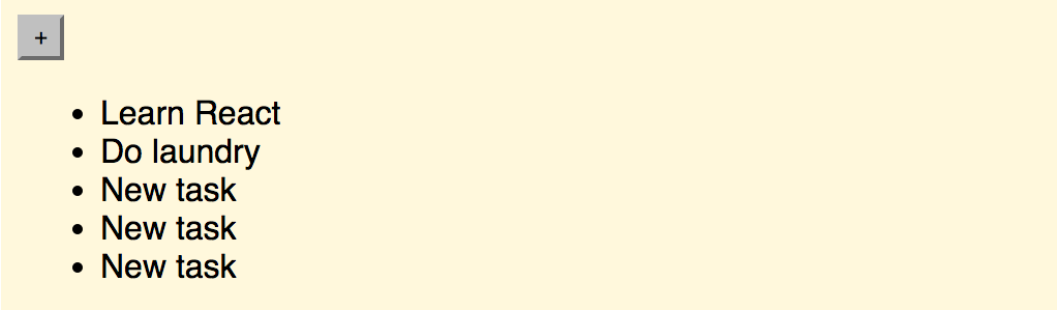
[7] https://mobxjs.github.io/mobx/

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={() => console.log('add note')}>+</button>
        <button onClick={this.addNote}>+</button>
        <Notes notes={notes} />
      </div>
    );
  }
  addNote = () => {
    // Es posible escribir esto de forma imperativa, es decir,
    // a través de `this.state.notes.push` y, después,
    // `this.setState({notes: this.state.notes})`.
    //
    // Suelo favorecer el estilo funcional cuando tiene sentido.
    // Incluso cuando es necesario escribir más código, ya que
    // prefiero los beneficios (facilidad para razonar, no
    // efectos colaterales) que trae consigo.
    //
    // Algunas librerias, como Immutable.js, van un paso más allá.
    this.setState({
      notes: this.state.notes.concat([{
        id: uuid.v4(),
        task: 'New task'
```

```
      }])
    });
  }
}
```

Dado que, en este punto, estamos enlazando una instancia, la recarga en caliente no se dará cuenta del cambio. Para probar la nueva funcionalidad tienes que refrescar el navegador y pulsar sobre el botón +. Deberías ver algo:



Notas con un más

Si estuviésemos utilizando un backend podríamos lanzar una consulta y capturar el id de la respuesta. De momento es suficiente con generar una entrada y un id aleatorio.

Podríamos utilizar `this.setState({notes: [...this.state.notes, {id: uuid.v4(), task: 'New task'}]})` para conseguir el mismo resultado. Este operador de propagación[8] puede ser utilizado también con funciones recibidas como parámetros. Mira el apéndice *Características del Lenguaje* para más información.

[8] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Operadores/Spread_operator

El autobind-decorator[9] puede ser una alternativa válida a la hora de inicializar propiedades. En ese caso podríamos utilizar la anotación `@autobind` a nivel de clase o de método. Para aprender más sobre decoradores echa un vistazo al apéndice *Entendiendo los Decoradores*.

3.5 Conclusión

De momento tenemos sólo una aplicación primitiva, y hay dos funcionalidades críticas que necesitamos: la edición y el borrado de notas. Es un buen momento de centrarnos en ellas a partir de ahora. Primero comenzaremos con el borrado y, tras ello, con la edición.

[9] https://www.npmjs.com/package/autobind-decorator

4. Borrado de `Notas`

Una forma sencilla de permitir el borrado de notas consiste en mostrar un botón con una "x" en cada `Nota`. Cuando este botón sea pulsado, simplemente deberemos borrar la nota en cuestión de la estructura de datos. Tal y como hicimos antes podemos comenzar añadiendo borradores, éste puede ser un buen lugar en el que separar el concepto de `Nota` del componente `Notas`.

A menudo trabajarás de esta forma con React. Generarás componentes de los que más adelante te darás cuenta que están compuestos por otros componentes que pueden ser extraidos. Este proceso de separación es fácil, y a veces puede incluso incrementar el rendimiento de tu aplicación puesto que la estás optimizando al renderizar partes más pequeñas.

4.1 Separación de `Nota`

Para mantener una lista de `Nota` que mantengan el mismo aspecto podemos modelarla utilizando un `div` de este modo:

app/components/Note.jsx

```
import React from 'react';

export default ({task}) => <div>{task}</div>;
```

Recuerda que esta declaración es equivalente a:

```
import React from 'react';

export default (props) => <div>{props.task}</div>;
```

Como puedes ver, destructurar reduce la cantidad de ruido del código y permite que la implementación sea simple.

Para hacer que nuestra aplicación utilice el nuevo componente tenemos que hacer cambios también en Notas:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';

export default ({notes}) => (
  <ul>{notes.map(note =>
    <li key={note.id}>{note.task}</li>
    <li key={note.id}><Note task={note.task} /></li>
  )}</ul>
)
```

La aplicación debe tener el mismo aspecto que ya tenía antes de hacer los cambios, pero hemos hecho hueco para poder meter más cosas más adelante.

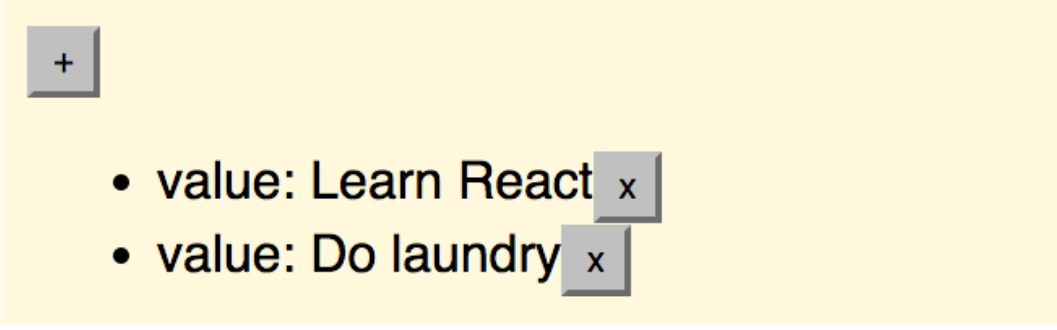
4.2 Añadir un Esqueleto para la Llamada a `onDelete`

Necesitamos extender las capacidades de Nota para capturar la intención de borrarla incluyendo una acción que se ejecute al llamar a onDelete. Presta atención al siguiente código:

app/components/Note.jsx

```
import React from 'react';

export default ({task}) => <div>{task}</div>;
export default ({task, onDelete}) => (
  <div>
    <span>{task}</span>
    <button onClick={onDelete}>x</button>
  </div>
);
```

Deberias ver una pequeña “x” después de cada nota:

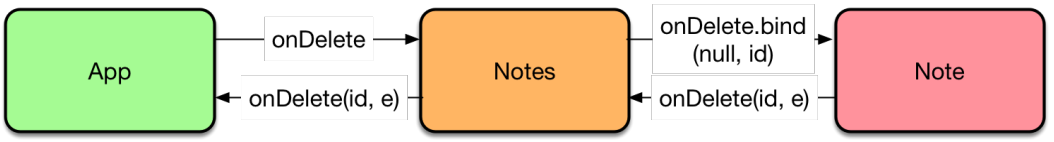


Notas con controles de borrado

Todavía no harán nada, arreglarlo es el siguiente paso.

4.3 Comunicar el Borrado a App

Ahora que tenemos los controles que necesitamos podemos comenzar a pensar en cómo conectarlos con los datos de App. Para poder borrar una Nota necesitamos conocer su id. Tras ello podremos implementar la lógica que se encarga de borrarlas en App. Para que te hagas una idea, queremos encontrarnos en una situación como la siguiente:



flujo de `onDelete`

La `e` representa un evento DOM al que deberias acostumbrarte. Podemos hacer cosas como parar la propagación de eventos con él. Esto se volverá más útil a medida que queramos tener más control sobre el comportamiento de la aplicación.

bind[1] nos permite definir el contexto de la función (el primer parámetro) y los argumentos (el resto de parámetros). Esta técnica se conoce con el nombre de **aplicación parcial**.

Para conseguir todo esto vamos a necesitar una nueva propiedad en `Notas`. También necesitaremos enlazar con `bind` el identificador de cada nota con la llamada a `onDelete` para hacer obrar la magia. Aquí tienes la implementación completa de `Notas`:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';

~~export default ({notes}) => (~~
~~  <ul>{notes.map(note =>~~
~~    <li key={note.id}><Note task={note.task} /></li>~~
~~  )}</ul>~~
~~)~~
export default ({notes, onDelete=() => {}}) => (
  <ul>{notes.map(({id, task}) =>
```

[1] https://developer.mozilla.org/es/docs/Web/JavaScript/Referencia/Objetos_globales/Function/bind

```
    <li key={id}>
      <Note
        onDelete={onDelete.bind(null, id)}
        task={task} />
    </li>
  )}</ul>
)
```

He definido un valor de retorno ficticio para evitar que nuestro código falle si no se proporciona un `onDelete`. Otra buena manera de conseguirlo es mediante el uso de `propTypes`, tal y como se muestra en el capítulo *Tipado con React*.

Ahora que tenemos las cosas en su lugar podemos usarlas con `App`:

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={this.addNote}>+</button>
        <Notes notes={notes} />
        <Notes notes={notes} onDelete={this.deleteNote} />
      </div>
    );
  }
  addNote = () => {
```

```
    ...
  }
  deleteNote = (id, e) => {
    // Dejar de procesar eventos para poder editar
    e.stopPropagation();

    this.setState({
      notes: this.state.notes.filter(note => note.id !== id)
    });
  }
}
```

Deberías poder borrar notas una vez hayas refrescado el navegador. Anticipándome al futuro he añadido la linea extra `e.stopPropagation()`. La idea subyacente es la de indicar al DOM que tiene que dejar de procesar eventos. En resumidas cuentas, vamos a evitar que se lancen otros eventos desde cualquier sitio que puedan afectar a la estructura si estamos borrando notas.

4.4 Conclusión

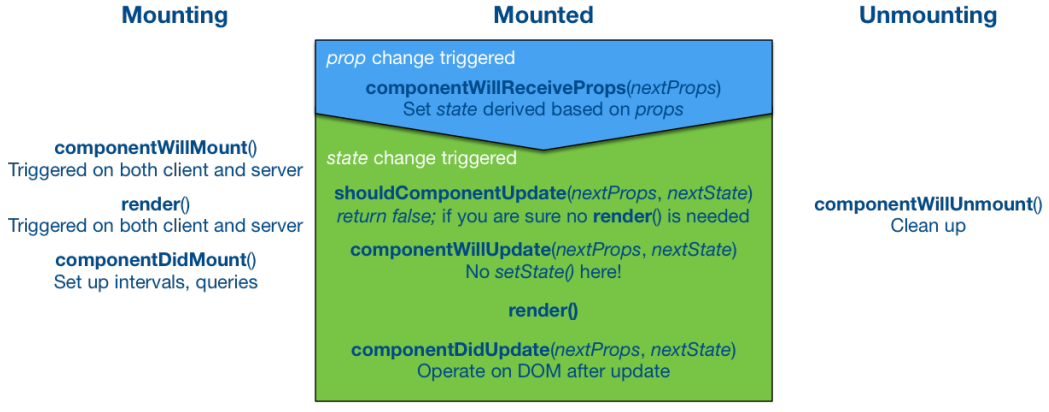
Trabajar con React a menudo es esto: identificar los componentes y flujos en base a tus necesidades. Aquí hemos necesitado modelar una `Nota` y hemos diseñado un flujo que nos permite tener el control sobre ella en el lugar correcto.

Todavía nos falta implementar una funcionalidad con la que terminar la primera parte del Kanban. La edición es la funcionalidad más difícil de todas. Una forma de implementarla es mediante un *editor en línea*. Implementar un componente adecuado ahora nos ayudará a ahorrar tiempo cuando tengamos que editar algo más. Antes de continuar con la implementación vamos a echar un vistazo más en detalle a los componentes de React para entender mejor qué tipo de funcionalidad nos pueden dar.

5. Comprendiendo los Componentes de React

Hasta donde hemos visto, los componentes de React son muy sencillos: Tienen un `estado` interno y aceptan propiedades (`props`). Aparte de esto, React facilita puntos de enganche que te permitirán hacer cosas más avanzadas, entre las que que se incluyen los métodos que gestionan el ciclo de vida y las referencias (`refs`). También existe una serie de propiedades y de métodos de los que te conviene ser consciente.

5.1 Métodos del Ciclo de Vida



Métodos del ciclo de vida

En la imagen superior puedes ver que cada componente de React tiene tres fases durante su ciclo de vida. Puede estar **montándose**, **montado** o **desmontado**. Cada una de esas tres fases tiene una serie de métodos relacionados.

Durante la fase de montaje puedes acceder a los siguientes:

- `componentWillMount()` se ejecutará una vez antes de renderizar nada. Una forma de utilizarlo puede ser cargar datos de forma asíncrona y forzar un renderizado a través de `setState`. `render()` verá que el estado ha sido actualizado y se ejecutará. La ejecución tendrá lugar durante el renderizado en el servidor.
- `componentDidMount()` se ejecutará tras el renderizado inicial. Es en este punto donde puedes acceder al DOM. Puedes usar este método, por ejemplo, para utilizar jQuery en un componente. Esta ejecución **no tendrá lugar** cuando se está renderizando en el servidor.

Una vez que un componente ha sido montado y está en funcionamiento puedes manipularlo con los siguientes métodos:

- `componentWillReceiveProps(object nextProps)` se ejecuta cuando el componente recibe propiedades nuevas. Puedes usarlo, por ejemplo, para modificar el estado de tu componente según estas propiedades que has recibido.
- `shouldComponentUpdate(object nextProps, object nextState)` te permite optimizar el renderizado, ya que devuelve `false` si detecta que no hay ningún cambio que aplicar tras comprobar las propiedades y el estado. Es aquí donde Immutable.js[1] y otras librerías similares te serán muy útiles a la hora de comprobar equidades. La documentación oficial[2] entra en más detalles.
- `componentWillUpdate(object nextProps, object nextState)` se ejecuta tras `shouldComponentUpdate` y antes de `render()`. No es posible utilizar `setState` aquí pero puedes, por ejemplo, cambiar propiedades de los estilos.
- `componentDidUpdate(object nextProps, object nextState)` se ejecuta tras el renderizado. En este punto puedes modificar el DOM. Puede ser útil para hacer que otro código funcione con React.

Para terminar, hay una enganche más que puedes utilizar cuando un componente está desmontándose:

- `componentWillUnmount()` se ejecuta justo antes de que un componente se desconecte del DOM. Es el lugar perfecto para limpiar recursos (por ejemplo, borrar temporizadores, elementos DOM personalizados, y cosas así).

[1] https://facebook.github.io/immutable-js/

[2] https://facebook.github.io/react/docs/optimizing-performance.html#shouldcomponentupdate-in-action

A menudo `componentDidMount` y `componentWillUnmount` van emparejados. Si configuras algo relacionado con el DOM o creas un listener en `componentDidMount` tendrás que recordar quitarlo con `componentWillUnmount`.

5.2 Refs

Los refs[3] de React te permiten acceder al DOM que hay por debajo fácilmente. Al utilizarlos podrás enlazar tu código a la página web.

Las referencias necesitan una instancia que les dé soporte, lo que significa que sólo funcionan con `React.createClass` o con definiciones de clases basadas en componentes. La idea principal es la siguiente:

```
<input type="text" ref="input" />

...

// Accede en cualquier lugar
this.refs.input
```

Aparte de a cadenas de texto, las referencias permiten que realices una llamada una vez que el componente sea montado. Puedes inicializar algo en este punto o capturar la referencia:

```
<input type="text" ref={element => element.focus()} />
```

5.3 Propiedades y Métodos Propios

Más allá del ciclo de vida y de las referencias hay una gran cantidad de propiedades y métodos[4] de los cuales deberías ser consciente, especialmente si vas a utilizar `React.createClass`:

[3] https://facebook.github.io/react/docs/more-about-refs.html
[4] https://facebook.github.io/react/docs/component-specs.html

- `displayName` - Es preferible establecer un `displayName` ya que nos permitirá depurar mejor. Para las clases de ES6 este valor se genera automáticamente a partir del nombre de la clase. Puedes ponerle también un `displayName` a un componente basado en una función anónima.
- `getInitialState()` - Se puede conseguir lo mismo con clases utilizando el `constructor`.
- `getDefaultProps()` - En clases las estableces dentro del `constructor`.
- `render()` - Es la piedra angular de React. Debe devolver un único nodo[5] ya que si devuelves varios no funcionará.
- `mixins` - `mixins` contiene un array de mixins que aplicar a los componentes.
- `statics` - `statics` contiene propiedades estáticas y métodos para un componente. Con ES6 puedes asignárselos a la clase del siguiente modo:

```
class Note {
  render() {
    ...
  }
}
Note.willTransitionTo = () => {...};

export default Note;
```

También puedes escribirlo así:

[5] https://facebook.github.io/react/tips/maximum-number-of-jsx-root-nodes.html

```
class Note {
  static willTransitionTo() {...}
  render() {
    ...
  }
}

export default Note;
```

Algunas librerías, como React DnD, se apoyan en métodos estáticos para facilitar enganches de transición. Esto te permite controlar qué ocurre cuando un componente se muestra o se oculta. Por definición todo lo estático se encuentra disponible en la propia clase.

Los componentes de React te permiten documentar la interfaz de tu componente utilizando `propTypes` de este modo:

```
const Note = ({task}) => <div>{task}</div>;
Note.propTypes = {
  task: React.PropTypes.string.isRequired
}
```

Lee el capítulo *Tipado con React* Para saber más sobre `propTypes`.

5.4 Convenciones de los Componentes de React

Prefiero tener el contructor primero, seguido de los métodos del ciclo de vida, `render()` y, finalmente, los métodos usados por `render()`. Esta aproximación de arriba a abajo me hace más sencillo leer el código. Hay una convención opuesta que deja `render` como último método. Las convenciones con respecto a los nombres también varían. Tendrás que encontrar aquellas convenciones que te hagan sentir más cómodo.

Puedes obligarte a utilizar una convención utilizando un linter (un analizador de código) como ESLint[6]. El uso de linters decrementa la cantidad fricción que puede aparecer al trabajar sobre el código de otros. Incluso en proyectos personales, el uso de herramientas que te permitan verificar la sintaxis y los estándares son muy útiles. No sólo reduce la cantidad y la gravedad de los errores sino que además te permite encontrarlos cuanto antes.

Si configuras un sistema de integración contínua podrás realizar pruebas contra muchas plataformas y detectar errores de regresión pronto. Esto es especialmente importante si estás utilizando rangos de versiones no muy definidos, ya que a veces la gestioń de dependencias pueden acarrear problemas y es bueno detectarlos.

5.5 Conclusión

No sólo la definición de componentes de React es muy sencilla, sino que además es pragmática. Las partes más avanzadas pueden llevar tiempo hasta que se llegan a dominar, pero es bueno saber que están allí.

En el próximo capítulo continuaremos con la implementación que permitirá a los usuarios editar sus notas individualmente.

[6] http://eslint.org/

6. Edición de `Notas`

La edición de notas supone un problema similar al del borrado, ya que el flujo de datos es exactamente el mismo. Necesitamos definir qué hacer tras invocar a `onEdit` y hacer una asociación con `bind` al identificador de la nota dentro de `Notas` que está siendo editado.

Lo que hace que este escenario sea más difícil son los requisitos a nivel de interfaz de usuario. No es suficiente con tener un botón, necesitamos encontrar la manera de permitir al usuario introducir un nuevo valor y de persistirlo en el modelo de datos.

Una forma de conseguirlo es mediante la implementación de algo llamado **edición en línea**. La idea es que cuando un usuario pulse sobre una nota se muestre una caja de texto. Cuando el usuario haya terminado la edición, bien pulsando *enter* o bien pulsando fuera del campo (lanzando un evento de tipo blur), capturaremos el valor y lo actualizaremos.

6.1 Implementación de `Editable`

Vamos a mantener este comportamiento dentro de un componente conocido como `Editable` con el fin de mantener la aplicación limpia. El componente nos dará un API como el siguiente:

```
<Editable
  editing={editing}
  value={task}
  onEdit={onEdit.bind(null, id)} />
```

Este es un ejemplo de componente **controlado**. Controlaremos explícitamente el estado de la edición desde el exterior del componente, lo cual no sólo nos dará más poder sobre él, sino que nos permitirá que `Editable` sea más sencillo de utilizar.

Suele ser buena idea nombrar las llamadas con el prefijo on. Esto nos permitirá distinguirlas de otras propiedades y mantener el código un poco más limpio.

Diseño Controlado vs. Diseño no Controlado

Una forma alternativa de gestionar esto es dejar el control del estado editing a Editable. Esta **forma no controlada** de diseño puede ser válida si no quieres que el estado de componente pueda ser modificado desde fuera.

Utilizar ambos diseños es posible. Puedes tener un componente controlado que tenga elementos no controlados dentro. En nuestro caso tendremos un diseño no controlado para la caja de texto que Editable contendrá en este ejemplo.

Editable estará formado por dos partes bien separadas. Por un lado necesitamos mostrar el valor mientras no estamos editando, pero por otro querremos mostrar un componente de Edición en caso de que estemos editando.

Antes de entrar en detalles podemos implementar un pequeño esqueleto y conectarlo con la aplicación, lo que nos dará la estructura básica que necesitaremos para hacer crecer el resto. Para comenzar, haremos una marca en la jerarquía de componentes para hacer que sea más fácil implementar el esqueleto.

La documentación oficial de React entra en los componentes controlados[1] en más detalle.

6.2 Extrayendo el Renderizado de Nota

Ahora mismo la Nota controla qué se va a renderizar dentro de ella. Podríamos incluir Editable dentro de ella y hacer que todo funcione mediante la interfaz Nota. A pesar de que podría ser una forma válida de hacerlo, podemos mover la responsabilidad del procesamiento a un nivel superior.

[1] https://facebook.github.io/react/docs/forms.html

Tener el concepto de Nota será especialmente útil cuando queramos llevar la aplicación un paso más allá, así que no hay necesidad de borrarla. En su lugar, podemos darle el control sobre su renderizado a Notas.

React tiene una propiedad conocida como children que nos permitirá conseguir esto. Modifica Nota and Notas como se muestra a continuación para llevar el control de renderizado de Nota a Notas:

app/components/Note.jsx

```
import React from 'react';

~~export default ({task, onDelete}) => (~~
~~  <div>~~
~~    <span>{task}</span>~~
~~    <button onClick={onDelete}>x</button>~~
~~  </div>~~
~~);~~
export default ({children, ...props}) => (
  <div {...props}>
    {children}
  </div>
);
```

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';

export default ({notes, onDelete=() => {}}) => (
  <ul>{notes.map(({id, task}) =>
    <li key={id}>
~~      <Note~~
~~        onDelete={onDelete.bind(null, id)}~~
~~        task={task} />~~
      <Note>
```

```
        <span>{task}</span>
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Ahora que tenemos espacio para trabajar podemos definir un esqueleto para `Editable`.

6.3 Inclusión del Esqueleto `Editable`

Podemos definir un punto por el que comenzar basándonos en la especificación que sigue. La idea es que hagamos una cosa u otra basándonos en la propiedad `editing` y que hagamos lo necesario para implementar nuestra lógica:

app/components/Editable.jsx

```
import React from 'react';

export default ({editing, value, onEdit, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
  }

  return <span {...props}>value: {value}</span>;
}

const Edit = ({onEdit = () => {}, value, ...props}) => (
  <div onClick={onEdit} {...props}>
    <span>edit: {value}</span>
  </div>
);
```

Para ver el esqueleto en acción todavía necesitamos conectarlo con nuestra aplicación.

