UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO DE INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

EVALUACIÓN DEL USO DE LIBRERÍAS MODELO-VISTA-CONTROLADOR PARA EL DESARROLLO DE SERVICIOS WEB. CASO DE ESTUDIO: REACT

DANIEL REVILLA TWOSE

2016

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

Título: Evaluación del uso de librerías Modelo-Vista-Controlador para el desarrollo de servicios Web. Caso de estudio: React.

Autor: D Daniel Revilla Twose.

Tutor: D. Santiago Pavón Gómez.

Ponente: D. Daniel Revilla Twose.

Departamento: Departamento de Ingeniería de Sistemas Telemáticos.

MIEMBROS DEL TRIBUNAL

Presidente: D. Santiago Pavón Gómez.

Vocal: D. Joaquín Luciano Salvachúa Rodríguez.

Secretario: D. Gabriel Huecas Fernández-Toribio.

Suplente: D. Juan Quemada Vives.

Los miembros del tribunal arriba nombrados acuerdan otorgar la calificación de:

Madrid, a de Enero de 2016

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

GRADO DE INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

EVALUACIÓN DEL USO DE LIBRERÍAS MODELO-VISTA-CONTROLADOR PARA EL DESARROLLO DE SERVICIOS WEB.

CASO DE ESTUDIO: REACT.

DANIEL REVILLA TWOSE

2016

RESUMEN

El proyecto consiste en analizar los problemas de las librerías MVC (Model-View-Controller) existentes y que se usan para el desarrollo de servicios web, evaluando las características y aplicabilidad de las nuevas librerías que han aparecido.

Se realizará una evaluación de las distintas librerías JavaScript y se compararán con las emergentes identificando qué problemas resuelven. Se estudiarán las características, las ventajas, así como las tecnologías que podemos encontrar alrededor de React y Flux ilustrándolas con un ejemplo sencillo.

Finalmente, se ilustrará mediante un ejemplo práctico como funcionan estas tecnologías en una versión simplificada del proyecto Quiz de la asignatura Computación en Red concluyendo que ventajas e inconvenientes presenta en una aplicación real.

SUMMARY

The aim of this project is to analyse the problems found in the existing MVC (Model-View-Controller) libraries and that are use to develop Web services, evaluating the characteristics and applicability of the new libraries that have emerged.

An evaluation of the different JavaScript libraries will be performed as well as a comparison with the emergent libraries identifying which problems they solve. The characteristics, advantages as well as the technologies that can be found around React and Flux will be studied.

Finally, an example will be made showing how these technologies work in a simplified version of the Quiz project from the subject Network Computing showing in the conclusions the advantages and disadvantages found in a real application.

PALABRAS CLAVE

JavaScript, Web, React, Flux, Redux, MVC, Model, View, Controller, Servicios Web, Cliente, Servidor.

KEYWORDS

JavaScript, Web, React, Flux, Redux, MVC, Model, View, Controller, Web Services, Client, Server.

ÍNDICE DEL CONTENIDO

1. INTRODUCCIÓN Y OBJETIVOS 1

1.1. Introducción 1

1.2. Objetivos 4

2. LIBRERÍAS MVC 5

2.1. Problemas en librerías MVC existentes 5

2.2. Nuevas librerías 10

3. REACT 14

3.1. Origen y Características 14

3.2. Ecosistema 20

3.3. Caso Práctico: 3 en raya 23

4. FLUX 24

4.1. Origen y Características 24

4.2. Caso Práctico: 3 en raya. 27

5. CASO DE USO: QUIZ 30

5.1. Descripción 30

5.2. Implementación 34

5.3. Resultados 44

6. CONCLUSIONES 46

6.1. Conclusion 46

6.2. Líneas futuras 47

7. BIBLIOGRAFÍA 48

1. INTRODUCCIÓN Y OBJETIVOS

1.1. INTRODUCCIÓN

JavaScript [1], que no debemos de confundir con Java, fue creado por Brendan Eich (en Netscape) en Mayo de 1995. Aunque no siempre fue conocido como JavaScript, se fundó con el nombre de Mocha cambiándose en Septiembre de 1995 por LiveScript y finalmente en Diciembre de ese mismo año se adoptó el nombre de JavaScript.

En 2005, Jesse James Garrett, quien acuñó el término de Ajax, describió un gran conjunto de tecnologías en las cuales JavaScript era la columna vertebral para la creación de aplicaciones Web en las que los datos puede ser cargados en segundo plano (background) evitando la necesidad de cargar la página por completo y realizando, por tanto, aplicaciones más dinámicas. Fue un periodo de renacimiento para JavaScript encabezado por el uso de librerías Open Source (Código Abierto) y de comunidades que surgieron alrededor de las mismas, librerías como Prototype [2], jQuery [3], Dojo [4] y Mootools [5].

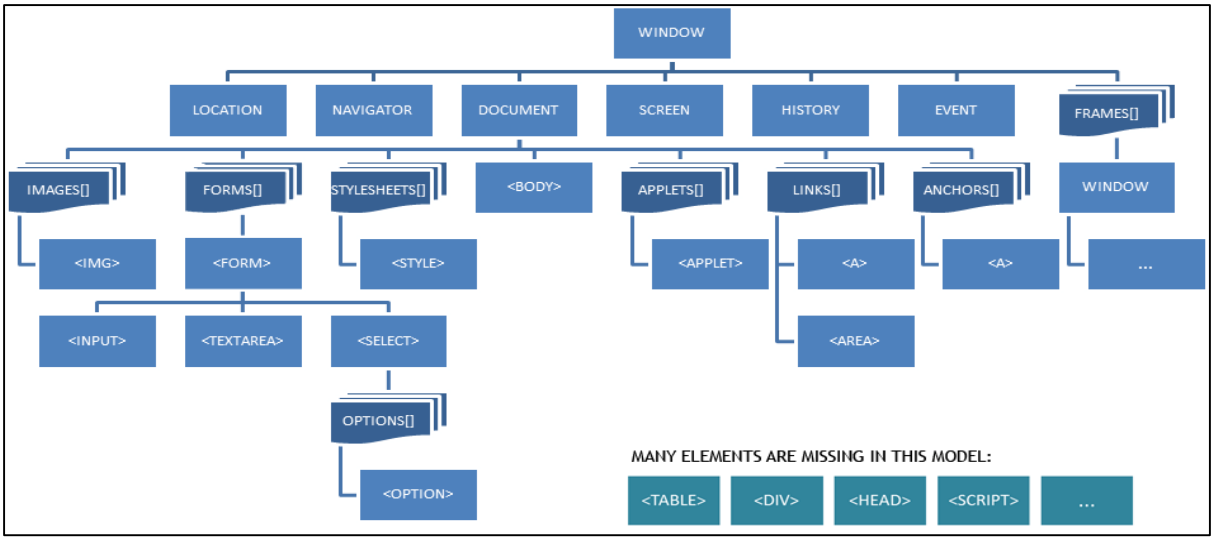
A día de hoy JavaScript está en una constante evolución, innovación y normalización con nuevos desarrollos como la plataforma NodeJS, que nos permite utilizar JavaScript en el lado del servidor, y APIs de HTML5 para controlar los medios de comunicación de los usuarios. Por parte del contenido multimedia ha habido un tremendo desarrollo en el ámbito JavaScript gracias al desarrollo de HTML5 (etiqueta canvas) y OpenGL 2.0. La evolución ha sido tal que existe una enorme variedad de librerías JavaScript que facilitan el esfuerzo y disminuyen los tiempos de desarrollo, afortunadamente encontramos sitios web como JSDB [6] donde se recopilan dichas librerías. En JSDB se tienen indexadas cientos de librerías categorizadas por el tipo de herramientas que permiten desarrollar animaciones, formularios, etc… incluyendo el repositorio en Github y el lugar oficial, un diccionario que siempre es recomendable tener a mano.

Hay que recalcar que no sólo ha cambiado la forma de programar en JavaScript sino que la tecnología ha sido el principal motor de este cambio, ya que cuando se fundó JavaScript los navegadores y las conexiones eran lentos y alojar un sitio web era costoso. El principal motivo por el que se impulsó JavaScript fue que gracias a este lenguaje se permitía ejecutar código del lado cliente evitando el coste de procesar la información en el servidor. Haciendo de los sitios web mucho mas dinámicos para el usuario y menos costoso para los servidores en términos de tráfico.

Un punto clave en la historia de JavaScript fue cuando los navegadores empezaron a soportar e implementar el árbol DOM (Document Object Model ó Documento Objeto Modelo) que permitía una interacción mucho más enriquecedora con las páginas web. El documento representa la información que es presentada en el navegador, el objeto son los párrafos,

cabeceras, enlaces, etc…, en definitiva, elementos individuales. Finalmente, el modelo es una copia o una representación interpretada del documento original pero de una forma distinta.

Mediante la utilización del DOM podemos tener acceso a cualquier elemento del documento y poder manipular su contenido y atributos así como crear nuevos elementos cuando y donde se quiera. Si unimos esto al manejador de eventos conseguimos una forma más reactiva pudiendo renderizar un contenido u otro. En la siguiente imagen [7] si ilustra un árbol DOM.



Las principales características que presenta JavaScript y que le han hecho atractivo son:

- **Dinámico:**

Como en la mayoría de lenguajes de scripting están asociados con valores y no variables. Por ejemplo, una variable “x” puede estar asociada a un número pero luego se la puede reasociar a una cadena de texto (string).

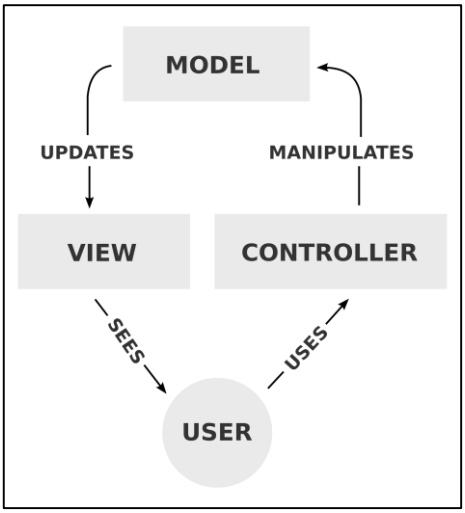
- **Basado en objetos:**

Los objetos son entidades que tienen un determinado estado (están definidos por uno o varios atributos), comportamiento (métodos) e identidad (propiedad que le diferencia del resto, identificador). Un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso a objetos de una misma clase. Por tanto un lenguaje basado en objetos también está basado en varias técnicas como herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento.

- **Funcional:**

Son objetos en sí mismos. Las funciones son creadas y se las puede invocar. Básicamente una función de JavaScript es un bloque de código diseñado para realizar una tarea particular.

A la hora de describir e implementar aplicaciones software surgió el llamado MVC [8] (Model View Controller, o en castellano, Modelo-Vista-Controlador), un patrón de arquitectura software que separa los datos y la lógica de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello a través de la vista se representa la información al usuario capturando el controlador las interactuaciones del mismo que puede provocar modificaciones del modelo y por tanto de la nueva vista.



Alrededor de JavaScript han surgido unas librerías muy populares como son Bakcbone [9], AngularJS [10], EmberJS [11], Bootstrap [12], etc… Librerías que han aprovechado la potencia de JavaScript para simplificar aún más la tareas más típicas que se realizan en la web. Eliminando por ejemplo la manipulación del DOM de forma que se actualice automáticamente cuando el modelo (Model, la M de MVC) cambie, actualizando la vista (View, la V de MVC) y viceversa.

Sin embargo, estas librerías tan populares, han ido teniendo problemas, y aunque se hayan actualizado para solventarlos, estos errores han dado paso a una nueva generación de librerías JavaScript muy prometedoras entre las cuales encontramos: React [13], Flux [14], Webpack [15], IO.js [16] y su unión con NodeJS [17], etc…

En definitiva, nos encontramos actualmente en una constante evolución en el mundo JavaScript que está dando a la web nuevas funcionalidades que hace unos años no hubiera sido posible de imaginar.

1.2. OBJETIVOS

Este trabajo se centra en la evaluación del uso de librerías MVC (Modelo-Vista-Controlador) para el desarrollo de servicios web. Los objetivos que se han planteado son los siguientes:

- Evaluación de las librerías MVC existentes y los problemas que han surgido en las mismas así como el surgimiento de nuevas librerías. Describiendo las ventajas e inconvenientes de las mismas.

- Se estudiará React como librería MVC reciente, las características que presenta así como el ecosistema de tecnologías que podemos encontrar alrededor del mismo.

- Se abordará Flux como arquitectura actual que tiene una buena integración junto con React, su origen y fundación además de estudiar las características que presenta.

- Se abordará un caso de estudio con estas dos últimas tecnologías (React y Flux) para la elaboración de una aplicación web simplificada y basada en el Quiz de la asignatura Computación en Red incluyendo Flux y React.

- En el caso de estudio se analizarán las herramientas empleadas, la implementación que ha sido llevada a cabo y los resultados que ofrecen estas herramientas. Elaborando un resumen de las ventajas e inconvenientes encontrados.

- Finalmente se darán las conclusiones sobre los casos de estudio y los siguientes pasos del futuro de las librerías MVC de JavaScript.

2. LIBRERÍAS MVC

2.1. PROBLEMAS EN LIBRERÍAS MVC EXISTENTES

En los últimos meses se han experimentado grandes cambios en el entorno de JavaScript. Con lenguajes como React, Flux, Redux, Webpack, IO.js, etc…está más que claro que son tiempos de cambio y que mantienen la alerta por si el status quo cambia. Entre los motivos que han propiciado estos cambios nos encontramos con una gran variedad de problemas en las principales librerías MVC ya existentes de JavaScript como pueden ser Bakcbone [9], AngularJS [10], EmberJS [11], Bootstrap [12]. Estas librerías han sido las librerías más populares y que más uso han conseguido estos últimos años.

Estos problemas han dado lugar a la aparición de librerías nuevas que solucionan estos problemas, así como el trabajo que han realizado las mismas librerías para renovarse, solucionando estos problemas. A continuación se ilustrarán los problemas encontrados en las principales librerías.

- **Bootstrap:**

Bootstrap fue desarrollado por Mark Otto y Jacob Thornton inicialmente con el nombre de Twitter Blueprint como un framework para desarrollar aplicaciones front-end (aplicaciones con las que el usuario interactúa directamente). Bootstrap contiene HTML y CSS (la semántica y el estilo, respectivamente) así como JavaScript para la creación de sitios web dinámicos y aplicaciones web multiplataforma. Ofrece una gran variedad de plantillas para facilitar un desarrollo rápido y con buenos resultados, sin embargo, han surgido problemas.

Uno de los principales problemas es que no sigue una buena praxis ya que al final se acaba trabajando con un árbol DOM atestado de clases rompiendo con una de las reglas fundamentales del buen desarrollo web, el HTML ya no es una representación semántica y la presentación o estilo no se encuentra separado del contenido. Todo esto complica la escalabilidad, reusabilidad y mantenimiento, un reto muy difícil que encuentran los puristas de aplicaciones front-end verdaderamente molesto. En definitiva, la web se separa en la semántica, la estructura que tiene, y que se define en HTML, el estilo se define en el CSS y finalmente la funcionalidad se define mediante JavaScript. Por ello la unión de dos partes o incluso las tres en un lenguaje (en vez de tenerlas separadas) producen errores más difíciles de encontrar y depurar.

Se producen colisiones con configuraciones ya existentes, como conflictos en el HTML, CSS y JavaScript generado y que ahora deben de pasar por un proyecto monstruoso y hacer un gran esfuerzo por remover o remplazar scripts y estilos. En definitiva, Bootstrap no es una buena solución si ya se dispone de un proyecto, ya que al pasarlo a Bootstrap puede suponer un gran trabajo a la hora de depurar y encontrar fallos.

Se trata, además, de un lenguaje pesado ya que incluye archivos CSS de 126KB y archivos JavaScript de 29KB lo que dificulta los tiempos de carga de una aplicación lo cual es un problema si la conexión de la que se dispone no es rápida. Bootstrap permite crear sitios web atractivos y interactuables pero se debe de fijar bien el mercado objetivo ya que este tipo de aplicaciones suponen un drenaje continuo de batería en el caso de dispositivos móviles y de tiempos de carga lentos.

No dispone de soporte de SASS (Syntactically Awesome StyleSheets) y este puede ser uno de los principales problemas ya que Bootstrap está construido en Less [18] que no provee ningún tipo de soporte a Compass [19] y SASS que a día de hoy están considerados mejores entornos para CSS. Finalmente nos encontramos que los sitios web que desarrollamos son muy similares a los de los demás, debido a las plantillas que proporciona Bootstrap y a su popularidad y sencillez. Además, si se quiere salir del estilo que da Bootstrap para personalizarlo nos encontraremos con un gran número de restricciones, no dejando mucho espacio a la creatividad.

- **AngularJS:**

AngularJS es un framework de código abierto para aplicaciones web mantenido principalmente por Google y una comunidad de desarrolladores individuales o grupos que cooperan con Google para solucionar varios de los problemas encontrados al desarrollar SPA (Single-Page Applications). Su objetivo es simplificar tanto la parte de desarrollo como la de pruebas para arquitecturas MVC y MVVM (Model-View-ViewModel) [20]. Además ahora con la aparición de React, AngularJS ha tenido que incorporar la visión de páginas basadas en componentes, desarrollando los suyos propios.

Angular crea una buena primera impresión debido a la implementación de la lógica a través de etiquetas que se actualizan por sí solas, sin embargo, se han encontrado varios defectos. Una de las reglas fundamentales en programación es que siempre se prefiere la forma explícita a la implícita como ocurre con la invocación de los manejadores de eventos de AngularJS. El two-way data binding significa simplemente que cuando las propiedades del "Model" se actualizan, la UI (User Interface) también se actualiza y cuando los elementos de la UI se actualizan los cambios se propagan hasta el "Model".

AngularJS emplea two-way data binding lo que provoca que cuando se cambia cualquier cosa en la aplicación se disparan cientos de eventos (que llaman a funciones). Esto provoca una cierta lentitud que desemboca en un mal rendimiento sobre todo con las plataformas móviles cuando se trata de aplicaciones complejas. Por eso Angular ha cambiado esto permitiendo, a día de hoy, poder realizar one-way data binding o incluso ha publicado numerosos artículos de cómo acelerar su rendimiento.

Como se puede encontrar en este artículo [21], uno de los principales problemas que se le recriminan a Angular es la introducción de cinco nuevas entidades (provider, service, factory, value y constant) que alegan que son distintas una de la otra pero que en esencia son lo mismo, complicando a los programadores las diferencias [22]. Así mismo se han encontrado varios problemas a la hora de debugear (busca de errores) debido a eventos que no parecen que salten pero que modifican el comportamiento de alguna variable.

