MEDAC
Instituto Oficial de Formación Profesional

DISEÑO DE INTERFACES WEB

TÉCNICO EN DESARROLLO DE APLICACIONES WEB

Biblioteca para la creación de interfaces (II)

13

ÍNDICE

OBJETIVOS

Utilizar y valorar distintas aplicaciones para el diseño de un documento web.

Utilizar marcos, tablas y capas para presentar la información de manera ordenada.

Crear y utilizar plantillas de diseño.

Interpretar guías de estilo.

/ 1. Introducción y contextualización práctica

Como vimos en el tema anterior, **React JS** es una librería de código abierto implementada sobre **JavaScript**, la cual permite el diseño e implementación de interfaces web. Para completar el contenido visto en el capítulo anterior, en este tema, analizaremos el uso de eventos sobre React, elementos clave para el desarrollo de cualquier sitio web y su correspondiente interfaz gráfica.

Otro de los aspectos importantes en el diseño de un sitio web es la incorporación de mecanismos que permiten modelar las rutas para que el usuario pueda navegar por un sitio. Por ejemplo, imaginaremos que hemos desarrollado una tienda virtual con varias secciones (*home*, tienda y galería de imágenes), *React Router* nos permitirá esta implementación basada en rutas para las aplicaciones que se han desarrollado sobre React JS.

El uso de patrones de diseño está cada vez extendido. En el tema anterior, se expuso el modelo MVC, pero, en este caso, también se usará Flux, una nueva arquitectura para el modelo *software* desarrollado por *F.lux Software*, que pretende optimizar ciertos aspectos, como la creación de un flujo unidireccional.

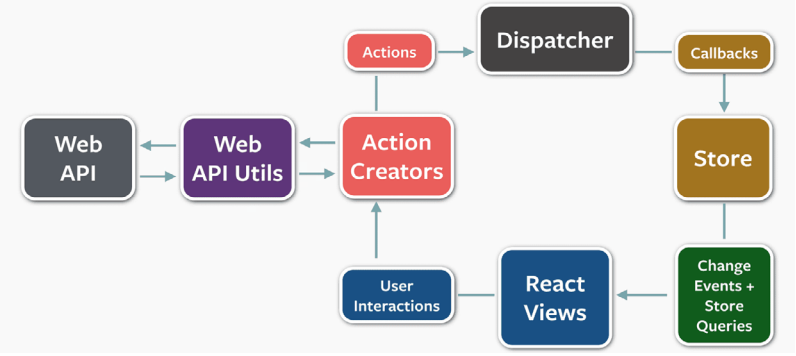

Fig. 1. Flujo de transición en Flux.

Escucha el siguiente audio donde planteamos la contextualización práctica de este tema, encontrarás su resolución en el apartado Resumen y resolución del caso práctico.

Audio intro. "Patrón Flux. Flujo unidireccional"
https://bit.ly/2JtJCbE

/ 2. Eventos en React.js

En el diseño de interfaces para sitios web la implementación de un comportamiento interactivo es esencial. Uno de los elementos que permite dotar de interacción a una interfaz son los eventos.

La **creación de eventos en React** se basa en el uso de dos elementos principales:

- En primer lugar, es necesario indicar el **tipo** de evento que se quiere tratar.
- También se indica el **método** que va a tratar dicho evento o, más bien, el **código** que se va a ejecutar cuando ocurre un determinado evento. **Esta implementación se lleva a cabo dentro de la función render().**

```
render () {
return (
//Código en JSX
<...tipoEvento={this.funcionTratamientoEvento}>
...
</...>
);
}
```

Código 1. Método render().

La implementación del código JSX es similar al que se utilizaba en HTML, salvo que ahora, en lugar de realizar el llamamiento de la función poniendo el nombre de esta seguido de paréntesis, será la función de tratamiento del evento la que se sitúe dentro de los símbolos de llave como se muestra a continuación:

```
<elemento tipoEvento={funcionTratamientoEvento}>

...

</elemento>
```

Código 2. Llamada función desde JSX.