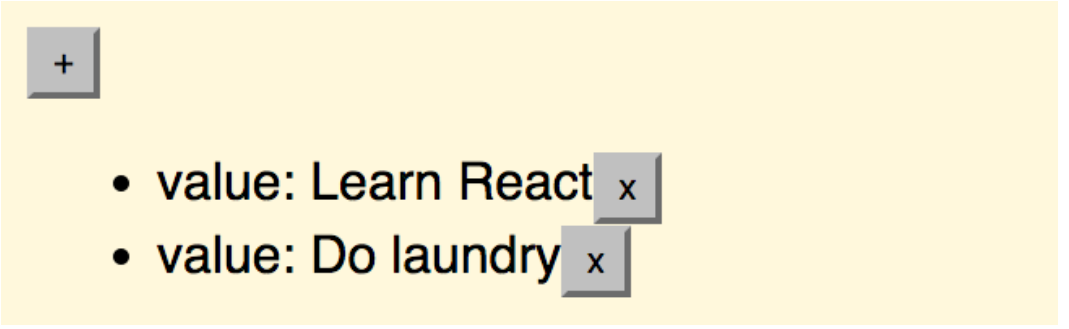
6.4 Conectando `Editable` con `Notas`

Todavía necesitamos cambiar las partes relevantes del código para que apunten a `Editable`. Hay más propiedades que conectar:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({notes, onDelete=() => {}}) => (
export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul>{notes.map(({id, task}) =>
    <li key={id}>
      <Note>
        <span>{task}</span>
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
  <ul>{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note onClick={onNoteClick.bind(null, id)}>
        <Editable
           editing={editing}
           value={task}
           onEdit={onEdit.bind(null, id)} />
        <button onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Si todo fue bien deberias ver algo como sigue:



Editable conectado

6.5 Haciendo un Seguimiento del Estado `editing` de `Nota`

Todavía nos falta la lógica necesaria para controlar `Editable`. Dado que el estado de nuestra aplicación está siendo mantenido en `App`, necesitaremos hacer cosas allí. Debería marcar el valor `editable` de una nota a `true` cuando comience con la edición y a `false` cuando el proceso de edición termine. También debería ajustar el valor de `task` al nuevo valor. De momento sólo estamos interesados en conseguir que el valor de `editable` funcione correctamente. Realizamos los siguientes cambios:

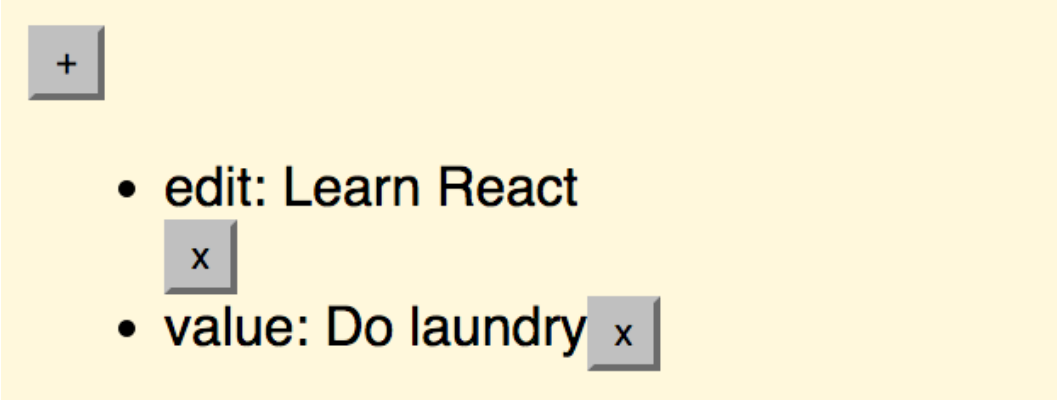
app/components/App.jsx

```
...

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
```

```
      <div>
        <button onClick={this.addNote}>+</button>
        <Notes notes={notes} onDelete={this.deleteNote} />
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
          />
      </div>
    );
  }
  addNote = () => {
    ...
  }
  deleteNote = (id, e) => {
    ...
  }
  activateNoteEdit = (id) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = true;
        }

        return note;
      })
    });
  }
  editNote = (id, task) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = false;
          note.task = task;
```

```
        }

        return note;
      })
    });
  }
}
```

Si tratas de editar una Nota ahora verás algo como lo siguiente:



Seguimiento del estado `editing`

Si pulsas en Nota dos veces para confirmar la edición verás un error llamado Uncaught Invariant Violation en la consola del navegador. Este ocurre porque todavía no hemos terminado de gestionar task correctamente. Esto es algo que deberemos arreglar a continuación.

Si usamos una estructura de datos normalizada (por ejemplo, {<id>: {id: <id>, task: <str>}}), es posible implementar las operaciones con Object.assign y evitar la mutación.

Para tener el código más limpio puedes extraer un método que contenga la lógica compartida por activateNoteEdit y por editNote.

6.6 Implementación de Edit

Nos falta algo que haga que esto funcione. Incluso aunque ahora podemos gestionar el estado de editing de cada Nota, todavía no podemos editarlas. Para ello necesitamos expandir Edit y hacer que muestre una caja de texto.

En este caso estaremos utilizando un diseño **no controlado** y obtendremos el valor de la caja de texto del árbol DOM sólo si lo necesitamos.

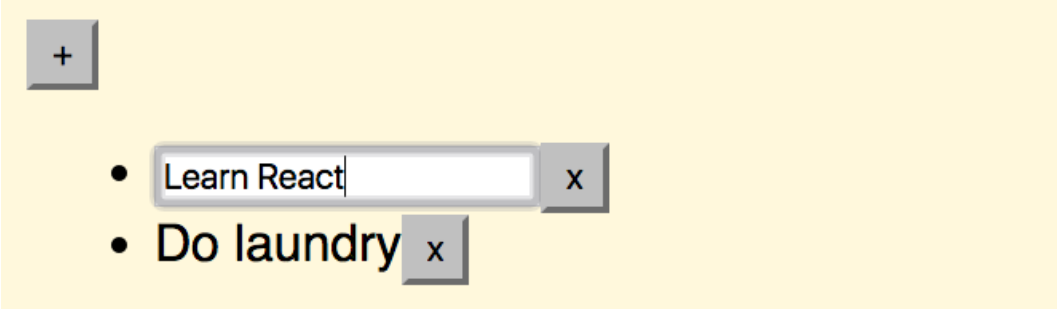
Fíjate en el código siguiente para ver la implementación completa. Observa cómo estamos gestionando el fin de la edición, capturamos onKeyPress y comprobamos si han pulsado Enter para confirmar la edición. También tenemos en cuenta al evento onBlur para saber cuándo la entrada de texto pierde el foco.

app/components/Editable.jsx

```
...


export default ({editing, value, onEdit, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
  }

  return <span {...props}>value: {value}</span>;
  return <span {...props}>{value}</span>;
}

const Edit = ({onEdit = () => {}, value, ...props}) => (
  <div onClick={onEdit} {...props}>
    <span>edit: {value}</span>
  </div>
);
class Edit extends React.Component {
  render() {
    const {value, onEdit, ...props} = this.props;
```

```
    return <input
      type="text"
      autoFocus={true}
      defaultValue={value}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter}
      {...props} />;
  }
  checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  }
  finishEdit = (e) => {
    const value = e.target.value;

    if(this.props.onEdit) {
      this.props.onEdit(value);
    }
  }
}
```

Si refrescas y editas una nota deberías ver lo siguiente:



Editando una Nota

6.7 Sobre los Componentes y el Espacio de Nombres

Podríamos haber abordado `Editable` de una forma diferente. En una edición anterior de este libro lo creé como un único componente. Lo hice mostrando el valor y el control de edición a través de métodos (esto es, mediante `renderValue`). A menudo, el nombrado de métodos como el anterior es una pista de que es posible refactorizar el código y extraer componentes como hicimos anteriormente.

Puedes ir un paso más adelante y colocar las partes de los componentes en un espacio de nombres[2]. De este modo habría sido posible definir los componentes `Editable.Value` y `Editable.Edit`. Mejor todavía, podríamos haber permitido al usuario intercambiar ambos componentes entre sí mediante props. Dado que la interfaz es la misma, los componentes deberían funcionar. Esto nos da una dimensión extra de personalización.

Llevándolo a la implementación, podemos tener algo como lo siguiente haciendo uso del espacio de nombres:

app/components/Editable.jsx

```
import React from 'react';

// Podemos conseguir que la edición y la presentación del valor se i\
ntercambien mediante props
const Editable = ({editing, value, onEdit}) => {
  if(editing) {
    return <Editable.Edit value={value} onEdit={onEdit} />;
  }

  return <Editable.Value value={value} />;
};

Editable.Value = ({value, ...props}) => <span {...props}>{value}</sp\
an>
```

[2] https://facebook.github.io/react/docs/jsx-in-depth.html#namespaced-components

```
class Edit extends React.Component {
  ...
}
Editable.Edit = Edit;

// También podemos exportar componentes individuales para permitir l\
a modificación
export default Editable;
```

Puedes utilizar una aproximación similar para definir otros componentes más genéricos. Considera algo como `Form`, puedes fácilmente tener `Form.Label`, `Form.Input`, `Form.Textarea`, etcétera. Cada uno contendrá un formato concreto y la lógica que necesite. Es una forma de hacer que tus diseños sean más flexibles.

6.8 Conclusión

Nos ha llevado algunos pasos, pero ya podemos editar notas. Lo mejor de todo es que `Editable` debería ser útil en cualquier lugar donde necesitemos editar alguna propiedad. Podríamos haber extraído la lógica más adelante si hubiésemos visto duplicidad, pero ésta también es una forma de hacerlo.

Aunque la aplicación hace lo que se espera de ella todavía es bastante fea. Haremos algo al respecto en el próximo capítulo a medida que le vayamos añadiendo estilos básicos.

7. Dando Estilo a la Aplicación de Notas

Estéticamente hablando, nuestra aplicación se encuentra en un estado bastante precario. Algo tendremos que hacer, ya que las aplicaciónes más divertidas de utilizar son aquellas que son visualmente más atractivas. En nuestro caso vamos a utilizar un estilo a la vieja usanza.

Para ello esparciremos algunas clases CSS y aplicaremos estilo en base a los selectores. El capítulo *Dando estilo a React* discute otras aproximaciones con mayor detalle.

7.1 Aplicando Estilo sobre el Botón "Añadir Nota"

Para dar estilo al botón "Añadir Nota" primero tenemos que asignarle una clase:

app/components/App.jsx

```
...

export default class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;

    return (
      <div>
        <button onClick={this.addNote}>+</button>
```

```
        <button className="add-note" onClick={this.addNote}>+</butto\
n>
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
          />
      </div>
    );
  }
  ...
}
```

También necesitamos añadir el estilo correspondiente:

app/main.css

```
...

.add-note {
  background-color: #fdfdfd;

  border: 1px solid #ccc;
}
```

Una forma más general de gestionar esto podría ser crear un nuevo componente `Botón` y darle estilo. Esto nos permitirá tener botones con estilo en toda la aplicación.

7.2 Aplicando estilos sobre `Notas`

Actualmente la lista de `Notas` está en crudo. Podemos mejorarla ocultando los estilos específicos de las listas. También podemos ajustar el ancho de `Notas` para que la interfaz del usuario aguante bien si un usuario introduce una tarea larga. Un buen primer paso es incluir algunas clases en `Notas` que la hagan más fácil de estilizar:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul>{notes.map(({id, editing, task}) =>
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note onClick={onNoteClick.bind(null, id)}>
      <Note className="note" onClick={onNoteClick.bind(null, id)}>
        <Editable
          className="editable"
          editing={editing}
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button onClick={onDelete.bind(null, id)}>x</button>
        <button
          className="delete"
          onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Para eliminar los estilos específicos de las listas podemos aplicar las reglas siguientes:

app/main.css

```
...

.notes {
  margin: 0.5em;
  padding-left: 0;

  max-width: 10em;
  list-style: none;
}
```

7.3 Aplicando Estilos sobre Notas Individuales

Todavía quedan cosas relacionadas con `Notas` pendientes de estilizar. Antes de insertar reglas debemos asegurarnos de que tenemos unos buenos puntos de enganche para ello en `Editable`:

app/components/Editable.jsx

```
import React from 'react';
import classnames from 'classnames';

export default ({editing, value, onEdit, ...props}) => {
export default ({editing, value, onEdit, className, ...props}) => {
  if(editing) {
    return <Edit value={value} onEdit={onEdit} {...props} />;
    return <Edit
      className={className}
      value={value}
      onEdit={onEdit}
      {...props} />;
  }

  return <span {...props}>{value}</span>;
  return <span className={classnames('value', className)} {...props}>
    {value}
```

```
  </span>;
}

class Edit extends React.Component {
  render() {
    const {value, onEdit, ...props} = this.props;
    const {className, value, onEdit, ...props} = this.props;

    return <input
      type="text"
      className={classnames('edit', className)}
      autoFocus={true}
      defaultValue={value}
      onBlur={this.finishEdit}
      onKeyPress={this.checkEnter}
      {...props} />;
  }
  ...
}
```

Puede que `className` sea difícil de manejar ya que sólo acepta un string y quizá queramos insertarle más de una clase. En este punto puede ser útil un paquete conocido como classnames[1]. Este paquete acepta muchos tipos de entradas y los convierte a un único string para resolver el problema.

Hay suficientes clases para diseñar el resto ahora. Podemos mostrar una sombra debajo de la nota si ponemos el ratón por encima. También es un buen momento para mostrar el control de borrado al mover el cursor por encima. Por desgracia estos estilos no se mostrarán en interfaces táctiles, pero son lo suficientemente buenos para esta demo:

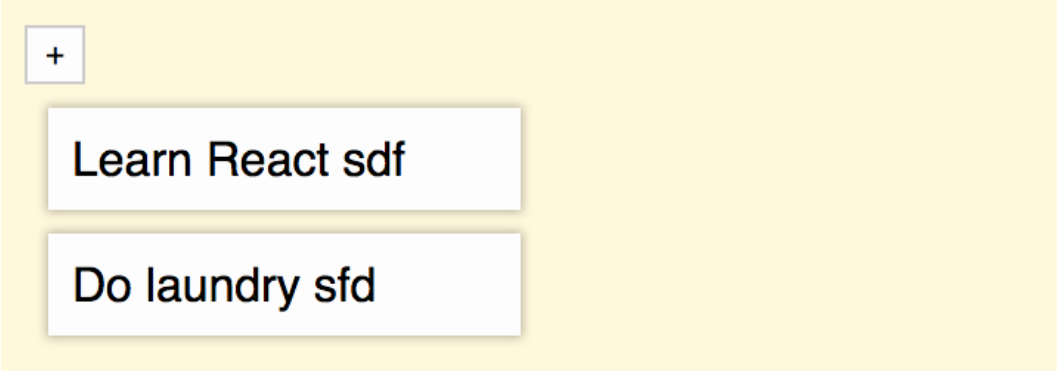
app/main.css

[1] https://www.npmjs.org/package/classnames

```
...

.note {
  overflow: auto;

  margin-bottom: 0.5em;
  padding: 0.5em;

  background-color: #fdfdfd;
  box-shadow: 0 0 0.3em .03em rgba(0,0,0,.3);
}
.note:hover {
  box-shadow: 0 0 0.3em .03em rgba(0,0,0,.7);

  transition: .6s;
}

.note .value {
  /* force to use inline-block so that it gets minimum height */
  display: inline-block;
}

.note .editable {
  float: left;
}
.note .delete {
  float: right;

  padding: 0;

  background-color: #fdfdfd;
  border: none;

  cursor: pointer;
```

```
  visibility: hidden;
}
.note:hover .delete {
  visibility: visible;
}
```

Si todo ha ido bien tu aplicación debería tener el siguiente aspecto:



La aplicación de Notas con estilo

7.4 Conclusión

Esta no es más que una forma de aplicar estilos sobre una aplicación de React. Delegar en clases como hemos hecho hasta ahora puede acarrear problemas a medida que la aplicación crezca. Éste es el motivo por el cual hay alternativas con las que poder aplicar estilos a la vez que se resuelve este problema en particular. El capítulo *Dando estilo a React* muestra muchas de esas técnicas.

Puede ser una buena idea probar un par de alternativas con el objetivo de encontrar alguna con la que te encuentes cómodo. Particularmente creo que los **Módulos CSS** prometen ser capaces de resolver el problema fundamental de CSS - el problema de que el ámbito de todo es global. Esta técnica te permite aplicar estilos para cada componente de forma local.

Ahora que tenemos una aplicación de Notas sencilla funcionando podemos comenzar a hacer un Kanban completo. Requerirá de un poco de paciencia ya que necesitaremos

mejorar la forma en la que estamos gestionando el estado de la aplicación. También necesitaremos añadir algo de estructura que nos falta y estar seguros de que podremos arrastrar y soltar notas por aquí y por allá. Todos ellos son objetivos jugosos para la siguiente parte del libro.

II Implementando Kanban

En este apartado convertiremos nuestra aplicación de notas en una aplicación Kanban. Durante el proceso aprenderás lo más básico de React. Dado que React es sólo una librería de vistas debatiremos sobre otras tecnologías relacionadas. Configuraremos una solución con la que gestionar los estados dentro de nuestra aplicación y también veremos cómo usar React DnD para incluir la funcionalidad de arrastrar y soltar en el tablero Kanban.

8. React y Flux

Puedes llegar bastante lejos guardándolo todo en componentes, lo cual es una forma completamente válida de comenzar. El problema comenzará cuando añadas estado a tu aplicación y necesites compartirlo en distintos sitios. Por este motivo han sido varios los gestores de estado que han aparecido. Cada uno de ellos trata de resolver el problema a su manera.

La arquitectura Flux de aplicaciones[1] fué la primera solución al problema. Te permite modelar tu aplicación mediante el uso de **Acciones**, **Almacenes** y **Vistas**. También tiene una parte conocida como **Dispatcher** con la que gestionar acciones y permitirte modelar dependencias entre las distintas llamadas.

Esta forma de oganización es particulamente útil cuando estas trabajando en equipos grandes. El flujo unidireccional hace fácil poder saber qué está pasando. Este es un elemento común de varias soluciones de gestión de estados disponibles en React.

8.1 Breve Introducción a Redux

Existe una solución llamada Redux[2] que rescata las ideas principales de Flux y las moldea para tener una forma concreta. Redux es más que una simple guía que sirve para que le dés a tu aplicación cierta estructura ya que te empuja a modelar todo lo relacionado con la gestión de los datos de una manera concreta. Mantendrás el estado de tu aplicación en una estructura de árbol que modificarás usando funciones puras (las cuales no tienen efectos secundarios) mediante reductores.

Puede parecer un poco complicado pero, en la práctica, Redux hace que el flujo de tus datos sea explícito. Flux estándar no es dogmático en algunas cosas. Creo que entender Flux de forma básica antes de produndizar en Redux es una buena idea para ver cosas que ambos tienen en común.

[1] https://facebook.github.io/flux/docs/overview.html
[2] http://redux.js.org/

8.2 Breve Introducción a MobX

MobX[3] tiene un punto de vista totalmente diferente sobre la gestión de datos. Si Redux te ayuda a modelar tu flujo de datos de manera explícita, MobX hace el esfuerzo de que sea implícita. No te obliga a seguir determinada estructura. En su lugar, tendrás que anotar tus estructuras de datos como **observable** y dejar a MobX gestionar cuándo se actualizan tus vistas.

Mientras que Redux adopta el concepto de inmutabilidad a través de la idea de los reductores, MobX hace justo lo contrario y apoya la mutación. Esto implica que ciertos asuntos como la gestión de referencias pueden ser extraordinariamente sencillos en MobX mientras que en Redux te verás forzado a normalizar tus datos para que puedas manipularlos fácilmente con reductores.

Tanto Redux como MobX son valiosos a su manera. No hay una solución correcta cuando de la gestión de datos se trata. Estoy seguro de que aparecerán más alternativas a medida que pase el tiempo. Cada solución tiene sus propias ventajas e inconvenientes. Entendiendo las alternativas tendrás una mejor capacidad de seleccionar la solución que mejor encaje con lo que necesites llegado el momento.

8.3 ¿Qué Sistema de Gestión de Estados Debería Utilizar?

El mapa de gestores de estados está cambiando constantemente. En la actualidad Redux[4] es muy popular, pero hay buenas alternativas a la vista. voronianski/flux-comparison[5] muestra una comparativa entre algunos de los más populares.

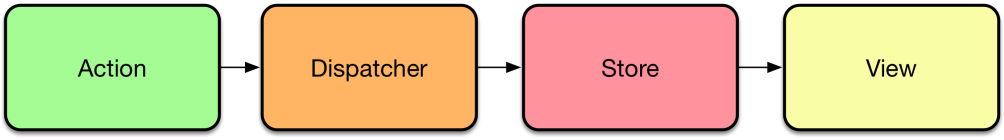
La elección de una librería está condicionada por tus propias preferencias personales. Tienes que tener en cuenta factores como API, funcionalidades, documentación y soporte. Comenzar con una de las alternativas más populares puede ser una buena idea. Podrás hacer elecciones que se ajusten más a lo que quieres a medida que vayas conociendo mejor la arquitectura.

[3] https://mobxjs.github.io/mobx/
[4] http://redux.js.org
[5] https://github.com/voronianski/flux-comparison

Para esta aplicación vamos a utilizar una implementación de Flux conocida con el nombre de Alt[6]. Su API está bien diseñado y es suficiente para nuestro propósito. Como extra, Alt ha sido diseñado teniendo en mente el renderizado isomórfico (renderiza de igual manera tanto en servidor como en cliente). Si conoces Flux tendrás un buen punto de partida con el que comprender mejor las alternativas.

El libro no cubre todas las soluciones alternativas en detalle todavía, pero diseñaremos la aplicación de tal forma que podamos utilizar alternativas más adelante. La idea es que podamos aislar la vista de la gestión de datos para que podamos intercambiar esta gestión sin tener que cambiar código de React. Es una forma de diseñar pensando en el cambio.

8.4 Introducción a Flux



Flujo de Datos Unidirectional de Flux

De momento sólo hemos trabajado con vistas. La arquitectura Flux introduce un par de conceptos nuevos, los cuales son acciones, dispatchers y almacenes. Flux implementa un flujo unidireccional, al contrario que otros frameworks populares como Angular o Ember. Aunque los flujos bidireccionales puedan ser convenientes, éstos tienen un coste. Puede ser difícil saber qué está pasando y por qué.

Acciones y Almacenes

Flux no es totalmente trivial ya que hay algunos conceptos que tener en cuenta. En nuestro caso modelaremos `NoteActions` y `NoteStore`. `NoteActions` facilita operaciones concretas que podremos realizar sobre nuestros datos. Por ejemplo, podremos tener `NoteActions.create({task: 'Aprender React'})`.

[6] http://alt.js.org/

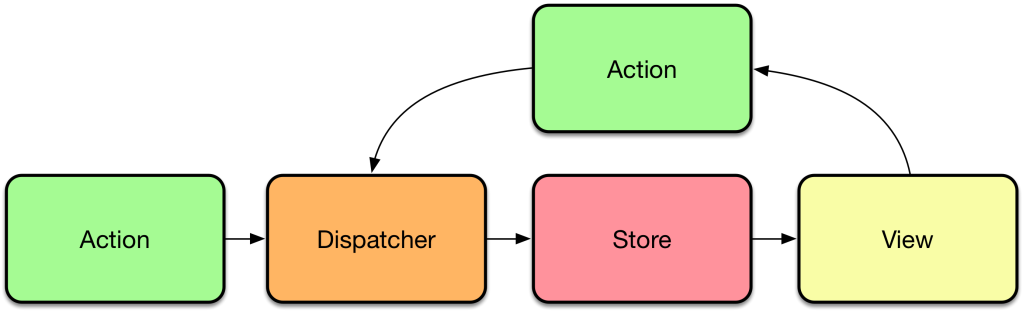
Dispatcher

El dispatcher percibirá que hemos ejecutado una acción. No sólo eso, sino que el dispatcher será capaz de lidiar con las posibles dependencias que existan entre almacenes. Es probable que cierta acción necesite ser ejecutada después de otra, el dispacher nos permitirá lograr todo ello.

El almacén detectará que el dispacher ha procesado una acción y se invocará. En nuestro caso se notificará a `NoteStore`. Como resultado, será capaz de actualizar su estado interno y, tras hacer esto, notificará del nuevo estado.

El Flujo de Datos de Flux

Esto completa el flujo de flujo unidireccional, aunque lineal, básico de Flux. Por regla general el proceso unidireccional tiene un flujo cíclico que no necesariamente termina. El siguiente diagrama ilustra un flujo más común. Es la misma idea de nuevo, pero incluye un ciclo de retorno. Para terminar, los componentes se actualizan a través de este proceso de bucle dependiendo de los datos de nuestro almacén.