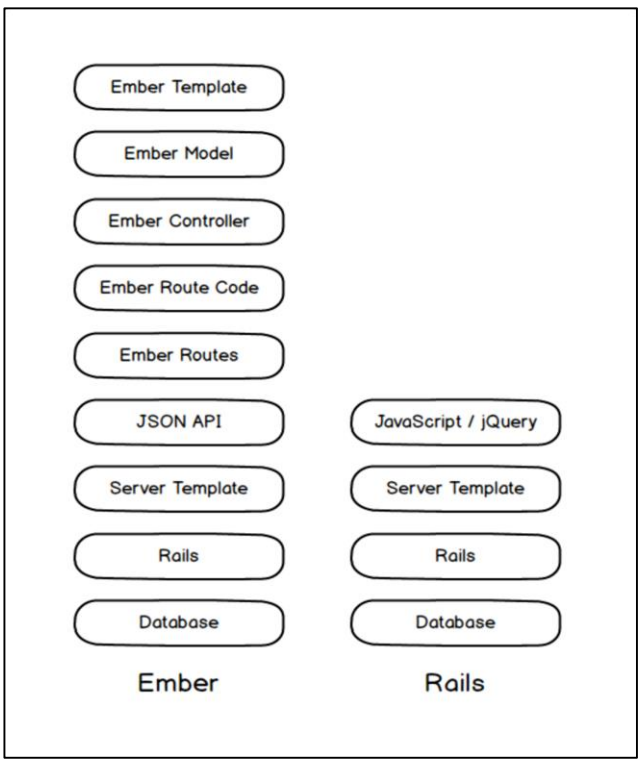
Se han encontrado varios problemas en el desarrollo de aplicaciones para el lado del servidor puesto que se tiene que añadir lógica al HTML. AngularJS también escribe lógica en el HTML lo que no separa semántica de funcionalidad y da lugar a los denominados "códigos spaghetti".

- **EmberJS:**

EmberJS es un framework de código abierto para aplicaciones JavaScript basadas en el patrón Model-View-Controller que permite al desarrollador crear SPA(Single-Page Applications) escalables. EmberJS permite escribir en menos líneas de código gracias a su entorno de trabajo permitiendo programar en distintos lenguajes de programación para que el desarrollador solo se centre en el desarrollo de la aplicación.

Entre los problemas más comunes en EmberJS nos encontramos errores silenciosos, como pasaba con Bootstrap donde al producirse muchos disparadores de eventos a veces alguno pasaba desapercibido cambiando información sensible y por tanto siendo muy difíciles de rastrear. Así mismo EmberJS proporciona la posibilidad de enrutar nuestra aplicación así cada vista puede tener una única URL, sin embargo esto ofrece problemas de escalabilidad cuando nuestra aplicación crece apareciendo enlaces que arbitrariamente dejan de funcionar.

No existen addons ni plugins para la paginación en EmberJS que es en definitiva un proceso muy costoso no solo en recursos sino especialmente en tiempo. Así mismo dentro de las librerías mencionadas anteriormente como de las más populares, a diferencias de las demás, resulta más complicado el encontrar ejemplos ya realizados. Al usar EmberJS se añaden más capas, por ejemplo en Rails, como se puede apreciar en la siguiente imagen [23].



En definitiva EmberJS resulta complicado para programadores que se están iniciando en este lenguaje y para aquellos que desean realizar aplicaciones de mayores tamaño, así como presentar algunos de los problemas que se encuentran en las librerías anteriormente mencionadas.

- **Backbone:**

Backbone proporciona la estructura a las aplicaciones web proveyendo modelos con el vínculo clave-valor, eventos propios, funciones enumeradas con una gran API. Backbone es una herramienta de desarrollo/API para JavaScript con una interfaz REST (Representational State Transfer) por JSON, basada en el paradigma de diseño de aplicaciones MVC. En concreto, está diseñada para desarrollar aplicaciones SPA (Single-Page Applications ó Aplicaciones de una única página) y para mantener las diferentes partes de las aplicaciones web sincronizadas.

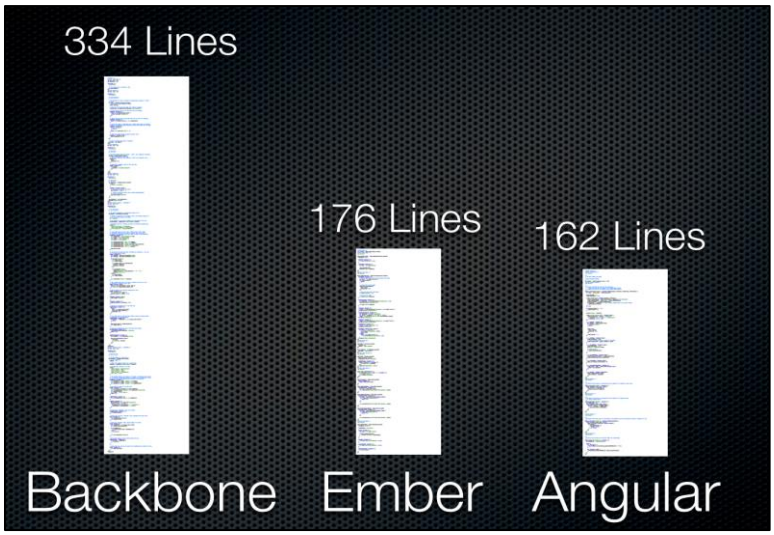
Respecto a Backbone nos encontramos al igual que ocurría con Bootstrap problemas a la hora de utilizar el DOM y a la hora de renderizar varias veces y mantener el estado correcto de la aplicación. Backbone no incluye ningún tipo de testing, siendo bastante complicado la depuración de las vistas, tal es el nivel de complejidad que no se realiza testing sobre esto.

Como se ha comentado en las anteriores librerías, Backbone también tiene en común el uso no tan eficiente de la memoria y esto conlleva a que el renderizar una página se convierta en algo muy lento. Para más información se puede consultar [24].

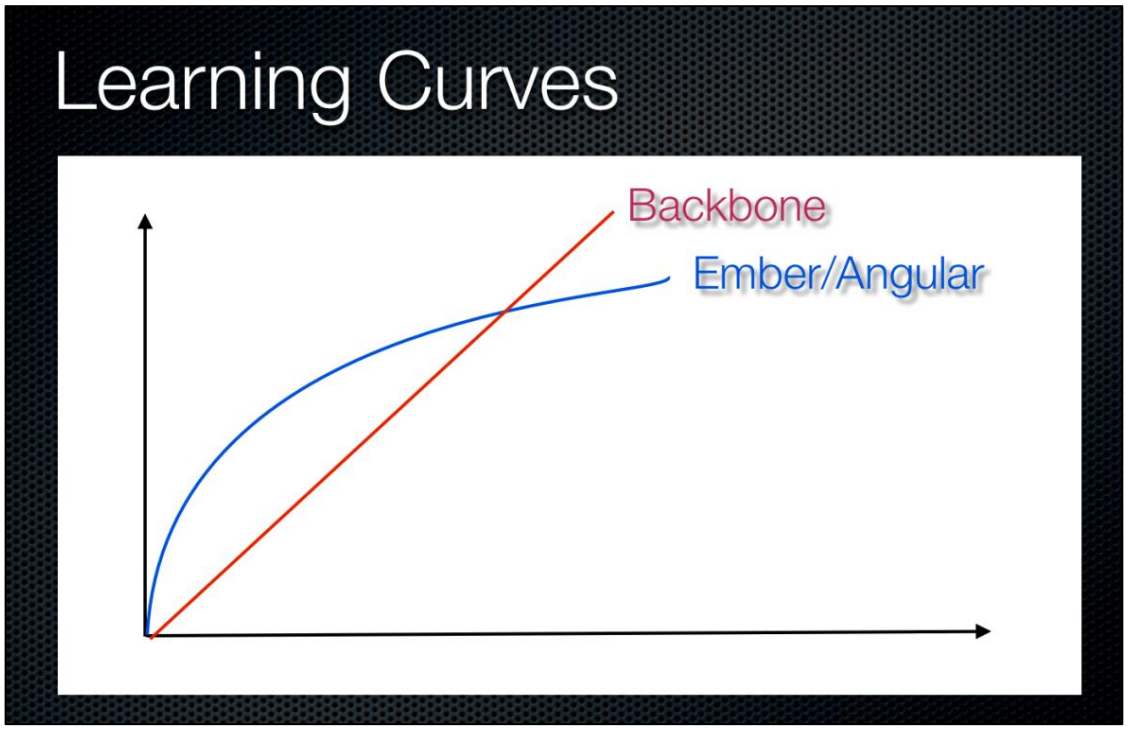
- **Comparativa y problemas comunes:**

Como se han ido comentando, los principales problemas se encuentran referenciados al DOM al cual no se le da un uso adecuado o bien surgen problemas al renderizar varias veces la aplicación no guardando correctamente el estado de la misma. Así mismo el uso de two-way data binding que pese a mostrar algunas ventajas para aplicaciones grandes se ha demostrado que presenta una solución difícilmente escalable.

Uno de los aspectos de los que presumen muchos lenguajes de programación es la cantidad de código que se requiere para realizar una misma aplicación. A continuación, se muestra una comparativa [24] de las librerías más populares en la aplicación de TO-DO [25].



Pero no debemos de confundirnos ni influenciarnos por la imagen ya que hay otros aspectos a valorar como por ejemplo las curvas de aprendizaje.



2.2.NUEVAS LIBRERÍAS

Como hemos visto los principales problemas encontrados se deben a un mal uso o al uso ineficiente del Document Object Model (DOM), así como problemas de escalabilidad en el uso de two-way data binding. Los problemas encontrados en el apartado anterior en las librerías más populares de JavaScript han abierto camino para dar origen a una gran variedad de librerías que intentan solventar dichos problemas.

No solo han surgido librerías de JavaScript resolviendo estos problemas sino herramientas, las también denominadas "plugins", para facilitar el desarrollo en JavaScript. La finalidad de estos es simplificar la programación en JavaScript, realizando diferentes funciones como la agrupación de distintos módulos (escritos con diferentes librerías) en unos pocos (escritos en menos librerías JavaScript), como Webpack, hasta una compilación ligera de código JavaScript para desplegar en el navegador (lado cliente) sin necesidad de tener que levantar un servidor.

Así mismo también las librerías más populares, ante sus fallos, se han actualizado intentado arreglar dichos errores, como puede ser el caso de AngularJS y su versión AngularJS 2.0 [26], que se encuentra ahora mismo en fase beta. No obstante, los desarrolladores de AngularJS ante estos cambios tan drásticos y que han cambiado en gran medida la forma de programar en dichos lenguajes se han decantado por otras librerías nuevas como es el caso de React o Flux.

Se debe mencionar la necesidad actual de tener herramientas como "Módulos Web" (Web Modules) que resultan mecanismos muy útiles que resuelven problemas como la creciente complejidad de código así como su unión e interfuncionalidad entre distintos lenguajes. Todo esto es debido a que ya no se habla de sitios web sino más bien de aplicaciones web y por tanto se necesita herramientas para facilitar su programación y despliegue. Entre los más famosos encontramos a Webpack[27], RequireJS [28] o Browserify [29].

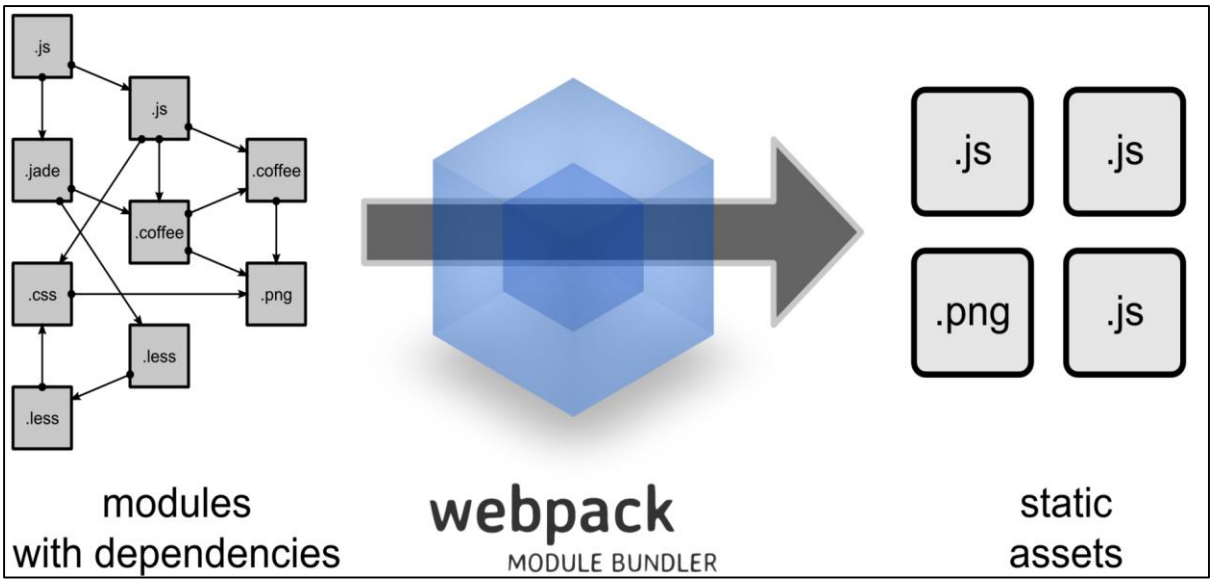
Puesto que React, Browserify y Flux serán el objeto de un caso práctico de este Trabajo Fin de Grado se les dedicará un apartado a parte. A continuación, se describirán algunas de las librerías y herramientas que han surgido o que han ido adquiriendo mayor protagonismo en los últimos meses.

- **Webpack:**

Webpack, es una herramienta muy potente, que ha aprendido de otras como Browserify y RequireJS pero que nunca ha intentado garantizar su compatibilidad con NodeJS así como con CommonJS por lo que implica un trabajo extra para adaptarlo a estos entornos. Esta herramienta se ha construido desde cero para ayudar el manejo de activos estáticos (active assets) para aplicaciones front-end.

Con Webpack se solucionan algunos problemas que se mencionaron anteriormente como es el mantener el tiempo de carga bajo y sobre todo se adapta para proyectos de gran tamaño evitando así los problemas de escalabilidad que tenían algunas librerías. También permite la interactuación e integración de librerías de terceros como módulos, así como disponer de todos los activos estáticos como módulos.

Webpack tiene dos tipos de dependencias en su árbol de dependencias: síncrona y asíncrona. La dependencias asíncronas actúan como un punto de división formando un bloque nuevo. Una vez el árbol de dependencias se encuentra optimizado se emite un archivo nuevo para cada bloque. Webpack solo puede procesar JavaScript de forma nativa, los "cargadores" (loaders) se encargan de transformar otros recursos de JavaScript como si se tratasen de módulos. En definitiva para resumir el funcionamiento de Webpack se puede ilustrar con la imagen que tienen en la página oficial [27]:



A partir de varios ficheros escritos en distintos lenguajes, gracias a Webpack, se puede unificar en pocos ficheros escritos en JavaScript de forma que la interacción entre ambos se simplifica y se reduce los tiempos de carga. Básicamente trata de unificar diferentes módulos en uno solo. Similar al comportamiento de Browserify que unifica varios ficheros JavaScript que exportan módulos, en uno único capaza de mantener toda la aplicación de una SPA en un único archivo, evitando tiempos de carga elevados y mejorando la experiencia del usuario.

Como también se ha mencionado anteriormente RequireJS también realizar funciones similares a Webpack. RequireJS es un archivo JavaScript y "cargador de módulos" (Module Loader) que está optimizado para el uso en navegadores y que puede ser usado en otros entornos JavaScript como Rhino o NodeJS. Gracias a su escritura modular la velocidad y calidad del código es más elevada y eficiente.

- **IO.js:**

IO.js es un proyecto de código abierto (Open Source) dirigido por una comunidad de desarrolladores. Fue iniciado por los principales contribuidores de NodeJS y en sus primeros meses atrajo a más desarrolladores activos que los que tuvo NodeJS en toda su historia. Se trata de un reemplazo intercambiable para NodeJS y que es compatible con casi todos los módulos de NPM (Node Package Manager).

IO.js garantiza estabilidad, transparencia (depuración y "tracing"), mejores streams, soporte a largo plazo (LTS, Long Term Support), localización y adoptar estándares en curso. Al disponer de una de las APIs de JavaScript más grandes, se tiene como objetivo el garantizar estabilidad en las futuras versiones de IO.js para que sigan siendo compatibles con versiones anteriores. Pudiendo llegar a surgir nuevas APIs centradas en ES6/7 sin romper viejos módulos y API.

Sin embargo, el hecho más importante que atañe a IO.js es su unión junto con NodeJS en su versión 4.0.0 que ha tenido un número record de contribuciones por parte de compañías e individuos al juntar estas dos tecnologías en una misma base bajo la dirección de la fundación Node.js. Esta versión 4.0.0 incluye algunas actualizaciones de la versión 3.0.0 de IO.js así como algunas características extra para los usuarios de NodeJS. Además se incluyen algunos rasgos de ES6 entre los que se encuentran arrays tipados, colecciones (Map, Set, etc…), símbolos, programación en bloques y varias funciones más.

Toda la información que se encuentra sobre NodeJS y sus nuevas características vienen resumidas en la siguiente referencia [30], en donde se anuncia la unión de IO.js y NodeJS así como la inclusión de los repositorios para realizar contribuciones a este proyecto.

- **Polymer:**

Polymer [31], es un lenguaje bastante similar a React, el cual explicaremos con más detalle. Este lenguaje está basado en uno de los principios de React que son los componentes. Un concepto que ha surgido nuevo en la web y se basa en la creación de bloques de códigos reusables y actualizables de forma sencilla. De esta forma se palia uno de los grandes problemas encontrados en las librerías JavaScript más populares como es la escalabilidad y la rapidez.

El concepto de componente o elemento en la web se podría asociar al concepto que tenemos de las etiquetas de HTML (o mejor dicho de HTML5). Las etiquetas de HTML5 por ejemplo <nav> definen una barra de navegación. Lo que se pretende con lenguajes de programación que construyen componentes (o elementos) se basa en el mismo principio. A partir de la definición del mismo generar un código asociado a dicho elemento, por ejemplo, un comentario que tiene un autor, la fecha y el contenido del mismo puede ser renderizado de golpe con la etiqueta <comment> (realizando el símil con HTML).