Finalmente, solo quedaría implementar el código del método que se va a ejecutar cuando se captura un evento. En este primer caso, se define una función que no recibe ningún parámetro de entrada.

```
funcionTratamientoEvento () {

//código que se va a ejecutar ante  la ocurrencia de un evento

}
```

Código 3. Función modelo tratamiento de eventos().

Las funciones para el tratamiento de eventos también pueden recibir valores por parámetro, por ejemplo, en el caso anterior, si la función se implementa para que pueda recibir valores por parámetro, quedaría de la siguiente forma: **funcionTratamientoEvento (valor1) {...}.**

2.1. Eventos de ratón y teclado

En las siguientes tablas, se muestran algunos de los eventos más utilizados que se pueden tratar, agrupados en función del dispositivo sobre el que se produce la interacción. En primer lugar, encontramos los **eventos asociados al ratón y al teclado.**

Los eventos de **ratón** se producen debido a la interacción entre el usuario y la interfaz de la aplicación, utilizando como dispositivo el puntero del ratón.

Habitualmente, se detectan eventos tales como la pulsación sobre un botón del ratón cuando este está colocado sobre algún elemento de la interfaz.

Eventos de ratón	
Tipo	**Descripción**
onClick	Se invoca cuando se pulsa el botón izquierdo de un ratón.
onMouseOut	Se invoca cuando se saca el puntero del ratón del elemento en el que se está realizando la escucha.
onMouseOver	Se invoca cuando pasamos con el puntero del ratón sobre el elemento en el que se está realizando la escucha.

Tabla 1. Eventos de ratón.

Los eventos de **teclado** más comunes son los que se producen debido a la acción del usuario sobre alguna de las teclas del dispositivo, siempre que el foco se encuentre sobre el elemento que está realizado la escucha del evento. Es decir, si, por ejemplo, nos colocamos encima de un elemento botón y se pulsa la letra 'a' en el teclado, si el elemento botón no tiene ningún evento de escucha de teclado, este evento no será detectado y no se desencadenará ninguna acción.

Eventos de teclado	
Tipo	**Descripción**
onKeyPress	Se invoca cuando se pulsa cualquier tecla del teclado.

Tabla 2. Eventos de teclado.

En el siguiente código, se produce la detección de la pulsación de un botón. Cuando esto ocurre, se imprime un mensaje por pantalla.

```
<button onClick={funcionTratamientoEvento}>
  Pulsa aquí
</button>
```

Código 4. Ejemplo evento sobre botón con JSX.

2.2. Eventos de foco y formulario

Además de los eventos anteriores, encontramos otros dos grandes bloques: **los eventos de atención relativos al foco y los eventos de formulario.**

Los **eventos de atención en el foco** no se centran tanto en una acción de tipo físico como las de ratón y de teclado, sino que evalúan **si se produce algún cambio en el elemento** (como, por ejemplo, cuando se escoge una opción en un menú desplegable) o **si se ha puesto el foco en un elemento** (por ejemplo, cuando nos colocamos encima de una caja de texto y esta se marca de otro color), independientemente del dispositivo con el que esta acción se realice.

Eventos de atención	
Tipo	**Descripción**
onFocus	Se invoca cuando se pone el foco sobre un elemento.
onBlur	Se invoca cuando se quita el foco de un elemento.
onChange	Se invoca cuando se modifica el contenido del elemento, por ejemplo, en un select.

Tabla 3. Eventos de atención en el foco.

Finalmente, encontramos los **eventos de formulario.** Los formularios permiten recoger diferentes tipos de información a través de cajas de texto, *checkbox* o menús desplegables, entre otros. Cada uno de los elementos que constituyen un formulario podrán implementar diferentes eventos en función del tipo elemento.

Ahora bien, los formularios presentan algunos elementos que son característicos de los mismos. Hablamos del **botón *Submit*** (utilizado para el envío de los datos completados en los campos del formulario) y, aunque menos común, el **botón *Reset*** (utilizado para limpiar el formulario y comenzar a rellenarlo de nuevo). A continuación se muestran los eventos asociados a cada uno de estos botones.

Eventos de formulario	
Tipo	**Descripción**
onSubmit	Se invoca cuando se pulsa el botón de envío de formulario Submit.
onReset	Se invoca cuando se pulsa el botón de envío de formulario Reset.