Flujo de Datos Cíclico de Flux

Parece que se dan muchos pasos para conseguir algo tan simple como crear una nueva `Nota`. Esta aproximación, sin embargo, conlleva sus propios beneficios. Es muy fácil de trazar y de depurar puesto que el flujo siempre actúa en una única dirección. Si algo está mal, está en algún lugar del ciclo.

Mejor todavía: podemos ver los datos que recorren nuestra aplicación. Tan sólo conecta tu vista a tu almacén y ya lo tienes. Ésta es uno de las mayores ventajas

de utilizar una solución de gestión de estados.

Ventajas de Flux

Aunque todo esto suene complicado, esta forma de trabajar dará flexibilidad a tu aplicación. Podemos, por ejemplo, implementar comunicación con un API, cachés, e internacionalización fuera de nuestras vistas. De esta forma se mantienen lejos de la lógica a la vez que las aplicaciones siguen siendo fáciles de entender.

Implementar una arquitectura Flux en tu aplicación incrementará la cantidad de código de alguna manera. Es importante comprender que la meta de Flux no es minimizar la cantidad de código escrito. Ha sido diseñado para que haya productividad en equipos grandes. Siempre se puede decir que explícito es mejor que implícito.

8.5 Migrando a Alt

Alt

En Alt trabajarás con acciones y almacenes. El dispatcher está oculto, pero puedes acceder a él si lo necesitas. Comparado con otras implementaciones, Alt oculta mucho código reutilizable. Tiene algunas características especiales que te permitirán guardar y recuperar el estado de la aplicación, lo cual es útil para implementar tanto persistencia como renderizado universal.

Hay un par de pasos que debemos seguir para permitir que Alt gestione el estado de nuestra aplicación:

1. Configurar una instancia de Alt para que siga las acciones, los almacenes y coordine la comunicación.

2. Conectar Alt con las vistas.
3. Dejar nuestros datos en un almacén.
4. Definir acciones que permitan manipular el almacén.

Lo iremos haciendo paso a paso. Las partes específicas de Alt van después de los adaptadores. El enfoque de adaptadores nos permite cambiar fácilmente de idea más adelante así que vale la pena implementarlos.

Configurando una Instancia de Alt

Todo en Alt comienza desde una instancia de Alt. Hace un seguimiento de las acciones y los almacenes y permite que la comunicación fluya. Para hacer que todo sea sencillo trataremos a todos los componentes de Alt como si fueran singleton[7]. Gracias a este patrón podremos reutilizar la misma instancia dentro de nuestra aplicación.

Para conseguirlo podemos dejarlo en un módulo y referenciarlo desde cualquier sitio. Configúralo tal y como sigue:

app/libs/alt.js

```
import Alt from 'alt';

const alt = new Alt();

export default alt;
```

Esta es la forma estándar de implementar *singletons* usando la sintaxis de ES6. Cachea el módulo de tal forma que te devolverá la misma instancia la próxima vez que importes Alt desde donde sea.

Observa que `alt.js` debe ir bajo `app/libs`, ¡no en el directorio `libs` del raíz!

[7] https://es.wikipedia.org/wiki/Singleton

El patrón singleton garantiza que habrá una y sólo una instancia. Este es precisamente el comportamiento que queremos ahora.

Uniendo Alt con las Vistas

Por normal general los gestores de estados facilitan dos cosas que pueden ser usadas para conectar con una aplicación React. Esto son el `Proveedor` y una función de alto nivel `conectar` (una función que devuelve una función que genera un componente). El `Proveedor` configura un contexto de react[8].

Los contextos son una característica avanzada que puede ser utilizada para enviar datos de forma implícita a través de la jerarquía de componentes sin utilizar props. La función `conectar` utiliza el contexto para cavar un hueco en el que enviar los datos al componente.

Es posible utilizar `conectar` a través de la invocación de funciones o de un decorador como veremos pronto. El apéndice *Entendiendo los Decoradores* entra más en profundidad en este patrón.

Para permitir que la arquitectura de nuestra aplicación sea sencilla de modificar necesitaremos configurar dos adaptadores, uno para el `Proveedor` y otro para `conectar`. Nos enfrentaremos con los detalles específicos de Alt en ambos sitios.

Configurando un `Proveedor`

Voy a utilizar una configuración especial que nos permitirá que nuestro `Proveedor` sea flexible. Lo envolveremos en un módulo que eligirá un `Proveedor` u otro dependiendo de nuestro entorno. Esto nos permitirá usar herramientas de desarrollo sin incluirlas en el pack de producción. Es necesario hacer algo de configuración extra pero merece la pena puesto que así tendremos un resultado más limpio.

El punto de partida de esto está en el index del módulo. CommonJS escoje por defecto el fichero **index.js** del directorio cuando hacemos un import de ese directorio. No podemos dejarlo en mano de los módulos de ES6 dado que queremos un comportamiento dinámico.

[8] https://facebook.github.io/react/docs/context.html

La idea es que nuestro componente reescriba el código dependiendo de la variable `process.env.NODE_ENV`, la cual servirá para seleccionar el módulo que queramos incluir. Aquí tenemos el punto de entrada de nuestro `Proveedor`:

app/components/Provider/index.js

```
if(process.env.NODE_ENV === 'production') {
  module.exports = require('./Provider.prod');
}
else {
  module.exports = require('./Provider.dev');
}
```

También necesitaremos los ficheros a los cuales está apuntando el fichero index. La primera parte es sencilla. Aquí necesitamos apuntar a nuestra instancia de Alt, conectarlo con un componente conocido como `AltContainer` y renderizar nuestra aplicación con él. Es aquí donde `props.children` entran en juego. Es la misma idea de antes.

`AltContainer` nos permitirá conectar los datos de nuestra aplicación a nivel de componente cuando implementemos `conectar`. Para ello, aquí tienes la implementación a nivel de producción:

app/components/Provider/Provider.prod.jsx

```
import React from 'react';
import AltContainer from 'alt-container';
import alt from '../../libs/alt';
import setup from './setup';

setup(alt);

export default ({children}) =>
  <AltContainer flux={alt}>
    {children}
  </AltContainer>
```

La implementación de `Proveedor` puede cambiar dependiendo del gestor de estados que estemos utilizando. Es posible que finalmente no haga nada, lo cual es aceptable. La idea es que tengamos un punto de extensión donde poder modificar nuestra aplicación si lo necesitamos.

Todavía nos estamos dejando una parte, la relacionada con la configuración de desarrollo. Será como la de producción con la excepción de que esta vez podremos habilitar herramientas específicas de desarrollo. Es una buena oportunidad de mover la configuración de *react-addons-perf* aquí desde el *app/index.jsx* de la aplicación. También estoy habilitando las herramientas de debug de Chrome para Alt[9]. Tendrás que instalar Chrome si quieres utilizarlas.

Aquí tienes el código completo del proveedor de desarrollo:

app/components/Provider/Provider.dev.jsx

```
import React from 'react';
import AltContainer from 'alt-container';
import chromeDebug from 'alt-utils/lib/chromeDebug';
import alt from '../../libs/alt';
import setup from './setup';

setup(alt);

chromeDebug(alt);

React.Perf = require('react-addons-perf');

export default ({children}) =>
  <AltContainer flux={alt}>
    {children}
  </AltContainer>
```

El módulo `setup` permite hacer toda la configuración relacionada con Alt que sea común tanto para el entorno de producción como para el entorno de desarrollo. De momento con que no haga nada es suficiente:

[9] https://github.com/goatslacker/alt-devtool

app/components/Provider/setup.js

```
export default alt => {}
```

Todavía necesitamos conectar el `Proveedor` con nuestra aplicación modificando *app/index.jsx*. Haz los siguientes cambios:

app/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';
import Provider from './components/Provider';

if(process.env.NODE_ENV !== 'production') {
  React.Perf = require('react-addons-perf');
}

ReactDOM.render(
  <App />,
  <Provider><App /></Provider>,
  document.getElementById('app')
);
```

Si miras la salida de Webpack verás que hay nuevas dependencias que se han instalado en el proyecto. Es lo que podemos esperar dados los cambios. El proceso puede tardar un rato en terminar, una vez que lo haga, refresca el navegador.

Dado que no hemos cambiado la lógica de la aplicación de ninguna manera, todo debería tener el mismo aspecto de antes. Un buen paso con el que continuar es implementar un adaptador que conecte los datos con nuestras vistas.

Puedes encontrar una idea similar en react-redux[10]. MobX no necesita un `Proveedor` para nada. En este caso nuestra implementación únicamente devolverá `children`.

[10] https://www.npmjs.com/package/react-redux

8.6 Entendiendo `conectar`

La idea de `conectar` es la de permitirnos incrustar datos y acciones concretas a componentes. Es así como podemos conectar los datos de los carriles y las acciones con `App`:

```
@connect(({lanes}) => ({lanes}), {
  laneActions: LaneActions
})
export default class App extends React.Component {
  render() {
    return (
      <div>
        <button className="add-lane" onClick={this.addLane}>+</butto\
n>
        <Lanes lanes={this.props.lanes} />
      </div>
    );
  }
  addLane = () => {
    this.props.laneActions.create({name: 'New lane'});
  }
}
```

Se puede escribir lo mismo sin decoradores. Esta es la sintáxis que utilizaremos en nuestra aplicación:

```
class App extends React.Component {
  ...
}

export default connect(({lanes}) => ({lanes}), {
  LaneActions
})(App)
```

En caso de que necesites aplicar varias funciones de alto nivel contra un componente, puedes utilizar una utilidad como `compose` y usarla con `compose(a, b)(App)`. Esto será igual a `a(b(App))` y se puede leer mejor.

En los ejemplos mostrados `compose` es una función que devuelve una función. Por ello lo llamamos Función de Alto Nivel. Al final obtendremos un componente de ella. Este envoltorio nos permite manejar todo lo relacionado con la conexión con los datos.

Podemos utilizar una función de alto nivel para anotar nuestros componentes y darles además otras propiedades especiales. Veremos esta idea cuando implementemos la funcionalidad de arrastrar y soltar. Los decoradores brindan una forma sencilla de incluir estos tipos de anotaciones. El apéndice *Entendiendo los Decoradores* profundiza más en este asunto.

Ahora que entendemos básicamente cómo debería funcionar `conectar` podemos implementarlo.

Configurando `conectar`

Voy a utilizar un `conectar` personalizado para remarcar un par de ideas clave. La implementación no es óptima en terminos de rendimiento pero será suficiente para esta aplicación.

Es posible optimizar el rendimiento con un trabajo posterior. Puedes utilizar uno de los conectores comunes en lugar de desarrollar el tuyo propio. He aquí una razón por la cual tener el control de `Proveedor` y `conectar` es útil, permite personalizar más adelante y entender cómo funciona el proceso.

app/libs/connect.jsx

```
import React from 'react';

export default (state, actions) => {
  if(typeof state === 'function' ||
    (typeof state === 'object' && Object.keys(state).length)) {
    return target => connect(state, actions, target);
  }

  return target => props => (
    <target {...Object.assign({}, props, actions)} />
  );
}

// Conectar con Alt a través del contexto. Esto no ha sido optimizado
// para nada. Si Alt almacena los cambios se forzará el renderizado.
//
// Ver *AltContainer* y *connect-alt* para encontrar soluciones ópti\
mas
function connect(state = () => {}, actions = {}, target) {
  class Connect extends React.Component {
    componentDidMount() {
      const {flux} = this.context;

      flux.FinalStore.listen(this.handleChange);
    }
    componentWillUnmount() {
      const {flux} = this.context;

      flux.FinalStore.unlisten(this.handleChange);
    }
    render() {
      const {flux} = this.context;
      const stores = flux.stores;
      const composedStores = composeStores(stores);
```

```
      return React.createElement(target,
        {...Object.assign(
          {}, this.props, state(composedStores), actions
        )}
      );
    }
    handleChange = () => {
      this.forceUpdate();
    }
  }
  Connect.contextTypes = {
    flux: React.PropTypes.object.isRequired
  }

  return Connect;
}

// Convierte {store: <AltStore>} en {<store>: store.getState()}
function composeStores(stores) {
  let ret = {};

  Object.keys(stores).forEach(k => {
    const store = stores[k];

    // Combina el estado del almacén
    ret = Object.assign({}, ret, store.getState());
  });

  return ret;
}
```

Dado que `flux.FinalStore` no está disponible por defecto, necesitamos cambiar nuestra instacia de Alt para que la contenga. Tras ello podremos acceder a ella donde la necesitemos:

app/libs/alt.js

```
import Alt from 'alt';
import makeFinalStore from 'alt-utils/lib/makeFinalStore';

const alt = new Alt();

export default alt;
class Flux extends Alt {
  constructor(config) {
    super(config);

    this.FinalStore = makeFinalStore(this);
  }
}

const flux = new Flux();

export default flux;
```

Podemos incrustar algunos datos de prueba en `App` y renderizarlos para ver `conectar` en acción. Haz los cambios siguientes para enviar datos a `App` y, después, mira cómo se muestran en la interfaz de usuario:

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';
import connect from '../libs/connect';

export default class App extends React.Component {
class App extends React.Component {
  constructor(props) {
    ...
  }
  render() {
    const {notes} = this.state;
```

```
    return (
      <div>
        {this.props.test}
        <button className="add-note" onClick={this.addNote}>+</butto\
n>
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
          />
      </div>
    );
  }
  ...
}

export default connect(() => ({
  test: 'test'
}))(App)
```

Refresca el navegador para mostrar el texto. Deberías poder ver ahora el texto que hemos conectado con `App`.

8.7 Usando el Dispatcher en Alt

Aunque hayamos llegado lejos sin utilizar el dispatcher de Flux, puede ser útil que sepamos algo sobre ello. Alt facilita dos formas de utilizarlo. Si quieres guardar una traza de todo lo que pase por la instancia de `alt` puedes utilizar un trozo de código como `alt.dispatcher.register(console.log.bind(console))`. También puedes lanzar `this.dispatcher.register(...)` en un constructor del almacén. Estos mecanismos te permitirán generar trazas de forma efectiva.

Otros gestores de estado ofrecen puntos de enganche similares. Es posible interceptar el flujo de datos de muchas formas e incluso crear una lógica personalizada encima de ello.

8.8 Conclusión

En este capítulo hemos debatido la idea básica de qué es la arquitectura Flux y hemos comenzado a migrar nuestra aplicación a ella. Hemos dejado todo lo relacionado con la gestión del estado tras un adaptador para poder modificar el código sin tener que cambiar nada relacionado con la vista. El paso siquiente será definir un almacén para nuestra aplicación y definir las acciones que lo puedan manipular.

9. Implementando `NoteStore` y `NoteActions`

Ahora que hemos movido todo lo relacionado con la gestión de los datos al lugar correcto podemos centrarnos en implementar las partes que faltan - `NoteStore` (Almacén de Notas) y `NoteActions` (Acciones sobre las Notas). Ambas encapsularán tanto los datos de la aplicación como la lógica.

No importa qué gestor de estados acabes usando, siempre encontrarás equivalencias en los demás. En Redux puedes usar acciones que provocarán un cambio de estado mediante un reductor. En MobX puedes modelar una acción en una clase ES6. La idea es que manipules los datos dentro de la clase y que ésto provoque que MobX refresque los componentes cuando sea necesario.

La idea aquí es similar: configuraremos acciones que acabarán invocando métodos en el estado que modificarán este estado. Cuando el estado cambia las vistas se actualizan. Para comenzar podemos implementar un `NoteStore` y definir la lógica para manipularlo. Una vez hayamos hecho eso, habremos migrado nuestra aplicación a la arquitectura Flux.

9.1 Configurando un `NoteStore`

De momento mantenemos el estado de la aplicación en `App`. El primer paso para llevarlo a Alt es definir un almacén y utilizar el estado desde allí. Esto romperá con la lógica de nuestra aplicación de forma temporal ya que necesitamos llevar el estado también a Alt. Sin embargo, crear este almacén inicial es un buen paso para cumplir con nuestro objetivo.

Para configurar un almacén necesitamos llevar a cabo tres pasos. Necesitaremos configurarlo, conectarlo con Alt en el `Proveedor` y, finalmente, conectarlo con `App`.

Los almacenes se modelan en Alt usando clases ES6. Aqui tienes una implementación mínima modelada con nuestro estado actual.

app/stores/NoteStore.js

```
import uuid from 'uuid';

export default class NoteStore {
  constructor() {
    this.notes = [
      {
        id: uuid.v4(),
        task: 'Apender React'
      },
      {
        id: uuid.v4(),
        task: 'Hacer la Colada'
      }
    ];
  }
}
```

El siguiente paso es conectar el almacén con el `Proveedor`. Es aquí donde el módulo `setup` se vuelve útil:

app/components/Provider/setup.js

```
~~export default alt => {}~~
import NoteStore from '../../stores/NoteStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);
}
```

Podemos ajustar `App` para consumir los datos desde el almacén y así comprobar que lo que hemos hecho funciona. Esto romperá la lógica que tenemos, pero lo arreglaremos en la próxima sección. Cambia `App` como sigue para hacer que `notas` esté disponible:

app/components/App.jsx

```
...

class App extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      notes: [
        {
          id: uuid.v4(),
          task: 'Aprender React'
        },
        {
          id: uuid.v4(),
          task: 'Hacer la Colada'
        }
      ]
    }
  }
  render() {
    const {notes} = this.state;
    const {notes} = this.props;

    return (
      <div>
        {this.props.test}
        <button className="add-note" onClick={this.addNote}>+</butto\
n>
        <Notes
          notes={notes}
          onNoteClick={this.activateNoteEdit}
          onEdit={this.editNote}
          onDelete={this.deleteNote}
          />
      </div>
```

```
    );
  }
  ...
}

export default connect(() => ({
  test: 'test'
}))(App)
export default connect(({notes}) => ({
  notes
}))(App)
```

Si refrescas la aplicación verás exactamente lo mismo que antes. Esta vez, sin embargo, estaremos consumiendo los datos desde nuestro almacén. Como resultado nuestra lógica está rota. Esto es algo que tendremos que arreglar más adelante mientras definimos `NoteActions` y llevamos la manipulación del estado a `NoteStore`.

Dado que `App` no dependerá más del estado, es posible convertirlo a un componente basado en funciones. A menudo la mayoría de tus componentes estarán basados en funciones precisamente por esta razón. Si no estás utilizando estado o referencias es seguro convertirlos a funciones.

9.2 Entendiendo las Acciones

Las acciones son uno de los conceptos principales de la arquitectura Flux. Para ser exactos, es una buena idea separar **acciones** de **creadores de acciones**. A menudo estos términos son intercambiables, pero hay una diferencia considerable.

Los creadores de acciones son literalmente funciones que *lanzan* acciones. El contenido de la acción será repartido a los almacenes que estén interesados. Puede ser útil pensar en ellos como mensajes dentro de un envoltorio que son repartidos.

Esta división es útil si quieres hacer acciones asíncronas. Puedes, por ejemplo, querer recuperar los datos iniciales de tu tablero Kanban. La operación puede ir bien o ir

mal, lo cual te dará tres acciones distintas que lanzar. Puedes lanzar acciones cuando comienzas la consulta y cuando recibes una respuesta.

Todos estos datos son valiosos si te permiten controlar la interfaz del usuario. Puedes mostrar una barra de progreso mientras la consulta se está realizando y actualizar el estado de la aplicación una vez llegan los datos del servidor. Si la consulta falla puedes hacer que el usuario lo sepa.

Este asunto es igual en otros gestores de estados. A menudo modelas una acción como una función que devuelve una función que lanza acciones individuales como puede ser el seguimiento del progreso de las consultas. En un ingenuamente síncrono caso es suficiente con devolver directamente el resultado de la acción.

La documentación oficial de Alt cubre las acciones asíncronas[1] con más detalle.

9.3 Configurando `NoteActions`

Alt tiene un pequeño método de utilidades conocido como `alt.generateActions` que puede generar creadores de acciones simples por nosotros. Estos generadores símplemente enviarán los datos que les pasemos, así que conectaremos estas acciones con los almacenes relevantes. En este caso, estamos hablando del `NoteStore` que definimos anteriormente.

Con respecto a la aplicación, es suficiente con que modelemos las operaciones CRUD básicas (Crear, Leer, Actualizar y Borrar). Podemos saltarnos la lectura ya que es implícita, pero es útil tener las demás disponibles como acciones. Configura `NoteActions` usando `alt.generateActions` como sigue:

app/actions/NoteActions.js

[1] http://alt.js.org/docs/createActions/

```
import alt from '../libs/alt';

export default alt.generateActions('create', 'update', 'delete');
```

Esto no hace mucho por sí mismo, aunque es un buen sitio para conectar las acciones con App para poder lanzarlas. Nos empezaremos a preocupar sobre las acciones individuales una vez hagamos que nuestro almacén sea más grande. Modifica App del siguiente modo para conectar las acciones:

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import Notes from './Notes';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';

class App extends React.Component {
  ...
}

~~export default connect(({notes}) => ({~~
~~  notes~~
~~}))(App)~~
export default connect(({notes}) => ({
  notes
}), {
  NoteActions
})(App)
```

Esto nos permitirá ejecutar cosas como this.props.NoteActions.create para poder lanzar acciones.

9.4 Conectando NoteActions con NoteStore

Alt facilita un par de formas útiles con las que conectar acciones con almacenes:

- `this.bindAction(NoteActions.CREATE, this.create)` - Enlaza una acción específica con un método específico.
- `this.bindActions(NoteActions)`- Enlaza todas las acciones con métodos por convención. Es decir, la acción `create` se enlazará con un método llamado `create`.
- `reduce(state, { action, data })` - Es posible implementar un método conocido como `reductor`, el cual imita la forma de trabajar de los reductores de Redux. La idea es devolver un nuevo estado basado en el estado actual y unos datos.

Utilizaremos `this.bindActions` en caso de que confiar en la convención sea suficiente. Modifica el almacén como sigue para conectar las acciones y añadir datos iniciales a la lógica:

app/stores/NoteStore.js

```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    this.bindActions(NoteActions);

    this.notes = [
      {
        id: uuid.v4(),
        task: 'Learn React'
      },
      {
        id: uuid.v4(),
        task: 'Do laundry'
      }
    ];
  }
  create(note) {
    console.log('create note', note);
```

```
  }
  update(updatedNote) {
    console.log('update note', updatedNote);
  }
  delete(id) {
    console.log('delete note', id);
  }
}
```

Para poder verlo en funcionamiento necesitamos conectar nuestras acciones con App y adaptar la lógica.

9.5 Migrando App.addNote a Flux

App.addNote es un buen punto en el que comenzar. El primer paso es lanzar la acción asociada (NoteActions.create) desde el método y comprobar si podemos ver algo en la consola del navegador. Si podemos, entonces podemos manipular el estado. Lanza una acción con la siguiente:

app/components/App.jsx

```
...

class App extends React.Component {
  render() {
    ...
  }
  addNote = () => {
    // Es posible escribir esto de forma imperativa, es decir,
    // a través de `this.state.notes.push` y, después,
    // `this.setState({notes: this.state.notes})`.
    //
    // Suelo favorecer el estilo funcional cuando tiene sentido.
    // Incluso cuando es necesario escribir más código, ya que
    // prefiero los beneficios (facilidad para razonar, no
```

```
    // efectos colaterales) que trae consigo.
    //
    // Algunas librerias, como Immutable.js, van un paso más allá.
    this.setState({
      notes: this.state.notes.concat([{
        id: uuid.v4(),
        task: 'New task'
      }])
    });
    this.props.NoteActions.create({
      id: uuid.v4(),
      task: 'New task'
    });
  }
  ...
}

...
```

Si refrescas el navegador y pulsas sobre el botón "añadir nota" deberías ver mensajes como el siguiente en la consola del navegador:

```
create note Object {id: "62098959-6289-4894-9bf1-82e983356375", task\
: "New task"}
```

Esto significa que tenemos los datos que necesitamos en el método `create` de `NoteStore`. Aún necesitamos manipular los datos, tras lo cual habremos cerrado el ciclo y deberíamos ver notas nuevas a través de la interfaz de usuario. Aquí Alt tiene un API similar al de React. Considera la siguiente implementación:

app/stores/NoteStore.js