Polymer nos proporciona componentes propios llamados componentes primitivos y da, obviamente, la capacidad para crear elementos propios tanto partiendo desde cero o basándose en dichos componentes primitivos. Como no todos los navegadores soportan los componentes se proporciona la librería Polyfill que adapta dichos componentes web mediante la implementación de una API JavaScript.

Además de la programación de componentes web, Polymer ofrece otra característica típica de React y Flux como es el "one-way data binding" que facilita una sintaxis más declarativa. Pudiendo construir componentes (elementos) más complejos y reutilizables con menos código y facilitando la actualización de dichos componentes y por tanto ahorrando tiempo de carga.

No obstante, una de las desventajas frente a React, es el árbol DOM ya que emplea uno local y los tiempos de acceso a distintos elementos son un poco más elevados en Polymer. Se encuentra una gran variedad de ejemplos así como tutoriales desde la página oficial [31]. Esta librería es junto con React y Flux una de las librerías modernas y más atractivas que soluciona varios problemas que se encontraban en otras librerías JavaScript. Así como poder encontrar una gran variedad de información, desde ejemplos, APIs, documentación hasta código en GitHub muy bien comentado.

Como el objeto de este Trabajo Fin de Grado es React y Flux, para estas dos tecnologías se les dedica un apartado entero explicando en mayor detalle algunas de las soluciones que proporcionan las nuevas librerías JavaScript y un proyecto práctico donde ver su aplicación.

3. REACT

3.1. ORIGEN Y CARACTERÍSTICAS.

Hace dos años el presentador Christopher Chedeau de Facebook anunció ReactJS como una librería JavaScript para desarrollar interfaces de usuario. ReactJS comenzó como un puerto JavaScript de XHP, una versión de PHP que lanzó Facebook hace seis años. XHP surgió por la necesidad de minimizar los ataques XSS (Cross Site Scripting) en los cuales un usuario malicioso proporcionaba contenido dañino que se encuentra incrustado y oculto en JavaScript (lenguaje que corren todos los navegadores web) pudiendo robar información o comprometiendo el contenido visto por el usuario.

XHP solventaba este tipo de ataques pero por el contrario generaba otros problemas en aplicaciones web dinámicas ya que estas necesitaban muchas peticiones y respuestas del servidor. Un ingeniero de Facebook negoció con su manager una solución a partir de XHP y a los seis meses nació ReactJS. En la historia de JavaScript, en sus despliegues el acceso y manipulación de árbol DOM (Document Object Model) es una operación muy costosa.

ReactJS ha asumido el coste de manipular el árbol DOM y ha llegado a una conclusión obvia, la optimización del uso del árbol DOM permite la creación de aplicaciones con respuestas rápidas y pequeñas en cuanto a código. React almacena de forma interna el árbol DOM y su contenido en el navegador de está forma solo se vuelve a re-renderizar el contenido que ha sufrido cambios.

Aunque la implementación es similar a la dada por AngularJS, este la realiza a través de two-way data binding mientras que ReactJS lo lleva a un paso más adelante ya que React conoce el estado de la aplicación (sabiendo cuando se han producido o no cambios) lo que conlleva a one-way data flow. Este es un punto clave sobre todo cuando hablamos de aplicaciones de gran tamaño como es el caso de Facebook.

La complejidad de la operación “diff” que es un termino para referirse a una operación que devuelve los elementos que han cambiado en un documento se estimó del orden de O(n^3), en termino de tiempos se estima en 17 minutos en encontrar los cambios en un árbol DOM de 10 000 nodos con una máquina de 1 GHz. Los ingeniero de Facebook mediante ReactJS y el uso de un mapa hash mejoraron estos tiempos hasta llegar al orden de O(n), un logro enorme y que hace de este lenguaje una alternativa perfecta a las librerías JavaScript más populares.

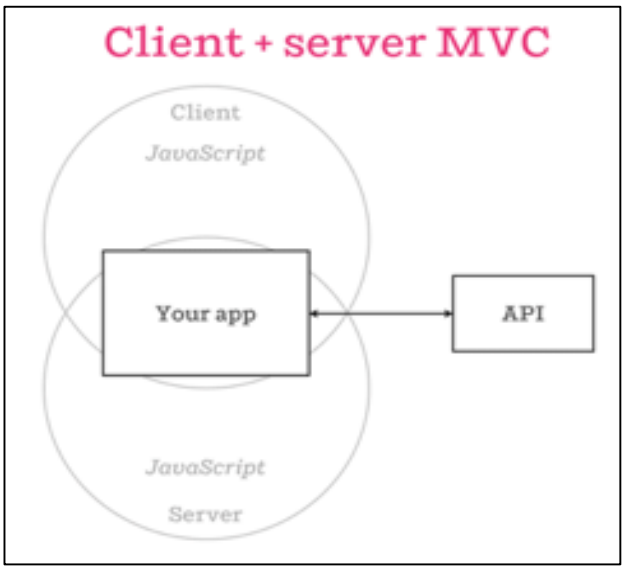
Una vez concluido el “por qué”, React o también llamado React.js o ReactJS es una biblioteca de JavaScript de código abierto para crear interfaces de usuario. Está sostenido por Facebook, Instagram y una comunidad de desarrolladores individuales y corporativos. React pretende ayudar a los desarrolladores a construir aplicaciones de gran tamaño usando información que cambia a lo largo del tiempo. Su principal objetivo es en definitiva ser

sencillo, declarativo e inequívoco (claro). El objetivo de React es renderizar el contenido del lado del servidor en la primera carga para dar la sensación de una SPA (Single Page Application) al usuario.

Se necesitaba algo que pudiera renderizar tanto en del lado del cliente como en del servidor y que compartiera el estado de la aplicación entre ambos lados. De este modo el cliente podría continuar en el punto en el que el servidor haya terminado.

Para implementar este tipo de arquitectura el código base tiene que ser común en el lado del servidor y en del cliente (Browserify / Webpack) y que las aplicaciones fueran capaces de renderizar objetos en ambos lados (servidor y cliente).

Y esta es la mentalidad de Browserify que cita: *"Browserify lets you require('modules') in the browser by bundling up all of your dependencies."* - **browserify.org**.



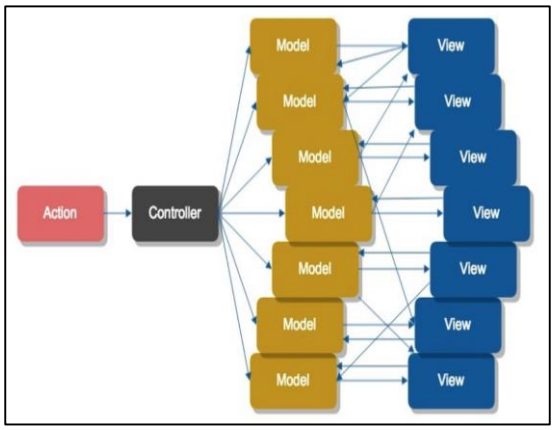
Isomorphic JavaScript architecture, Source: AirBnb Nerds

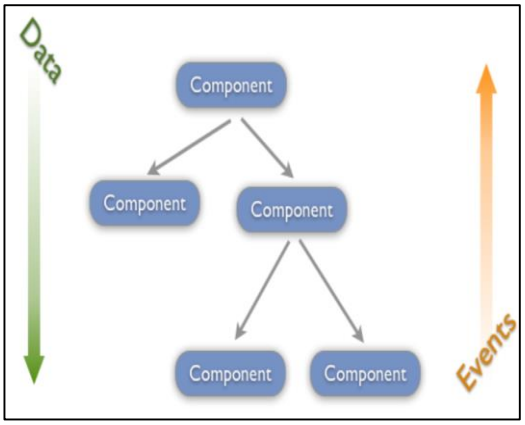
Facebook e Instagram son el mayor soporte económico a este proyecto pero tenemos que aclarar que sin otras comunidades, como por ejemplo StackOverflow [32], GitHub, SubReddit [33], etc, este proyecto no hubiera podido haber sido realizado. A pesar de que React es un lenguaje prácticamente nuevo, el soporte que ha recibido es tal que podemos encontrar un gran abanico de recursos como "Online Playgrounds" en los que se puede desarrollar y experimentar con React de forma online. Como por ejemplo: JSFiddle [34] que integra JSX, lenguaje que detallaremos posteriormente.

Como ya hemos mencionado con anterioridad React es una herramienta para construir aplicaciones web basadas en componentes que brillan por su rapidez y sencillez debido al uso inteligente de su DOM virtual. Los componentes en React están escritos en un JSX que es una mezcla entre JavaScript y XML pero que es compilado en JavaScript usando las herramientas de compilación de React. ReactJS es, en definitiva, la V del modelo MVC (Model-View-Controller) que tiene las siguientes características que le han llevado a ser una de las principales alternativas a las librerías más populares de JavaScript.

- **One-way Data Flow:**

React elimina la complejidad del doble flujo de información mediante un único flujo de información. Cuando las propiedades (“props”) de un componente de React se actualizan, este componente se vuelve a renderizar. Hay que distinguir que el propio componente no puede modificar directamente sus propiedades pero si puede llamar de vuelta a una función (“callback”) para que cambie sus propiedades. A este mecanismo se le expresa como “properties flow down and actions flow up”, es decir, las propiedades fluyen hacia abajo y las acciones hacia arriba. Este comportamiento lo comparte también la arquitectura de Flux y se describirá más en detalle en su respectivo apartado. Un ejemplo para entender el mecanismo de “one-way data flow” es un carrito de la compra donde tenemos varios componentes que puede ser productos de la compra con el resultado total a pagar y este total no se puede modificar, pero si se elimina un producto, esta acción puede llamar de vuelta a una función que modifique dichas propiedades, además de eliminar dicho componente. A la izquierda two-way data binding y a la derecha one-way data binding.





- **Virtual DOM:**

React tiene un árbol virtual DOM (Document Object Model) propio, antes de confiar únicamente en el DOM del navegador. Esto permite a la librería elegir qué partes del DOM se han cambiado mediante “diffing” (Diff es un método de comparación de información). La nueva versión se almacena en el DOM virtual y utiliza “diffing” para determinar si resulta eficaz actualizar el DOM del navegador. En otras palabras, React compara los cambios realizados para decidir si re-renderiza un sitio web (por ejemplo). Volviendo al ejemplo del carrito de la compra, si se elimina un producto del carrito esto afecta únicamente al importe total (y a que efectivamente se ha eliminado dicho componente del carrito) y por tanto solo es necesario actualizar dichos elementos. En otras librerías de JavaScript que se han analizado esto no ocurría de esta forma y se renderizaba toda la página suponiendo un coste más elevado, así como una solución poco escalable, ya que cada vez que se modifique un elemento debería de renderizarse la página entera lo que supone un gasto muy elevado.

- **Server-Side Rendering (JavaScript Isomorphism):**

La representación o también llamado la renderización del lado servidor es una característica importante y única de React. El isomorfismo de React es la principal diferenciación con Angular.js. Esta característica es realmente importante cuando encontramos páginas web muy transitadas donde la experiencia del usuario debe ser la prioridad. Compañías como Netflix o PayPal usan el isomorfismo de ReactJS. Veremos herramientas (como Browserify) en los apartados posteriores que realizan una transpilación o compilación ligera de código JavaScript para disponer de aplicaciones que se ejecutan únicamente en el lado cliente sin necesidad de llamar al servidor varias veces.

- **JSX:**

Cuando usamos React también podemos usar JSX. JSX es una extensión de la sintaxis de JavaScript muy similar a la de XML. No es obligatorio utilizar JSX junto con React ya que también podemos utilizar JavaScript puro en su lugar. Pero gracias a la sintaxis de JSX se define de forma óptima el árbol estructural y sus atributos de forma más eficiente que con JavaScript puro. Gracias a XML se nos permite crear árboles más grandes y fáciles de leer que las llamadas a funciones de JavaScript. React puede renderizar etiquetas HTML (“Strings”) u otros componentes de React (“Classes"). Para renderizar etiquetas HTML se emplean las minúsculas mientras que para otro componente React se suele emplear mayúsculas.

- **Browserify:**

Esta última no es una característica como tal de React pero si una herramienta muy útil y casi imprescindible, además de poder utilizar Watchify como herramienta alternativa para un desarrollo mas constante. Watchify se diferencia respecto a Browserify en que realiza la compilación nada mas se produzcan cambios, ahorrando el ejecutar constantemente Browserify cada vez que realizamos cambios. Browserify permite a los desarrolladores escribir modelos en CommonJS separados, que son compatibles con el módulo del sistema usado por Node.js teniendo como objetivo el compilarlos en un único fichero para el navegador. Browserify también incluye versiones especificas del navegador de módulos básicos de Node.js y es usado por React para crear versiones distribuidas de su librería. Es una buena herramienta para utilizar en cualquier entorno de JavaScript porque con las decisiones adecuadas te permite reutilizar código.

- **Componentes:**

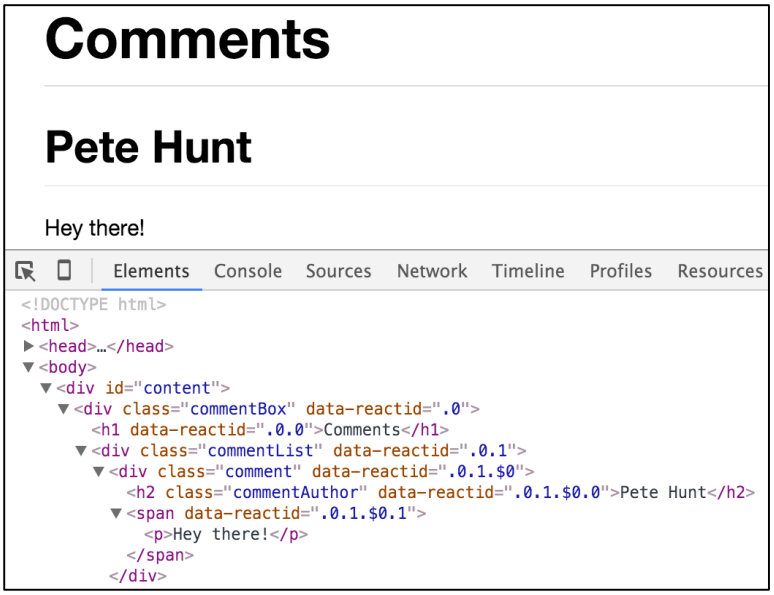
Un componente es, en esencia, similar a una etiqueta HTML. Por ejemplo, con la etiqueta <nav> se define una barra de navegación en HTML, de igual forma ocurre cuando creamos un componente. En el hipotético caso que tengamos un comentario podemos mostrarlo al usuario mediante la etiqueta <Comment>, y dentro de esta etiqueta se encontrará el contenido que se renderizará visualmente. Una apreciación sintáctica que se debe de resaltar es que al

igual que en HTML las etiquetas siempre van en minúsculas, los componentes en React siempre empiezan con la primera letra en mayúscula. Para crear un componente con React basta con llamar a su método constructor "React.createClass(…);". Por ejemplo un sencillo ejemplo donde creamos un comentario:

```
var CommentBox = React.createClass({
    render: function() {
        return (
            <div className="commentBox">
                Hello, world! I am a CommentBox.
            </div>
        );
    }
});

ReactDOM.render(
    <CommentBox />,
    document.getElementById('content');
);
```

Creamos una clase comentario mediante la función mencionada anteriormente pero además a esta clase la podemos definir métodos. El más importante es el método "render" que devuelve un árbol de componentes React que eventualmente entrega a HTML. Finalmente el método ReactDOM.render(…) instancia el componente raíz, es decir, el objeto cuya id es 'content' en nuestro caso y va insertando las relaciones ("markup") en un DOM vacío que se provee como segundo argumento.

Como se puede ver el código HTML que recogemos cuando vemos funcionando una vista en React son etiquetas . Estas etiquetas no son nodos DOM, son instancias de los componentes de React. Son al fin y al cabo marcadores o piezas de datos que React sabe como tratarlos de forma segura.



Una vez visto la creación de componentes y como se instancian en el árbol virtual DOM que usa React, vamos a pasar a hablar del uso de las propiedades ("props") de los componentes. Para ello vamos a seguir el tutorial que se encuentra en la página oficial de React [13]. Vamos a crear un componente que sea un comentario muy sencillo en el cual queremos que se muestre el autor del comentario y contenido en sí del propio comentario. Es como sigue:

```
var Comment = React.createClass({
    render: function() {
        return (
            <div className="comment">
                <h2 className="commentAuthor">
                    {this.props.author}
                </h2>
                {this.props.children}
            </div>
        );
    }
});
```

Este componente se enmarca dentro de otro que será una lista de comentarios (CommentList), que renderizará varios componentes <Comment>. Los datos pasados desde el componente padre (en este caso CommentList) estarán accesibles como una propiedad ("props") para el componente hijo (en este caso Comment). Estas propiedades puede ser accedidas usando "this.props.NombreProp" donde "NombreProp" lo declaramos nosotros en el elemento padre. Ahora bien si miramos el código de CommentList.

```
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        <Comment author="Pete Hunt">This is one comment</Comment>
        <Comment author="Jordan Walke">This is another comment</Comment>
      </div>
    );
  }
});
```

Por tanto, hemos definido en CommentList, dos componentes del tipo Comment, y les hemos pasado como propiedades un autor ("author"). Por eso en el componente hijo recogemos el valor del autor con "this.props.author", y en este caso recogeremos "Pete Hunt" ó "Jordan Walke". Mientras que con la sentencia "this.props.children" (siendo "children" una palabra reservada para React), recogeremos el contenido que exista entre las etiquetas <Comment> </Comment>, en este caso "This is one comment".

React también tiene métodos reservados para realizar varias acciones los cuales pueden ser consultados desde la API. En la página web oficial de React vienen separados según el tipo de componente. Por ejemplo con el método getInitialState(), podemos definir el estado inicial de la aplicación ya que cuando se ejecute la vista de React, es decir, la primera carga, se cargara con el estado que definamos en ese método. Este método se ejecutará una única vez. También otros métodos como setState() donde podemos definir el estado de la aplicación en un momento dado de la misma.

3.2. ECOSISTEMA

Alrededor de React han surgido varias librerías de JavaScript formando un ecosistema muy rico alrededor de esta tecnología y con resultados increíbles. En definitiva el mundo de JavaScript está experimentando una autentica revolución. A continuación, enumeraremos algunas de las librerías JavaScript que complementan a React y que en conjunto conforman un "ecosistema" muy productivo y novedoso.

