MEDAC
Instituto Oficial de Formación Profesional

DISEÑO DE INTERFACES WEB
TÉCNICO EN DESARROLLO DE APLICACIONES WEB

Librerías para la creación de interfaces (II)

15

ÍNDICE

OBJETIVOS

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

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

Crear y utilizar plantillas de diseño.

Interpretar guías de estilo.

Utilizar la librería Mobx, comprender y analizar sus bloques principales.

/ 1. Introducción y contextualización práctica

Como ya se vio en el tema anterior, existen librerías en JavaScript que permiten agilizar y simplificar el diseño e implementación de aplicaciones e interfaces web. En este último capítulo, analizaremos en detalle el funcionamiento de Mobx.

Esta librería permite acceder a los datos de una aplicación, es decir, permite alterar el estado de la aplicación, puesto que este viene definido por el valor de sus datos. Si bien puede utilizarse con cualquier tipo de proyecto desarrollado con JavaScript, la combinación más actual aparece con React.js, como se verá más adelante.

De manera práctica, estudiaremos los conceptos claves para el funcionamiento de Mobx: hablamos de los estados, las acciones y las derivaciones.

El uso de librerías en el diseño de interfaces y aplicaciones web es una realidad en constante evolución, por lo que, en la actualidad, un desarrollador siempre estará aprendiendo nuevas formas de actualizar sus herramientas de desarrollo.

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

Fig. 1. Logotip Mobx.

Audio intro. "Redux y Mobx. Similitudes y diferencias"

https://bit.ly/3mxeR44

/ 2. Introducción Mobx

Mobx es una librería escrita en JavaScript que nos permite acceder y modificar los estados de la aplicación de una forma sencilla y escalable. Es posible utilizarlo en cualquier proyecto implementado en JavaScript, aunque lo más frecuente, en la actualidad, es utilizarlo en la combinación Mobx y React.

Una de las principales diferencias que Mobx muestra con Redux es que el código resulta más sencillo y fácil de entender. La página oficial de esta librería está accesible desde el enlace.

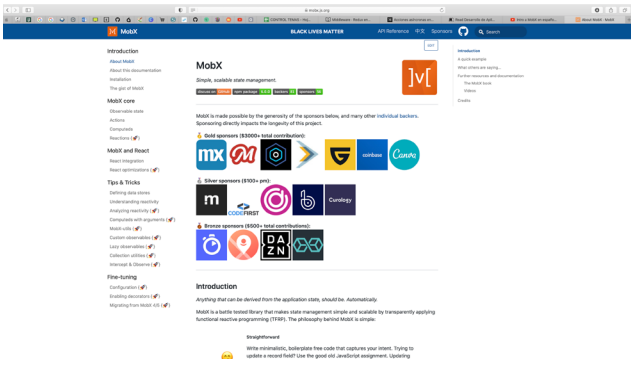

Fig. 2. Sitio oficial Mobx.

En Mobx, podemos distinguir **tres conceptos clave** para su desarrollo: el estado (*state*), las acciones (*actions*) y las derivaciones (*derivations*).

- ***state***: es el estado de la aplicación, uno de los pilares fundamentales de este modelo. Los estados son los datos de la aplicación, como, por ejemplo, los *arrays*, objetos o primitivas, entre otros.
- ***derivations***: es cualquier valor que se pueda calcular de forma automática partiendo del estado de la aplicación. Será posible obtener tanto valores simples como el resultado de un contador, y hasta representaciones HTML.
- ***actions***: las acciones son aquellos fragmentos de código que permiten modificar el estado de la aplicación.