```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    ...
  }
  create(note) {
    console.log('create note', note);
    this.setState({
      notes: this.notes.concat(note)
    });
  }
  ...
}
```

Si intentas añadir una nota, la actualización funcionará. Alt es quien mantiene el estado ahora y la edición se mantiene gracias a la arquitectura que hemos configurado. Todavía tenemos que repetir el proceso para el resto de métodos que faltan para poder completar el trabajo.

9.6 Migrando `App.deleteNote` a Flux

El proceso es exáctamente el mismo para `App.deleteNote`. Necesitamos conectarlo con nuestra acción y adaptar el código. He aquí la parte de `App`:

app/components/App.jsx

```
...

class App extends React.Component {
  ...
  deleteNote = (id, e) => {
    // Avoid bubbling to edit
    e.stopPropagation();

    this.setState({
      notes: this.state.notes.filter(note => note.id !== id)
    });
    this.props.NoteActions.delete(id);
  }
  ...
}

...
```

Si refrescas y tratas de borrar una nota verás un mensaje como el siguiente en la consola del navegador:

```
delete note 501c13e0-40cb-47a3-b69a-b1f2f69c4c55
```

Para finalizar la migración necesitamos mostrar la lógica de `setState` al método `delete`. Recuerda borrar `this.state.notes` y reemplazarlo simplemente por `this.notes`:

app/stores/NoteStore.js

```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  ...
  delete(id) {
    console.log('delete note', id);
    this.setState({
      notes: this.notes.filter(note => note.id !== id)
    });
  }
}
```

Tras este cambio deberías poder borrar notas como antes. Todavía hay un par de métodos que migrar.

9.7 Migrando `App.activateNoteEdit` a Flux

`App.activateNoteEdit` es básicamente una operación de actualización. Necesitamos cambiar el flag `editing` de la nota a `true`, lo cual iniciará el proceso de edición. Como siempre, deberemos migrar `App` primero:

app/components/App.jsx

```
...

class App extends React.Component {
  ...
  activateNoteEdit = (id) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = true;
        }
```

```
        return note;
      })
    });
    this.props.NoteActions.update({id, editing: true});
  }
  ...
}

...
```

Si refrescas y tratas de editar una nota verás un mensaje como el siguiente en la consola del navegador:

```
update note Object {id: "2c91ba0f-12f5-4203-8d60-ea673ee00e03", edit\
ing: true}
```

Todavía necesitamos aplicar el cambio para hacer que esto funcione. La lógica es la misma que la que teniamos anteriormente en App con la excepción de que lo hemos generalizado usando Object.assign:

app/stores/NoteStore.js

```
import uuid from 'uuid';
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  ...
  update(updatedNote) {
    console.log('update note', updatedNote);
    this.setState({
      notes: this.notes.map(note => {
        if(note.id === updatedNote.id) {
          return Object.assign({}, note, updatedNote);
        }
```

```
        return note;
      })
    });
  }
  ...
}
```

Ahora debería ser posible comenzar a editar notas, aunque si terminas de editarlas verás un error como `Uncaught TypeError: Cannot read property 'notes' of null`. Esto se debe a que nos falta la parte final de la migración: cambiar `App.editNote`.

9.8 Migrando `App.editNote` a Flux

Esta parte final es sencilla. Ya tenemos la lógica que necesitamos, es sólo cuestión de conectar `App.editNote` correctamente con ella. Necesitaremos invocar a nuestro método `update` de una forma adecuada:

app/components/App.jsx

```
...

class App extends React.Component {
  ...
  editNote = (id, task) => {
    this.setState({
      notes: this.state.notes.map(note => {
        if(note.id === id) {
          note.editing = false;
          note.task = task;
        }

        return note;
      })
    });
```

```
    this.props.NoteActions.update({id, task, editing: false});
  }
}

...
```

Tras refrescar el navegador deberías ser capaz de modificar tareas de nuevo y la aplicación debería funcionar exactamente igual que antes. Modificar `NoteStore` incluyendo acciones ha provocado una cascada de actualizaciones sobre `App` que han hecho que todo se actualice mediante `setState`, lo que hará que el componente invoque a `render`. Este es el flujo unidireccional de Flux en acción.

Realmente ahora tenemos más código que antes, pero eso no importa. `App` está un poco más limpio y su desarollo es más fácil de continuar como veremos pronto. Lo más importante es que nos hemos apañado para implementar la arquitectura Flux en nuestra aplicación.

Nuestra implementación actual es ingenua en el sentido de que no valida parámetros de ninguna forma. Puede ser una buena idea validar la forma de los objetos para evitar problemas durante el desarrollo. Flow[2] facilita una forma gradual de hacerlo. Aparte, puedes hacer tests que prueben el sistema.

¿Para qué sirve?

Integrar un gestor de estados supone mucho esfuerzo, pero no es en vano. Ten en cuenta las siguiente preguntas:

1. Supón que quieres almacenar las notas en el `localStorage`. ¿Dónde implementarías esta funcionalidad?. Una aproximación puede ser el módulo `setup` del `Proveedor`.
2. ¿Qué ocurre si tenemos varios componentes que quieran utilizar los datos? Podemos consumirlos usando `connect` y mostrarlos.

[2] http://flowtype.org/

3. ¿Qué ocurre si tenemos muchas listas de notas separadas para distintos tipos de tareas?. Podemos crear otro almacén para hacer un seguimiento de esas listas. Ese almacén podrá referenciar las notas por id. Haremos algo parecido en el próximo capítulo.

Adoptar un gestor de estados puede ser útil en el momento en el que tu aplicación React crezca. Esta abstracción tiene el coste de que tienes que escribir más código pero, por otro lado, si lo haces bien, acabarás con algo que será más fácil de razonar y de desarrollar más adelante. Cabe destacar que el flujo unidireccional utilizado por estos sistemas ayudan mucho tanto a la depuración como al testing.

9.9 Conclusión

Hemos visto en este capítulo cómo migrar una aplicación sencilla a una arquitectura Flux. Durante el proceso hemos aprendido más acerca de las **acciones** y los **almacenes** de Flux. Llegados a este punto estamos listos para añadir más funcionalidad a nuestra aplicación. Añadiremos persistencia basada en el `localStorage` a nuestra aplicación y realizaremos una pequeña limpieza por el camino.

10. Implementando Persistencia en `localStorage`

Ahora mismo nuestra aplicación no puede mantener su estado si la página se refresca. Una buena forma de solucionar este problema es almacenar el estado de la aplicación en el localStorage[1] y recuperarlo cuando ejecutemos la aplicación de nuevo.

Esto no es un problema si estamos trabajando contra un backend, aunque incluso en ese caso tener una caché temporal en `localStorage` puede ser útil, únicamente estate seguro de que no almacenas información sensible ya que es fácil acceder a ella.

10.1 Entendiendo `localStorage`

`localStorage` es una parte de Web Storage API. La otra mitad, el `sessionStorage`, funciona sólo cuando el navegador está en funcionamiento mientras `localStorage` persiste incluso más allá. Ambos comparten el mismo API[2] que se muestra a continuación:

- `storage.getItem(k)` - Devuelve la cadena de texto almacenada en la clave enviada como parámetro.
- `storage.removeItem(k)` - Elimina el dato que coincida con la clave.
- `storage.setItem(k, v)` - Guarda el valor recibido en base a la clave indicada.
- `storage.clear()` - Borra el contenido del almacén.

Es conveniente operar con el API utilizando las herramientas de desarrollador del navegador. En Chrome, la pestaña *Recursos* es útil y te permite tanto inspeccionar los datos como realizar operaciones directas contra ellas. Puedes utilizar incluso los

[1] https://developer.mozilla.org/en/docs/Web/API/Window/localStorage

[2] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

atajos storage.key y storage.key = 'value' en la consola para hacer pequeñas pruebas.

localStorage y sessionStorage pueden utilizar hasta un máximo de 10 MB entre las dos, que aunque es algo que debería estar bien soportado por los navegadores, puede fallar.

10.2 Implementando un Envoltorio para localStorage

Para hacer que las cosas sean simples y manejables vamos a implementar un pequeño envoltorio sobre el almacén que nos permita limitar la complejidad. El API consistirá en un método get(k) para recuperar elementos del almacenamiento y set(k,v) para establecerlos. Dado que el API que está por debajo funciona con cadenas de texto, usaremos JSON.parse y JSON.stringify para la serialización. Tendremos que tener en cuenta que JSON.parse puede fallar. Considera la siguiente implementación:

app/libs/storage.js

```
export default storage => ({
  get(k) {
    try {
      return JSON.parse(storage.getItem(k));
    }
    catch(e) {
      return null;
    }
  },
  set(k, v) {
    storage.setItem(k, JSON.stringify(v));
  }
})
```

Esta implementación es suficiente para cumplir nuestros propósitos. No funcionará siempre y fallará si ponemos demasiados datos en el almacén. Para superar estos

problemas sin tener que arreglarlos por tí mismo es posible utilizar un envoltorio como localForage[3] para ocultar la complejidad.

10.3 Persistiendo la Aplicación usando `FinalStore`

El que tengamos los medios para poder escribir en el `localStorage` no es suficiente. Todavía necesitamos poder conectarlo con nuestra aplicación de alguna manera. Los gestores de estados tienen puntos de enganche para este propósito y a menudo encontrarás una forma de interceptarlos de alguna manera. En el caso de Alt, esto ocurre gracias a un almacén predefinido conocido como `FinalStore`.

El caso es que ya lo tenemos configurado en nuestra instancia de Alt. Lo que nos queda es escribir el estado de la aplicación en el `localStorage` cuando éste cambie. También necesitaremos cargar el estado cuando arranquemos la aplicación. Estos procesos en Alt se conocen como **snapshotting** y **bootstrapping**.

Una forma alternativa de gestionar el almacenamiento de los datos es hacer un snapshot sólo cuando se cierre el navegador. Hay una llamada a nivel de ventana llamado `beforeunload` que puede ser utilizado para ello. Sin embargo, esta aproximación es algo frágil. ¿Qué ocurrirá si ocurre algo inesperado y el evento no se llega a lanzar por algún motivo?, que perderás datos.

10.4 Implementando la Lógica de Persistencia

Podemos gestionar la lógica de persistencia en un módulo separado que se dedique a ello.

Puede ser una buena idea implementar un flag de `debug` ya que puede ser útil deshabilitar el snapshotting de forma temporal. La idea es dejar de almacenar datos si el flag está puesto a true.

[3] https://github.com/mozilla/localForage

Esto es especialmente útil para poder eliminar el estado de la aplicación de forma drástica durante el desarrollo y poder dejarlo en un estado en blanco mediante `localStorage.setItem('debug', 'true')` (`localStorage.debug = true`), `localStorage.clear()` y, finalmente, refrescar el navegador.

Dado que el bootstrapping puede fallar por algún motivo desconocido debemos ser capaces de capturar el error. Puede ser una buena idea seguir con el arranque de la aplicación aunque algo horrible haya pasado llegado ese punto.

La siguiente implementación ilustra estas ideas:

app/libs/persist.js

```
export default function(alt, storage, storageName) {
  try {
    alt.bootstrap(storage.get(storageName));
  }
  catch(e) {
    console.error('Failed to bootstrap data', e);
  }

  alt.FinalStore.listen(() => {
    if(!storage.get('debug')) {
      storage.set(storageName, alt.takeSnapshot());
    }
  });
}
```

Puede que acabes con algo similar usando otros gestores de estado. Necesitarás encontrar puntos de enganche similares con los que inicializar el sistema con datos cargados desde el `localStorage` y poder escribir el estado cuando algo haya cambiado.

10.5 Conectando la Lógica de Persistencia con la Aplicación

Todavía nos falta una pieza para hacer que esto funcione. Necesitamos conectar la lógica con nuestra aplicación. Por suerte hay un sitio indicado para ello, la configuración. Déjala como sigue:

app/components/Provider/setup.js

```
import storage from '../../libs/storage';
import persist from '../../libs/persist';
import NoteStore from '../../stores/NoteStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);

  persist(alt, storage(localStorage), 'app');
}
```

Si refrescas el navegador ahora, la aplicación debería mantener su estado. Puesto que esta solución es genérica, añadir más estados al sistema no debería suponer un problema. También podemos integrar un backend que facilite estos puntos de enganche si queremos.

Si tuviésemos un backend real podríamos incluir el resultado en el HTML y devolverlo al navegador, lo que nos ahorraría un viaje. Si además renderizamos el HTML inicial de la aplicación acabaremos implementando una aproximación básica al **renderizado universal**. El renderizado universal es una técnica muy poderosa que permite usar React para mejorar el rendimiento de tu aplicación a la vez que sigue funcionando el SEO.

Nuestra implementación no está falta de fallos. Es fácil llegar a una situación en la que el localStorage contenga datos inválidos debido a cambios que hayamos hecho en el modelo de datos. Esto te acerca al mundo de los esquemas de bases de datos y sus migraciones. Lo que debes aprender aquí es que cuánto más estado tengas en tu aplicación, más complicado se volverá manejarlo.

10.6 Limpiando NoteStore

Antes de continuar es una buena idea limpiar NoteStore. Todavía queda algo de código de experimentos anteriores. Dado que la persistencia ya funciona, puede que queramos arrancar desde un estado en blanco. Incluso si queremos tener datos iniciales, puede que sea mejor gestionarlos a alto nivel, por ejemplo al arrancar la aplicación. Cambia NoteStore de este modo:

app/stores/NoteStore.js

```
~~import uuid from 'uuid';~~
import NoteActions from '../actions/NoteActions';

export default class NoteStore {
  constructor() {
    this.bindActions(NoteActions);

~~    this.notes = [~~
~~      {~~
~~        id: uuid.v4(),~~
~~        task: 'Learn React'~~
~~      },~~
~~      {~~
~~        id: uuid.v4(),~~
~~        task: 'Do laundry'~~
~~      }~~
~~    ];~~
    this.notes = [];
  }
  ...
}
```

Es suficiente de momento. Nuestra aplicación debería arrancar desde cero.

10.7 Implementaciones Alternativas

Que hayamos usado Alt en esta implementación inicial no significa que sea la única opción. Para poder comparar varias arquitecturas he implementado la misma aplicación utilizando técnicas diferentes. A continuación presento una breve comparación:

- Redux[4] es una arquitectura inspirada en Flux diseñada con la recarga en caliente como primer objetivo a cumplir. Redux se basa en un único árbol de estado, el cual se manipula con *funciones puras* conocidas como reductores. Redux te fuerza a profundizar en la programación funcional. la implementación es muy parecida a la de Alt. - Demo de Redux[5]
- Comparado con Redux, Cerebral[6] tiene un enfoque diferente. Fue desarrollado para permitir ver *cómo* la aplicación cambia su estado. Cerebral guía más cómo hacer el desarrollo y, como resultado, viene con las pilas más cargadas. - Demo de Cerebral[7]
- MobX[8] te permite tener estructuras de datos observables. Las estructuras pueden estar conectadas con componentes de React así que cuando las estructuras cambian, también lo hacen los componentes. La implementación del Kanban es sorprendentemente simple ya que se pueden utilizar referencias reales entre componentes. - Demo de MobX[9]

10.8 ¿Relay?

Comparado con Flux, Relay[10] de Facebook mejora la recepción de datos. Permite llevar los requisitos sobre los datos a nivel de vista. Puede ser utilizado de forma independiente o con Flux dependiendo de lo que necesites.

[4] http://rackt.org/redux/
[5] https://github.com/survivejs/redux-demo
[6] http://www.cerebraljs.com/
[7] https://github.com/survivejs/cerebral-demo
[8] https://mobxjs.github.io/mobx/
[9] https://github.com/survivejs/mobx-demo
[10] https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html

No lo vamos a cubrir en este libro por ser una tecnología que todavía no está madura. Relay tiene algunos requisitos especiales, como un API compatible con GraphQL. Sólo lo explicaré si pasa a ser adoptado por la comunidad.

10.9 Conclusión

En este capítulo hemos visto cómo configurar el `localStorage` para almacenar el estado de la aplicación. Es una técnica pequeña a la vez que útil. Ahora que hemos solucionado la persistencia, estamos listos para tener un tablero de Kanban estupendo.

11. Gestionado Dependencias de Datos

Hasta ahora hemos desarrollado una aplicación que mantiene las notas en el `localStorage`. Para hacer algo más parecido a Kanban necesitamos modelar el concepto de `Carril`. Un `Carril` es algo que debe ser capaz de almacenar muchas notas y conocer su orden. Una forma de modelar esto es simplemente crear un `Carril` que contenga un array de identificadores de `Nota`.

Sin embargo, esta relación puede invertirse. Una `Nota` puede tener referencia a un `Carril` utilizando un identificador y almacenando cuál es su posición dentro del `Carril`. En nuestro caso vamos a utilizar el primer enfoque ya que permite reordenar con más facilidad.

11.1 Definiendo `Carriles`

Como hicimos anteriormente, podemos utilizar la idea de tener dos componentes aquí. Habrá un componente de más alto nivel (en este caso `Carriles`) y otro de más bajo nivel (`Carril`). El componente de más alto nivel se encargará de la ordenación de los carriles. Un `Carril` se renderizará a sí mismo y tendrá las reglas de manipulación básicas.

Al igual que con `Notas`, vamos a necesitar un conjunto de acciones. De momento es suficiente con crear nuevos carriles así que podemos crear la acción correspondiente como sigue:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions('create');
```

Además vamos a necesitar un `LaneStore` (almacén de carriles) y un método para poder crearlos. La idea es muy similar al `NoteStore` que teníamos anteriormente. La función `create` añadirá un nuevo carril a la lista de carriles. Tras ello, el cambio se propagará a los listeners (es decir, a `FinalStore` y sus componentes).

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    // Si no hay notas creamos un array vacío
    lane.notes = lane.notes || [];

    this.setState({
      lanes: this.lanes.concat(lane)
    });
  }
}
```

Para unir `LaneStore` con nuestra aplicación usaremos `setup`:

app/components/Provider/setup.js

```
import storage from '../../libs/storage';
import persist from '../../libs/persist';
import NoteStore from '../../stores/NoteStore';
import LaneStore from '../../stores/LaneStore';

export default alt => {
  alt.addStore('NoteStore', NoteStore);
  alt.addStore('LaneStore', LaneStore);

  persist(alt, storage(localStorage), 'app');
}
```

También necesitaremos un contenedor en el que mostrar nuestros carriles:

app/components/Lanes.jsx

```
import React from 'react';
import Lane from './Lane';

export default ({lanes}) => (
  <div className="lanes">{lanes.map(lane =>
    <Lane className="lane" key={lane.id} lane={lane} />
  )}</div>
)
```

Finalmente crearemos un pequeño esqueleto para `Carril` con el que asegurarnos que nuestra aplicación no se cuelga cuando conectamos `Carriles` con ella. Más adelante moveremos aquí mucha de la lógica que ahora está presente en `App`:

app/components/Lane.jsx

```
import React from 'react';

export default ({lane, ...props}) => (
  <div {...props}>{lane.name}</div>
)
```

11.2 Conectando `Carriles` con `App`

El paso siguiente es hacer hueco para `Carriles` en `App`. Simplemente reemplazaremos las referencias a `Notas` por `Carriles` y configuraremos las acciones de carriles y su almacén, lo que significa que mucho código antiguo desaparecerá. Cambia `App` por el código siguiente:

app/components/App.jsx

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import Lanes from './Lanes';
import LaneActions from '../actions/LaneActions';

const App = ({LaneActions, lanes}) => {
  const addLane = () => {
    LaneActions.create({
      id: uuid.v4(),
      name: 'New lane'
    });
  };

  return (
    <div>
      <button className="add-lane" onClick={addLane}>+</button>
      <Lanes lanes={lanes} />
    </div>
  );
```

```
};

export default connect(({lanes}) => ({
  lanes
}), {
  LaneActions
})(App)
```

Si pruebas esta implementación en el navegador verás que no hace mucho. Deberías poder añadir nuevos carriles en el Kanban y poder ver el texto "New lane" (nuevo carril) pero eso es todo. Para recuperar la funcionalidad que teníamos con las notas tendremos que centrarnos en modelar `Carril` más adelante.

11.3 Modelando `Carril`

`Carril` mostrará un nombre y `Notas` asociadas. El ejemplo que sigue tiene muchos cambios desde nuestra implementación inicial de `App`. Cambia el contenido del fichero y déjalo como sigue:

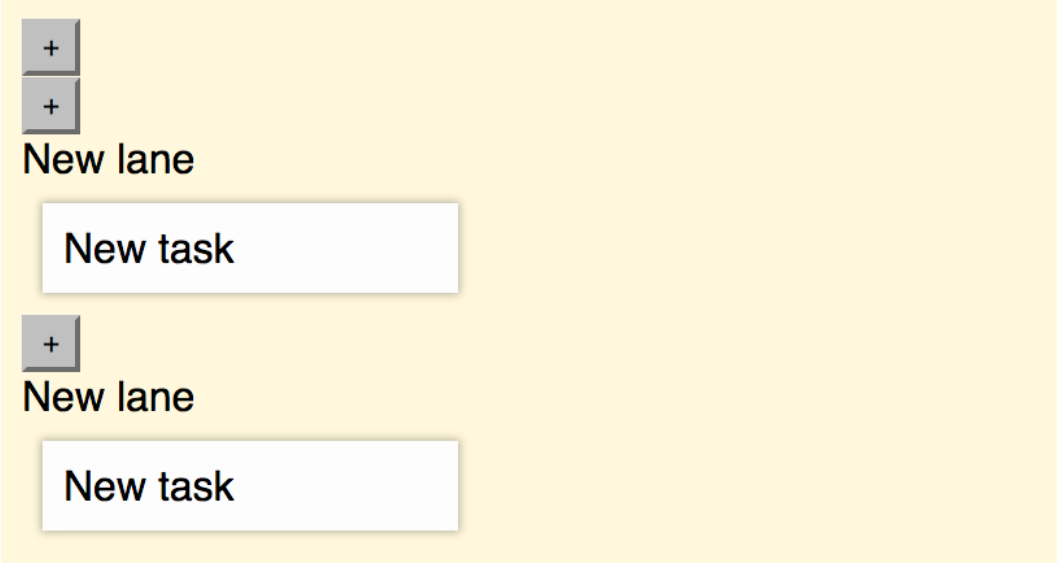
app/components/Lane.jsx

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import Notes from './Notes';

const Lane = ({
  lane, notes, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    NoteActions.update({id, task, editing: false});
  };
  const addNote = e => {
    e.stopPropagation();
```

```
    const noteId = uuid.v4();

    NoteActions.create({
      id: noteId,
      task: 'New task'
    });
  };
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (
    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}>+</button>
        </div>
        <div className="lane-name">{lane.name}</div>
      </div>
      <Notes
        notes={notes}
        onNoteClick={activateNoteEdit}
        onEdit={editNote}
        onDelete={deleteNote} />
    </div>
  );
};

export default connect(
```

```
  ({notes}) => ({
    notes
  }), {
    NoteActions
  }
)(Lane)
```

Si ejecutas la aplicación e intentas añadir notas nuevas verás que algo va mal. Cada nota que añades es compartida por todos los carriles. Si se modifica una nota, los otros carriles se modifican también.



Duplicar notas

El motivo de por qué ocurre esto es sencillo. Nuestro `NoteStore` es un singleton, lo que significa que todos los componentes que estén escuchando `NoteStore` recibirán los mismos datos. Necesitamos resolver este problema de alguna manera.

11.4 Haciendo que `Carriles` sea el Responsable de `Notas`

Ahora mismo nuestro `Carril` sólo contiene un array de objetos. Cada uno de estos objetos conoce su *id* y su *nombre*. Vamos a necesitar algo más sotisficado.

Cada `Carril` necesita saber qué `Notas` le pertenecen. Si un `Carril` contiene un array de identificadores de `Nota` podrá filtrar y mostrar sólo las `Notas` que le pertenecen. En breve implementaremos un esquema que permita esto.

Entendiendo `attachToLane` (añadir al carril)

Cuando añadimos una nueva `Nota` al sistema usando `addNote`, debemos asegurarnos que está asociada a un `Carril`. Esta asociación puede ser modelada mediante un método, como por ejemplo `LaneActions.attachToLane({laneId: <id>, noteId: <id>})`. He aquí un ejemplo de cómo podría funcionar.