- **React Native:**

El 28 de enero, Facebook anunció React Native [35] como un puente entre su framework de ReactJS y los dos sistemas operativos móviles líderes (iOS y Android). Con React Native se puede utilizar la plataforma estándar de componentes como la "UITabBar" en iOS y "Drawer" en Android. Esto le da a las aplicación un aspecto coherente con el resto del ecosistema manteniendo su calidad.

Todas las operaciones entre el código de la aplicación de JavaScript y la plataforma nativa se realizan de forma asíncrona , y los módulos nativos también pueden hacer uso de las hebras adicionales. React Native implementa un sistema de gran alcance para negociar toques en vistas complejas jerarquizadas y proporciona componentes de alto nivel como "TouchableHighlight" que se integran adecuadamente con vistas de desplazamiento y otros elementos sin ninguna configuración adicional.

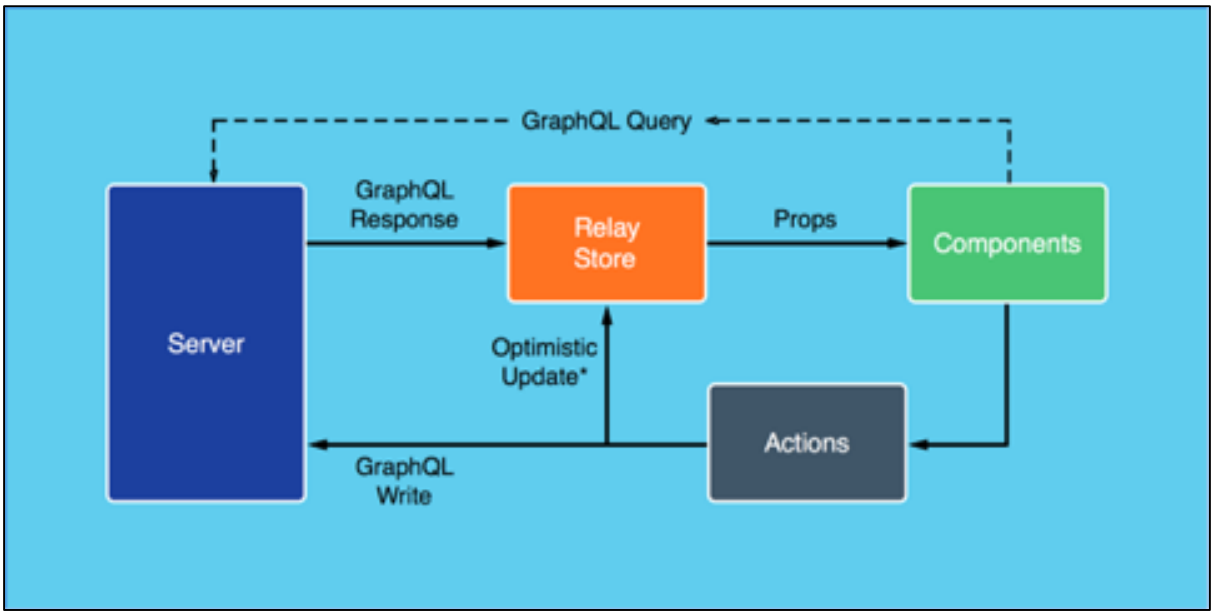
Además existe la posibilidad de utilizar NPM para instalar las librerías JavaScript que van a trabajar encima de la funcionalidad dentro de React Native como por ejemplo "XMLHttpRequest", "window.requestAnimationFrame", y "navigator.geolocation". Para más información sobre el alcance de React Native se puede consultar la página oficial (que está incluida en la Bibliografía) donde además se pueden encontrar ejemplos que ilustran cada uno de los puntos mencionados anteriormente.

- **Relay & GraphQL:**

Relay [36], es el nuevo "framework" de Facebook que provee "data-fetching" (recogida de información) para las aplicaciones de React. Cada componente especifica su propia información de forma declarativa usando un lenguaje de query (consultas) llamado GraphQL. La información se hace disponible a los componentes mediante las propiedades "this.props".

Los desarrolladores componen los componentes en React de forma natural y Relay se encarga de traducir estas peticiones de información para proporcionar a cada componente de React la información que ha pedido (ni nada más ni nada menos). Actualiza los componentes cuando la información cambia y mantiene un almacén (Store) en el lado cliente de esta información (caché).

Por otro lado encontramos a GraphQL que es un lenguaje de consultas de información (Query). El ecosistema que forman estos dos lenguajes es complementario ya que GraphQL se encarga de procesar las peticiones que hacen a los componentes y de responder a Relay para que este actualice la información a los componentes. Como podemos ver ilustrado en la siguiente imagen.



El uso conjunto de estos dos lenguajes nos beneficia en los siguientes puntos:

- Búsqueda previa automática y eficiente de datos para toda una jerarquía de vistas en una sola solicitud de red .
- Paginación Trivial (Pagination) para ir a buscar sólo los elementos adicionales.
- Permite que la vista sea reutilizable para que compare la información que ya contiene y la nueva información sin que tenga que cambiar el componente entero.
- Subscripciones automáticas para que los componentes se re-renderizen si la información cambia. Evitando que re-renderizen componentes no afectados.
- Relay aprovecha al máximo el modelo declarativo de los componentes de React evitando líneas adicionales para recoger información.

- **Styling en React:**

Hay múltiples opciones a la hora de dar estilo como por ejemplo Radium [37], ReactStyle [38] o ReactInline [39]. Estas opciones son capaces de dar estilo a las aplicaciones de React sin la necesidad de utilizar CSS. Aunque siempre nos será posible utilizar ficheros CSS para dar estilo a nuestra aplicación. Todas las opciones propuestas son muy extensas y por ello se especifica en la bibliografía los enlaces donde se detalla cada tecnología.

- **Gráficos con D3.js:**

D3 es un lenguaje que tiene como objetivo crear gráficos asombrosos como podemos ver en el siguiente tutorial [40]. Tanto React como D3 comparten el mismo enfoque: “dame un conjunto de información, dime como la tengo que renderizar y yo me las apañaré para saber que partes del árbol DOM tengo que actualizar”. De esta forma ocurre con D3, cualquier modificación solo afectara a los componentes involucrados siendo un lenguaje muy eficaz para tener información sincronizada de forma gráfica.

- **Animaciones con Haskell (React-Haskell):**

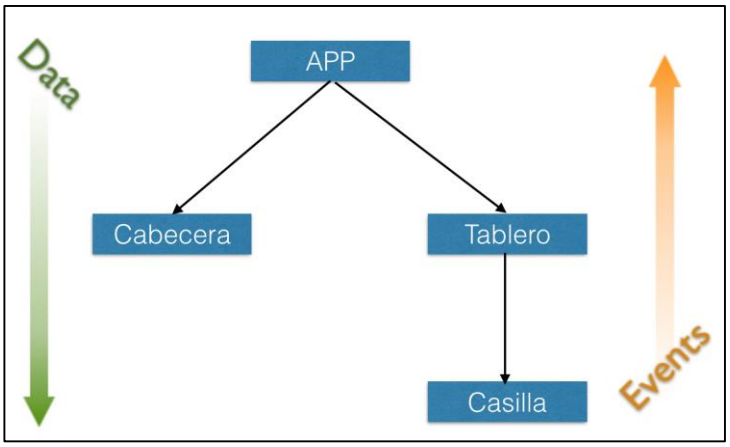
Las animaciones son un problema que aún no esta resuelto en React y Haskell esta empezando a ser usado para UI web convirtiéndose en un cambio rompedor en este terreno. No solo realiza animaciones el punto que debemos de resaltar es su simplicidad en el proceso de actualización. Además Haskell ofrece una amplia variedad de funciones de transición (efecto de transición ó easing) así como temporización, cambios de anchura, altura, etc… Para mas información sobre los efectos consultar la página web oficial donde se muestran las animaciones (verdaderamente increíble).

3.3. CASO PRÁCTICO: 3 EN RAYA

Se ilustrará React y su funcionamiento con un caso práctico desarrollado por Enrique Barra para las clases de Ingeniería Web y que ha sido adaptado así como la adición de un estilo. Esta solución se puede encontrar en el siguiente repositorio de GitHub, con el código correctamente comentado e ilustrado, así de cómo proceder para su instalación en el archivo Readme.md. El repositorio es el siguiente:

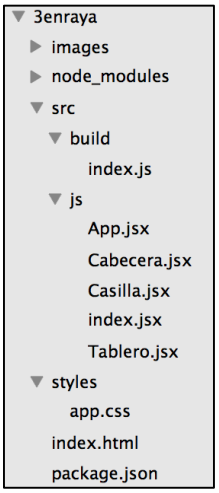
https://github.com/revilla-92/3enraya.

En primer lugar, vamos a describir la estructura que sigue esta aplicación a través del siguiente diagrama. Para seguir esta estructura, prime-ro debemos de empezar por los componentes más bajos e ir hacia los componentes más elevados. Para ello empezaremos con el componente de Casilla.



Este componente será el más pequeño de la aplicación y tendrá unos valores (“X”, “0” ó “-“) y para modificar estos valores se accede a través de las “props” de dicho componente “this.props.valor” para cambiar la propiedad del valor de este componente. Así mismo también podemos definirle eventos como es que se le haga click y asociarle una función para que realice unas determinadas acciones. Una vez más y haciendo referencia al diagrama anterior, los eventos irán llamándose desde los componentes más pequeños hacia los componentes más grandes. De este modo el evento click aunque se inicie al pulsar sobre una casilla será el componente APP el encargado de gestionar este evento y de modificar la información (data) hacia los componentes descendientes. Respecto a la información que es cambiada al hacer click, se debe de modificar el texto del componente cabecera indicando el turno del jugador y así mismo se deberá cambiar el valor (this.props.valor) de la casilla en cuestión.

La estructura que se ha seguido en cuanto a los directorios se encuentra en la imagen a la derecha. Con esta jerarquía se quiere ayudar a entender de que se ha utilizado JSX para cada componente de React pero que el componente padre es el “*index.jsx*” y es el fichero que posteriormente se compilará (transpilar) con Browserify mediante el comando:



browserify -t babelify src/js/index.jsx > src/build/index.js

Para que al final se añada un único fichero JavaScript al HTML de la aplicación.

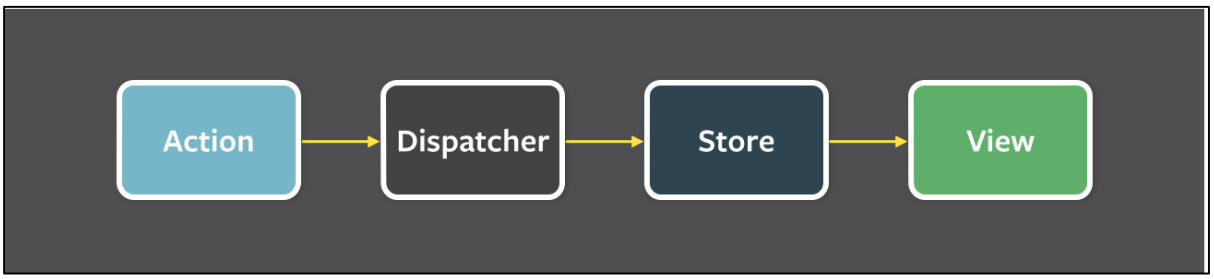
4. FLUX

4.1. ORIGEN Y CARACTERÍSTICAS

Flux es la arquitectura de la aplicación que utiliza Facebook para construir aplicaciones del lado cliente. Complementa perfectamente a React y su visión de "componentes" utilizando "one-way data flow". La historia de Flux está compartida con la de React, ya que Flux nació a partir de la idea que supuso React para complementarla y para a partir de esta construir aplicaciones de forma sencilla y escalable, en definitiva, con la misma mentalidad con la que se creó React.

Se recomienda para llegar a entender correctamente la arquitectura ver el vídeo de la presentación oficial de Flux [41]. En esta presentación se describe como el two-way data binding es una forma sencilla de desarrollar aplicaciones pero que no es muy escalable cuando se dispone de múltiples Vistas (Views) o Modelos (Models) ya que en este momento se puede disponer de muchos flujos en los que una vista actualiza varios modelos y estos a su vez varias vistas e incluso poder llegar a un bucle infinito.

Como se ha mencionado en puntos anteriores esto es el principal problema de la gran mayoría de las librerías más populares en JavaScript y por ello lo que se pretende con Flux es una arquitectura en la que se utiliza el one-way data binding. Como se ilustra en el siguiente diagrama de la página oficial [42].

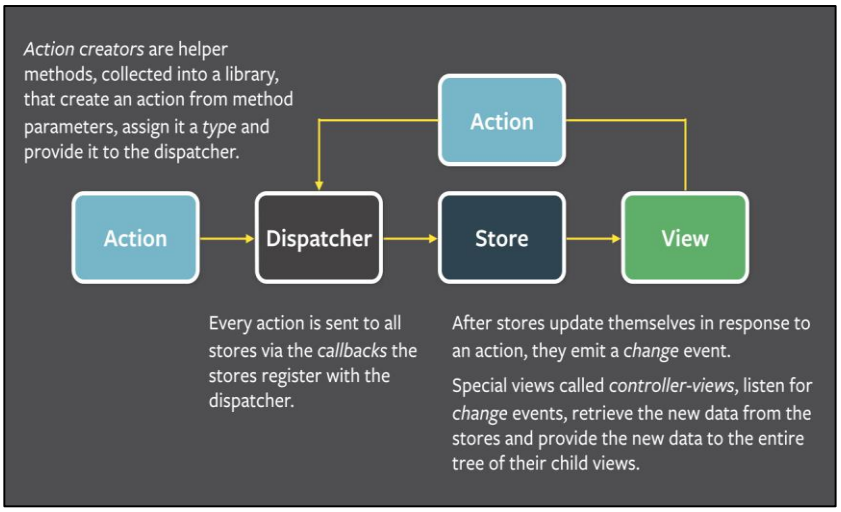


Las aplicaciones de Flux tienen tres partes fundamentales, que vienen descritas en la imagen anterior, que son el "Dispatcher", "Stores" y "Views", estando las vistas en React (Componentes). Esta arquitectura no debería confundirse con el MVC (Model-View-Controller). Los controladores existen en Flux pero son denominados View-Controllers, en otras palabras, vistas que se encuentran en lo más alto de la jerarquía que recogen la información de las "Stores" y la pasan a sus descendientes. Por otro lado, las "Actions" describen todos los cambios que son posibles dentro de la aplicación y aunque anteriormente no se les ha considerado como un parte fundamental, podría ser la cuarta parte fundamental dentro de Flux.

Flux evita el patrón MVC a favor de un flujo unidireccional de información. Cuando un usuario interacciona con una vista (realizada en React), la vista propaga la acción hacia el

"Dispatcher" y este a las "Stores" donde se tiene toda la lógica de las aplicaciones en Flux y que se encarga a su vez de actualizar las vistas que se han visto afectadas por dicha acción. Esto funciona especialmente bien con React y su estilo de programación declarativo , que permite a la "Store" enviar actualizaciones sin especificar cómo hacer la transición entre los estados dentro de las vistas (ya que de esto se encarga React).

Esta arquitectura funciona, no únicamente por el flujo unidireccional de información, sino también porque nada fuera de las "Stores" tiene idea de cómo se gestiona la información. Esta separación ayuda a que ningún agente externo manipule la lógica y el estado de la aplicación. A continuación se analizará cada parte fundamental de Flux.



- **Un Único Dispatcher:**

El "Dispatcher" es el eje central que administra todo el flujo de datos en una aplicación en Flux. Se trata esencialmente de un registro de llamadas de vuelta ("callbacks") hacia las "Stores". Cada "Store" se registra y proporciona una llamada de vuelta y cuando el "Dispatcher" es invocado en un método que genera una acción, se envía hacia las "Stores" a través de las llamadas de vuelta ("callbacks") la carga útil (payload) de estas acciones. En otras palabras, después de que se desata una acción se transmite los cambios del estado de los componentes hacia el Dispatcher y este se encarga de propagarlos hacia las "Stores" (en forma de payload, o de carga útil) que se encargan de modificar convenientemente los componentes afectados por estas acciones. Cuando una aplicación crece este elemento se vuelve de vital importancia ya que el "Dispatcher" puede manejar las dependencias entre varias "Stores" invocando las llamadas de vuelta ("callbacks") de una forma ordenada.

- **Stores:**

Las "Stores" contienen la lógica y el estado de una aplicación. El papel que juegan es algo similar al que juega el modelo (model) en el patrón MVC. Las "Stores" manejan el estado de varios objetos a diferencia del model del patrón MVC que maneja instancias de un objeto. Como se ha mencionado anteriormente, una "Store" se registra a sí misma junto con el "Dispatcher" y le proporciona una llamada de vuelta ("callback"). Esta llamada de vuelta recibe carga útil (información en forma de payload) como parámetro. Esta carga útil contiene una acción con un atributo que identifica el tipo de acción y en las "Stores" se tiene un

conmutador entre estos tipos de acción que interpreta la carga útil realizando las modificaciones en la lógica y en el estado de la aplicación. Esto se ilustrará mucho más claro en el caso práctico, pero en esencia, se alterna por el tipo de acción que ha sido disparada por una acción en la vista (componente) y se recoge la información que se manda a través del componente y que gestiona el "Dispatcher" para modificar el estado de los componentes emitiendo dicho cambio a la aplicación (a los demás componentes).

- **Vistas y Vistas-Controladores:**

En inglés Views y Views-Controllers. Estas vistas se diseñan o programan en React que proporciona la visión de componentes que encaja a la perfección con Flux. Las "View-Controllers" sirven de pegamento entre las "Stores" y la cadena de vistas descendientes que tiene , actualizando las vistas descendientes cuando se dispara un evento como si de una cadena se tratase. El componente que se encuentra en lo alto de esta jerarquía accede al estado de la aplicación mediante los métodos accesores de las "Stores". En primer lugar se actualiza la vista en lo alto de la jerarquía y a continuación en su método render(), pasa los cambios que han sido accedidos desde las "Stores" hacia las vistas o componentes descendientes para que se actualicen del mismo modo. En algunas ocasiones se suele pasar en un objeto toda la información que se debe pasar a toda la cadena de descendientes para que cada uno de ellos utilice la información que necesite. Esto ultimo permite que el numero de propiedades ("props") que tenemos que manejar se reduzca considerablemente. De vez en cuando, si la jerarquía se expande se requiere añadir otro View-Controller más abajo en la jerarquía para mantener los componentes lo más simples posibles y para encapsular una determinada sección de la aplicación. De este modo es más sencillo depurar y encontrar errores. No obstante, esto tiene una contrapartida y es que se puede violar o romper el "one-way data binding" pudiendo introducir puntos en los que el flujo de información sea bidireccional.