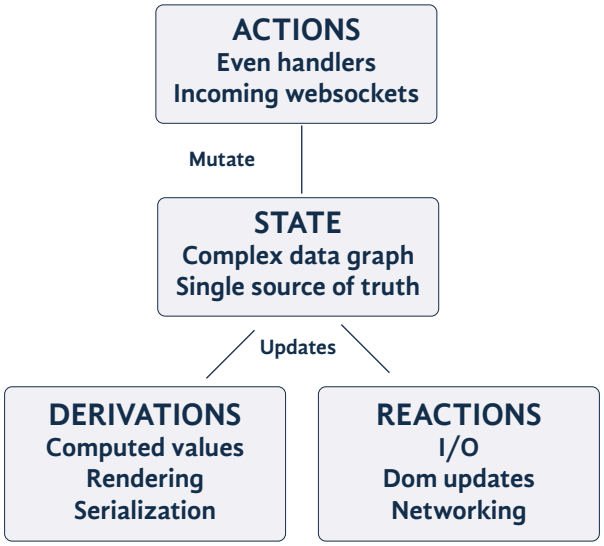

Fig. 3. Diagrama de conceptos clave. Fuente: https://mobx.js.org/getting-started

/ 3. Instalación de Mobx

Para implementar un nuevo proyecto utilizando **Mobx, al igual que el capítulo anterior, en primer lugar, es necesaria la creación de un nuevo proyecto**. En Mobx, se utilizaría la siguiente instrucción para crear un nuevo proyecto llamado app-**mobx**:

```
npx create-react-app app-mobx
```

Código 1. Creación de proyecto Mobx.

```
Success! Created app-mobx at /Users/diana/Desktop/pruebas/app-mobx
Inside that directory, you can run several commands:

  npm start
    Starts the development server.

  npm run build
    Bundles the app into static files for production.

  npm test
    Starts the test runner.

  npm run eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd app-mobx
  npm start

Happy hacking!
iMac-de-Diana:pruebas diana$
```

Fig. 4. Salida del terminal de instalación Mobx.

A continuación, se realiza la **instalación de los módulos de Mobx: mobx y mobx-react**.

```
npm install --save mobx mobx-react
```

Código 2. Instalación de mobx.

Para utilizar la librería Mobx en un proyecto, se utiliza la siguiente instrucción que permite importar el contenido de dicha librería:

```
import { configure } from "mobx"
configure ({useProxies:"never"})
```

Código 3. Sentencias para importar mobx en el nuevo proyecto.

Como recordatorio:

El comando **npx** es un comando de **Node.js, una tecnología en JavaScript que permite la instalación de programas implementados en lenguaje de programación JavaScript.**

La descarga del paquete está disponible desde su sitio web: https://nodejs.org/en/download/.

Para realizar la instalación de Node.js en cualquier sistema operativo, tras la descarga del ejecutable correspondiente, basta con hacer clic sobre el ejecutable y se abrirá un instalador. El proceso de instalación es muy sencillo, simplemente se pulsará *Next* hasta completar la instalación. No necesita ninguna configuración específica.

La estructura de ficheros creada en el directorio de la nueva aplicación debería ser similar a la siguiente:

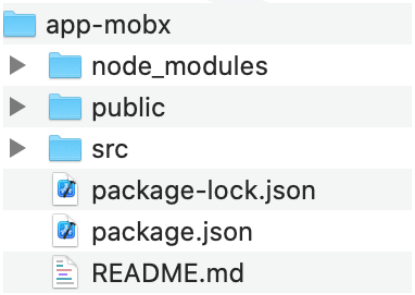

Fig. 5. Estructura de ficheros en app-mobx.

/ 4. Estados

Los **estados son los datos que va a manejar una aplicación**. En **Mobx, deben ser observables**, por lo tanto, será necesario implementar algún tipo de mecanismo que permita hacerlos así. Para ello,se utiliza la palabra **makeObservable**. En un apartado posterior, mostraremos cómo convertir un estado en observable.

Ahora bien, cuando se crean los estados en una determinada aplicación, estos deben tomar un valor de los que se muestran a continuación, y que indican si el estado va a ser observable, calculado (*computed*), o será una acción.

- **observable**: permite definir un campo rastreador que almacena un estado
- ***computed***: indica una acción que modifica el estado y almacena la salida en memoria
- ***action***: indica la acción que va a modificar el estado

Comenzaremos desde este apartado la implementación de una aplicación completa utilizando Mobx, que ilustrará todo el proceso. En primer lugar, se implementa el nombre de la clase en la que se define una lista de tareas y, posteriormente, el estado de cada una de ellas, por ejemplo, tareas completadas o tareas sin completar.