```
const addNote = e => {
  e.stopPropagation();

  const noteId = uuid.v4();

  NoteActions.create({
    id: noteId,
    task: 'New task'
  });
  LaneActions.attachToLane({
    laneId: lane.id,
    noteId
  });
}
```

Esta es sólo una forma de gestionar `noteId`. Podemos llevar la lógica de generación a `NoteActions.create` y devolver el identificador generado desde allí. Podemos

hacerlo mediante una Promesa[1], lo cual puede ser muy útil si añadimos un backend a nuestra implementación. Así es como quedaría:

```
const addNote = e => {
  e.stopPropagation();

  NoteActions.create({
    task: 'New task'
  }).then(noteId => {
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId: noteId
    });
  })
}
```

Hemos declarado una dependencia clara entre `NoteActions.create` y `LaneActions.attachToLane`. Esto podría ser una alternativa válida si, especialmente, quieres llevar la implementación más lejos.

Puedes modelar el API usando parámetros de forma posicional y terminar teniendo `LaneActions.attachToLane(laneId, note.id)`. Yo prefiero pasar el objeto porque se lee bien y no hay que tener cuidado con el orden.

Otra forma de gestionar el problema de la dependencia puede ser utilizar una característica del dispacher de Flux conocida como waitFor[2]. Es mejor evitarlo si puedes. Además, gestores de estado como Redux hacen que sea redundante. Usar `Promesas` como antes nos puede ayudar.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
[2] http://alt.js.org/guide/wait-for/

Configurando `attachToLane`

Para comenzar debemos añadir `attachToLane` a las acciones como hicimos antes:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'attachToLane'
);
```

Para poder implementar `attachToLane` tenemos que buscar un carril que coincida con el identificador del carril que hemos recibido y asociarle el identificador de la nota. Es más, cada nota sólo debe pertenecer a un carril cada vez. Podemos hacer una pequeña comprobación:

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  attachToLane({laneId, noteId}) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.notes.includes(noteId)) {
          lane.notes = lane.notes.filter(note => note !== noteId);
        }

        if(lane.id === laneId) {
          lane.notes = lane.notes.concat([noteId]);
        }

        return lane;
      })
```

```
    });
  }
}
```

Ser únicamente capaz de incluir notas en un carril no es suficiente. También vamos a necesitar poder sacarlas, lo cual ocurre cuando borramos notas.

En este punto podemos mostrar un mensaje de advertencia cuando tratemos de incluir una nota en un carril que no exista. `console.warn` será tu amigo en este caso.

Configurando `detachFromLane`

Podemos modelar de forma parecida la operación contraria `detachFromLane` usando un API como el siguiente:

```
LaneActions.detachFromLane({noteId, laneId});
NoteActions.delete(noteId);
```

Al igual que con `attachToLane`, podemos modelar el API usando parámetros de forma posicional para dejarlo de este modo: `LaneActions.detachFromLane(laneId, noteId)`.

De nuevo debemos configurar la acción:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'attachToLane', 'detachFromLane'
);
```

La implementación se parece a `attachToLane`. En este caso, borraremos las notas que se puedan encontrar dentro.

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  detachFromLane({laneId, noteId}) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.id === laneId) {
          lane.notes = lane.notes.filter(note => note !== noteId);
        }

        return lane;
      })
    });
  }
}
```

Dado que tenemos la lógica en su lugar, podemos comenzar la conexión con la interfaz de usuario.

Es posible que `detachFromLane` no desvincule nada. Si se detecta este caso puede ser una buena idea usar `console.warn` para hacer consciente al desarrollador de lo que ocurre.

Conectando `Carril` con la Lógica

Para hacer que esto funcione necesitamos hacer cambios en un par de sitios:

- Cuando añadimos una nota, necesitamos **vincularla** con el carril actual.
- Cuando borramos una nota, necesitamos **desvincularla** del carril actual.
- Cuando renderizamos un carril necesitamos **seleccionar** las notas que le pertenecen. Es importante renderizar las notas en el orden en el cual pertenezcan al carril. Esto requiere de algo de lógica extra.

Estos cambios implican modificar `Carril` como sigue:

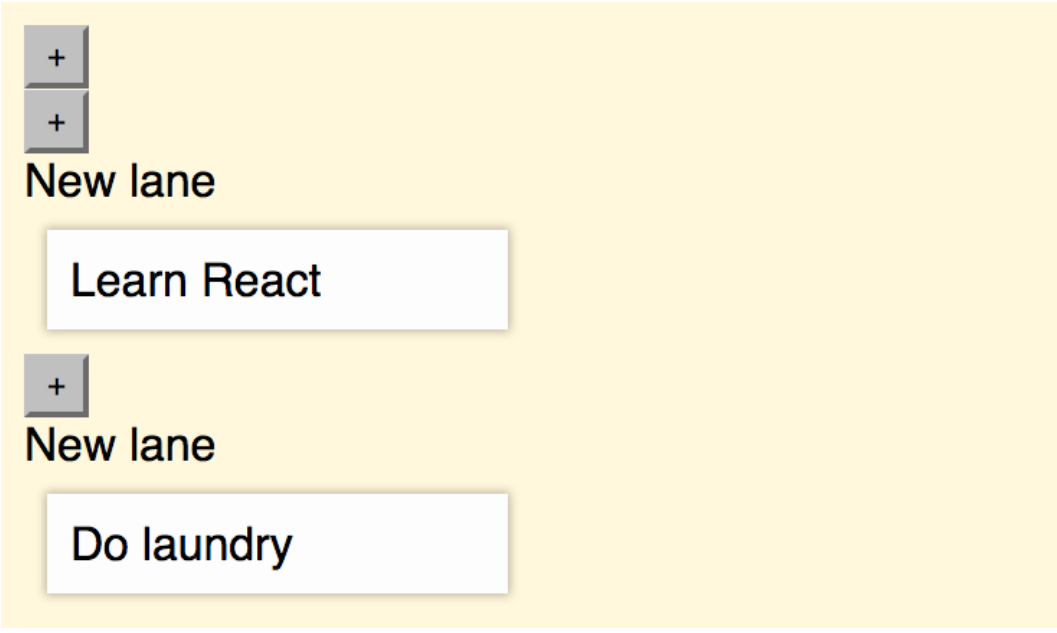
app/components/Lane.jsx

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';

const Lane = ({
  lane, notes, NoteActions, ...props
  lane, notes, LaneActions, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    ...
  };
  const addNote = e => {
    e.stopPropagation();

    const noteId = uuid.v4();

    NoteActions.create({
      id: noteId,
```

```
      task: 'New task'
    });
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId
    });
  };
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

    LaneActions.detachFromLane({
      laneId: lane.id,
      noteId
    });
    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (
    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}>+</button>
        </div>
        <div className="lane-name">{lane.name}</div>
      </div>
      <Notes
        notes={notes}
        notes={selectNotesByIds(notes, lane.notes)}
        onNoteClick={activateNoteEdit}
        onEdit={editNote}
        onDelete={deleteNote} />
    </div>
```

```
  );
};

function selectNotesByIds(allNotes, noteIds = []) {
  // `reduce` es un método poderoso que nos permite
  // agrupar datos. Puedes implementar filter` y `map`
  // dentro de él. Nosotros lo estamos usando para
  // concatenar notas cuyos id coincidan
  return noteIds.reduce((notes, id) =>
    // Concatena ids que encajen al resultado
    notes.concat(
      allNotes.filter(note => note.id === id)
    )
  , []);
}

export default connect(
  ({notes}) => ({
    notes
  }), {
    NoteActions
    NoteActions,
    LaneActions
  }
)(Lane)
```

Si intentas utilizar la aplicación ahora verás que cada carril es capaz de mantener sus propias notas:



Separate notes

La estructura actual nos permite mantener el singleton y una estructura de datos plana. Lidiar con las referencias es un tanto tedioso, pero es consistente con la arquitectura Flux. Puedes ver el mismo problema en la implementación con Redux[3]. La implementación con MobX[4] evita el problema completamente.

`selectNotesByIds` pudo haber sido escrita utilizando `map` y `find`. En ese caso podrías haber acabado utilizando `noteIds.map(id => allNotes.find(note => note.id === id));`. Sin embargo, tendrías que haber utilizado el polyfill `find` para que funcione en navegadores viejos.

Normalizar los datos puede hacer que `selectNotesByIds` sea trivial. Si estás usando una solución como Redux, la normalización puede hacer fáciles operaciones como ésta.

[3] https://github.com/survivejs-demos/redux-demo
[4] https://github.com/survivejs-demos/mobx-demo

11.5 Extrayendo `LaneHeader` (Cabecera de Carril) de `Carril`

`Carril` está empezando a ser un componente demasiado grande. Tenemos la oportunidad de partirlo para hacer que nuestra aplicación sea más fácil de mantener. En concreto, la cabecera del carril puede ser un componente por sí mismo. Para comenzar, define `LaneHeader` basándote en el código actual tal y como sigue:

app/components/LaneHeader.jsx

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';

export default connect(() => ({}), {
  NoteActions,
  LaneActions
})(({lane, LaneActions, NoteActions, ...props}) => {
  const addNote = e => {
    e.stopPropagation();

    const noteId = uuid.v4();

    NoteActions.create({
      id: noteId,
      task: 'New task'
    });
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId
    });
  };
```

```
  return (
    <div className="lane-header" {...props}>
      <div className="lane-add-note">
        <button onClick={addNote}>+</button>
      </div>
      <div className="lane-name">{lane.name}</div>
    </div>
  );
})
```

Necesitamos conectar el componente que hemos extraido con `Carril`:

app/components/Lane.jsx

```
import React from 'react';
~~import uuid from 'uuid';~~
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';
import LaneHeader from './LaneHeader';

const Lane = ({
  lane, notes, LaneActions, NoteActions, ...props
}) => {
  const editNote = (id, task) => {
    NoteActions.update({id, task, editing: false});
  };
~~  const addNote = e => {~~
~~    e.stopPropagation();~~

~~    const noteId = uuid.v4();~~

~~    NoteActions.create({~~
~~      id: noteId,~~
~~      task: 'New task'~~
```

```
    });
    LaneActions.attachToLane({
      laneId: lane.id,
      noteId
    });
  };
  const deleteNote = (noteId, e) => {
    e.stopPropagation();

    LaneActions.detachFromLane({
      laneId: lane.id,
      noteId
    });
    NoteActions.delete(noteId);
  };
  const activateNoteEdit = id => {
    NoteActions.update({id, editing: true});
  };

  return (
    <div {...props}>
      <div className="lane-header">
        <div className="lane-add-note">
          <button onClick={addNote}>+</button>
        </div>
        <div className="lane-name">{lane.name}</div>
      </div>
      <LaneHeader lane={lane} />
      <Notes
        notes={selectNotesByIds(notes, lane.notes)}
        onNoteClick={activateNoteEdit}
        onEdit={editNote}
        onDelete={deleteNote} />
    </div>
  );
```

```
};

...
```

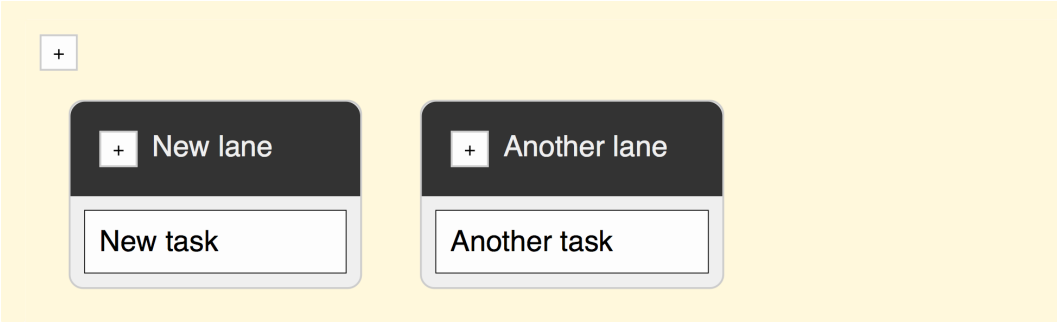
Tras estos cambios tendremos algo con lo que es un poco más fácil trabajar. Podría haber sido posible mantener todo el código en un único componente. A menudo reflexionarás y te darás cuenta de que hay mejores maneras de dividir tus componentes. A menudo la necesidad de reutilizar o de mejorar el rendimiento serán quienes te fuercen a realizar estas divisiones.

11.6 Conclusión

En este capítulo nos las hemos apañado para resolver el problema de gestión de dependencias de datos. Este problema aparece con frecuencia. Cada gestor de estados tiene su propia manera de lidiar con ello. Las alternativas basadas en Flux y Redux esperan que seas tú quien gestione las referencias. Soluciones como MobX integran la gestión de referencias. La normalización de los datos puede hacer que este tipo de operaciones sean más sencillas.

En el próximo capítulo nos centraremos en añadir una funcionalidad que no tenemos en la aplicación: la edición de carriles. También haremos que la aplicación tenga un mejor aspecto. Por suerte mucha de la lógica que necesitamos ya ha sido desarrollada.

12. Editando los Carriles



Tablero Kanban

Todavia tenemos mucho trabajo por hacer para conseguir que esto sea un Kanban real como el que se muestra ahí arriba. La aplicación todavía carece de algo de lógica y de estilo. Es en eso en lo que nos vamos a centrar ahora.

El componente `Editable` que implementamos antes nos será útil ahora. Podremos usarlo para que sea posible cambiar el nombre de los carriles. La idea es la misma que para las notas.

También debemos hacer que sea posible eliminar carriles. Para ello necesitaremos añadir un control en el UI e incluir algo de lógica. De nuevo, la idea es similar a lo de antes.

12.1 Implementando la Edición de Nombres de `Carril`

Para editar el nombre de un `Carril` necesitaremos algo de lógica y puntos de enganche con el UI. `Editable` puede encargarse del UI, la lógica nos dará algo más de trabajo. Para comenzar, deja `LaneHeader` como sigue:

app/components/LaneHeader.jsx

```
import React from 'react';
import uuid from 'uuid';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Editable from './Editable';

export default connect(() => ({}), {
  NoteActions,
  LaneActions
})(({lane, LaneActions, NoteActions, ...props}) => {
  const addNote = e => {
    ...
  };
  const activateLaneEdit = () => {
    LaneActions.update({
      id: lane.id,
      editing: true
    });
  };
  const editName = name => {
    LaneActions.update({
      id: lane.id,
      name,
      editing: false
    });
  };

  return (
    <div className="lane-header" {...props}>
    <div className="lane-header" onClick={activateLaneEdit} {...prop\
s}>
      <div className="lane-add-note">
        <button onClick={addNote}>+</button>
      </div>
```

```
      <div className="lane-name">{lane.name}</div>
      <Editable className="lane-name" editing={lane.editing}
        value={lane.name} onEdit={editName} />
    </div>
  );
})
```

La interfaz de usuario debería tener el mismo aspecto tras este cambio. Todavía necesitamos implementar `LaneActions.update` para hacer que esto funcione.

Igual que antes, tenemos que hacer cambios en dos sitios, en la definición de la acción y en `LaneStore`. Aquí tenemos la parte de la acción:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'attachToLane', 'detachFromLane'
);
```

Para añadir la lógica que falta, modifica `LaneStore` de este modo. La idea es la misma que la de `NoteStore`:

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    ...
```

```
  }
  update(updatedLane) {
    this.setState({
      lanes: this.lanes.map(lane => {
        if(lane.id === updatedLane.id) {
          return Object.assign({}, lane, updatedLane);
        }

        return lane;
      })
    });
  }
  ...
}
```

Tras estos cambios deberías ser capaz de editar los nombres de los carriles. El borrado de carriles es una buena característica con la que seguir.

12.2 Implementando el Borrado de `Carril`

El borrado de carriles en un problema parecido. Necesitamos poner más cosas en la interfaz de usuario, añadir una acción y asociarle lógica.

La interfaz del usuario es un lugar natural en el que comenzar. A menudo es una buena idea añadir algunos `console.log` en ciertos lugares para estar seguro de que los manejadores se ejecutan cuando esperas. Puede ser incluso mejor que escribas tests para ellos. De este modo acabarás con una especificación ejecutable. Aquí tienes un esqueleto con el que poder borrar carriles:

app/components/LaneHeader.jsx

```
...

export default connect(() => ({}), {
  NoteActions,
  LaneActions
})(({lane, LaneActions, NoteActions, ...props}) => {
  ...
  const deleteLane = e => {
    // Evita que se ejecuten los eventos naturales de javascript del\
 componente
    e.stopPropagation();

    LaneActions.delete(lane.id);
  };

  return (
    <div className="lane-header" onClick={activateLaneEdit} {...prop\
s}>
      <div className="lane-add-note">
        <button onClick={addNote}>+</button>
      </div>
      <Editable className="lane-name" editing={lane.editing}
        value={lane.name} onEdit={editName} />
      <div className="lane-delete">
        <button onClick={deleteLane}>x</button>
      </div>
    </div>
  );
});
```

De nuevo, necesitamos agrandar nuestra definición de acción:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'delete', 'attachToLane', 'detachFromLane'
);
```

Y, para finalizar con la implementación, tenemos que añadir algo de lógica:

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  constructor() {
    this.bindActions(LaneActions);

    this.lanes = [];
  }
  create(lane) {
    ...
  }
  update(updatedLane) {
    ...
  }
  delete(id) {
    this.setState({
      lanes: this.lanes.filter(lane => lane.id !== id)
    });
  }
  ...
}
```

Si todo ha ido correctamente ahora deberías ser capaz de borrar carriles enteros.

La implementación actual tiene un problema. Aunque estamos borrando las referencias con los carriles, las notas todavía existen. Esto lo podemos arreglar de dos

maneras: creando una papelera donde ir dejando esta basura y borrarla cada cierto tiempo o podemos borrar las notas junto con el carril. Sin embargo, para el ámbito de esta aplicación vamos a dejarlo como está, es una mejora de lo que tendremos que ser conscientes.

12.3 Dando Estilo al Tablero Kanban

El estilo se ha estropeado un poco al añadir `Carriles` a la aplicación. Cambia lo siguiente para que quede un poco mejor:

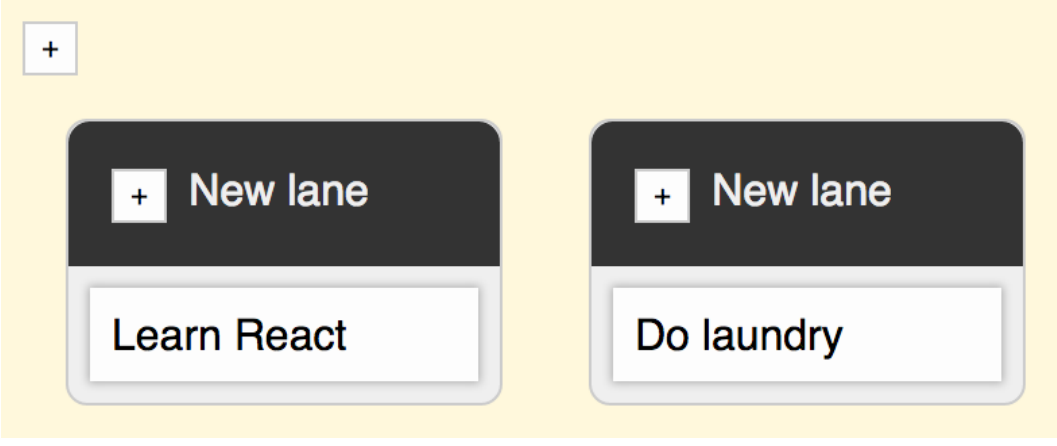
app/main.css

```
body {
  background-color: cornsilk;

  font-family: sans-serif;
}

.add-note {
  background-color: #fdfdfd;

  border: 1px solid #ccc;
}

.lane {
  display: inline-block;

  margin: 1em;

  background-color: #efefef;
  border: 1px solid #ccc;
  border-radius: 0.5em;

  min-width: 10em;
  vertical-align: top;
```

```
}

.lane-header {
  overflow: auto;

  padding: 1em;

  color: #efefef;
  background-color: #333;

  border-top-left-radius: 0.5em;
  border-top-right-radius: 0.5em;
}

.lane-name {
  float: left;
}

.lane-add-note {
  float: left;

  margin-right: 0.5em;
}

.lane-delete {
  float: right;

  margin-left: 0.5em;

  visibility: hidden;
}
.lane-header:hover .lane-delete {
  visibility: visible;
}
```

```
.add-lane, .lane-add-note button {
  cursor: pointer;

  background-color: #fdfdfd;
  border: 1px solid #ccc;
}

.lane-delete button {
  padding: 0;

  cursor: pointer;

  color: white;
  background-color: rgba(0, 0, 0, 0);
  border: 0;
}

. . .
```

Deberías ver algo como esto:



Kanban con Estilo

Podemos dejar el CSS en un sólo fichero ya que el nuestro es un proyecto pequeño. En caso de que comience a crecer, habrá que considerar el partirlo en varios ficheros.

Una forma de hacer esto es extraer el CSS de cada componente y referenciarlo desde él (por ejemplo, `require('./lane.css')` en `Lane.jsx`). Puedes incluso considerar el utilizar **Módulos CSS** para hacer que las CSS funcionen en un ámbito local. Lee el capítulo *Dando Estilo a React* para más información.

12.4 Conclusión

Aunque nuestra aplicación empieza a tener un buen aspecto y tiene una funcionalidad básica todavia carece de una característica vital: aún no podemos mover notas entre carriles. Esto es algo que solucionaremos en el próximo capítulo cuando implementemos el drag and drop (arrastrar y soltar).

13. Implementado Arrastrar y Soltar

Nuestra aplicación de Kanban es casi utilizable. Tiene un buen aspecto y cierta funcionalidad básica. En este capítulo integraremos la funcionalidad de arrastrar y soltar utilizando React DnD[1].

Al terminar este capítulo deberías ser capaz de arrastrar notas entre carriles. Aunque parezca sencillo implica realizar algo de trabajo por nuestra parte ya que tendremos que anotar los componentes de la forma correcta y crear la lógica necesaria.

13.1 Configurando React DnD

Para comenzar necesitaremos conectar React DnD con nuestro proyecto. Vamos a utilizar un backend de arrastrar y soltar basado en el de HTML5. Existen backends específicos para testing y tacto[2].

Para configurarlo, necesitaremos utilizar el decorador `DragDropContext` y facilitarle el backend de HTML5. Voy a utilizar `compose` de redux para evitar revestimientos innecesarios y mantener el código más limpio:

app/components/App.jsx

[1] https://gaearon.github.io/react-dnd/
[2] https://github.com/yahoo/react-dnd-touch-backend

```
import React from 'react';
import uuid from 'uuid';
import {compose} from 'redux';
import {DragDropContext} from 'react-dnd';
import HTML5Backend from 'react-dnd-html5-backend';
import connect from '../libs/connect';
import Lanes from './Lanes';
import LaneActions from '../actions/LaneActions';

const App = ({LaneActions, lanes}) => {
  const addLane = () => {
    LaneActions.create({
      id: uuid.v4(),
      name: 'New lane'
    });
  };

  return (
    <div>
      <button className="add-lane" onClick={addLane}>+</button>
      <Lanes lanes={lanes} />
    </div>
  );
};

export default connect(({lanes}) => ({
  lanes
}), {
  LaneActions
})(App)
export default compose(
  DragDropContext(HTML5Backend),
  connect(
    ({lanes}) => ({lanes}),
    {LaneActions}
```

```
  )
)(App)
```

Tras este cambio la aplicación debería tener el mismo aspecto de antes, pero ahora estamos preparados para añadir la funcionalidad.

13.2 Permitiendo que las Notas sean Arrastradas

Permitir que las notas puedan ser arrastradas es un buen comienzo. Antes de ello, necesitamos configurar una constante de tal modo que React DnD sepa que hay distintos tipos de elementos arrastrables. Crea un fichero con el que poder indicar que quieres mover elementos de tipo `Nota` como sigue:

app/constants/itemTypes.js

```
export default {
  NOTE: 'note'
};
```

Esta definición puede ser extendida más adelante incluyendo nuevos tipos, como `CARRIL`, al sistema.

A continuación necesitamos decirle a nuestra `Nota` que es posible arrastrarla. Esto se puede conseguir usando la anotación `DragSource`. Modifica `Nota` con la siguiente implementación:

app/components/Note.jsx