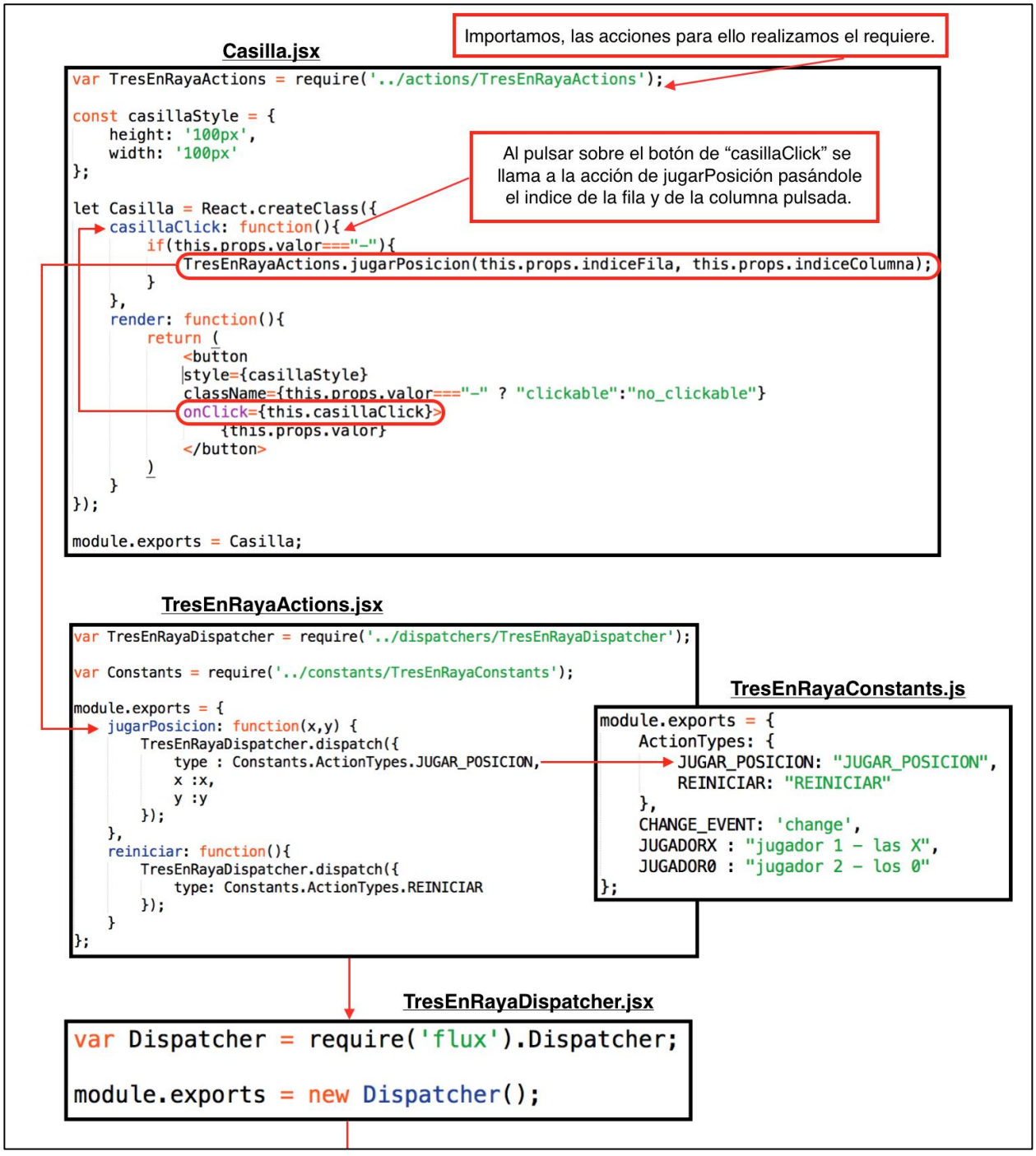
- **Actions:**

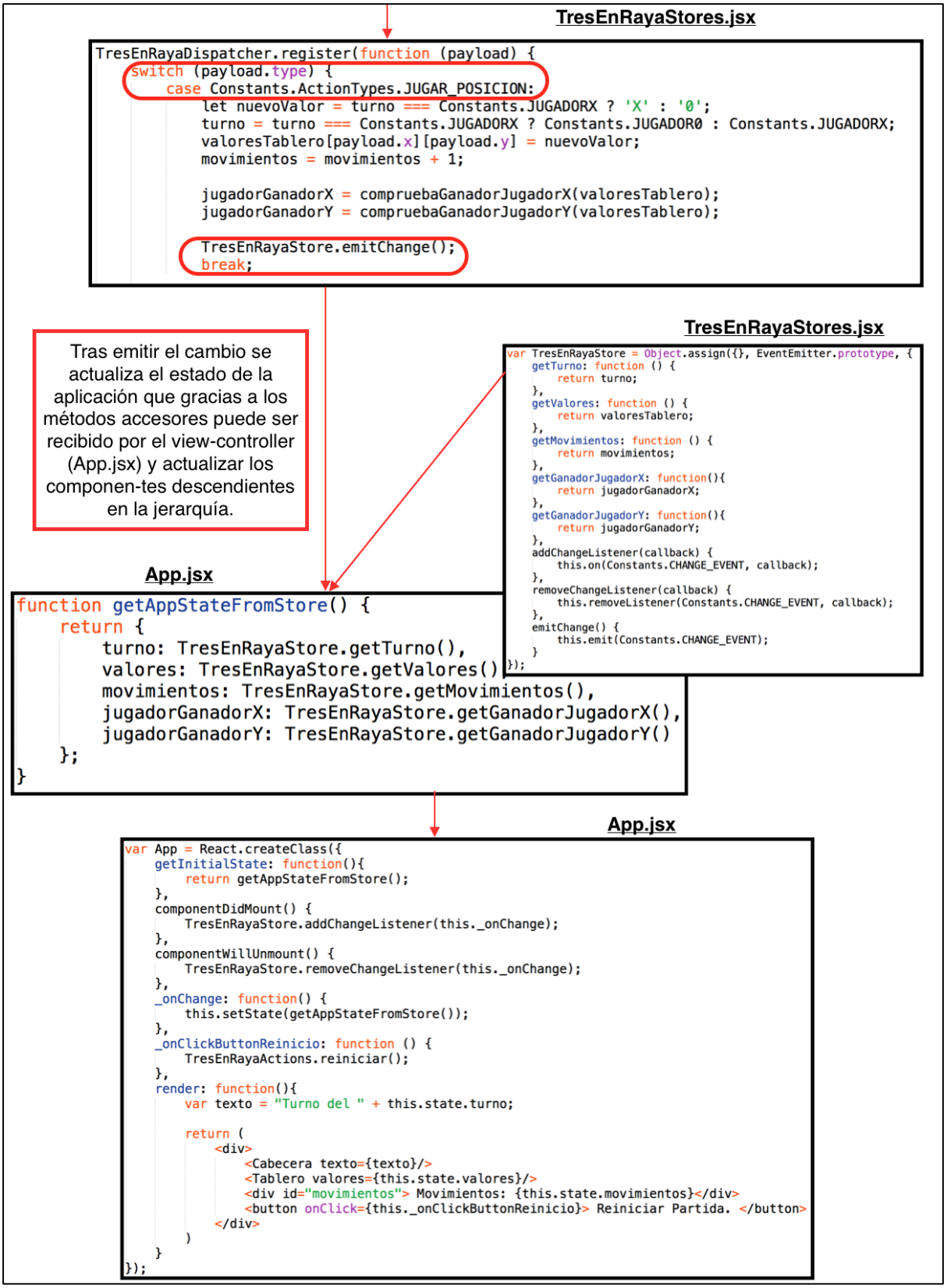
Las acciones son en definitiva un método que se invoca cuando se interacciona con un componente y este manda la acción hacia el "Dispatcher". En esta llamada se le puede pasar información con el cambio que ha solicitado el usuario y que se pasa a través del "Dispatcher" hacia las "Stores". Obviamente no se nos debe de olvidar declarar el método en el componente y asociar esa acción al componente. También es posible tener acciones provenientes de otros lugares como por ejemplo de un servidor. Esto suele pasar cuando durante la inicialización o cuando el servidor devuelve un error o incluso cuando el servidor tiene que actualizar la aplicación.

4.2. CASO PRÁCTICO: 3 EN RAYA.

Como ya ilustramos para el caso de React basándonos en el mismo ejemplo hemos traslado el mismo caso práctico a la arquitectura que proporciona Flux. En primer lugar describiremos la arquitectura aplicada en este ejemplo. Se ha usado un View-Controller que llamaremos "app.jsx" y que se encarga de recoger la información de la "Store" y pasarla a sus descendientes. Así mismo la misma cadena de componentes se puede aplicar en Flux, por lo que la imagen que ilustra dicha jerarquía en React es igual de aplicable en este ejemplo.

Lo que interesa ilustrar es el flujo de eventos en Flux a través del código:





En esencia, vemos como en el componente “Casilla” tenemos una acción que se dispara cuando se pulsa sobre dicho componente (“onClick”). Al disparar este evento se llama a una acción que hemos definido como “jugarPosicion” a la cual se le necesita pasar como

parámetros el índice de la fila (x) y el de la columna (y), los cuales se pasan mediantes las propiedades ("props") de dicho componente.

Así mismo, como paréntesis en el flujo de información, se define una serie de constantes, por ejemplo los tipos de acciones para que a cada acción se le asigne un determinado tipo de acción. De este modo cuando llegue la información a la Store será mucho más sencillo filtrar el payload, ya que éste contendrá el tipo de acción y por tanto por cada tipo de acción nos encontraremos un "caso" dentro del "switch".

A continuación, estos cambios llegan al "Dispatcher", que en nuestro caso es muy transparente y que únicamente pasa el "callback" a la "Store" donde se encuentra almacenada la lógica de la aplicación. Mediante una sentencia "switch" se procesa el payload correspondiente a esta acción para que en consecuencia se realicen unas determinadas acciones que cambien el estado de la aplicación. Finalmente se emite el cambio para que el app.jsx obtenga los cambios procedentes de la "Store" mediante sus métodos accesores y propague los cambios hacia los demás componentes descendientes en la jerarquía.

Adicionalmente y como mejora se han añadido, además de un estilo propio, dos componentes, en los que se cuenten el número de movimientos (o de acciones, del tipo de "jugarPosicion") y un botón para reiniciar la partida (volver a recoger el estado inicial de la aplicación). Todo este código se encuentra comentado en GitHub en el siguiente repositorio:

https://github.com/revilla-92/3enrayaFlux

La estructura que se ha seguido en cuanto a los directorios se encuentra en la imagen a la derecha. Con esta jerarquía se quiere ayudar a entender de que se ha utilizado JSX para cada componente de React pero que el componente padre es el "*index.jsx*" y es el fichero que posteriormente se compilará (transpilar) con Browserify mediante el comando:

browserify -t babelify src/js/index.jsx > src/build/index.js

Para que al final se añada un único fichero JavaScript al HTML de la aplicación. La estructura de directorios además ayuda a entender los elementos que componen la arquitectura de Flux así como el flujo de información que sigue y se ha descrito anteriormente.

5. CASO DE USO: QUIZ

5.1. DESCRIPCIÓN

El caso práctico se ha basado en la misma idea que el proyecto Quiz-2015 de la asignatura de Computación en Red, más concretamente se ha escogido la idea de tener Quiz como un conjunto formado por una pregunta y una respuesta. El proyecto no abarca en tanta profundidad los conceptos que se desarrollaron en la asignatura y se centra más en la aplicación de Flux y en la creación de los Quizes. Se utilizará por tanto la plataforma para almacenar repositorios GitHub, repasando los comandos básicos para su utilización, NPM (Node Package Manager), Flux y React así como la integración de estas tecnologías en un proyecto completo.

En primer lugar, tras registrarnos en GitHub [43], nos creamos un repositorio con un determinado nombre, una descripción opcional e indicamos que queremos inicializarlo con un fichero Readme.md. Clonamos dicho proyecto que se encuentra vacío en cualquier ubicación de nuestro ordenador ejecutando desde la terminal el comando:

git clone https://github.com/NOMBRE_USUARIO/NOMBRE_PROYECTO.git

Una vez clonado el proyecto, desde el directorio donde se encuentra clonado y teniendo instalado Node.js y NPM (Node Package Manager) [44], podemos empezar a realizar nuestra aplicación. Mediante el comando "npm init" nos saltará un asistente en terminal crearnos un fichero JSON (JavaScript Object Notation). Una vez creado podremos instalar los paquetes NPM que vayamos a emplear en nuestra aplicación. Para instalar dependencias (paquetes NPM) ejecutaremos el siguiente comando:

npm install --save NOMBRE_PAQUETE

Con la opción "--save", indicamos que se instalará de forma local en las dependencias de nuestro proyecto. Si seguido de "--save", ponemos "-dev", se añadirá a las dependencias de desarrollo (devDependencies). A continuación, se realizará una breve descripción de los módulos NPM (Node Package Manager) empleados para la realización de este proyecto y los cuales vienen definidos en el fichero package.json. Se describe la función de cada paquete pero su utilización se detalla en la bibliografía y en el propio ejemplo.

El paquete principal de nuestra aplicación es express [45] ya que esta se construirá con dicho módulo. Presenta unas características que le hacen una buena opción como framework (entorno de trabajo). Lo que consigue es construir de forma rápida un servidor HTTP, con un enrutamiento muy fuerte, permite disponer de redireccionamiento, captura de errores además de soportar una gran variedad de motores de plantillas (template-engine).

Una de las principales características de muchas aplicaciones web es que siempre disponen de una apariencia homogénea cambiando el contenido pero manteniendo elementos como la barra de navegación, el pie de página, etc.. Con express-partials [46] se permite definir un diseño común e ir variando el contenido según que vista estemos renderizando de esta manera únicamente debemos de diseñar el código específico de cada vista.

El paquete express-flash [47] es secundario pero muy útil para mostrar mensajes de información adicional como por ejemplo mostrar errores a la hora de rellenar un formulario. Resulta una herramienta visual para añadir mensajes de cualquier índole en la aplicación. Con serve-favicon [50], un sencillo paquete de Node.js, se nos permite pasar la ruta de cual queremos que sea el icono de nuestra aplicación.

Con el paquete express-session [48] se nos permite gestionar las sesiones de los usuarios. Junto con el paquete cookie-parser [49] nos permite, por ejemplo mantener una sesión activa cuando un usuario se "logea" a la aplicación. Además nos permite crear cookies en el navegador para identificar a la aplicación mediante un nombre.

En realidad únicamente es necesario Flux [51] para importar el "Dispatcher" en la parte de la aplicación encargada de añadir los Quizes. En cuanto a React [52] y React-Dom [53] no son necesarios ya que estos se pueden incorporar incluyendo una etiqueta <script> en la vista del layout. Más concretamente estos dos scripts:

```
<script src="https://fb.me/react-0.14.0.js"></script>
<script src="https://fb.me/react-dom-0.14.0.js"></script>
```

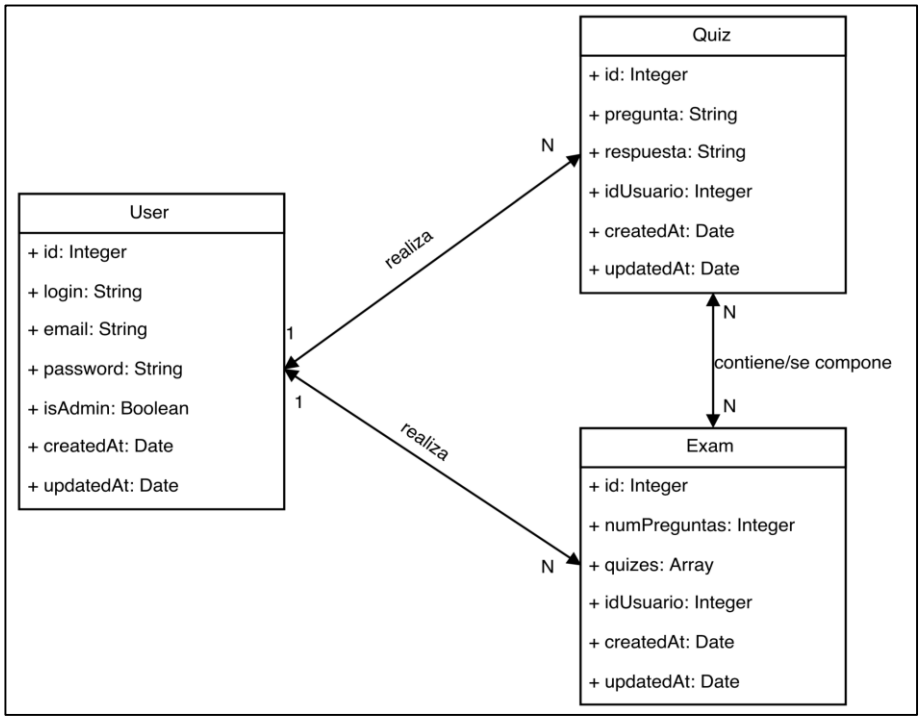
Gracias a la importación de estos dos paquetes podemos emplear los métodos de React y ReactDom, para crear elementos y renderizarlos a la vista. Browserify [54] y Babelify [55] se emplean para convertir todos los ficheros que se generan en la aplicación creada en Flux para "transpilarlos" (compilación ligera) en un único fichero JavaScript el cual incluimos dentro de la vista que contiene la aplicación en Flux. El comportamiento de estas dos herramientas en conjunto es similar al de Webpack descrito con anterioridad.

El paquete fixed-data-table [56] es verdaderamente indispensable para representar información en forma de array en una tabla. Se emplea en la aplicación de Flux para mostrar los quizes que se van creando y añadiendo a la tabla. Viene con plantillas de estilo por defecto que son impresionantes y es muy intuitivo de utilizar.

Los paquetes Sequelize [57] y SQLite3 [58] son los relacionados con las bases de datos. Con Sequelize podemos definir nuestros "esquemas" de las bases de datos y el lenguaje que hemos escrito para que luego se genere la base de datos es SQLite.

Los demás paquetes que no se describen pero sin embargo se encuentran incluidos en el fichero package.json, bien no se emplean o no son muy relevantes dentro de la aplicación. Una vez más aclarar que las descripciones dadas son muy someras y que en los enlaces de la bibliografía vienen con mucho más detalle y con casos prácticos muy útiles.