En este caso, los datos de aplicación van a ser la lista de tareas, y por lo tanto, dicha lista va a ser el estado de la aplicación.

```
class ListaTareasStore  {
        listatareas=[];
}
```

Código 4. Desarrollo de class ListaTareasStore (I).

En el siguiente diagrama, se muestra el flujo seguido cuando se utiliza la librería Mobx. En este caso, al producirse una acción que altera y modifica un dato, es decir, el estado de la aplicación.

Tras este cambio, se lanzarán las funciones diseñadas para el cálculo de valores que al mismo tiempo alimentan las denominadas *reactions*. Tanto los valores calculados como las *reactions* pertenecen a las derivaciones propias del modelo Mobx.

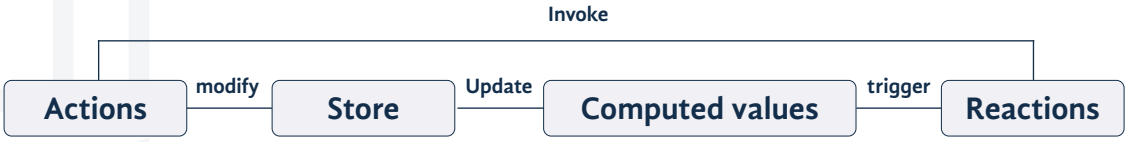

Fig. 6. Diagrama de flujo Mobx.

Vídeo 1. "Análisis del modelo Mobx"
https://bit.ly/2JgDG68

/ 5. Caso práctico 1: "Crea una aplicación que permita añadir asignaturas a la matrícula de un curso"

Planteamiento: Se quiere implementar una nueva aplicación que permita añadir nuevas asignaturas a la matrícula del nuevo curso. Cuando estas asignaturas son aprobadas, se marcan como tal en la aplicación para mostrar el progreso total del curso.

Diseña las siguientes funciones en la clase ListaAsignaturasStore:

- contadorAsignaturasSuperadas
- informeProgreso
- addAsignaturaMatricula

Nudo: Para la creación de cada una de las funciones anteriores, se utilizar la librería Mobx. En primer lugar, será necesario crear el proyecto de la siguiente forma:

```
npx create-react-app app-mobx-asignaturas
```

Código 5. Creación de proyecto Mobx

Desenlace: El código solución se muestra a continuación y en él, se implementan todas las funciones descritas para la clase ListaAsignaturasStore.

```
class ListaAsignaturasStore {
listaasignaturas=[];
//Calcula el número de asignaturas superadas
get contadorAsignaturasSuperadas () {
return this.listaasignaturas.filter( asignatura => asignatura.superada ===true).length
}
/*Indica el número de asignaturas superadas sobre el número total de matriculadas*/
informeProgreso () {
return 'Informe progreso curso:
${this. contadorAsignaturasSuperadas} /
${this.listaasignaturas.length}'
}
/*Añade nuevas asignaturas al listado de asignaturas matriculadass*/
addAsignaturasMatricula(asignatura) {
this.listaasignaturas.push({
asignatura: asignatura,
superada: false
})
}
}
```

Código 6. Solución del caso práctico 1.

/ 6. Derivaciones

Las derivaciones son **cualquier elemento** que puede derivarse de los estados de la aplicación sin ningún tipo de interacción adicional.

Encontramos dos tipos de derivaciones en Mobx:

- **Valores calculados** - ***computed value***: funciones que devuelven un valor que depende del estado. Por ejemplo, si para el caso anterior se quisiera implementar el código necesario para devolver el número de tareas que están completadas, se realizará el siguiente desarrollo:

```
class ListaTareasStore  {
        listatareas=[];
        get contadorTareasCompletadas () {
                return this.listatareas.filter( tarea => tarea.completada ===true).length
        }
}
```

Código 7. Desarrollo de class ListaTareasStore (II). Valores calculados.