```
import React from 'react';
import {DragSource} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, children, ...props
}) => {
  return connectDragSource(
    <div {...props}>
      {children}
    </div>
  );
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};

export default DragSource(ItemTypes.NOTE, noteSource, connect => ({
  connectDragSource: connect.dragSource()
}))(Note)
```

Deberías ver algo como esto en la consola del navegador al tratar de mover una nota:

```
begin dragging note Object {className: "note", children: Array[2]}
```

Ser capaz sólo de mover notas no es suficiente. Necesitamos anotarlas para que puedan ser soltadas. Esto nos permitirá lanzar cierta lógica cuando tratemos de soltar una nota encima de otra.

Observa que React DnD no soporta recarga en caliente perfectamente. Puede que necesites refrescar el navegador para ver los mensajes de log que esperas.

13.3 Permitiendo a las Notas que Detecten Notas que Pasan por Encima

Podemos anotar notas de tal modo que detecten que otra nota les está pasando por encima de un modo similar al anterior. En este caso usaremos la anotación `DropTarget`:

app/components/Note.jsx

```
import React from 'react';
~~import {DragSource} from 'react-dnd';~~
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  ~~connectDragSource, children, ...props~~
  connectDragSource, connectDropTarget,
  children, ...props
}) => {
  ~~return connectDragSource(~~
  return compose(connectDragSource, connectDropTarget)(
    <div {...props}>
      {children}
    </div>
  );
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
};
```

```
const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();

    console.log('dragging note', sourceProps, targetProps);
  }
};

export default DragSource(ItemTypes.NOTE, noteSource, connect => ({
  connectDragSource: connect.dragSource()
}))(Note)
export default compose(
  DragSource(ItemTypes.NOTE, noteSource, connect => ({
    connectDragSource: connect.dragSource()
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
    connectDropTarget: connect.dropTarget()
  }))
)(Note)
```

Si pruebas a arrastrar una nota por encima de otra deberías ver mensajes como el siguiente en la consola:

```
dragging note Object {} Object {className: "note", children: Array[2\
]}
```

Ambos decoradores nos dan acceso a las propiedades de `Nota`. En este caso estamos usando `monitor.getItem()` para acceder a ellas en `noteTarget`. Esta es la clave para hacer que todo funcione correctamente.

13.4 Desarrollando el API `onMove` para `Notas`

Ahora que podemos mover las notas podemos comenzar a definir la lógica. Se necesitan los siguientes pasos:

1. Capturar el identificador de `Nota` en `beginDrag`.
2. Capturar el identificador de la `Nota` objetivo `hover`.
3. Lanzar la llmada a `hover` cuando se ejecute `onMove` par que podamos incluir la lógica en algún sitio. `LaneStore` puede ser el mejor lugar para ello.

Siguiendo la idea anterior podemos pasar el identificador de la `Nota` mediante una propiedad. También necesitaremos crear un esqueleto para la llamada a `onMove` y definir `LaneActions.move` y `LaneStore.move`.

Aceptando `id` y `onMove` en `Nota`

Podemos aceptar las propiedades `id` y `onMove` en `Nota` como sigue:

app/components/Note.jsx

```
...

const Note = ({
  connectDragSource, connectDropTarget,
  children, ...props
  onMove, id, children, ...props
}) => {
  return compose(connectDragSource, connectDropTarget)(
    <div {...props}>
      {children}
    </div>
  );
};

const noteSource = {
  beginDrag(props) {
    console.log('begin dragging note', props);

    return {};
  }
```

```
};
const noteSource = {
  beginDrag(props) {
    return {
      id: props.id
    };
  }
};

const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();

    console.log('dragging note', sourceProps, targetProps);
  }
};
const noteTarget = {
  hover(targetProps, monitor) {
    const targetId = targetProps.id;
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    if(sourceId !== targetId) {
      targetProps.onMove({sourceId, targetId});
    }
  }
};

...
```

Tener esas propiedades no es útil si no pasamos nada a `Notas`. Ese será nuestro siguiente paso.

Pasando `id` y `onMove` desde `Notes`

Pasar el `id` de una nota y `onMove` es sencillo:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" onClick={onNoteClick.bind(null, id)}>
      <Note className="note" id={id}
        onClick={onNoteClick.bind(null, id)}
        onMove={({sourceId, targetId}) =>
          console.log('moving from', sourceId, 'to', targetId)}>
        <Editable
          className="editable"
          editing={editing}
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button
          className="delete"
          onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Si mueves una nota encima de otra verás mensajes por consola como el siguiente:

```
moving from 3310916b-5b59-40e6-8a98-370f9c194e16 to 939fb627-1d56-4b\
57-89ea-04207dbfb405
```

13.5 Añadiendo Acciones en el Movimiento

La lógica de arrastrar y soltar funciona como sigue. Supón que tienes un carril que contiene las notas A, B y C. En caso de que sitúes A detrás de C el carríl contendrá B, C y A. Si tienes otra lista, por ejemplo D, E y F, y movemos A al comienzo de ésta lista, acabaremos teniendo B y C y A, D, E y F.

En nuestro caso tendremos algo de complejidad extra al soltar notas de carril en carril. Cuando movamos una `Nota` sabremos su posición original y la posición que querramos que tenga al final. El `Carril` sabe qué `Notas` le pertenecen por sus ids. Vamos a necesitar decir al `LaneStore` de alguna forma que debe realizar algo de lógica sobre las notas que posee. Un buen punto de partida es definir `LaneActions.move`:

app/actions/LaneActions.js

```
import alt from '../libs/alt';

export default alt.generateActions(
  'create', 'update', 'delete',
  'attachToLane', 'detachFromLane',
  'move'
);
```

Debemos conectar esta acción con el punto de enganche `onMove` que acabamos de definir:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';
import LaneActions from '../actions/LaneActions';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" id={id}
        onClick={onNoteClick.bind(null, id)}
        onMove={({sourceId, targetId}) =>
          console.log('moving from', sourceId, 'to', targetId)}>
        onMove={LaneActions.move}>
        <Editable
          className="editable"
          editing={editing}
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button
          className="delete"
          onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Puede ser una buena idea refactorizar `onMove` y dejarla como propiedad para hacer que el sistema sea más flexible. En nuestra implementación el componente `Notas` está acoplado con `LaneActions`, lo cual no es particularmente útil si quieres poder usarlo en otro contexto.

También debemos definir un esqueleto en `LaneStore` para ver que lo hemos cableado

todo correctamente:

app/stores/LaneStore.js

```
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  detachFromLane({laneId, noteId}) {
    ...
  }
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
}
```

Deberías ver los mismos mensajes de log de antes.

A continuación vamos a añadir algo de lógica para conseguir que esto funcione. Hay dos casos de los que nos tenemos que preocupar: mover notas dentro de un mismo carril y mover notas entre distintos carriles.

13.6 Implementando la Lógica de Arrastrar y Soltar Notas

El movimiento dentro de un mismo carril es complicado. Cuando estás basando las operaciones en ids y haces las operaciones una a una, tienes que tener en cuenta que puede hacer alteraciones en el índice. Como resultado estoy usando update[3] de React para solucionar el problema de una pasada.

Es posible solucionar el caso de mover notas entre carriles usando splice[4]. Primero obtenemos la nota a mover, y después la incorporamos al carril destino. De nuevo,

[3] https://facebook.github.io/react/docs/update.html

[4] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

`update` puede ser útil aquí, aunque en este caso `splice` está bien. El siguiente código muestra una posible solución:

app/stores/LaneStore.js

```
import update from 'react-addons-update';
import LaneActions from '../actions/LaneActions';

export default class LaneStore {
  ...
  move({sourceId, targetId}) {
    console.log(`source: ${sourceId}, target: ${targetId}`);
  }
  move({sourceId, targetId}) {
    const lanes = this.lanes;
    const sourceLane = lanes.filter(lane => lane.notes.includes(sour\
ceId))[0];
    const targetLane = lanes.filter(lane => lane.notes.includes(targ\
etId))[0];
    const sourceNoteIndex = sourceLane.notes.indexOf(sourceId);
    const targetNoteIndex = targetLane.notes.indexOf(targetId);

    if(sourceLane === targetLane) {
      // las mueve en bloque para evitar complicaciones
      sourceLane.notes = update(sourceLane.notes, {
        $splice: [
          [sourceNoteIndex, 1],
          [targetNoteIndex, 0, sourceId]
        ]
      });
    }
    else {
      // elimina la nota del origen
      sourceLane.notes.splice(sourceNoteIndex, 1);

      // y la mueve al objetivo
```

```
      targetLane.notes.splice(targetNoteIndex, 0, sourceId);
    }

    this.setState({lanes});
  }
}
```

Si pruebas la aplicación ahora verás que puedes arrastrar notas y que el comportamiento debería ser el correcto. Arrastrar a carriles vacíos no funcionará y la presentación puede ser mejorada.

Podría ser mejor si indicásemos la localización de la nota arrastrada de forma más clara. Podemos conseguirlo ocultándola de la lista. React DnD nos dá los puntos de enganche que necesitamos para conseguirlo.

Indicando Dónde Mover

React DnD tiene una cualidad conocida como monitores de estado. Con ellos podemos usar `monitor.isDragging()` y `monitor.isOver()` para detectar qué `Nota` es la que estamos arrastrando. Podemos configurarlo como sigue:

app/components/Note.jsx

```
import React from 'react';
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, connectDropTarget,
  onMove, id, children, ...props
  connectDragSource, connectDropTarget, isDragging,
  isOver, onMove, id, children, ...props
}) => {
  return compose(connectDragSource, connectDropTarget)(
    <div {...props}>
```

```
      {children}
    </div>
    <div style={{
      opacity: isDragging || isOver ? 0 : 1
    }} {...props}>{children}</div>
  );
};

...

export default compose(
  DragSource(ItemTypes.NOTE, noteSource, connect => ({
    connectDragSource: connect.dragSource()
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
    connectDropTarget: connect.dropTarget()
  }))
  DragSource(ItemTypes.NOTE, noteSource, (connect, monitor) => ({
    connectDragSource: connect.dragSource(),
    isDragging: monitor.isDragging()
  })),
  DropTarget(ItemTypes.NOTE, noteTarget, (connect, monitor) => ({
    connectDropTarget: connect.dropTarget(),
    isOver: monitor.isOver()
  }))
)(Note)
```

Si arrastras una nota por un carril, la nota arrastrada se mostrará en blanco.

Hay un pequeño problema con nuestro sistema. Todavía no podemos arrastrar notas sobre un carril vacío.

13.7 Arrastrando Notas sobre Carriles Vacíos

Para arrastrar notas sobre carriles vaciós necesitamos permitirles el poder recibir notas. Al igual que antes, podemos configurar una lógica basada en `DropTarget` para

ello. Antes de nada, necesitamos capturar el hecho de arrastrar en `Carril`:

app/components/Lane.jsx

```
import React from 'react';
import {compose} from 'redux';
import {DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';
import connect from '../libs/connect';
import NoteActions from '../actions/NoteActions';
import LaneActions from '../actions/LaneActions';
import Notes from './Notes';
import LaneHeader from './LaneHeader';

const Lane = ({
  lane, notes, LaneActions, NoteActions, ...props
  connectDropTarget, lane, notes, LaneActions, NoteActions, ...props
}) => {
  ...

  return (
  return connectDropTarget(
    ...
  );
};

function selectNotesByIds(allNotes, noteIds = []) {
  ...
}

const noteTarget = {
  hover(targetProps, monitor) {
    const sourceProps = monitor.getItem();
    const sourceId = sourceProps.id;

    // Si el carril destino no tiene notas
```

```
    // le damos la nota.
    //
    // `attachToLane` hace la limpieza necesaria
    // por defecto y garantiza que una nota sólo
    // pueda pertenecar a un carril
    if(!targetProps.lane.notes.length) {
      LaneActions.attachToLane({
        laneId: targetProps.lane.id,
        noteId: sourceId
      });
    }
  }
};

export default connect(
  ({notes}) => ({
    notes
  }), {
    NoteActions,
    LaneActions
  }
)(Lane)
export default compose(
  DropTarget(ItemTypes.NOTE, noteTarget, connect => ({
    connectDropTarget: connect.dropTarget()
  })),
  connect(({notes}) => ({
    notes
  }), {
    NoteActions,
    LaneActions
  })
)(Lane)
```

Debería ser capaz de poder arrastrar notas a carriles vacios una vez hayas añadido esta lógica.

Nuesta implementación de `attachToLane` hace gran parte del trabajo duro por nosotros. Si no garantizase que una nota sólo puede pertenecer a un carril nuestra lógica debería ser modificada. Es bueno tener este tipo de certezas dentro del sistema de gestión de estados.

Solucionando el Modo de Edición durante el Arrastre

La implementación actual tiene un pequeño problema. Puedes arrastrar una nota mientras esta está siendo editada. Esto no es conveniente ya que no es lo que la mayoría de la gente espera poder hacer. No puedes, por ejemplo, hacer doble click en la caja de texto para seleccionar todo su contenido.

Por suerte es fácil de arreglar. Necesitamos usar el estado `editing` de cada `Nota` para ajustar su comportamiento. Lo primero que necesitamos es pasar el estado `editing` a una `Nota` individual:

app/components/Notes.jsx

```
import React from 'react';
import Note from './Note';
import Editable from './Editable';
import LaneActions from '../actions/LaneActions';

export default ({
  notes,
  onNoteClick=() => {}, onEdit=() => {}, onDelete=() => {}
}) => (
  <ul className="notes">{notes.map(({id, editing, task}) =>
    <li key={id}>
      <Note className="note" id={id}
        editing={editing}
        onClick={onNoteClick.bind(null, id)}
        onMove={LaneActions.move}>
        <Editable
          className="editable"
          editing={editing}
```

```
          value={task}
          onEdit={onEdit.bind(null, id)} />
        <button
          className="delete"
          onClick={onDelete.bind(null, id)}>x</button>
      </Note>
    </li>
  )}</ul>
)
```

Lo siguiente será tenerlo en cuenta a la hora de renderizar:

app/components/Note.jsx

```
import React from 'react';
import {compose} from 'redux';
import {DragSource, DropTarget} from 'react-dnd';
import ItemTypes from '../constants/itemTypes';

const Note = ({
  connectDragSource, connectDropTarget, isDragging,
~~  isOver, onMove, id, children, ...props~~
  isOver, onMove, id, editing, children, ...props
}) => {
  // Pass through if we are editing
  const dragSource = editing ? a => a : connectDragSource;

~~  return compose(connectDragSource, connectDropTarget)(~~
  return compose(dragSource, connectDropTarget)(
    <div style={{
      opacity: isDragging || isOver ? 0 : 1
    }} {...props}>{children}</div>
  );
};

...
```

Este pequeño cambio nos dá el comportamiento que queremos. Si tratas de editar una nota ahora, la caja de texto se comportará como esperas.

Mirando hacia atrás podemos ver que mantener el estado `editing` fuera de `Editable` fue una buena idea. Si no lo hubiésemos hecho así, implementar este cambio habría sido bastante más difícil ya que tendríamos que poder sacar el estado fuera del componente.

¡Por fin tenemos un tablero Kanban que es útil!. Podemos crear carriles y notas nuevas, y también podemos editarlas y borrarlas. Además podemos movar las notas. ¡Objetivo cumplido!

13.8 Conclusión

En este capítulo has visto cómo implementar la funcionalidad de arrastrar y soltar para nuestra pequeña aplicación. Puedes modelar la ordenación de carriles usando la misma técnica. Primero, marcas los carriles como arrastrables y soltables, los ordenas según sus identificadores y, finalmente, añades algo de lógica para hacer que todo funcione. Debería ser más sencillo que lo que hemos hecho con las notas.

Te animo a que hagas crecer la aplicación. La implementación actual debería servir de punto de entrada para hacer algo más grande. Más allá de la implementación de arrastrar y soltar, puedes tratar de añadir más datos al sistema. También puedes hacer algo con el aspecto gráfico. Una opción puede ser usar varias de las aproximaciones de aplicación de estilos que se discuten en el capítulo *Dando Estilo a React.*

Para conseguir que sea difícil romper la aplicación durante el desarrollo, puedes implementar tests como se indica en *Probando React. Tipando con React* discute más modos aún de endurecer tu código. Aprender estas aproximaciones puede merecer la pena. A veces es realmente útil diseñar antes los tests de las aplicaciones, ya que es una aporximación valiosa que te permite documentar lo que vas asumiendo a medida que haces la implementación.

III Técnicas Avanzadas

Hay multitud de técnicas avanzadas de React de las que es bueno que seas consciente. Haciendo buenas pruebas y escribiendo bien tu código harás que éste sea más robusto ante cambios. Será más sencillo de desarrollar si tienes el andamiaje correcto sujetando tu aplicación.

Aplicar estilos en React es, por si mismo, un asunto complicado. Hay muchas formas de conseguirlo y no existe ningún consenso claro acerca de cuál es la forma correcta de hacerlo con React. Te daré una buena idea de cuál es la situación actual.

Debido a que no hay una forma correcta de estructurar proyectos hechos con React, te daré algunas ideas de cómo conseguirlo. Es mejor ser pragmático y aproximar una estructura que sea la más correcta para tí.

14. Probando React

He decidido publicar una versión TL;DR (demasiado largo, no lo leas) de este capítulo para la comunidad para así animar a la gente a apoyar mi trabajo. Esto me permitirá desarrollar más contenido, así que realmente es una estrategia ganar-ganar.

Puedes acceder al capítulo completo comprando una copia a través de Leanpub[1]. Los siguientes puntos te darán una idea del contenido del capítulo.

14.1 TL;DR

- Técnicas básicas de testing, incluido test unitario, test de aceptación, test basado en propiedades, y test basado en mutaciones.
- Los tests unitarios nos permiten *determinar* ciertas verdades.
- Los tests de aceptación nos permiten probar aspectos cualitativos de nuestro sistema.
- Los tests basados en propiedades (echa un vistazo a QuickCheck[2]) son más genéricos y nos permiten cubrir un mayor rango de valores con más facilidad. Esos tests son más difíciles de probar.
- Los tests basados en mutaciones permiten probar los tests. Desafortunadamente, todavía no es una técnica particularmente popular en JavaScript.
- La aplicación[3] de Cesar Andreu tiene una buena configuración de test (Mocha/Karma/Istanbul).
- La cobertura del código nos permiten saber qué partes del código no están siendo probadas. Sin embargo, esto no nos da ninguna medida de la calidad de nuestros tests.

[1] https://leanpub.com/survivejs_react
[2] https://hackage.haskell.org/package/QuickCheck
[3] https://github.com/cesarandreu/web-app

- React Test Utilities[4] nos brinda una buena manera de escribir tests unitarios para nuestros componentes. Hay APIs más sencillas, como jquense/react-testutil-query[5].
- Alt tiene buenos mecanismos para probar acciones[6] y almacenes[7].
- El testing te dá confianza, lo cual se convierte en algo particularmente importante a medida que la base del código crece, ya que se vuelve más difícil romper cosas sin darte cuenta.

[4] https://facebook.github.io/react/docs/test-utils.html
[5] https://github.com/jquense/react-testutil-query
[6] http://alt.js.org/docs/testing/actions/
[7] http://alt.js.org/docs/testing/stores/
[8] https://leanpub.com/survivejs-react

15. Tipado con React

He decidido publicar una versión TL;DR (demasiado largo, no lo leas) de este capítulo para la comunidad para así animar a la gente a apoyar mi trabajo. Esto me permitirá desarrollar más contenido, así que realmente es una estrategia ganar-ganar.

Puedes acceder al capítulo completo comprando una copia a través de Leanpub[1]. Los siguientes puntos te darán una idea del contenido del capítulo.

15.1 TL;DR

- propTypes[2] son estupendos. Utilízalos para mejorar la mantenibilidad de tu aplicación.
- `propTypes` te mostrarán algunos pequeños errores durante el desarrollo, pero se descartan en la compilación para producción para mejorar el rendimiento.
- Flow[3] va un paso más allá. Te da una sintaxis básica que te permite tipar gradualmente tu código JavaScript.
- Mientras que `flow` es un analizador estático que debes ejecutar de forma separada, babel-plugin-typecheck[4] puede hacer verificaciones durante el desarrollo.
- El lenguaje TypeScript[5] de Microsoft es otra alternativa más. Desde la versión 1.6 tiene soporte para JSX.

[1] https://leanpub.com/survivejs_react
[2] https://facebook.github.io/react/docs/reusable-components.html
[3] http://flowtype.org
[4] https://github.com/codemix/babel-plugin-typecheck
[5] http://www.typescriptlang.org/
[6] https://leanpub.com/survivejs-react

16. Aplicando Estilo a React

Históricamente las páginas web se han dividido en marcado HTML, estilo (CSS) y lógica (JavaScript). Gracias a React y a otras aproximaciones similares hemos empezado a cuestionar esta división. Todavía queremos separar los ámbitos de alguna manera, pero la forma de dividirlo parte de ejes diferentes.

Este cambio de mentalidad nos ha llevado a nuevas formas de pensar sobre cómo aplicar estilos. Con React todavía estámos tratando de encontrar la mejor manera, aunque algunos patrones iniciales hayan surgido ya. Como resultado, resulta difícil dar una recomendación definitiva en este momento. En su lugar, voy a mostrar algunas aproximaciones para que las conozcas y puedas elegir la que más te convenga.

16.1 Estilo a la Vieja Usanza

La aproximación a la vieja usanza consiste en dejar algunos *ids* y *classes* por ahí, configurar reglas en el CSS, y esperar lo mejor. En CSS todo tiene, por defecto, ámbito global. Las definiciones anidadas (por ejemplo, `.main .sidebar .button`) crean una lógica implícita en los estilos. Ambas características incrementan la complejidad a medida que el proyecto va creciendo. Esta aproximación puede ser aceptable al comenzar, pero a medida que vayas desarrollando, irás queriendo migrar a otra solución.

16.2 Metodologías CSS

¿Qué ocurre cuando tu aplicación comienza a crecer y se añaden nuevos conceptos? Los selectores CSS son globales. El problema se vuelve incluso peor si tienes que lidiar con el orden el que se cargan. Si hay varios selectores iguales, la última declaración es la que gana, a menos que haya un `!important` en algún lugar. Se vuelve complejo muy rápidamente.

Podemos luchar contra este problema haciendo que los selectores sean más específicos, usando reglas de nombrado, etc. Esto simplemente retrasa lo inevitable. Ya que han sido muchas las personas que han combatido contra este problema durante mucho tiempo, algunas metodologías han emergido.

Particularmente, OOCSS[1] (Object-Oriented CSS), SMACSS[2] (Scalable and Modular Approach for CSS), y BEM[3] (Block Element Modifier) son bien conocidas. Cada una de ellas soluciona los problemas de CSS a su propia manera.

BEM

El origen de BEM reside en Yandex. La meta de BEM es la de permitir que existan componentes reutilizables y compartir código. Sitios como Get BEM[4] te pueden ayudar a entender la metodología con más detalle.

El mantener nombres de clases largos tal y como BEM requiere puede ser duro. Es por ello que han aparecido varias librerías que pueden hacerlo más sencillo. Para React, algunas de ellas son react-bem-helper[5], react-bem-render[6], y bem-react[7].

Ten en cuenta que postcss-bem-linter[8] te permite analizar tu CSS para ver si cumple con BEM.

OOCSS y SMACSS

Al igual que BEM, tanto OOCSS como SMACSS tienen sus propias convenciones y metodologías. En el momento de escribir esto, no existen librerías específicas para React de OOCSS o SMACSS.

[1] http://oocss.org/
[2] https://smacss.com/
[3] https://en.bem.info/method/
[4] http://getbem.com/
[5] https://www.npmjs.com/package/react-bem-helper
[6] https://www.npmjs.com/package/react-bem-render
[7] https://www.npmjs.com/package/bem-react
[8] https://www.npmjs.com/package/postcss-bem-linter

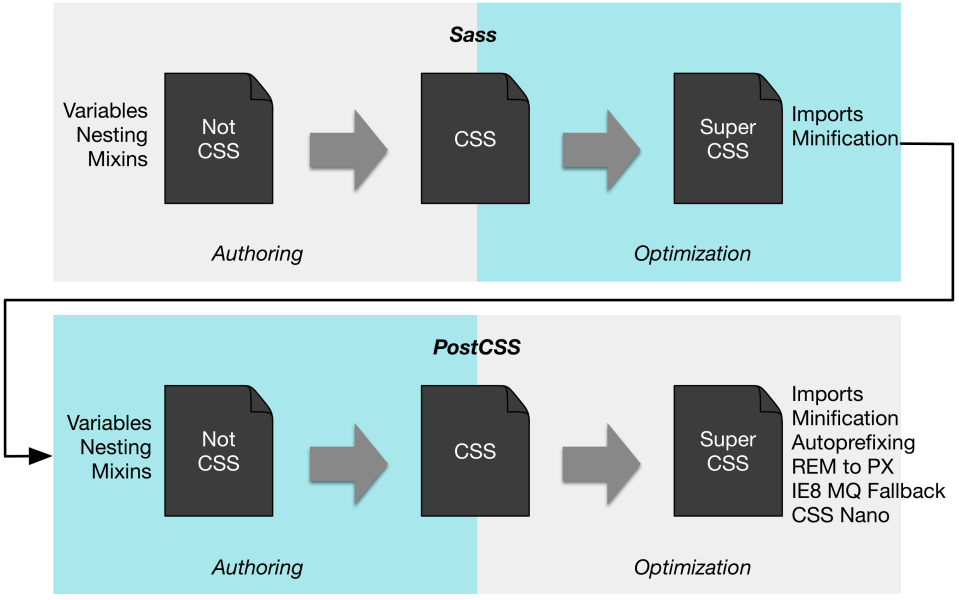
Pros y Contras

El beneficio principal de adoptar una metodología es que estructura tu proyecto. Las metodologías resuelven algunos problemas básicos y te ayudan a desarrollar buen software a largo plazo. Las convenciones que traen a un proyecto ayudan al mantenimiento del mismo y son menos propensas a provocar un desastre.

Por el contrario, una vez adoptas una, te será muy difícil migrar a otra.

Las metodologías también traen sus propias particularidades (p.e. esquemas de nombrado complejos). Esto puede hacer que algunas cosas se vuelvan más complicadas de lo que deberían ser. No necesariamente arreglan los mayores problemas sino que, a veces, simplemente los rodean.

16.3 Procesadores CSS



Procesadores CSS

El CSS plano carece de ciertas funcionalidades que podrían hacer que el mantenimiento fuese más sencillo. Considera algo básico como variables, anidamientos, mixins, operaciones matemáticas o funciones relacionadas con colores. Estaría bien poder olvidar los prefijos específicos para cada navegador. Son pequeñas cosas con las que te encuentras pronto y que hacen que sea molesto generar CSS plano.

A veces puede que veas términos como *preprocesador* o *postprocesador*. Stefan Baumgartner[9] llama a estas herramientas simplemente *procesadores de CSS*. La imagen anterior basada en el trabajo de Stefan muestra el asunto. Las herramientas operan tanto a nivel de autor como de optimización. Con nivel de autor nos referimos a que hace que sea fácil escribir CSS. Las características de optimización hacen que

[9] https://medium.com/@ddprrt/deconfusing-pre-and-post-processing-d68e3bd078a3

el CSS plano generado esté optimizado para los navegadores.

Lo interesante aquí es que puede que quieras utilizar varios procesadores de CSS. La imagen de Stefan ilustra cómo puedes escribir CSS fácilmente con Sass y aún así optimizarlo con PostCSS. Por ejemplo, puede hacer *autoprefix* de tu código CSS para que no te tengas que preocupar de poner prefijos por navegador nunca más.

Puedes usar procesadores comunes como Less[10], Sass[11], Stylus[12], o PostCSS[13] con React.

cssnext[14] es un plugin de PostCSS que te permite experimentar el futuro ahora. Hay algunas restricciones , pero puede merecer la pena probarlo. La ventaja de PostCSS y cssnext es que estarás programando literalmente en el futuro, A medida que los navegadores mejoren y adopten los estándares no tendrás que preocuparte de hacer migraciones.

Pros y Contras

Comparado con CSS plano, los procesadores dejan muchas cosa encima de la mesa. Lidian con ciertas molestias (p.e. el autoprefixing) a medida que mejoran tu productividad. PostCSS es más granular por definición y te permite utilizar únicamente las características que necesitas. Los procesadores como Less o Sass son muy útiles. Ambas aproximaciones pueden ser utilizadas juntas, de tal modo que puedes apoyarte en Sass y aplicar algunos plugins de PostCSS cuando sea necesario.

En nuestro projecto podemos aprovecharnos de cssnext incluso si no hemos hecho cambios en nuestro CSS. Gracias al autoprefixing, las esquinas redondeadas de los carriles se verán mejor en navegadores antiguos. Es más, podemos parametrizar estos estilos gracias al uso de parámetros.

[10] http://lesscss.org/
[11] http://sass-lang.com/
[12] https://learnboost.github.io/stylus/
[13] http://postcss.org/
[14] https://cssnext.github.io/

16.4 Aproximaciones Basadas en React

Existen algunas alternativas extra con React. ¿Qué ocurre si todo lo que habiamos ideado sobre estilos era erróneo?. CSS es poderoso, pero se puede volver un lío inmantenible sin algo de disciplina. ¿Dónde podemos trazar la separación entre CSS y JavaScript?

Hay varias aproximaciones para React que nos permiten aplicar estilos a nivel de componente. Puede parecer un sacrilegio, pero React, rebelde como es, nos puede llevar allí.

Estilos en Línea al Rescate

Irónicamente, la forma en la que las soluciones que utilizan React resuelven el problema de los estilos es a través de estilos en línea. Deshacerse de los estilos en línea fue una de las principales razones para el uso de múltipls archivos CSS separados, pero ahora han vuelto. Esto significa que, en vez de tener algo como esto:

```
render(props, context) {
  const notes = this.props.notes;

  return <ul className='notes'>{notes.map(this.renderNote)}</ul>;
}
```

y acompañarlo de CSS, tendremos algo como esto:

```
render(props, context) {
  const notes = this.props.notes;
  const style = {
    margin: '0.5em',
    paddingLeft: 0,
    listStyle: 'none'
  };

  return <ul style={style}>{notes.map(this.renderNote)}</ul>;
}
```

Como ocurre con los nombres de los atributos en HTML, usaremos la convención camel case para las propiedades CSS.

Ahora que estamos aplicando estilos a nivel de componente podemos implementar la lógica que modifica estos estilos fácilmente. Una forma clásica de hacer esto ha sido alterar los nombres de las clases basándonos en el aspecto que queremos tener. Ahora podemos ajustar las propiedades que queramos directamente.

Sin embargo, hemos perdido algo por el camino. Ahora nuestros estilos están fuertemente ligados a nuestro código JavaScript. Va a ser difícil hacer cambios de mucha envergadura sobre nuestra base del código ya que vamos a tener que modificar un montón de componentes para ello.

Podemos tratar de hacer algo para ello inyectando algunos estilos mediante props. Un estilo puede adaptar su propio estilo basado en uno que reciba. Esto se puede mejorar más adelante mediante convenciones que permitan que ciertas partes de la configuración de los estilos lleguen a ciertas partes específicas de los componentes. Es simplemente reinventar los selectores a una pequeña escala.

¿Qué hacemos con cosas como los media queries?. Esta inocente aproximación no se encarga de ello. Afortunandamente hay gente que ha desarrollado librerías que solucionan estos problemas por nosotros.

Según Michele Bertoli las características básicas de estas librerías son

- Autoprefixing - p.e., para `border`, `animation`, `flex`.
- Pseudo clases - p.e., `:hover`, `:active`.

- Media queries - p.e., `@media (max-width: 200px)`.
- Estilos como Objetos - Revisa el ejemplo anterior.
- Extracción de estilos CSS - Es útil para poder partir un fichero CSS grande en ficheros CSS pequeños que ayuden con la primera carga de la página. Esto evita que veamos la página sin estilos al entrar (FOUC).

Vamos a ver algunas de las librerías disponibles para que te hagas una idea de cómo funcionan. Echa un vistazo a la list de Michele[15] para tener una mejor visión de la situación.

Radium

Radium[16] tiene algunas ideas valiosas que merece la pena destacar. Lo más importante es que facilita las abstracciones necesarias para poder lidiar con media queries y pseudo clases (p.e. `:hover`). Expande la sintaxis básica como sigue:

```
const styles = {
  button: {
    padding: '1em',

    ':hover': {
      border: '1px solid black'
    },

    '@media (max-width: 200px)': {
      width: '100%',

      ':hover': {
        background: 'white',
      }
    }
  },
```

[15] https://github.com/MicheleBertoli/css-in-js
[16] http://projects.formidablelabs.com/radium/

```
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
};

...

<button style={[styles.button, styles.primary]}>Confirm</button>
```

Para que la propiedad `style` funcione deberás anotar tus clases usando el decorador `@Radium`.

React Style

React Style[17] utiliza la misma sintaxis que React Native StyleSheet[18]. Alarga la definición básica introduciendo algunas claves adicionales para cada fragmento.

```
import StyleSheet from 'react-style';

const styles = StyleSheet.create({
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  },
  button: {
    padding: '1em'
  },
```

[17] https://github.com/js-next/react-style
[18] https://facebook.github.io/react-native/docs/stylesheet.html#content

```
  // media queries
  '@media (max-width: 200px)': {
    button: {
      width: '100%'
    }
  }
});

...

<button styles={[styles.button, styles.primary]}>Confirm</button>
```

Como puedes ver, podemos uasr fragmentos individuales para tener el mismo efecto que teniamos con Radium. Además, los media queries también están soportados. React Style espera que manipules los estados del navegador (p.e. `hover`) mediante JavaScript. Las animaciones con CSS no funcionarán, es mejor usar alguna otra solución para ello.

El plugin de React Style para Webpack[19] puede extraer las declaraciones del CSS en un paquete aparte. Ahora estamos más cerca de lo que ha utilizado todo el mundo, pero no tenemos las cascadas, aunque mantenemos la declaración de los estilos a nivel de componente.

JSS

JSS[20] es un compilador de JSON a hoja de estilos. Puede ser una forma útil de representar estilos usando estructuras JSON ya que tiene un espacio de nombres sencillo. También es posible realizar tranformaciones en el JSON para obtener más funcionalidad, como el autoprefixing. JSS tiene una interfaz para plugins con la que hacer este tipo de cosas.

JSS puede utilizarse con React a través de react-jss[21]. Puedes usar *react-jss* de este modo:

[19] https://github.com/js-next/react-style-webpack-plugin
[20] https://github.com/jsstyles/jss
[21] https://www.npmjs.com/package/react-jss

```
...
import classnames from 'classnames';
import useSheet from 'react-jss';

const styles = {
  button: {
    padding: '1em'
  },
  'media (max-width: 200px)': {
    button: {
      width: '100%'
    }
  },
  primary: {
    background: 'green'
  },
  warning: {
    background: 'yellow'
  }
};

@useSheet(styles)
export default class ConfirmButton extends React.Component {
  render() {
    const {classes} = this.props.sheet;

    return <button
      className={classnames(classes.button, classes.primary)}>
        Confirm
      </button>;
  }
}
```

Hay un jss-loader[22] para Webpack.

React Inline

React Inline[23] es un enfoque intereasnte para aplicar estilos. Genera CSS basado en la prop `className` de los elementos que lo usen. El ejemplo anterior puede ser adaptado para React Inline de esta manera:

```
import cx from 'classnames';
...

class ConfirmButton extends React.Component {
  render() {
    const {className} = this.props;
    const classes = cx(styles.button, styles.primary, className);

    return <button className={classes}>Confirm</button>;
  }
}
```

Por desgracia, se basa en su propia herramienta personalizada para generar código de React y el CSS que necesita para trabajar.

jsxstyle

El jsxstyle[24] de Pete Hunt trata de mitigar algunos de los problemas React Style. Como has podido ver en los ejemplos anteriores, todavía tenemos las definiciones de los estilos separadas del lenguaje de marcado de los componentes. jsxstyle une ambos conceptos. Observa el siguiente ejemplo:

[22] https://www.npmjs.com/package/jss-loader
[23] https://github.com/martinandert/react-inline
[24] https://github.com/petehunt/jsxstyle

```
// PrimaryButton component
<button
  padding='1em'
  background='green'
>Confirm</button>
```

Esta aproximación todavía está en fase inicial. Por ejemplo, no hay soporte para media queries. En lugar de definir modificadores como antes, acabarás definiendo más componentes con los que dar cabida a tus casos de uso.

Al igual que con React Style, jsxstyle tieen un cargador de Webpack que puede extraer CSS en un fichero separado.

16.5 Módulos CSS

Como si no hubiera suficientes opciones de estilo para React, hay una más que merece la pena mencionar. Módulos CSS[25] parte de la premisa de que los estilos CSS deben ser locales por defecto. Los estilos globales deben ser tratadas como un caso especial. El post The End of Global CSS[26] de Mark Dalgleish entra en más detalles sobre esto.

De forma resumida, si se te hace difícil usar estilos globales, tienes que apañártelas para resolver el mayor problemas de las CSS. Esta aproximación te permite desarrollar CSS como hemos estado haciendo hasta ahora, solo que esta vez el ámbito se reduce a un contexto más seguro y localizado por defecto.

Esto en sí mismo soluciona una gran cantidad de los problemas que las librerias anteriores trataban de resolver a su propia forma. Si necesitas estilos globales todavía puedes tenerlos, es probable que queramos tener ciertos estilos que apliquen a alto nivel, después de todo. Esta vez seremos explícitos en ello.

Para qur te hagas una idea mejor, fíjate en el siguiente ejemplo:

style.css

[25] https://github.com/css-modules/css-modules

[26] https://medium.com/seek-ui-engineering/the-end-of-global-css-90d2a4a06284

```
.primary {
  background: 'green';
}

.warning {
  background: 'yellow';
}

.button {
  padding: 1em;
}

.primaryButton {
  composes: primary button;
}

@media (max-width: 200px) {
  .primaryButton {
    composes: primary button;

    width: 100%;
  }
}
```

button.jsx

```
import styles from './style.css';

...

<button className=`${styles.primaryButton}`>Confirm</button>
```

Como puedes ver, esta aproximación trata de encontrar el equilibrio entre aquello que a la gente le resulta familiar con librerías específicas para React. Es por ello que no

me sorprende que esta aproximación sea muy popular aunque todavía sea reciente. Echa un ojo a la demo de CSS Modules de Webpack[27] para ver más ejemplos.

Puedes usar procesadores, como Sass, junto con Módulos CSS, en caso de que busques tener más funcionalidad.

gajus/react-css-modules[28] hace que sea más sencillo todavía usar Módulos CSS con React. Con él, no tendrás que referencia al objeto `styles` nunca más, y no estás obligado a poner nombres usando camelCase.

Glen Maddern discute este tema con mucho más detalle en si artículo llamado CSS Modules - Bienvenidos al Futuro[29].

16.6 Conclusión

Es fácil probar varias aproximaciones de estilos con React. Puedes hacerlo, yendo desde CSS plano hasta configuraciones más complejas. Es sencillo probar distintas alternativas.

Las aproximaciones te permiten dejar los estilos a nivel de componente. Esto nos ofrece un contraste interesante con respecto a las aproximaciones convencionales en las que las CSS se mantienen separadas. Lidiar con la lógica específica del componente se vuelve más sencillo. Puede que pierdas algo del poder que te dan las CSS, pero obtendrás algo más sencillo de entender que puede ser más difícil de romper.

Los Módulos CSS buscan un equilibrio entre un enfoque convencional y un enfoque específico para React. Incluso aún siendo un recién llegado parece tener mucho

[27] https://css-modules.github.io/webpack-demo/
[28] https://github.com/gajus/react-css-modules
[29] http://glenmaddern.com/articles/css-modules

potencial. El mayor beneficio parece ser que no perderás mucho durante el proceso. Representa un paso adelante que ha sido comúnmente usado.

Todavía no existen buenas prácticas, y todavía estamos tratando de encontrar la mejor manera de aplicar estilos con React. Tendrás que experimentar un poco por tí mismo para hacerte una idea de qué es lo que mejor encaja en tu caso particular.

17. Estructurando Proyectos con React

React no fuerza a tener ninguna estructura de proyecto en particular. Lo bueno de ello es que te permite tener la estructura que mejor se adapte a lo que necesites. Lo malo es que no es posible tener una estructura ideal que funcione en todos los proyectos. Por tanto, voy a darte algo de inspiración que podrás usar cuando pienses en estructuras.

17.1 Un Directorio por Concepto

Nuestra aplicación Kanban tiene una estructura plana como la siguiente:

```
├── actions
│   ├── LaneActions.js
│   └── NoteActions.js
├── components
│   ├── App.jsx
│   ├── Editable.jsx
│   ├── Lane.jsx
│   ├── Lanes.jsx
│   ├── Note.jsx
│   └── Notes.jsx
├── constants
│   └── itemTypes.js
├── index.jsx
├── libs
│   ├── alt.js
│   ├── persist.js
│   └── storage.js
```

```
├── main.css
└── stores
    ├── LaneStore.js
    └── NoteStore.js
```

Es suficiente para nuestro propósito, pero hay algunas alternativas interesantes:

- Un fichero por concepto - Perfecto para prototipos pequeño. Puedes dividirlo a medida que la aplicación se vaya haciendo más seria.
- Un directorio por componente - Es posible dejar componentes en directorios que pasen a ser de su propiedad. Aunque quizá sea la aproximación más pesada, tiene algunas ventajas interesantes que veremos pronto.
- Un directorio por vista - Esta aproximación se vuelve relevante una vez quieres introducir enrutamiento en tu aplicación.

Hay más alternativas pero éstas cubren los casos más comunes. Siempre hay espacio para hacer ajustes en base a las necesidades de tu aplicación.

17.2 Un Directorio por Componente

Si dejamos nuestros componentes en directorios que pasen a ser de su propiedad podemos acabar teniendo algo como esto:

```
├── actions
│   ├── LaneActions.js
│   └── NoteActions.js
├── components
│   ├── App
│   │   ├── App.jsx
│   │   ├── app.css
│   │   ├── app_test.jsx
│   │   └── index.js
│   ├── Editable
```

```
|   |   ├── Editable.jsx
|   |   ├── editable.css
|   |   ├── editable_test.jsx
|   |   └── index.js
...
|   └── index.js
├── constants
|   └── itemTypes.js
├── index.jsx
├── libs
|   ├── alt.js
|   ├── persist.js
|   └── storage.js
├── main.css
└── stores
    ├── LaneStore.js
    └── NoteStore.js
```

Puede ser más pesada que la solución que tenemos actualmente. Los ficheros *index.js* sirven de punto de entrada para los componentes. Introducen ruido pero simplifican los imports.

Sin embargo, hay algunos beneficios interesantes de esta aproximación:

- Podemos utilizar tecnologías como los Módulos CSS para aplicar estilos en cada componente de forma independiente.
- Dado que cada componente tiene un pequeño "paquete" de sí mismo, puede ser más sencillo sacarlo del proyecto. De este modo puedes crear componentes genéricos en cualquier lugar y utilizarlos en muchas aplicaciones.
- Podemos definir tests unitarios a nivel de componente. Esto te anima a hacer tests, y todavía podemos hacer tests de alto nivel de la aplicación exactamente igual que antes.

Puede ser interesante tratar de dejar las acciones y los almacenes también en `components`. O pueden seguir un esquema de directorios similar. La ventaja de todo esto es que te permiten definir tests unitarios de una forma similar.

Esta configuración no es suficiente si quieres que la aplicación tenga varias vistas. Necesitamos algo más que nos ayude.

gajus/create-index[1] es capaz de generar los ficheros *index.js* automáticamente a medida que vas desarrollando.

17.3 Un Directorio por Vista

Tener varias vistas es un reto por sí mismo. Para comenzar, debes definir un esquema de enrutamiento. react-router[2] es una solución popular que cumple este propósito. Además de la definición del esquema, necesitarás definir qué quieres mostrar en cada vista. Puedes tener vistas separadas para la página principal de tu aplicación, otra para el registro, el tablero de Kanban, etc, enlanzándolas con cada ruta.

Estos requisitos implican nuevos conceptos que deben ser introducidos en nuestra estructura. Una forma de lidiar con el enrutado es crear un componente `Routes` que coordine qué vista hay que mostrar en base a la ruta actual. En lugar de `App` podemos tener varias vistas en su lugar. He aquí el aspecto que podría tener una posible estructura:

```
├── components
│   ├── Note
│   │   ├── Note.jsx
│   │   ├── index.js
│   │   ├── note.css
│   │   └── note_test.jsx
│   ├── Routes
│   │   ├── Routes.jsx
│   │   ├── index.js
│   │   └── routes_test.jsx
│   └── index.js
```

[1] https://github.com/gajus/create-index
[2] https://github.com/rackt/react-router

```
...
├── index.jsx
├── main.css
└── views
    ├── Home
    |   ├── Home.jsx
    |   ├── home.css
    |   ├── home_test.jsx
    |   └── index.js
    ├── Register
    |   ├── Register.jsx
    |   ├── index.js
    |   ├── register.css
    |   └── register_test.jsx
    └── index.js
```

La idea es la misma que antes, aunque esta vez tenemos más partes que coordinar. La aplicación comienza desde `index.jsx`, que invocará `Routes`, que decidirá qué vista mostrar, Tras esto el flujo sigue como hasta ahora.

Esta estructura puede escalar mejor, pero también tiene sus límites. Una vez el proyecto comience a crecer puede que quieras introducir nuevos componentes en él. Puede ser natural introducir un concepto, como "funcionalidad", entre las vistas y los componentes.

Por ejemplo, puede que quieras tener un `LoginModal` resultón que se muestre en ciertas vistas sólo si la sesión del usuario ha caducado. Puede estar compuesto por componentes de más bajo nivel. De nuevo, las características comunes pueden ser desplazadas fuera del proyecto como paquetes si ves que tienen potencial para ser reusadas.

17.4 Conclusión

No hay una forma única de estructurar tu proyecto con React. Dicho esto, es uno de esos aspectos en los que merece la pena pensar. Encontrar una estructura que nos

ayude merece la pena. Una estructura clara ayuda al mantenimiento y hace que tu proyecto sea más entendible por otros.

Puedes hacer que la estructura evolucione a medida que avanzas. Las estructuras muy pesadas puede que te retrasen. A medida que el proyecto evoluciona, debe hacerlo también su estructura. Es una de esas cosas en las que merece la pena meditar acerca de cómo afecta al desarrollo.

Apéndices

Como no todo lo que merece la pena discutir encaja en un libro como este, he recopilado material relacionado y lo he colocado en pequeños apéndices. Estos apéndices apoyan el contenido principal y explican algunos temas, tales como las características del lenguaje, en un mayor grado de detalle. Además. hay consejos y soluciones de problemas al final.

Características del Lenguaje

ES6 (o ES2015) ha sido sin lugar a dudas el mayor cambio en JavaScript en mucho tiempo. Como resultado, muchas funcionalidades nuevas han sido añadidas. El propósito de este apéndie es mostrar las características utilizadas en este libro de forma individual para que sea más fácil entender cómo funcionan. En lugar de ir a la especificación completa[3], me centraré únicamente en el subconunto de características usadas en el libro.

Módulos

ES6 introduce una declaración formal de módulos. Anteriormente había que utilizar soluciones ad hoc o cosas como AMD o CommonJS. Las declaraciones de módulos de ES6 son analizables estáticamente, lo cual es útil para no cargar código sin utilizar símplemente analizando la estructura de imports.

`import` y `export` Sencillos

Para mostrarte un ejemplo de cómo exportar directamente un módulo echa un vistazo al código siguiente:

persist.js

```
import makeFinalStore from 'alt-utils/lib/makeFinalStore';

export default function(alt, storage, storeName) {
  ...
}
```

index.js

[3] http://www.ecma-international.org/ecma-262/6.0/index.html

```
import persist from './persist';

...
```

`import` y `export` Múltiple

A menudo puede ser útil utilizar módulos como un espacio de nombres con varias funciones:

math.js

```
export function add(a, b) {
  return a + b;
}

export function multiply(a, b) {
  return a * b;
}

export function square(a) {
  return a * a;
}
```

De forma alternativa puedes escribir el módulo de la forma siguiente:

math.js

```
const add = (a, b) => a + b;
const multiple = (a, b) => a * b;

// Puedes omitir los () si quieres ya que tiene sólo un parámetro
const square = a => a * a;

export {
  add,
  multiple,
  // Puedes crear alias
  multiple as mul
};
```

El ejemplo utiliza la *sintaxis de la flecha gorda*. Esta definición puede ser consumida desde un import de la manera siguiente:

index.js

```
import {add} from './math';

// También podríamos usar todos los métodos de math con
// import * as math from './math';
// math.add, math.multiply, ...