En cuanto a las bases de datos, se ha utilizado el módulo de NPM Sequelize mientras que la base de datos en sí se encuentra en SQLite (módulo SQLite3). Como se trata de un proyecto ilustrativo se han obviado algunos componentes del proyecto original como los comentarios, los favoritos y los ficheros adjuntos. Esto nos deja principalmente con un diagrama relaciona de bases de datos muy sencillo a la vez que ilustrativo representado en UML (Unified Modeling Language).



Al emplear el módulo de Sequelize se define en cada base de datos los siguientes atributos de forma automática: id, fecha de creación y fecha de actualización. Estos parámetros no son necesarios definirlos y se rellenan por sí solos. Aclarar únicamente que el campo de id es un número entero y las fechas de creación y actualización son objetos del tipo “Date” de JavaScript.

Como en cualquier aplicación web encontramos que siempre es necesario estar registrado para poder acceder a los servicios que proporciona. Por tanto, se define la base de datos de “User” en la que se guardan los datos de los usuarios que se registran en la aplicación. Estos son simplemente el nombre con el que el usuario quiere acceder a la aplicación (“login”), la contraseña y el email. Con estos tres parámetros se define un usuario, ya que existe un parámetro adicional “isAdmin” que por defecto es falso y que permite al administrador de la aplicación realizar determinadas acciones que no están permitidas para el resto de usuarios.

El usuario, una vez registrado en el sistema, puede crear Quiz (pregunta y respuesta) mediante la aplicación de Flux a las cuales se les asigna el identificador del usuario que las ha creado. Hay, por tanto, una relación de 1 a N, es decir, un usuario puede crear varias preguntas o varias preguntas pertenecen a un usuario. Así mismo como función adicional se puede generar un Examen que no es más que una recopilación ("Array") de varios Quiz. Por tanto aquí existen varias relaciones. Una relación 1 a N con los usuarios, ya que un usuario puede crear varios exámenes y, por otro lado, una relación N a N con Quiz, ya que varios exámenes puede contener varias preguntas y viceversa (varios Quizes pueden estar contenido en varios Exámenes).

Cuando se produce una relación N a N es de buena praxis realizar una tabla (una base de datos) con el resultado de la intersección de ambas, es decir, la tabla "Join". Esto se realiza de forma muy sencilla mediante Sequelize y de forma completamente abstracta ya que basta con las sentencias:

```
Exam.belongsToMany(Quiz, {through: 'RelacionQuizExam'});
Quiz.belongsToMany(Exam, {through: 'RelacionQuizExam'});
```

Por tanto, la tabla "Join" se llamará "RelacionQuizExam" y se creará de forma automática con los atributos identificativos de ambas tablas, es decir, sus identificadores (id).

5.2. IMPLEMENTACIÓN

En cuanto a la implementación de la aplicación se puede separar en dos grandes bloques: la aplicación desarrollada en Node.js y la aplicación desarrollada en Flux así como su integración con la aplicación en Node.js. En primer lugar describiremos con detalle la aplicación desarrollada en Flux, seguido de su integración en el proyecto con Node.js. Como son partes separadas se ha desarrollado por separado cada aplicación así pues se dispone del código de la aplicación de Flux en el siguiente repositorio:

https://github.com/revilla-92/FluxQuiz

En este repositorio se incluye la misma versión que se añade a la aplicación en Node.js, con el estilo añadido y la funcionalidad al 100%, a excepción de la interactuación con la Base de Datos que se explicará al final. Análogamente a como se describió en el caso práctico de Flux, pero con más detalle se va a explicar cada componente de la aplicación en Flux: acciones, componentes, constantes, dispatchers y stores. Como aclaración, a partir de este momento, cuando se mencione Quiz se hará referencia al par de valores pregunta y respuesta, pudiendo nombrar estos valores por separados.

- **Constantes:**

```
module.exports = {
    ActionTypes: {
        ADD_QUIZ: "ADD_QUIZ",
        DELETE_QUIZ: "DELETE_QUIZ"
    },
    CHANGE_EVENT: 'CHANGE_EVENT'
};
```

Como constantes se han definido, y de forma similar a como ocurría en el 3 en raya, los tipos de acción que van a tener lugar que son en este caso, la creación y eliminación de un Quiz. Así mismo se crea la constante que utilizaremos cuando se produce un cambio. Los métodos, como por ejemplo, el de emitir el cambio tendrá un parámetro que comprobar cuando se dispara un evento. Estas constantes se harán referencias en los siguientes puntos de la arquitectura de la aplicación.

- **Acciones:**

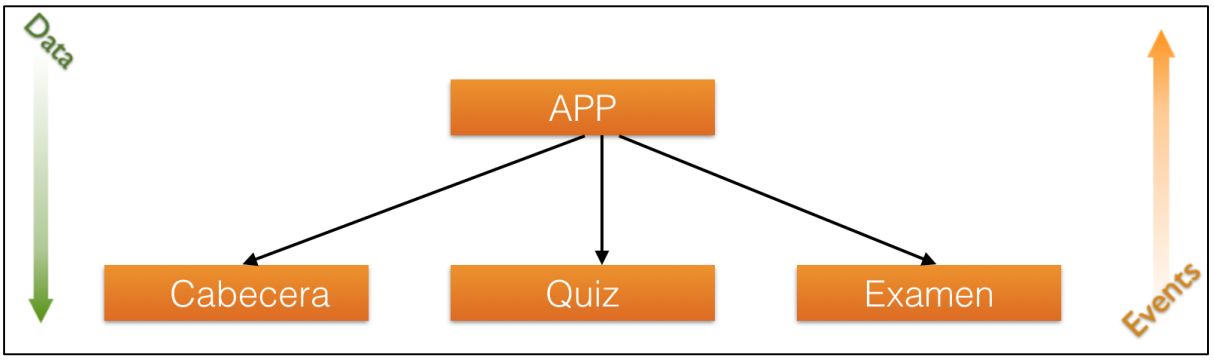
```
var QuizDispatcher = require('../dispatchers/QuizDispatcher');
var Constants = require('../constants/QuizConstants');

module.exports = {
    add_Quiz: function(question, answer) {
        QuizDispatcher.dispatch({
            type : Constants.ActionTypes.ADD_QUIZ,
            question :question,
            answer :answer
        });
    },
    delete_Quiz: function(id) {
        QuizDispatcher.dispatch({
            type : Constants.ActionTypes.DELETE_QUIZ,
            id : id
        });
    }
};
```

Principalmente se han definido dos acciones que son las de crear y eliminar Quizes. La forma que se tiene de representar los Quizes será una tabla, que se explicará más tarde, por tanto lo necesario para añadir un Quiz son los valores de la pregunta y la respuesta asociados al Quiz que se quiere introducir.

Mientras que para eliminar un Quiz, será una acción que se realizará desde la tabla. Valdrá con el índice de la tabla que ocupe, en otras palabras podría decirse que el índice de la fila que ocupe un Quiz en la tabla va a ser su "ID". Así mismo y como se describió en el apartado de Flux respecto a la práctica, se pasa al Dispatcher y este a su vez a la Store, el tipo de acción (que vienen identificados en las constantes). De esta forma se puede separar el payload que envía el Dispatcher a la Store por el tipo de acción que se ha disparado y filtrar con la sentencia "switch" en varios casos.

- **Componentes:**

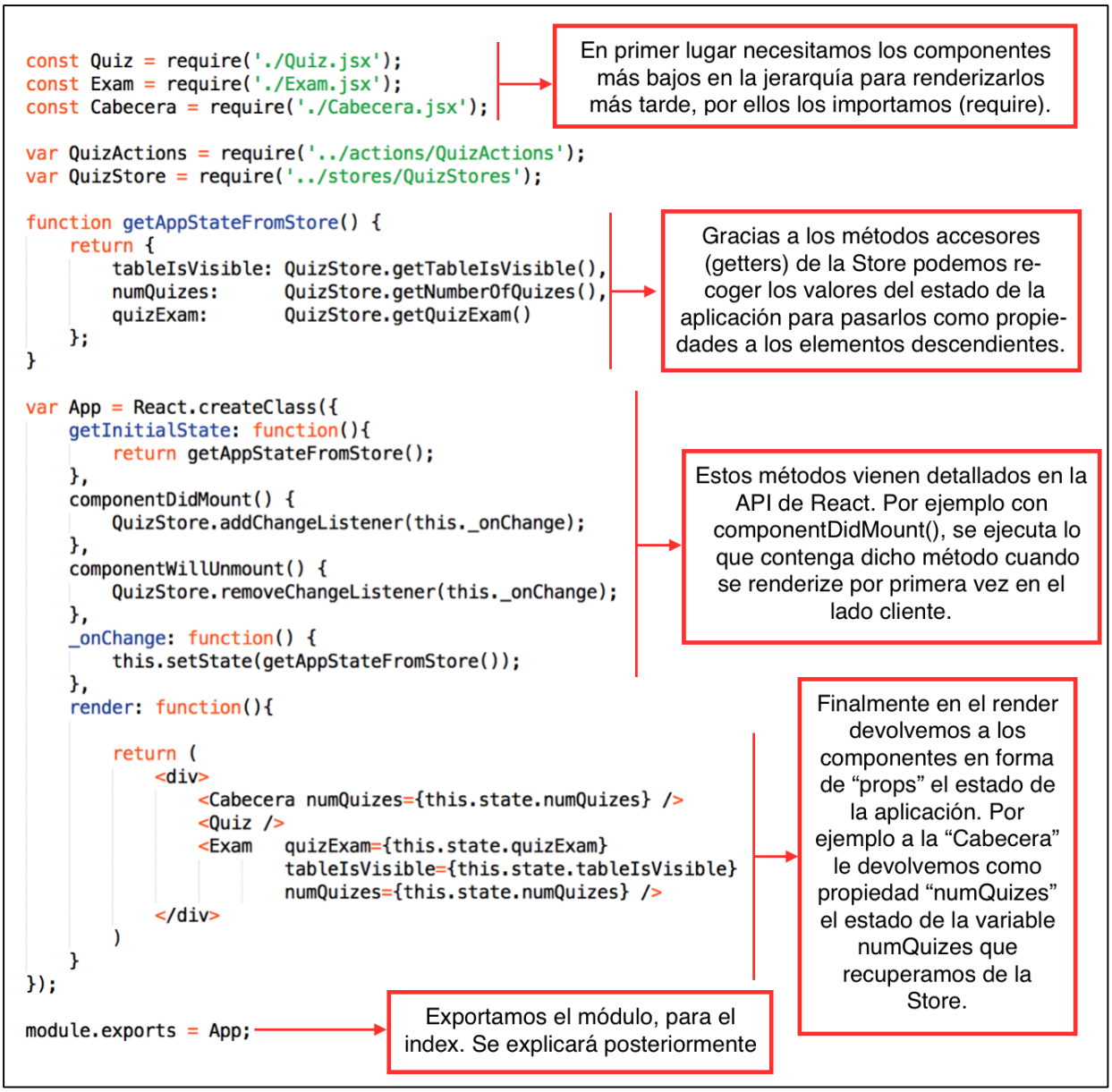
Los componentes o las vistas que componen nuestra aplicación son cuatro, tres "views" y un "view-controller". El "view-controller" es el fichero llamado app.jsx y el que se encarga, como hemos descrito anteriormente de gestionar y mandar el estado que recibe a sus descendientes en este caso a la Cabecera.jsx, Quiz.jsx y Exam.jsx.



Por tanto dentro de las vistas, podemos distinguir componentes pasivos, aquellos que no generan eventos. En este caso son la Cabecera, que muestra el numero de Quizes creado y una parte del componente Exam, que es la representación de los Quizes creados en una tabla. Por lo que los componentes activos o generadores de eventos son el Quiz y el botón que se incluye dentro de la tabla del componente examen que permite eliminar los Quizes. A continuación vamos a ir detallando cada componente.

En primer lugar, la App es el view-controller y por tanto el elemento más alto en la jerarquía de componentes. Se encarga de recoger, a través de los métodos accesores que proporciona la Store, el estado de la aplicación y propagarlo a los elementos por debajo en la jerarquía. En la página oficial de React [13] encontramos una API bien detallada y con ejemplos de cómo emplear los métodos de los componentes. En este caso, como App es el elemento view-controller necesita determinados métodos.

Con componentDidMount() y con componentWillUnmount() se ejecutarán cuando por primera vez se cargue la aplicación en Flux por primera vez y cuando se cierre, respectivamente. Mientras que con getInitialState() y con _onChange(), cogemos el estado de la aplicación en el momento inicial (valores por defecto asignados en las stores), y cada vez que se produzca un cambio, respectivamente.



A través del método render() propagamos dichos cambios a los demás componentes descendientes pasándoles el estado. Como viene detallado en la imagen. Por tanto, como se puede apreciar jugamos con varios estados en los que pasamos el número de Quizes creados al componente Exam y Cabecera ya que si este es cero estos componentes muestran un mensaje de que aún no se ha creado ningún Quiz o directamente no se muestran. Y finalmente el estado más importante es el denominado “quizExam” donde se pasa en forma de array bidimensional (pregunta, respuesta) los quizes a la vista del “Exam”.