Tabla 4. Eventos de formulario.

/ 3. *Routing*

Routing es el proceso de determinar el mejor camino para llegar a un determinado destino. Es decir, permite el encaminamiento desde un origen hasta el destino estableciendo, para ello, las rutas o caminos que se van a seguir.

Cuando se desea implementar un nuevo sitio web que requiere de la inserción de rutas para navegar entre las diferentes páginas y secciones del sitio, será necesario definir cada ruta, una a una.

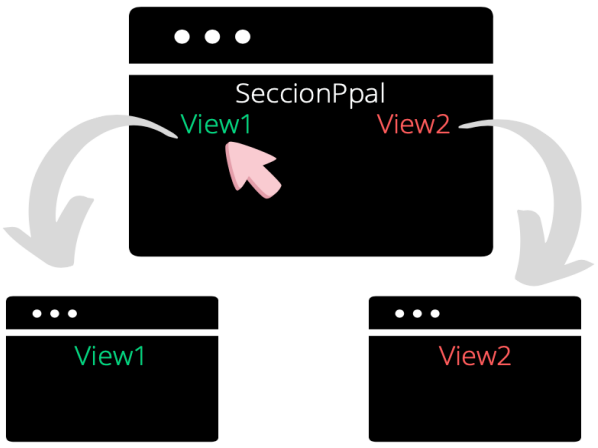

Fig. 2. Diagrama de rutas en diseño de interfaces.

Una implementación típica sería la siguiente, en la que el componente **SeccionPpal** muestra los enlaces a los componentes que quedarán enlazados a través de las rutas recogidas en **View1 y View2.**

```
import React from 'react';
import {BrowserRouter as Router, Route, Switch} from "react-router-dom";
import SeccionPpal from "./components/SeccionPpal";
import View1 from "./components/views/View1";
import View2 from "./components/views/View2";
export default function App (){
return(
<Router>
<SeccionPpal />
<Switch>
<Route path="view-1" component={View1}/>
<Route path="view-2" component={View2}/>
</Switch>
</Router>
);
}
```

Código 5. Implementación React con routing básico.

3.1. Optimización de rutas y React Router

a) Optimización

A medida que el número de rutas crece, también lo hará el código contenido colocado dentro de las etiquetas ***switch*** que aparecen en el código anterior (código 5).

Para evitar esta sobrecarga en el código, se va a crear un nuevo objeto para las rutas. Cada uno de estos nuevos objetos presentará dos propiedades: ***component y path.***

- ***component:*** indica el nombre del componente que va a ser asociado a una ruta.
- ***path:*** indica la cadena de texto que representa la ruta en la que el nuevo componente está disponible.

```
const NOMBREVISTA = {
component : nombreComponente,
path: "ruta componente"
};
```

Código 6. Sintaxis creación de objeto ruta.

Para el caso anterior, el código 5 quedaría optimizado de la siguiente forma para el caso de la ruta 1:

```
const View1 = {
component : views.View1,
path: "/view-1"
};
```

Código 7. Mejora código 5 Vista 1.

b) Instalación y configuración de *React Router*

React Router es una colección de componentes de navegación que permite la creación de rutas en una aplicación. En ella, se distingue una librería para el desarrollo web (*react-router-dom*) y otra para desarrollo en dispositivos móviles (*react-router-native*).

Fig. 3. Logotipo React Router.

La instalación de la librería *React Router* se realiza por línea de comandos:

1. Instalación de la versión de *React Router.* Tras la instalación, puede ser importado en cualquier módulo.

```
npm install --save react-router
```

Código 8. Instalación React Router.

2. Para importar la librería desde cualquier módulo, se importa de la siguiente forma:

```
import { Router, Route, browserHistory } from 'react-router';
```

Código 9. Importación librería en módulo de desarrollo.

Audio 1. "React Router. Enrutamiento estático y dinámico"
https://bit.ly/2KWQjnh

/ 4. Patrones de *software*. Flux

Flux es una arquitectura que permite definir el flujo de datos en una aplicación web. Se utiliza para escribir aplicaciones en el cliente, es decir, en el *front-end*.