...
```

Ya que la sintaxis de los módulos de ES6 es analizable estáticamente, es posible usar heramientas como analyze-es6-modules[4].

Imports con Alias

A veces puede ser útil hacer alias de imports. Por ejemplo:

[4] https://www.npmjs.com/package/analyze-es6-modules

```
import {actions as TodoActions} from '../actions/todo'

...
```

`as` te permite evitar conflictos de nombrado.

Webpack `resolve.alias`

Los empaquetadores, como Webpack, pueden dar funcionalidad más allá de esto. Puedes definir un `resolve.alias` para alguno de tus directorios de módulos, por ejemplo. Esto te permite usar un import como `import persist from 'libs/persist';` independientemente de dónde estés importando. Un simple `resolve.alias` puede ser algo como esto:

```
...
resolve: {
  alias: {
    libs: path.join(__dirname, 'libs')
  }
}
```

La documentación oficial describe las posibles alternativas[5] con todo lujo de detalles.

Clases

Al contrario de como ocurre con otros lenguajes ahí fuera, JavaScript utiliza una herencia basada en prototipos en lugar de herencia basada en clases. Ambas aproximaciones tienen sus ventajas. De hecho, puedes imitar un modelo basado en clases utilizando uno basado en prototipos. Las clases de ES6 simplemente son azúcar sintáctico de los mecanismos de JavaScript, ya que internamente sigue utilizando el sistema antiguo, solo que parece algo distinto para el programador.

React permite definición de componentes basados en clases. No todos estamos de acuerdo en que sea algo bueno. Dicho esto, la definición puede estar bien siempre que no abuses de ella. Para darte un ejemplo sencillo, observa el código siguiente:

[5] https://webpack.github.io/docs/configuration.html#resolve-alias

```
import React from 'react';

export default class App extends React.Component {
  constructor(props) {
    super(props);

    // This es una propiedad fuera del funcionamiento de React.
    // Si no necesitas lanzar render() cuando cambia puede funcionar.
    this.privateProperty = 'private';

    // Estado específico de React. Puedes cambiarlo con `this.setSta\
te`, lo
    // cual podrá llamar a `render()`.
    this.state = {
      name: 'Class demo'
    };
  }
  render() {
    // Use estas propiedades de alguna manera.
    const privateProperty = this.privateProperty;
    const name = this.state.name
    const notes = this.props.notes;

    ...
  }
}
```

Quizá la mayor ventaja de la aproximación basada en clases sea el hecho de que reduce algo de complejidad, especialmente cuando involucra los métodos del ciclo de vida de React.

Clases y Módulos

Como vimos antes, los módulos de ES6 permiten hacer `export` e `import` de uno o varios objetos, funciones o incluso clases. También puedes usar `export default`

`class` para exportar una clase anónima o exportar varias clases desde el mismo módulo usando `export class className`.

Note.jsx

```
export default class extends React.Component { ... };
```

Notes.jsx

```
import Note from './Note.jsx';
...
```

También puedes usar `export class className` para exportar varias clases nombradas de un único módulo.

Components.jsx

```
export class Note extends React.Component { ... };

export class Notes extends React.Component { ... };
```

App.jsx

```
import {Note, Notes} from './Components.jsx';
...
```

Se recomienda que tengas las clases separadas en módulos diferentes.

Propiedades de las Clases e Iniciadores de Propiedades

Las clases de ES6 no enlazan sus métodos por defecto. Esto puede suponer un problema a veces, ya que puede que quieras acceder a las propiedades de la instancia. Hay características experimentales conocidas como las propiedades de las clases y los iniciadores de propiedades[6] que arreglan este problema. Sin ellos podríamos escribir algo como:

[6] https://github.com/jeffmo/es-class-static-properties-and-fields

```
import React from 'react';

class App extends React.Component {
  constructor(props) {
    super(props);

    this.renderNote = this.renderNote.bind(this);
  }
  render() {
    ...

    return this.renderNote();
  }
  renderNote() {
    // Dado que renderNote ha sido enlazado, podemos usar `this` com\
o esperamos
    return <div>{this.props.value}</div>;
  }
}
App.propTypes = {
  value: React.PropTypes.string
};
App.defaultProps = {
  value: ''
};

export default App;
```

Utilizando propiedades de clases e iniciadores de propiedades podemos escribir algo más limpio en su lugar:

```
import React from 'react';

export default class App extends React.Component {
  // la definición de propType mediante propiedades estáticas de la \
clase
  static propTypes = {
    value: React.PropTypes.string
  }
  static defaultProps = {
    value: ''
  }
  render() {
    ...

    return this.renderNote();
  }
  // El iniciador de propiedades se encarga del `bind`
  renderNote = () => {
    // Dado que renderNote ha sido enlazado, podemos usar `this` com\
o esperamos
    return <div>{this.props.note}</div>;
  }
}
```

Ahora que nos hemos llevado la declaración a nivel de método el código se lee mejor. He decidido usar esta característica en este libro principalmente por este motivo. Hay menos de lo que preocuparse.

Funciones

JavaScript ha sido tradicionalmente muy flexible con respecto a las funciones. Para que te hagas una mejor idea, aquí tienes la implementación de `map`:

```
function map(cb, values) {
  var ret = [];
  var i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map(function(v) {
  return v * 2;
}, [34, 2, 5]); // salen [68, 4, 10]
```

En ES6 podríamos haberlo escrito de esta manera:

```
function map(cb, values) {
  const ret = [];
  const i, len;

  for(i = 0, len = values.length; i < len; i++) {
    ret.push(cb(values[i]));
  }

  return ret;
}

map((v) => v * 2, [34, 2, 5]); // salen [68, 4, 10]
```

La implementación de `map` es más o menos lo mismo. La parte interesante es la forma en la que lo llamamos. En concreto, `(v) => v * 2` es fascinante. En lugar de tener que escribir `function` por todos lados, la sintaxis de la flecha gorda nos da un pequeño y útil atajo. Para ver más ejemplos de uso echa un vistazo a lo que sigue:

```
// Todas son equivalentes
v => v * 2;
(v) => v * 2; // Prefiero esta opción en funciones cortas
(v) => { // Usa esta si necesitas ejecutar varias sentencias
  return v * 2;
}

// Podemos enlazarlo a una variable
const double = (v) => v * 2;

console.log(double(2));

// Si quieres usar un atajo y devolver un objeto
// necesitas encapsular el objeto.
v => ({
  foo: 'bar'
});
```

El contexto de la Función Flecha

Las funciones flecha son un tanto especiales ya que no tienen un `this` propio. En su lugar, `this` apunta al ámbito del objeto invocante. Fíjate en el siguiente ejemplo:

```
var obj = {
  context: function() {
    return this;
  },
  name: 'demo object 1'
};

var obj2 = {
  context: () => this,
  name: 'demo object 2'
};
```

```
console.log(obj.context()); // { context: [Function], name: 'demo ob\
ject 1' }
console.log(obj2.context()); // {} en Node.js, Window en el navegador
```

Como puedes ver en el código anterior, la función anónima tiene un `this` que apunta a la función `context` del objeto `obj`. En otras palabras, está enlazando el ámbito del objeto `obj` a la función `context`.

Esto es así porque `this` no apunta al ámbito del objeto que lo contiene, sino al ámbito del objeto que lo invoca, como puedes ver en el siguiente fragmento de código:

```
console.log(obj.context.call(obj2)); // { context: [Function], name:\
 'demo object 2' }
```

La función flecha en el objeto `obj2` no enlaza ningún objeto a su contexto, siguendo lo que serían las reglas normales de ámbitos resolviendo la referencia al ámbito inmediatemente superior.

Incluso cuando este comportamiento parece ser un poco extraño, en realidad es útil. En el pasado, si querías acceder al contexto de la clase padre necesitabas enlazarlo o relacionarlo en una variable del estilo `var that = this;`. La introducción de la sintaxis de la función flecha ha mitigado este problema.

Parámetros de las Funciones

Históricamente, lidiar con los parámetros de las funciones ha sido algo limitado. hay varios hacks, como `values = values || [];`, pero no son particularmente buenos y son propensos a errores. Por ejemplo, el uso de `||` puede causar problemas con ceros. ES6 soluciona este problema introduciendo parámetros por defecto. De este modo, podemos escribir simplemente `function map(cb, values=[])`.

Hay más que esto y los valores por defecto pueden depender unos de otros. También puedes pasar una cantidad arbitraria de parámetros mediante `function map(cb, ...values)`. En este caso, puedes llamar a la función usando `map(a => a * 2, 1, 2, 3, 4)`. Este API puede que no sea perfecto para `map`, pero puede tener más sentido en otro escenario.

También hay medios útiles para extraer valores de los objetos enviados. Esto es muy útil con los componentes de React que se definen como funciones:

```
export default ({name}) => {
  // Interpolación de strings en ES6. ¡Observa las tildes!
  return <div>{`Hello ${name}!`}</div>;
};
```

Interpolación de Strings

Antiguamente, lidiar con strings era algo doloroso en JavaScript. Por lo general se utilizaba una sintaxis del tipo `'Hello' + name + '!'`. Sobrecargar + para alcanzar este propósito quizá no era la mejor manera ya que podia provocar comportamientos extraños. Por ejemplo, `0 + ' world'` puede devolver el string `0 world` como resultado.

Aparte de ser más clara, la interpolación de strings de ES6 permite strings multilínea. Esto es algo que la anterior sintaxis no soportaba. Observa los siguientes ejemplos:

```
const hello = `Hello ${name}!`;
const multiline = `
multiple
lines of
awesomeness
`;
```

Puede que tardes un poco hasta que te acostumbres a la tilde, pero es poderosa y menos propensa a errores.

Destructuring

Eso de `...` está relacionado con la idea de destructuring. Por ejemplo, `const {lane, ...props} = this.props;` sacará `lane` fuera de `this.props` mientras que el resto del objeto se quedará en `props`. Esta sintaxis es todavía experimental en objetos. ES6 especifica una forma oficial de poder hacer lo mismo en arrays como sigue:

```
const [lane, ...rest] = ['foo', 'bar', 'baz'];

console.log(lane, rest); // 'foo', ['bar', 'baz']
```

El operador spread (...) es útil para concatenaciones. Verás sintaxis similar con frecuencia en ejemplos de Redux. Se basa en el experimental Object rest/spread syntax[7]:

```
[...state, action.lane];

// Esto es igual que
state.concat([action.lane])
```

La misma idea funciona en los componentes de React:

```
...

render() {
  const {value, onEdit, ...props} = this.props;

  return <div {...props}>Spread demo</div>;
}

...
```

Iniciadores de Objetos

ES6 facilita varias funcionalidades para hacer que sea más sencillo trabajar con objetos. Citando a MDN[8], fíjate en los siguientes ejemplos:

[7] https://github.com/sebmarkbage/ecmascript-rest-spread
[8] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Object_initializer

```
const a = 'demo';
const shorthand = {a}; // Lo mismo que {a: a}

// Métodos atajo
const o = {
  get property() {},
  set property(value) {},
  demo() {}
};

// Nombres de propiedades procesadas
const computed = {
  [a]: 'testing' // demo -> testing
};
```

`const`, `let`, `var`

En JavaScript, las variables son globales por defecto. `var` las enlaza a *nivel de función*, lo cual es un contraste con muchos otros lenguajes que implementan enlazamiento a *nivel de bloque*. ES6 introduce enlazamiento a nivel de bloque con `let`.

`const` también está soportado, lo que garantiza que la referencia a una variable no pueda ser cambiada. Esto, sin embargo, no significa que no puedas cambiar el contenido de la variable, así que si estás apuntando a un objeto, ¡todavía tendrás permitido cambiarlo!

Suelo utilizar `const` siempre que sea posible. Si necesito que algo sea mutable, `let` es estupendo. Es difícil encontrar un uso útil de `var` teniendo `const` y `let`. De hecho, todo el código de este libro, exceptuando el apéndice, utiliza `const`, lo que quizá pueda enseñarte lo lejos que puedes llegar con él.

Decoradores

Dado que los decoradores son todavía una funcionalidad experimental hay mucho que hablar de ellos. Hay un apéndice entero dedicado a este tema. Lee *Entendiendo los Decoradores* para más información.

Conclusión

Hay mucho más ES6 y más especificaciones que éstas. Si quieres entender la especificación mejor, ES6 Katas[9] es un buen punto en el que comenzar para aprender más. Únicamente teniendo una buena idea de lo básico podrás llegar lejos.

[9] http://es6katas.org/

Entendiendo los Decoradores

Si has usado lenguajes de programación antes, como Java o Python, puede que la idea te resulte familiar. Los decoradores son azúcar sintáctico que te permiten envolver y anotar clases y funciones. En su actual propuesta[10] (fase 1) sólo se permite la envoltura a nivel de clase y de método. Las funciones puede que sean soportadas en el futuro.

Con Babel 6 puedes habilitar este comportamiento mediante los plugins babel-plugin-syntax-decorators[11] y babel-plugin-transform-decorators-legacy[12]. El primero da soporte a nivel de sintáxis mientras que el segundo da el tipo de comportamiento que vamos a discutir ahora.

El mayor beneficio de los decoradores es que nos permiten envolver comportamiento en partes simples y reutilizables a la vez que reducimos la cantidad de ruido. Es totalmente posible programar sin ellos, sólo hacen que algunas de las tareas acaben siendo más agradecidas, como vimos con las anotaciones relacionadas con arrastrar y soltar.

Implementando un Decorador para Generar Logs

A veces es útil saber qué métodos han sido invocados. Por supuesto que puedes usar `console.log` pero es más divertido implementar `@log`. Es una forma mejor de tenerlo controlado. Observa el siguiente ejemplo:

[10] https://github.com/wycats/javascript-decorators

[11] https://www.npmjs.com/package/babel-plugin-syntax-decorators

[12] https://www.npmjs.com/package/babel-plugin-transform-decorators-legacy

```
class Math {
  @log
  add(a, b) {
    return a + b;
  }
}

function log(target, name, descriptor) {
  var oldValue = descriptor.value;

  descriptor.value = function() {
    console.log(`Calling "${name}" with`, arguments);

    return oldValue.apply(null, arguments);
  };

  return descriptor;
}

const math = new Math();

// los argumentos pasados deberían aparecer en el log
math.add(2, 4);
```

La idea es que nuestro decorador `log` envuelva la función original, lance un `console.log` y, finalmente, haga la invocación con los argumentos[13] originales. Puede que te parezca un poco extraño si nunca antes habías visto `arguments` o `apply`.

`apply` puede ser visto como otra forma de invocar una función pasándole su contexto (`this`) y sus parámetros como un array. `arguments` recibe de forma implícita todos los parámetros con los que se ha invocado a la función así que es ideal para este caso.

El logger puede ser movido a un módulo aparte. Tras ello, podemos usarlo en nuestra aplicación en aquellos lugares donde queramos mostrar el log de algunos métodos. Una vez han sido implementados, los decoradores se convierten en una herramienta poderosa.

[13] https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/arguments

El decorador recibe tres parámetros:

- `target` se relaciona con la instancia de la clase.
- `name` contiene el nombre del método que va a ser ejecutado.
- `descriptor` es la pieza más interesante ya que nos permite anotar un método y manipular su comportamiento. Puede tener el siguiente aspecto:

```
const descriptor = {
  value: () => {...},
  enumerable: false,
  configurable: true,
  writable: true
};
```

Como puedes ver, `value` hace que sea posible envolver el comportamiento. Lo demás te permite modificar el comportamiento a nivel de método. Por ejemplo, un decorador `@readonly` puede limitar el acceso. `@memoize` es otro ejemplo interesante ya que permite que los métodos implementen cacheo fácilmente.

Implementado `@connect`

`@connect` envolverá nuestro componente en otro componente. Se encargará de lidiar con la lógica de conexión (`listen/unlisten/setState`). Mantendrá internamente el estado del almacén y se lo pasará a los componentes hijos que estén siendo envueltos. Durante el proceso, enviará el estado mediante props. La siguiente implementación ilustra la idea:

app/decorators/connect.js

```
import React from 'react';

const connect = (Component, store) => {
  return class Connect extends React.Component {
    constructor(props) {
      super(props);

      this.storeChanged = this.storeChanged.bind(this);
      this.state = store.getState();

      store.listen(this.storeChanged);
    }
    componentWillUnmount() {
      store.unlisten(this.storeChanged);
    }
    storeChanged() {
      this.setState(store.getState());
    }
    render() {
      return <Component {...this.props} {...this.state} />;
    }
  };
};

export default (store) => {
  return (target) => connect(target, store);
};
```

¿Puedes ver la idea del decorador? Nuestro decorador vigila el estado del almacén. Tras ello, pasa el estado al componente contenido mediante props.

... es conocido como el operador spread[14]. Expande el objeto recibido para separar los pares clave-valor, o propiedades, como en este caso.

[14] https://github.com/sebmarkbage/ecmascript-rest-spread

Puedes conectar el decorador con `App` de este modo:

app/components/App.jsx

```
...
import connect from '../decorators/connect';

...

@connect(NoteStore)
export default class App extends React.Component {
  render() {
    const notes = this.props.notes;

    ...
  }
  ...
}
```

Llevar la lógica a un decorador nos permite mantener nuestros componentes sencillos. Ahora debería ser trivial poder añadir más almacenes y conectarlos a los componentes si quisiéramos. E incluso mejor, podriamos conectar varios almacenes a un único componente fácilmente.

Ideas para Decoradores

Podemos crear decoradores para varias desarrollar distintas funcionalidades, como es la de deshacer, de esta manera. Esto nos permite mantener nuestros componentes limpios y empujar la lógica común a algún lugar fuera de nuestra vista. Los decoradores bien diseñados pueden ser utilizados en varios proyectos.

El `@connectToStores` de Alt

Alt facilita un decorador similar conocido como `@connectToStores`. Se apoya en métodos estáticos. En lugar de ser métodos normales que están incluidos en una

instancia específica, se incluyen a nivel de clase. Esto significa que puedes llamarlos a través de la propia clase (p.e., `App.getStores()`). El siguiente ejemplo muestra cómo podemos integrar `@connectToStores` en nuestra aplicación:

```
...
import connectToStores from 'alt-utils/lib/connectToStores';

@connectToStores
export default class App extends React.Component {
  static getStores(props) {
    return [NoteStore];
  };
  static getPropsFromStores(props) {
    return NoteStore.getState();
  };
  ...
}
```

Esta aproximación es muy parecida a nuestra implementación. En realidad hace más ya que te permite conectar con varios almacenes a la misma vez. También te dá más control sobre la forma en la que puedes encajar el almacén de estados con las props.

Conclusión

Aunque todavía sean un tanto experimentales, los decoradores son una buena forma de llevar lógica allá donde pertenezca. Mejor todavía, nos dan un grado de reusabilidad mientras mantienen nuestros componentes ordenados y limpios.

Resolución de Problemas

He tratado de recopilar algunos problemas comunes aquí. Este capítulo crecerá a medida que se vayan encontrando más problemas comunes.

EPEERINVALID

Es probable que veas un mensaje como este:

```
npm WARN package.json kanban-app@0.0.0 No repository field.
npm WARN package.json kanban-app@0.0.0 No README data
npm WARN peerDependencies The peer dependency eslint@0.21 - 0.23 inc\
luded from eslint-loader will no
npm WARN peerDependencies longer be automatically installed to fulfi\
ll the peerDependency
npm WARN peerDependencies in npm 3+. Your application will need to d\
epend on it explicitly.

...

npm ERR! Darwin 14.3.0
npm ERR! argv "node" "/usr/local/bin/npm" "i"
npm ERR! node v0.10.38
npm ERR! npm  v2.11.0
npm ERR! code EPEERINVALID

npm ERR! peerinvalid The package eslint does not satisfy its sibling\
s' peerDependencies requirements!
npm ERR! peerinvalid Peer eslint-plugin-react@2.5.2 wants eslint@>=0\
.8.0
npm ERR! peerinvalid Peer eslint-loader@0.14.0 wants eslint@0.21 - 0\
```

```
.23

npm ERR! Please include the following file with any support request:
...
```

En lenguaje de los humanos significa que algún paquete, `eslint-loader` en este caso, tiene un requisito `peerDependency` demasiado estricto. Nuestro paquete ya tiene instalada una versión más reciente. Dado que la dependencia requerida de forma transitiva es más antigua que la nuestra, nos aparece este error en particular.

Hay un par de formas de solucionar esta situación:

1. Avisar al autor del paquete del problema y confiar en que incremente el rango de versiones que puede soportar.
2. Resolver el conflicto utilizando una versión que satisfaga la dependencia. En este caso, podríamos hacer que el `eslint` que utilizamos nosotros sea la versión `0.23` (`"eslint": "0.23"`), y todo el mundo debería ser feliz.
3. Hacer un fork del paquete, arreglar el rango de versiones, y utilizar tu propia versión. En este caso necesitarás tener una declaración de este tipo `"<paquete>": "<cuenta de github>/<proyecto>#<referencia>"` en tus dependencias.

Ten en cuenta que las dependencias se tratan de forma diferente a partir de npm 3. Tras esa versión, le toca al consumidor del paquete (es decir, a tí) lidiar con ello.

Warning: setState(...): Cannot update during an existing state transition

Puede que te salga esta advertencia mientras utilizas React. Es fácil que te salga si lanzas `setState()` dentro de métodos como `render()`. A veces puede ocurrir de forma indirecta. Una manera de provocar la advertencia es invocar un método en vez de enlazarlo. Por ejemplo: `<input onKeyPress={this.checkEnter()} />`. Si `this.checkEnter` utiliza `setState()`, este código fallará. En su lugar, deberías usar `<input onKeyPress={this.checkEnter} />`.

Warning: React attempted to reuse markup in a container but the checksum was invalid

Esta advertencia te puede salir de varias formas. Algunas causas comunes son:

- Has intentado montar React varias veces en el mismo contenedor. Comprueba su script de carga y asegúrate de que tu aplicación sólo se carga una vez.
- El marcado de tu plantilla no encaja con la que renderiza React. Esto puede ocurrir especialmente si estás renderizando el lenguaje de marcado inicial en un servidor.

Module parse failed

Cuando utilizas Webpack puede salir un error como este:

```
ERROR in ./app/components/Demo.jsx
Module parse failed: .../app/components/Demo.jsx Line 16: Unexpected\
 token <
```

Esto significa que hay algo que está impidiéndole a Webpack que interprete el fichero correctamente. Deberías comprobar la configuración del `loader` con cuidado. Asegúrate de que los cargadores correctos se aplican sobre los ficheros correctos. Si estás usando `include` deberías comprobar que el fichero que quieres cargar es alcanzable desde el `include`.

El Proyecto Falla al Compilar

Aún cuando todo en teoría debería funcionar, a veces los rangos de las versiones te pueden dificultar las cosas, a pesar de usar versionado semántico. Si un paquete principal se rompe, por ejemplo `babel`, y ejecutaste `npm i` en un mal momento, puede que acabes con un proyecto que no compila.

Un buen primer paso es ejecutar `npm update`. Comprobará tus dependencias y podrá las versiones más recientes. Si esto no arregla el problema puedes probar a eliminar `node_modules` (`rm -rf node_modules`) desde el directorio del proyecto y reinstalar las dependencias (`npm i`).

A menudo no estás sólo con tu problema. Por eso puede que merezca la pena mirar en los gestores de incidencias de los proyectos a ver qué puede estar pasando. Puede que encuentres una manera alternativa o una solución allí. Estos problemas suelen solucionarse rápido en los proyectos más populares.

En un entorno en producción, puede ser preferible bloquear las dependencias de producción usando `npm shrinkwrap`. La documentación oficial[15] entra más al detalle en este asunto.

[15] https://docs.npmjs.com/cli/shrinkwrap