En este caso, se devolverán en un *array* todas las tareas que tienen la propiedad completada como verdadera. Como al final se ha colocado la función ***length***, lo que se devuelve es la longitud de dicho *array*, y por tanto, el número de tareas que están completadas.

- ***Reaction***: funciones que no producen un valor. A diferencia del caso anterior, ahora se va a realizar un informe sobre las tareas que han sido completadas sobre el número de tareas totales.

En primer lugar, se crea una nueva función que va a devolver el informe descrito, en este caso, recibe el nombre de informe(), y en su interior, se añade una cadena de texto. **La función informe() utiliza el valor calculado descrito en el punto anterior, así como el valor calculado de la dimensión total de la lista de tareas**.

```
class ListaTareasStore  {
        listatareas=[];
        get contadorTareasCompletadas () {
                return this.listatareas.filter( tarea => tarea.completada ===true).length
        }
        informe () {
                return 'Informe progreso:
                ${this.contadorTareasCompletadas} /
                ${this.listatareas.length}'
        }
}
```

Código 8. Desarrollo de class ListaTareasStore (III). Reaction.

Audio 1. "MobX. Diseño reactivo"
https://bit.ly/3ocKj85

/ 7. Acciones

El tercer concepto clave de Mobx son las acciones (*actions*). Tratan, sobre todo, aquel código que va a modificar el estado de las aplicaciones. Todas estas van a tener acciones, puesto que, de lo contrario, solo tendríamos almacenes de datos sobre los que no se podría realizar ninguna acción.

Una acción es un fragmento de código que permite cambiar el estado de la aplicación. Habitualmente, las acciones tienen como desencadenante la ocurrencia de un **evento**, por ejemplo, cuando se pulsa un botón.

Para el caso del código de ejemplo que se está utilizando para ilustrar este capítulo, se añadiría una nueva acción que nos va a permitir incorporar tareas a nuestra lista de tareas.

```
class ListaTareasStore  {
  listatareas=[];
  get contadorTareasCompletadas () {
    return this.listatareas.filter( tarea => tarea.completada ===true).length
  }
informe () {
  return 'Informe progreso:
  ${this.contadorTareasCompletadas} /
  ${this.listatareas.length}'
}
addListaTareas(tarea) {
  this.listatareas.push({
    tarea: tarea,
      completada: false
    })
  }
}
```

Código 9. Desarrollo de class ListaTareasStore (IV). Acciones.

Para completar la implementación, se crearía una nueva instancia del store que nos permitirá ejecutar las acciones desarrollados para la aplicación.

```
let store = new ListaTareasStore()
console.log(store.informe())
store.addListaTareas('Estudiar')
console.log(console.log(store.informe())
store.listatareas[0]completada=true;
console.log(console.log(store.informe())
```

Código 10. Desarrollo de class ListaTareasStore (V). Instanciación y prueba.

La salida por pantalla del código anterior será:

Informe progreso: 0/0

Informe progreso: 0/1

Informe progreso: 1/1

/ 8. Estado observable

Para completar este diseño, sería deseable que cada vez que se produzca un cambio de estado, la llamada a la función **informe** se hiciera de forma automática. Es decir, se trata de imprimir solo la última versión del informe para que los datos intermedios no se muestren, y, concretamente, este es uno de los objetivos principales de Mobx: realizar la ejecución de código que depende de un estado de forma automática.

Fig. 7. Ilustración "observando estados observables".

Para conseguir esto, el **estado debe ser observable** y Mobx podrá rastrear todos los cambios que se producen sobre él. Sobre el código anteriormente mostrado, se crea un constructor, el cual utiliza **makeObservable**, además, se utilizan los tipos **observable** y ***computed*** para indicar cómo se coloca el foco sobre cada una de las propiedades. En concreto:

- **observable**: indica que el valor puede cambiar con el tiempo.
- ***computed***: indican que estos valores pueden derivarse del estado.

```
constructor() {
  makeObservable(this, {
    listatareas: observable,
    contadorTareasCompletadas: computed,
    informe: computed,
    addTarea: action,
  });
  autorun(() => console.log(this.informe));
 }
```

Código 11. Desarrollo del constructor observable. class ListaTareasStore (VI).