En el capítulo anterior, vimos el patrón MVC, Modelo-Vista-Controlador, pero Facebook desarrolló su propio patrón de *software*, Flux. El nuevo patrón utilizaba una lógica de datos en un solo sentido. Esta arquitectura se utiliza de forma conjunta con la librería de desarrollo React.js.

Fig. 4. Logotipo Flux.

El comportamiento de Flux como patrón de arquitectura de *software* permite diferenciar entre las siguientes "capas":

- ***Views:*** estos elementos son los que permiten la transformación de los componentes React en elementos "tradicionales" del DOM virtual. Estos elementos representan las interfaces web sobre las que los usuarios realizarán las acciones oportunas, lo que supondrá la activación de una determinada acción.

- ***Stores:*** estos elementos se encargan de la conexión de las vistas (*views*) y los elementos de acción (los métodos *actions*). Se trata de los almacenes de datos de los clientes y quedan unidos a la capa de acción a través de los disparadores.

- ***Actions:*** estos elementos son los encargados de modelar las funciones y operaciones que se realizan en una aplicación. Las acciones se producen ante la detección de un evento sobre alguno de los componentes de las *View*.

- ***Dispatcher:*** estos se encuentran entre los componentes *actions* y los *stores*. Reciben las peticiones de las acciones y estas son encaminadas hacia el store sobre el que van a actuar.

En el siguiente vídeo mosrtamos la ampliación de contenido en sitios *web flux y React Router*.

Vídeo 1. "Ampliación de contenido en sitios web Flux y React Router"
https://bit.ly/3lpqeJW

/ 5. Caso práctico 1: "Diseño web *routing*"

Planteamiento: Como se ha visto anteriormente, el diseño de rutas a través de *React Router* permite optimizar el diseño de código, puesto que cuando el número de enlaces es muy elevado no será óptimo colocar un *Router* de forma manual para cada uno de los enlaces. En este ejercicio se pide adaptar el código 5 del apartado de Routing, utilizando la creación de objetos ruta descritos en el apartado de Optimización de rutas.

Nudo: Para la implementación del nuevo código, utiliza la siguiente sintaxis base de creación de objetos ruta.

```
const NOMBREVISTA = {
component : nombreComponente,
path: "ruta componente"
};
```

Código 10. Sintaxis creación de objeto ruta.

Desenlace: A continuación, se añade el código resultante. De nuevo, se utilizará un elemento *switch*, puesto que se necesita recorrer todo el archivo de rutas, y así va creando un *Route* para cada uno de los objetos que encuentra en el fichero de rutas hacia el que apunta.

```
import React from 'react';
import {BrowserRouter as Router, Route, Switch} from "react-router-dom";
import SeccionPpal from "./components/SeccionPpal";
import routes from "./config/routing/routes";
export default function App (){
return(
<Router>
<SeccionPpal />
<Switch>
{routes.map(route => (
<Route
key={route.path}
path={route.path}
component={route.component}
/>
))}
</Switch>
</Router>
);
}
```

Código 11. Código completo acceso a objetos ruta.

/ 6. Capas de Flux: Vistas y acciones

A continuación, analizaremos todos los elementos que intervienen en la arquitectura Flux, construyendo paso a paso el flujo completo.

De manera general, el funcionamiento de esta arquitectura se basa en que los usuarios realizarán determinadas acciones sobre una vista, las cuales serán detectadas como eventos sobre los distintos componentes de la vista. Cuando ocurre una acción, esta es enviada al disparador (*dispatcher*), que envía la información al *store o stores* implicados. Finalmente, los *stores* transmiten los cambios realizados a las vistas para que muestren por pantalla la respuesta recibida.

a) Vistas

Las vistas o *Views,* aunque no son directamente una capa de la arquitectura Flux, sí que son una parte esencial en el diseño de la aplicación, puesto que será sobre la que se realiza la interacción necesaria entre el usuario y la aplicación, **poniendo de nuevo de manifiesto la importancia del diseño de interfaces web.**

Fig. 5. Arquitectura Flux (I). Vistas.

b) Acciones

A continuación, encontramos lo que se podría considerar la primera capa, la capa de acciones, *Actions*. Es en esta capa en la que se definen las funciones del sistema.