Finalmente respecto a la App hemos visto que exportamos dicho módulo que se utiliza en el index.js. Este no requiere pasar nada al componente App y es el fichero que se necesita para que mediante Browserify se “transpile” a un único fichero JavaScript el cual importaremos en nuestro documento

```
const App = require("./components/App.jsx");

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

HTML, mediante una etiqueta <script>. En esta etiqueta cargaremos el resultado de la transpilación realizada mediante Browserify y Babelify, mediante el siguiente comando:

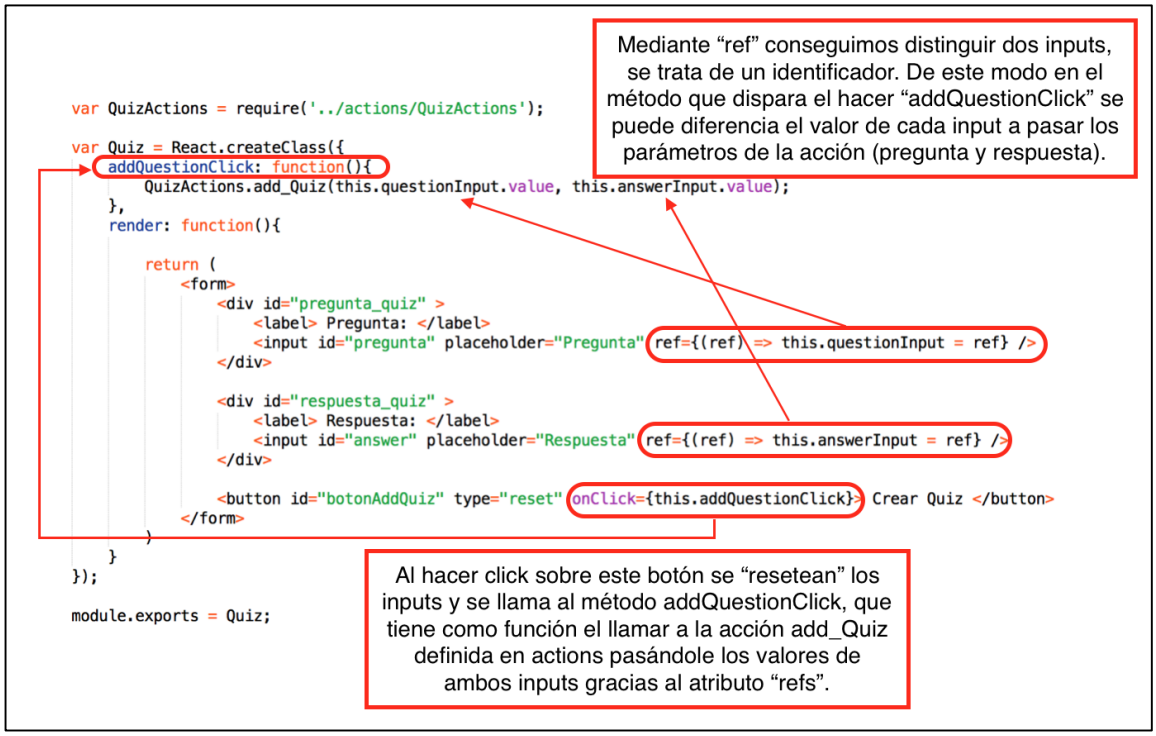
browserify -t babelify public/src/js/index.jsx > public/src/build/index.js

Por tanto al añadirlo a nuestro fichero HTML veremos la aplicación resultante, tratándose de una SPA (Single Page Application).

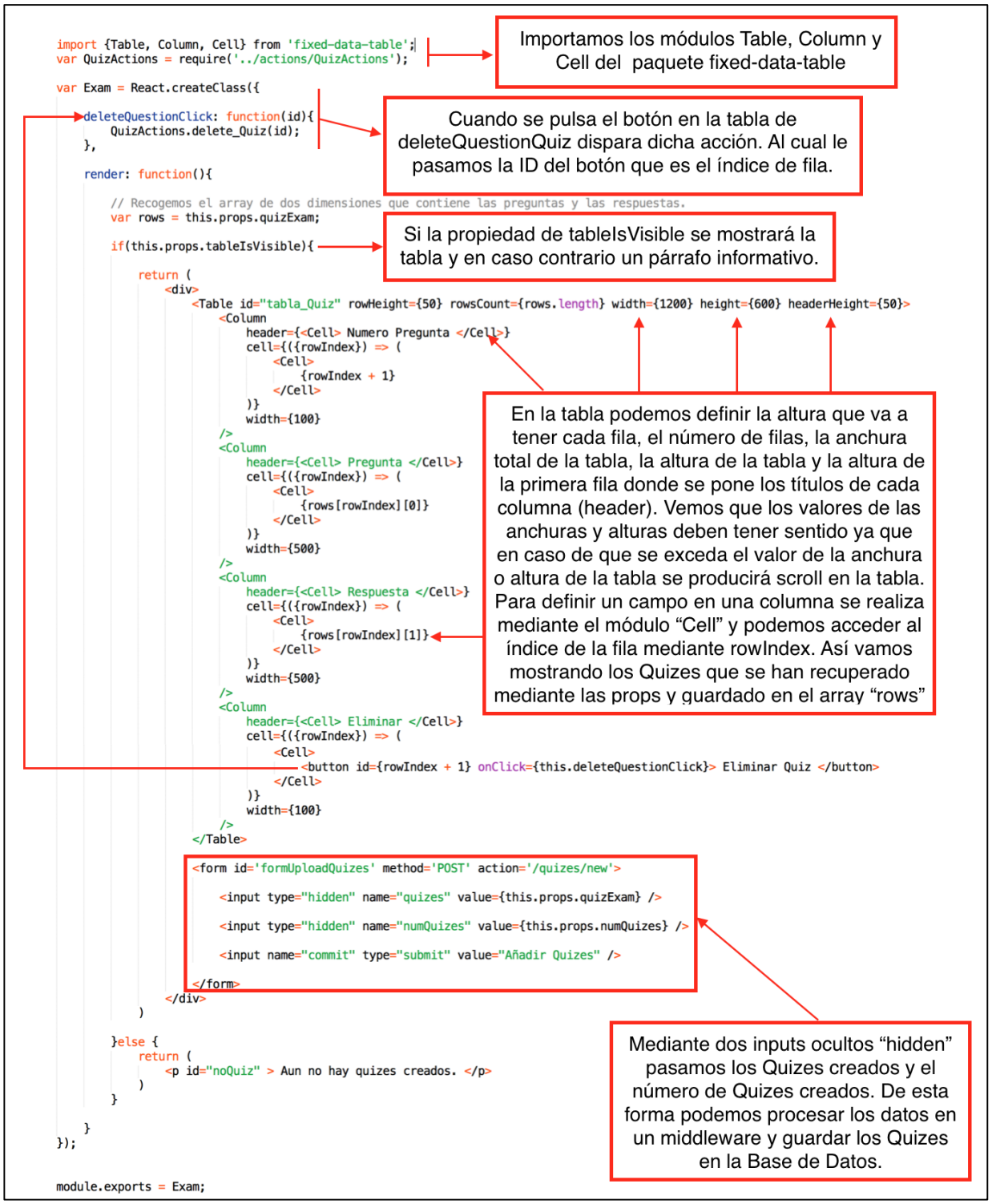
```
<div id="contenedor"></div>
<script type="text/javascript" src="/src/build/index.js"></script>
```

El componente mas secundario es el de la Cabecera el cual es muy similar al del ejemplo de 3 en raya ya que muestra el número de Quizes que se ha creado hasta el momento. Se trata de un componente completamente pasivo el cual muestra la información si el numero de Quizes creados es mayor que cero, y en caso contrario no muestra nada. Como vimos de la App, se pasa el número de preguntas en la propiedad "numQuizes" por tanto en el componente Cabecera accederemos a dicho valor mediante "this.props.numQuizes".

Un componente importante es el Quiz, que incluye el formulario para crear Quizes.



Del componente Quiz, se debe destacar las referencias a etiquetas input para distinguirlas entre sí y una llamada a una acción. Por último, el componente clave y que sirve de puente entre Flux y Node.js es el Exam. En este componente tenemos que destacar dos cosas principalmente, que son la representación de la información gracias al paquete fixed-data-table y la subida de dicha información a la base de datos de Quizes. Respecto al paquete fixed-data-table, necesitamos importar los componentes "Table", "Column" y "Cell". Con estos tres componentes podemos definir una table en la cual tenemos que tener en cuenta que en vez de definir cada fila como se realiza en HTML se definen las columnas.



En cada columna definimos la primera fila denominada "header" en la cual introducimos el título de lo que va ir representado en dicha columna (similar a la etiqueta <th> en HTML). Mediante el componente "Cell" introducimos el contenido que irá en cada fila por ello cogemos siempre la misma columna en la información que queremos representar.

Gracias a data-fixed-table, se puede de manera muy intuitiva construir una tabla representando la información en un array. En nuestro caso el array con la información de preguntas y respuestas se denomina "rows". Mediante los atributos de Table podemos definir

la altura de cada fila así como la anchura de cada columna, lo que no resulta una buena praxis, puesto que mezcla estilo con semántica pero que se encuentra muy bien implementado por el paquete data-fixed-table.

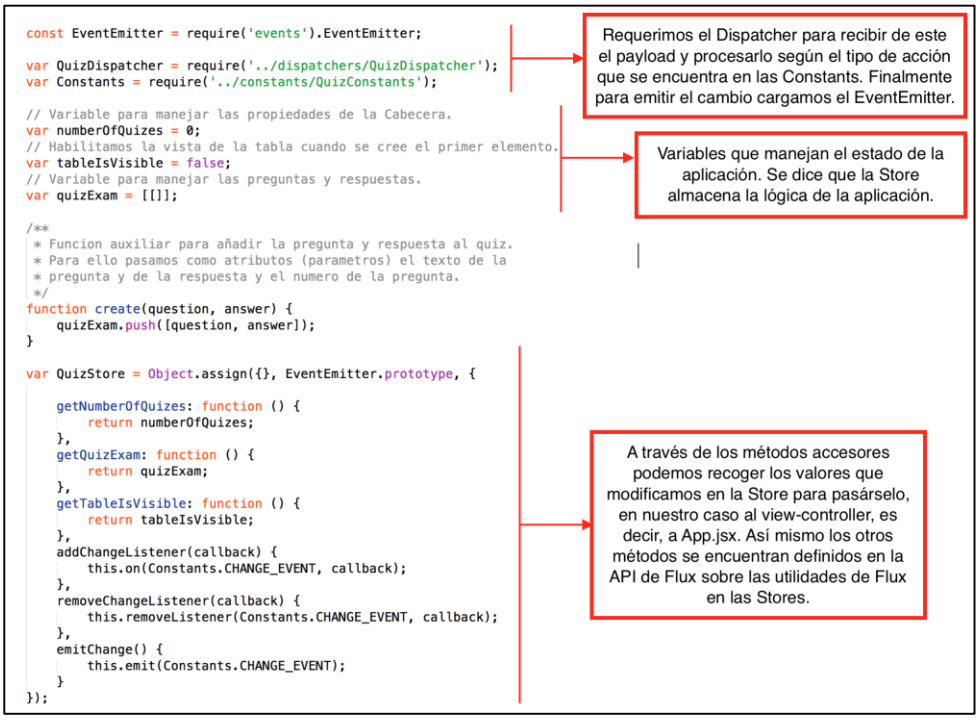
Por otro lado, en la última columna de dicha tabla se añade la opción de eliminar una fila entera de dicha tabla. Este método es mucho más sencillo que el de crear puesto que basta con suministrar el número de la fila a eliminar, su identificador, y eliminar dicha posición del array de Quizes. Finalmente, y acabando con los componentes, cabe destacar el formulario al final de dicho componente que realiza una petición post en el cual encontramos dos inputs del tipo "hidden" en los que pasamos el array de los Quizes y el número de preguntas creadas.

Una vez pulsado el botón de "Añadir Quizes" se procesa en el middleware correspondiente esta petición procesando el array bidimensional separándolo en dos arrays, el de preguntas y el de respuestas y finalmente añadiéndolas a la base de datos de Quizes.

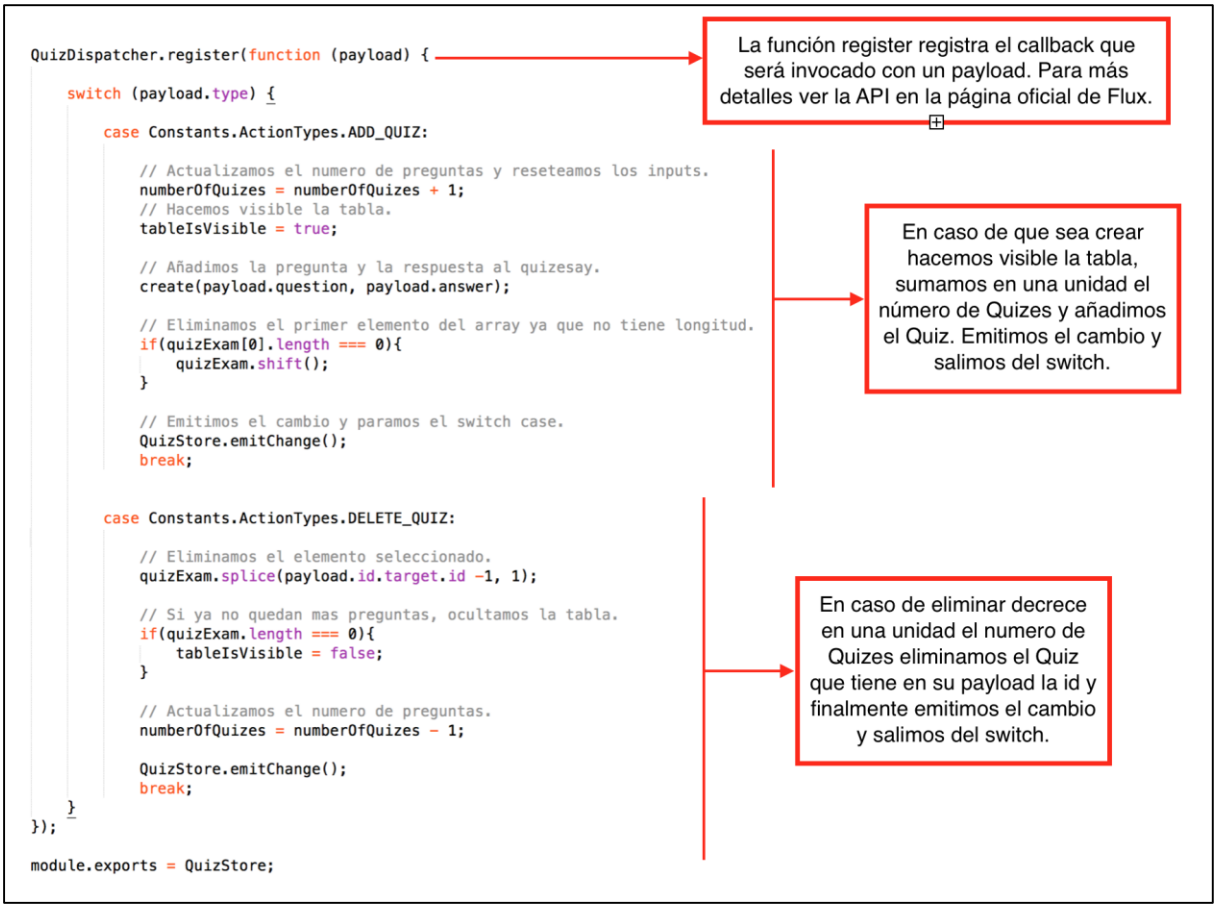
- **Stores:**

Con las Stores lo que realizamos es principalmente el manejo del array de Quizes ya que es aquí en donde se procesa y se almacena toda la lógica. Aquí manejamos las variables que pasamos al estado de los componentes como por ejemplo el hacer visible la tabla con los Quizes creados o el número de preguntas.

Aunque bien es cierto y como se comentó en el apartado de Flux se hubiera podido pasar toda la información en un mismo objeto. Esto resultaría una mejor praxis para esta arquitectura y en este ejemplo en concreto. No obstante, el haber desglosado cada variable por separado en vez de tenerlas en el mismo objeto ilustra de mejor forma como funcionaría cuando una aplicación crece y se hace inviable dicha praxis.



En la imagen anterior, se ilustra como se crean los métodos accesores para que luego a partir de estos la "View-Controller", es decir, la App.jsx pueda pasar la información manipulada por la Store a las vistas descendientes. Finalmente recalcar que cuando se procesan los cambios según qué acción es de vital importancia emitirlos y detener la sentencia "switch" con un "break".




```
QuizDispatcher.register(function (payload) {

    switch (payload.type) {

        case Constants.ActionTypes.ADD_QUIZ:

            // Actualizamos el numero de preguntas y reseteamos los inputs.
            numberOfQuizes = numberOfQuizes + 1;
            // Hacemos visible la tabla.
            tableIsVisible = true;

            // Añadimos la pregunta y la respuesta al quizesay.
            create(payload.question, payload.answer);

            // Eliminamos el primer elemento del array ya que no tiene longitud.
            if(quizExam[0].length === 0){
                quizExam.shift();
            }

            // Emitimos el cambio y paramos el switch case.
            QuizStore.emitChange();
            break;


        case Constants.ActionTypes.DELETE_QUIZ:

            // Eliminamos el elemento seleccionado.
            quizExam.splice(payload.id.target.id -1, 1);

            // Si ya no quedan mas preguntas, ocultamos la tabla.
            if(quizExam.length === 0){
                tableIsVisible = false;
            }

            // Actualizamos el numero de preguntas.
            numberOfQuizes = numberOfQuizes - 1;

            QuizStore.emitChange();
            break;
    }
});

module.exports = QuizStore;
```

Al igual que ocurre con otros lenguajes de programación, como el caso de Python, el código es muy entendible una vez se comprende el flujo de información que sigue una aplicación diseñada con Flux. Para comprender una vez más el flujo de la aplicación se recomienda ver nuevamente la imagen ilustrada en el apartado de Flux, ya que ocurre de forma análoga en este caso práctico.

Para concluir con la sección de Flux y React, se ha implementado una funcionalidad que merece ser desatacada y es la posibilidad de realizar peticiones AJAX (Asynchronus JavaScript And XML) al servidor para cargar los Quizes guardados en la Base de Datos en nuestra aplicación SPA sin tener que recargar la aplicación. Para ello en primer lugar debemos de importar JQuery en el layout de nuestra aplicación:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
```

El funcionamiento es sencillo. En primer lugar debemos de tener en cuenta que para recoger los Quizes de la base de datos primero tenemos que hacer una petición al servidor. Por ello

debemos de proporcionar una URL a la APP, en nuestro caso se ha escogido "/api/quiz". Por tanto y como se ha descrito como funciona el uso de "props" pasamos a la APP dicha URL para una vez en la APP realizar una petición AJAX al servidor.

```
ReactDOM.render(
    <App url="/api/quiz" />,
    document.getElementById('contenedor')
);
```

Una vez en la App, creamos una función para que esta sea invocada cuando se pulse sobre un botón (aunque bien podría ser cualquier otro tipo de evento). Realizamos una petición por tanto a "/api/quiz", sabiendo que el tipo de información que se va a recibir como respuesta es del tipo JSON (JavaScript Object Notation). Si la llamada ha sido realizada con éxito se devuelve la información recogida en el servidor como una variable JSON llamada "data". A continuación mediante el método "setState", establecemos el estado de una variable que también llamaremos "data" con la información recogida de dicha petición AJAX.

```
loadQuizesFromServer: function () {
    $.ajax({
        url: this.props.url,
        dataType: 'json',
        cache: false,
        success: function(data) {
            this.setState({data: data});
        }.bind(this),
        error: function(xhr, status, err) {
            console.error(this.props.url, status, err.toString());
        }.bind(this)
    });
},
```

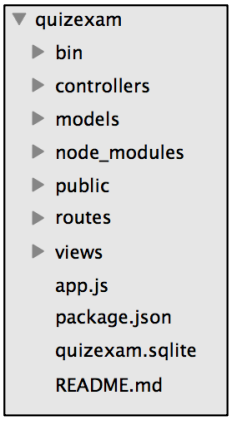
Una vez tenemos el resultado de la petición del estado en la aplicación simplemente debemos de pasar dicha información al componente Exam mediante sus props y de la misma que se ha explicado anteriormente mostramos dicha información en la tabla. Por el lado del servidor debemos de preparar el controlador para que al recibir la petición se envíe el resultado apropiado.

```
exports.loadQuizesToFlux = function(req, res, next) {

    models.Quiz.findAll()
    .map(function(quiz){
        return [quiz.id, quiz.pregunta, quiz.respuesta];
    })
    .then(function(quizes){
        console.log(quizes);
        res.send(quizes);
    })
    .catch(function(error){
        console.log("Error:", error);
        res.send([]);
    });
}
```

Por ello, cuando realizamos una petición del tipo “get” se encuentra en el fichero “routes/index.js” que cuando se realiza una petición “get a /api/quiz se encarga de ejecutar el middleware “loadQuizesToFlux” que encontramos en la imagen anterior. Con este middleware básicamente realizamos una petición a la base de datos de encontrar todos los Quizes creados. Mediante la función “map” hacemos que, en caso de que la promesa se cumpla, el formato con el que se devuelvan los datos de la petición de “findAll()” tengan el formato de array únicamente con los atributos id, pregunta y respuesta.

Finalmente si no se ha producido ningún error enviamos los “quizes” que recogerá adecuadamente la aplicación en Flux. Si por el caso contrario, se ha encontrado un error, al tratarse de una petición asíncrona devolveremos un array vacío. Las peticiones AJAX, son un complemento muy útil para SPA (Single Page Application). Estas proporcionan la sensación al usuario, sin tener la necesidad de cambiar de vista ni cargar contenido, tener interactuación también con el servidor.

Se da por finalizado la implementación de la parte de la aplicación relacionada con Flux. A continuación se comentará someramente la aplicación realizada con Node.js puesto que no es objeto de este Trabajo de Fin de Grado y ya han sido vistas en asignaturas como en Computación en Red. En primer lugar para la aplicación se ha seguido el patrón MVC (Model-View-Controller) que se ha explicado anteriormente. Comprender el funcionamiento del patrón MVC sirve para estructurar la jerarquía de directorios en nuestra aplicación como podemos ver en la siguiente imagen.



Se han explicado anteriormente los módulos que hemos empleado e instalado mediante NPM así como las bases de datos empleadas, lo que constituye los modelos (models). Por tanto queda por matizar las vistas y los controladores empleados así como la aplicación en general.

Comenzaremos por los niveles superiores de la aplicación, hablamos del fichero “app.js”, en este fichero es donde requerimos los módulos que hemos instalado y los iniciamos. Creamos en primer lugar la aplicación mediante express. Una vez creada empleamos en la aplicación módulos como express-sessions, express-partials y finalmente añadimos un “Router” una función que proporciona express para llevar el encaminamiento de nuestra aplicación. Esta aplicación se exporta al fichero “www” del directorio “bin” donde indicamos un puerto por defecto para que se arranque la aplicación cuando realicemos desde una terminal “npm start”.

En el fichero “routes/index.js”, se encuentran todas las posibles rutas que puede tener nuestra aplicación. En caso de que no se hallen la ruta se mostrará un error 404 (“Not Found”). Afortunadamente, debido a express-partials la aplicación mostrará en parte del contenido el error por lo que no sería un fallo abrupto y molesto. Las posibles acciones y más básicas son “get” y “post”. La primera de ellas se realiza principalmente cuando se va a renderizar una

vista mientras que la segunda es cuando se van a realizar cambios sobre la misma o sobre algún objeto del modelo.