Finalmente, para completar el código del constructor, se ha añadido la función **autorun()**, que imprime el informe resultante. Además, gracias a esta función, se produce una ejecución automática del código cada vez que cambia el valor del estado de cualquiera de los datos marcados como observables en el código.

/ 9. Referencias

Hasta ahora, hemos trabajado con un único estado, es decir, el estado de la aplicación se ha basado en las tareas y en el estado de las mismas. Pero ¿sería posible añadir nuevos *arrays* de datos a estos estados? Lo lógico es pensar que sí, puesto que, de lo contrario, todo desarrollo solo podría manipular unos pocos datos.

Imaginemos que se quiere añadir un nuevo dato (que también marcará el estado de la aplicación) para indicar si una tarea está asignada, recogiendo además el nombre de la persona a la que se ha asignado.

En primer lugar, sería necesario modificar el código de la función **addTarea**, puesto que, ahora, se añade el nuevo dato **asignado**.

```
addTarea(tarea) {
   this.listatareas.push({
     tarea: tarea,
     completada: false,
     asignado: null
   });
 }
```

Código 12. Ampliación de la función addTarea. Desarrollo de class ListaTareasStore (VII).

Para la creación de este nuevo objeto, es importante indicar que va a ser observable, tal y como se indica a continuación:

```
const empleadosStore = observable([
 { nombre: “Manuel” },
 { nombre: “José” },
 { nombre: “Beatriz” }
]);
```

Código 13. Creación de empleadosStore. Desarrollo de class ListaTareasStore (VIII).

Finalmente, se completa el desarrollo con el siguiente fragmento de instanciación. Prueba en la que se indica para cada una de las tareas de la lista de tareas (listatareas[posicion]) cuál de las referencias al *array* de empleados le va a corresponder, utilizando el dato asignado.

```
listaTareasStore.listatareas[0].asignado = empleadosStore[0];
listaTareasStore.listatareas[1].asignado = empleadosStore[1];
empleadosStore[0].nombre = “Manuel García”;
```

Código 14. Inserción de asignados. Desarrollo de class ListaTareasStore (IX).

/ 10. Acciones asíncronas

Como ocurría en el capítulo anterior, es deseable que la implementación de este tipo de aplicaciones sea de forma **asíncrona**, es decir, que las acciones se ejecuten asíncronamente. **En el caso de Mobx, el cambio en la aplicación se produce ante un cambio en el estado**.

Para ilustrar este apartado de forma práctica, tenemos una página web que permite realizar peticiones a nuestra aplicación, estas peticiones añadirán una nueva tarea a la lista de tareas. Por lo tanto, el estado de la aplicación irá cambiando tras actualizar en su almacén de tareas la nueva tarea.

Para implementar estas peticiones de tareas, en primer lugar, se actualiza el número de **solicitudes pendientes** (a continuación, se muestra dónde se debe incluir en el código para convertirla en observable y que, de esta forma, se pueda detectar un cambio en su estado).

Una vez ejecutada la acción, **una nueva tarea será incluida en la lista de tareas** y, finalmente, se **disminuye la lista de tareas pendientes**.

```
listaTareasStore.tareasPendientes++;
setTimeout(action(() => {
  listaTareas.addTarea('Añadir tarea con id: ' + Math.random());
  listaTareas.tareasPendientes--;
}), 2000);
```

Código 15. Peticiones pendientes a aplicación lista de tareas. Desarrollo de class ListaTareasStore (X).

Finalmente, será necesario modificar el contenido del constructor implementado en el código 11, añadiendo un nuevo dato bajo el nombre **tareas pendientes** y de tipo **observable**, lo que permitirá el diseño asíncrono descrito en este apartado.

```
constructor() {
  makeObservable(this, {
    listatareas: observable,
    tareasPendientes: observable,
    contadorTareasCompletadas: computed,
    informe: computed,
    addTarea: action,
  });
  autorun(() => console.log(this.informe));
 }
```

Código 16. Ampliación del constructor observable. Desarrollo de class ListaTareasStore (XI).