Dentro de la arquitectura Flux, las acciones suponen el comienzo del flujo. Los usuarios de una aplicación iniciarán la interacción produciendo algún evento sobre un objeto de las vistas. Esto supondrá la puesta en ejecución de la acción asociada.

Por lo tanto, podemos decir que la capa de acciones va a trabajar directamente con la capa de las vistas con la que interaccionará el usuario.

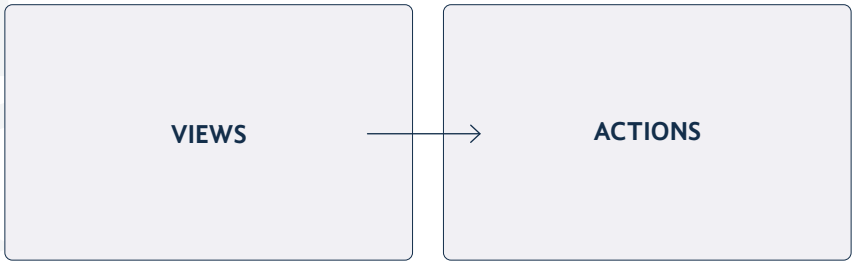

Fig. 6. Arquitectura Flux (II). Vistas + Actions.

Vídeo 2. "Análisis de flujo en Flux"
https://bit.ly/3obZEpz

/ 7. Capas de Flux. Disparadores y stores

a) Disparadores

A continuación, encontramos los disparadores o *dispatcher*. Esta capa se encarga de la distribución de las acciones hacia el *store* correcto. De esta forma se agiliza la ejecución, puesto que se asume que la distribución optimiza el proceso.

Fig. 7. Arquitectura Flux (III). Vistas + Actions + Dispatcher.

En cuanto a la implementación del código, serán necesarias las siguientes instrucciones:

```
var Dispatcher = require('flux').Dispatcher;
var AppDispatcher = new Dispatcher();
module.exports = AppDispatcher;
```

Código 12. Instrucciones disparadores.

Ahora bien, en muchas ocasiones, el cambio producido por una acción puede estar relacionado con varios *store* y la llamada a cada uno de estos desde los *dispatcher* debe hacerse en un orden concreto, ya que será habitual que para realizar un cambio B, antes se haya que realizar una actualización A. Para implementar esa espera, la clase *Dispatcher* proporciona el método *waitFor()*.

b) Store

Estos elementos son los encargados de almacenar los nuevos datos y estados de una aplicación. Las acciones que se llevan a cabo son dirigidas hacia el store sobre el que van a actuar, gracias a los disparadores.

Finalmente, los *stores* quedarán unidos también a la capa *Views*, puesto que será aquí donde se mostrarán los datos solicitados o las modificaciones realizadas, entre otras acciones posibles.

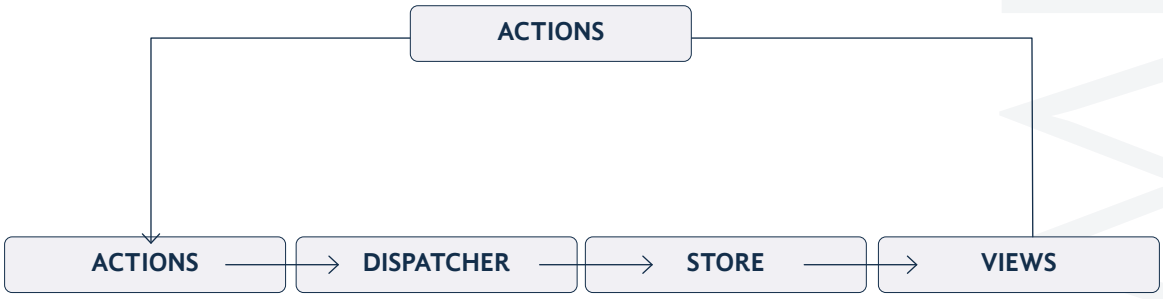

Fig. 8. Arquitectura Flux (IV).

De nuevo, como se puede ver en la figura anterior, se realizará una nueva detección de eventos sobre las vistas que conducirá a la ejecución de una nueva acción que hará comenzar el flujo.