La jerarquía de directorios tanto de las vistas como de los controladores es la misma lo que es de gran ayuda a la hora de organizar cuando y qué controlador actúa sobre cada vista. Comenzando por los controladores, se encargan de gestionar las peticiones que llegan de los usuarios mediantes el "Router" (routes). El controlador de los usuarios se encarga de listar, crear, eliminar, editar y mostrar las vistas que permiten realizar tales acciones. Es el elemento intermedio encargado de realizar también comprobaciones de parámetros y de gestionar errores para capturarlos y aislarlos.

Por su parte el controlador encargado de gestionar las sesiones (session_controller.js) se encarga de renderizar las vistas para autenticarse en la aplicación e iniciar una sesión. Tiene funciones como crear, requerir estar "logeado", destruir, autodestruir pasado un tiempo la sesión, etc… Esto es de vital importancia, ya que en el enrutador (routes) podemos poner una cadena de middlewares (controladores intermedios) antes de renderizar una vista o de realizar cualquier acción. Por ejemplo, antes de ejecutar la edición de un usuario se debe de comprobar que se está previamente logeado en el sistema, que el usuario a editar es el mismo que está logeado, etc…

En definitiva, los controladores sirven para realizar comprobaciones disparadas por las acciones de los usuarios y de gestionar las vistas, además de encargarse de modificar la información de las bases de datos. Por ello, operaciones como crear, editar y eliminar requieren acceso a la base de datos (al modelo, model). Gracias a Sequelize se disponen de métodos para modificar las bases de datos. Todos estos métodos devuelven una "promesa" que deberemos de comprobar a posteriori si lo que se devuelve de dicho método o "promesa" es lo que se estaba buscando y en caso contrario capturar el error y mostrarlo.

Finalmente hay que añadir que se debe de tener un manejo del lenguaje JavaScript en especial con los arrays y recalcar que es de vital importancia tener un código bien "traceado", es decir, que contenga trazas para saber en que punto de la aplicación da fallo. Siguiendo una buena praxis se puede depurar fallos de forma más rápida y eficiente así como saber que se recoge de las peticiones (request o req). Esto último resulta imprescindible para realizar la llamada "comprobación de parámetros" que es en muchas ocasiones puntos de fallo típicos en un gran número de aplicaciones.

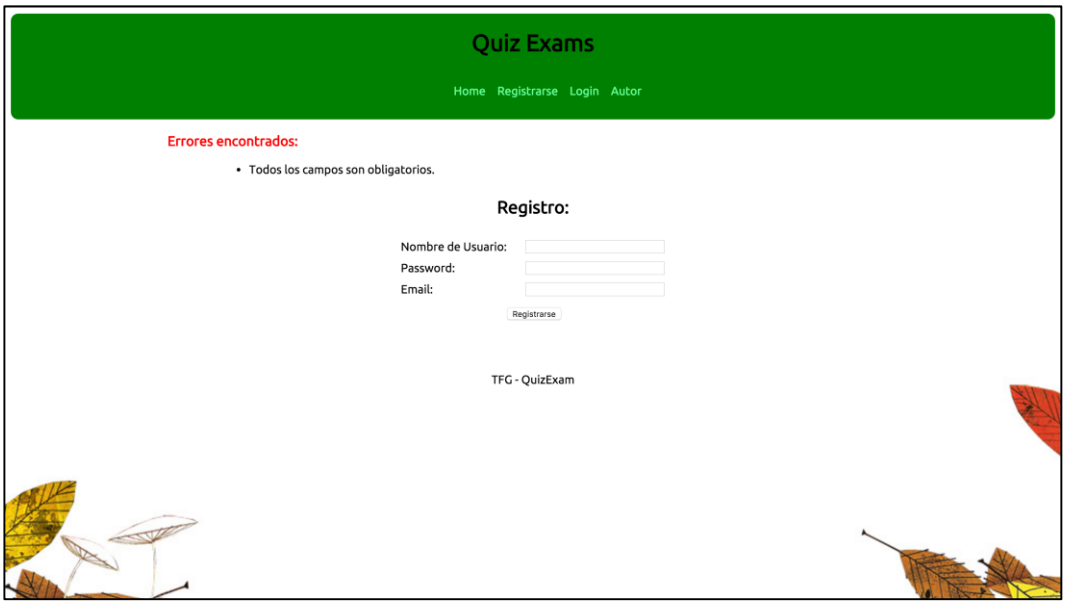
Todo el código con los comentarios incluidos se puede encontrar en el siguiente repositorio de GitHub:

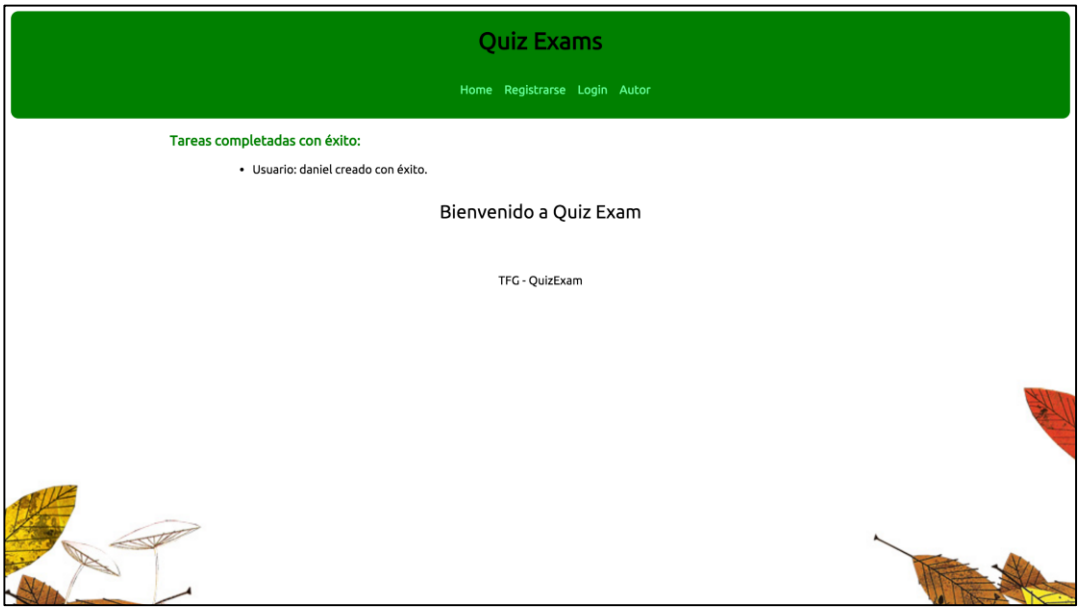
https://github.com/revilla-92/quizexam

5.3. RESULTADOS

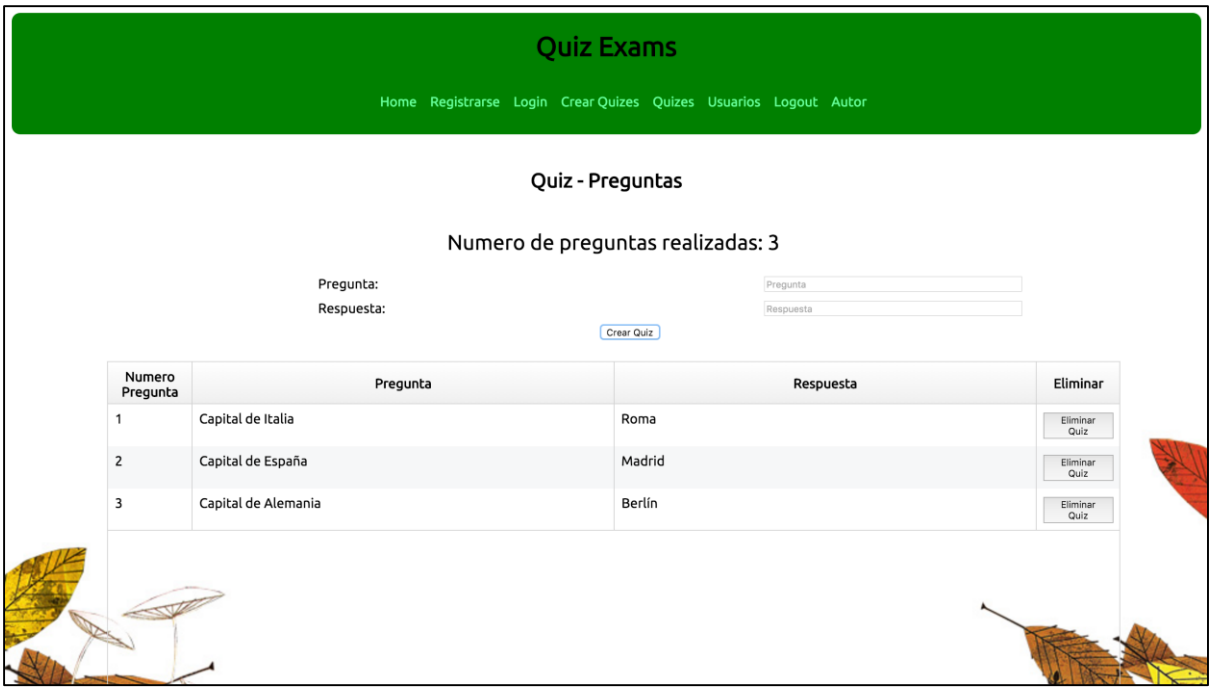
La aplicación ha sido dotada de un estilo, que se puede encontrar en el directorio público junto con las imágenes y ficheros JavaScript que son requeridos bien por las propias hojas de estilo o por las vistas. También se le ha añadido una fuente de Google Fonts [59]. Como se mencionaba durante la implementación de la aplicación al realizar comprobación de parámetros en caso de detectar errores, se notificaba al usuario cual había sido el error en lugar de detener la ejecución de la aplicación.

Como por ejemplo, cuando un usuario no introducía todos los campos en un formulario, o introducía erróneamente su nombre de "login" y contraseña. Para estos mensajes informativos se ha utilizado express-flash, con distintos identificadores, como por ejemplo en caso de error o de éxito.





Una vez más el resultado obtenido de emplear el módulo fixed-data-table es impresionante ya que proporciona estilos por defecto y que vienen muy bien detallados en la página web oficial. Dando un uso adecuado como se mencionó durante la implementación se consigue una aplicación muy vistosa con una reducción de trabajo considerable.



La aplicación desarrollada en Flux resulta, a ojos del usuario, muy interactiva ya que podemos ver cambios en la aplicación sin necesidad alguna de interactuar con el servidor, es decir, tenemos la sensación SPA (Single Page Application). La tendencia hacía este tipo de aplicaciones es evidente ya que proporciona una interacción increíble.

6. CONCLUSIONES

6.1. CONCLUSION

A lo largo de este proyecto se ha realizado un análisis de las distintas librerías JavaScript. Esta comparación me ha resultado muy útil ya que se ilustra los problemas que se encuentran en las mismas a la vez de contextualizar históricamente el origen de estas. La evolución de JavaScript ha sido vertiginosa cambiando el status quo constantemente, resultando intrigante las líneas futuras de este lenguaje a la vez que desesperante por el continuo cambio que implica un estudio rápido de las nuevas librerías.

Se puede extraer como conclusión que la tendencia, después de haber estudiado el panorama actual de JavaScript, es el desarrollo de SPA (Single Page Applications). Esta tendencia se justifica por dos sencillas razones. La primera es que permiten una mayor interacción en el lado cliente ya que no se depende de tiempos de carga con un servidor. La segunda, por otro lado, permite al servidor estar más "despreocupado" ya que solo se necesita el uso del mismo en momentos ocasionales aliviando la carga que tienen los servidores.

En cuanto a React, por sí solo, es un lenguaje que ofrece muchas posibilidades dentro de lo que es la vista (View). No obstante, a nivel personal, la realización de React se hace máxima cuando se junta con Flux que proporciona una arquitectura muy sencilla y que una vez entendida es muy sencillo llevar a la práctica. Este es, en esencia, el objetivo principal de este Trabajo de Fin de Grado, abordar de la forma más ilustrativa estas tecnologías. De igual forma me ha pasado durante la realización del mismo, al entender bien las características principales, he conseguido avanzar a nivel práctico más rápidamente.

En Flux, es de vital importancia el comprender el flujo que sigue la información, en lo cual se ha hecho especial hincapié, ya que resulta ser un lenguaje muy escalable y por tanto fácilmente de replicar otra acciones y nuevas funcionalidades. Las imágenes que muestran bloques de código seguidos con flechas representan este flujo de información, el one-way data binding y que muestra la abstracción conceptual e ilustrativa que se ha exigido en este proyecto.

El caso práctico de "QuizExam" me ha servido, personalmente, para profundizar a un nivel mayor del que se dio en la asignatura Computación en Red (CORE) puesto que ha sido realizado desde cero. Se ha simplificado su implementación para conseguir un ejemplo más ilustrativo y más profundo de los conocimientos adquiridos en las asignaturas del DIT (Departamento de Ingeniería de Sistemas Telemáticos). Asignaturas como FTEL (Fundamentos de los Sistemas Telemáticos), PROG (Programación), CORE (Computación en Red), IWEB (Ingeniería Web) e ISST (Ingeniería de Sistemas y Servicios Telemáticos.

6.2. LÍNEAS FUTURAS

En cuanto a los siguientes pasos solo el tiempo podrá determinar la evolución y la importancia que llegará a tener React y Flux. Personalmente, debido al apoyo que reciben desde Facebook e Instagram y las simplificaciones que permiten tanto en el código como a nivel de flujo de aplicación, creo que tendrán una impresionante evolución. Desde luego su acogida inicial ha sido enorme. Conceptualmente, es verdad que se irán migrando aplicaciones con el "two-way data binding" hacia el "one-way data binding" y que este es un concepto más duradero en el tiempo.

Así mismo, el concepto de "Componente" en la web empieza a ser más popular, ya no solo lo encontramos en React sino también en otras librerías nuevas como Polymer. Sin embargo, en este concepto creo que si puede haber muchos cambios mientras que en la arquitectura, es decir en Flux, se producirán en menor medida. En cuanto al futuro de las librerías más populares en JavaScript (y también más antiguas), estas se encuentran en la necesidad de evolucionar para mantenerse, como es el caso de AngularJS 2.0.

En cuanto al caso práctico, si bien es cierto que se han abarcado las funcionalidades básicas, se pueden añadir más. Por ejemplo, funciones para la edición de los Quizes dentro de la aplicación de Flux que se implementaría sencillamente con un botón que al pulsarlo enviase a los "inputs" los valores de la pregunta y la respuesta a editar y que al guardarlos se actualice el estado. Así mismo, una opción muy atractiva es la de "examinar", es decir, generar un JSON con el estado, en otras palabras, un examen, que respondiera un alumno y que al terminarlo supiera la nota obtenida.

Por otro lado, en cuanto a la aplicación de Node.js, se podría integrar el resto de funcionalidades que para este Trabajo Fin de Grado se han obviado. Hablamos de la integración de los comentarios tanto en los quizes como en los exámenes así como la carga de archivos adjuntos (imágenes) en los exámenes. Se podría añadir la relación que puede existir entre un usuario y los exámenes como puede ser la de "favoritos". Así como la inclusión de estadísticas obtenidas sobre la realización de los exámenes, como las calificaciones más altas, nota media obtenida en los exámenes, etc…

Igualmente, en mi opinión, recomiendo, la integración de la aplicación de Flux en Computación en Red para ilustrar en buena medida el "one-way data binding" ya que es sin duda alguna la tendencia que va a seguir las aplicaciones web. Así mismo en un proyecto tan completo como resulta el Quiz de CORE y en el cual se proporciona gran parte de la funcionalidad realizada, resultaría muy ilustrativo la integración de Flux y React en dicho proyecto a un nivel sencillo y explicativo como se ha seguido en este Trabajo de Fin de Grado.

7. BIBLIOGRAFÍA

[1] https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript

[2] http://prototypejs.org/

[3] http://jquery.com/

[4] http://www.dojofoundation.org/

[5] http://mootools.net/

[6] http://www.javascripting.com/

[7] https://www.ikdoeict.be/leercentrum/slides/javascript/02_dom.html#/dom-full-version

[8] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[9] http://backbonejs.org/

[10] https://angularjs.org/

[11] http://emberjs.com/

[12] http://getbootstrap.com/

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

[14] http://facebook.github.io/flux/

[15] https://webpack.github.io/

[16] https://iojs.org/

[17] https://nodejs.org/en/

[18] https://en.wikipedia.org/wiki/Less_(stylesheet_language)

[19] http://compass-style.org/

[20] https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel

[21] https://medium.com/@mnemon1ck/why-you-should-not-use-angularjs-1df5ddf6fc99#.18ple2arn

[22] http://stackoverflow.com/questions/14324451/angular-service-vs-angular-factory

[23] http://aokolish.me/blog/2014/11/16/8-reasons-i-won't-be-choosing-ember.js-for-my-next-app/

[24] http://blog.shinetech.com/2013/09/06/backbone-is-not-enough/

[25] http://todomvc.com/

[26] https://angular.io/

[27] http://webpack.github.io/docs/

[28] http://requirejs.org/

[29] http://browserify.org/

[30] https://nodejs.org/en/blog/announcements/foundation-v4-announce/

[31] https://www.polymer-project.org/1.0/

[32] http://stackoverflow.com/questions/tagged/reactjs

[33] https://www.reddit.com/r/reactjs

[34] https://jsfiddle.net/reactjs/69z2wepo/

[35] https://facebook.github.io/react-native/

[36] https://facebook.github.io/react/blog/2015/02/20/introducing-relay-and-graphql.html

[37] https://github.com/FormidableLabs/radium

[38] https://github.com/js-next/react-style

[39] https://github.com/martinandert/react-inline

[40] http://nicolashery.com/integrating-d3js-visualizations-in-a-react-app/

[41] https://www.youtube.com/watch?v=nYkdrAPrdcw

[42] https://facebook.github.io/flux/

[43] https://github.com/

[44] https://nodejs.org/en/download/

[45] https://www.npmjs.com/package/express

[46] https://www.npmjs.com/package/express-partials

[47] https://www.npmjs.com/package/express-flash

[48] https://www.npmjs.com/package/express-session

[49] https://www.npmjs.com/package/cookie-parser

[50] https://www.npmjs.com/package/serve-favicon

[51] https://www.npmjs.com/package/flux

[52] https://www.npmjs.com/package/react

[53] https://www.npmjs.com/package/react-dom

[54] https://www.npmjs.com/package/browserify

[55] https://www.npmjs.com/package/babelify

[56] https://www.npmjs.com/package/fixed-data-table

[57] https://www.npmjs.com/package/sequelize

[58] https://www.npmjs.com/package/sqlite3

[59] https://www.google.com/fonts