/ 8. Caso práctico 2: "Implementando un proyecto con arquitectura Flux"

Planteamiento: Se pide adaptar e implementar el código necesario para cada una de las capas ***actions y dispatcher*** que componen la **arquitectura Flux.**

Nudo: En primer lugar, será necesario crear dos nuevas carpetas (***actions y stores***) en el directorio **principal src.** Seguidamente, se crean los ficheros que se describen a continuación y se colocan en las carpetas señaladas.

Desenlace: Los ficheros resultantes quedan de la forma:

- **dispatcher.js se coloca en src.**

```
import { Dispatcher } from "flux";
const dispatcher = new Dispatcher();
export default dispatcher;
```

Código 13. Fichero dispatcher.js del caso práctico 2.

- **actionTypes.js (se definen las constantes) se coloca en src/actions.**

```
export default {
  GET_POSTS: "GET_POSTS",
};
```

Código 14. Fichero actionTypes.js del caso práctico 2.

- **postActions.js se coloca en src/actions.**

```
import dispatcher from "../appDispatcher";
import actionTypes from "./actionTypes";
import data from "../db.json";
export function getPosts() {
  dispatcher.dispatch({
    actionTypes: actionTypes.GET_POSTS,
    posts: data["posts"],
  });
}
```

Código 15. Fichero postActions.js del caso práctico 2.

En este caso, se muestra una breve exposición del código de desarrollo necesario para implementar una aplicación con arquitectura Flux. El código de desarrollo completo esta tomado de la fuente que se cita a continuación.

Fuente: https://github.com/Sharvin26/DummyBlog/tree/master/src

/ 9. Resumen y resolución del caso práctico de la unidad

En esta unidad, hemos podido comprobar que en el diseño de interfaces para sitios web, la implementación de un comportamiento interactivo es esencial. Uno de los elementos que permite dotar de interacción a una interfaz son los eventos. La **creación de eventos en React** se basa en el uso de dos elementos principales: el tipo de evento que se quiere tratar y el método que va a tratar dicho evento.

Existen diferentes tipos de eventos: **eventos asociados al ratón, eventos de teclado, los eventos de atención relativos al foco y los eventos de formulario.**

Otro de los puntos claves estudiados en este capítulo es el ***routing***, proceso que permite determinar las rutas a seguir para implementar la navegación del usuario por las diferentes páginas de un mismo sitio web. Para evitar esta sobrecarga en el código a la hora de incluir elementos de *router*, se va a crear un nuevo objeto para las rutas. Cada uno de estos nuevos objetos presentará dos propiedades: ***component y path.***

Finalmente, se ha analizado el patrón Flux, una arquitectura que permite definir el flujo de datos en una aplicación web. El comportamiento de Flux como patrón de arquitectura de *software* permite diferenciar entre las siguientes "capas": ***views, stores, actions y dispatcher.***

Resolución del caso práctico inicial

La arquitectura Flux, desarrollada por Facebook, para evitar la comunicación bidireccional entre los controladores y los modelos, implementa el flujo unidireccional que se describe a continuación.

1. Desde la capa de vista ***(views)***, mediante la interacción sobre un evento, se envía una acción que provocará un cambio en el estado de la aplicación.

2. La acción ***(actions)*** queda conectada al disparador ***(dispatcher)*** que determinará hacia dónde se encamina la acción, la cual contiene el tipo y los datos.

3. El **disparador** envía la acción al *store* oportuno.

4. El **store** recibe la acción y en función del tipo del que se trate, se actualiza el estado.

5. La interfaz, es decir, las vistas ***(views)*** reciben el cambio que se haya producido en el estado y modifican lo que se muestra al usuario, si así procede.

La acción volvería a comenzar desde el principio, pero siempre en un único sentido.

/ 10. Bibliografía

García-Miguel, D. (2019). *Diseño de Interfaces Web* (1.a ed.). Madrid, España: Síntesis.

/ 11. Webgrafía

React.js. Recuperado de: https://es.reactjs.org
React Router. Recuperado de: https://reactrouter.com
Flux. Recuperado de: https://facebook.github.io/flux/