Vídeo 2. "Análisis del modelo Mobx. Código"
https://bit.ly/2L0BU9L

/ 11. Caso práctico 2: "Acciones asíncronas y elementos observables"

Planteamiento: Utilizando como base el ejercicio resuelto para el caso práctico 1, se desea que el funcionamiento de esta aplicación cumpla los criterios necesarios para presentar un comportamiento asíncrono.

Por lo tanto, en este ejercicio, se pide modificar el código añadiendo un constructor que permita definir el valor de los estados de forma adecuada.

Nudo: Se añade la función **constructor**, la cual permite determinar qué datos/estados van a ser definidos como observables, y por tanto, se ha de realizar la "escucha" sobre esto para cambiar la interfaz de la aplicación cuando cambie el estado del dato.

Desenlace: El código solución se muestra a continuación y, en él, se añade la nueva función.

```
class ListaAsignaturasStore  {
  listaasignaturas=[];
//Constructor
constructor() {
  makeObservable(this, {
    listaasignaturas: observable,
    asignaturasPendientes: observable,
    contadorAsignaturasSuperadas: computed,
    informeProgreso: computed,
    addAsignatura: action,
  });
  autorun(() => console.log(this.informeProgreso));
 }
  //Calcula el número de asignaturas superadas
  get contadorAsignaturasSuperadas () {
    return this.listaasignaturas.filter( asignatura => asignatura.superada ===true).length
 }
 /*Indica el número de asignaturas superadas sobre el número total de matriculadas*/
 informeProgreso () {
return 'Informe progreso curso:
    ${this. contadorAsignaturasSuperadas} /
    ${this.listaasignaturas.length}'
  }
  /*Añade nuevas asignaturas al listado de asignaturas matriculadass*/
   addAsignaturasMatricula(asignatura) {
    this.listaasignaturas.push({
      asignatura: asignatura,
      superada: false
    })
  }
}
```

Código 17. Solución del caso práctico 2.

/ 12. Resumen y resolución del caso práctico de la unidad

En esta unidad, hemos conocido a **Mobx**, una librería escrita en JavaScript que nos permite acceder y modificar los estados de la aplicación de una forma sencilla y escalable. Es posible utilizarla en cualquier proyecto implementado en JavaScript, aunque lo más frecuente, en la actualidad, es utilizarla en la combinación Mobx y React. **En Mobx, se distinguen tres conceptos clave para su desarrollo: el estado (*state*), las acciones (*actions*) y las derivaciones (*derivations*)**.

Los **estados son los datos que va a manejar una aplicación**, en **Mobx, deben ser observables**, por lo tanto, será necesario implementar algún tipo de mecanismo que permita hacer estos estados observables. Para ellos, se utiliza la palabra **makeObservable**. Ahora bien, cuando se crean los estados en una determinada aplicación, estos deben tomar un valor que indican si el estado va a ser observable, calculado (*computed*) o será una acción.

Hemos aprendido que las **derivaciones** pueden derivarse de los estados de la aplicación sin ningún tipo de interacción adicional. Encontramos dos tipos de derivaciones en Mobx: ***computed value y reaction***.

El tercer concepto clave de Mobx que hemos estudiado son las acciones. **Una acción es un fragmento de código que permite cambiar el estado de la aplicación**. Habitualmente, las acciones tienen como **desencadenante** la ocurrencia de un **evento**, por ejemplo, cuando se pulsa un botón.

Resolución del caso práctico inicial

Aunque los resultados podrían resultar similares y algunas características de diseño son parecidas, existen ciertas diferentes entre Redux y MobX que conviene tener en cuenta a la hora de seleccionar una librería o la otra.

Fig. 8. Mobx vs Redux.

1. Mobx es **más fácil de aprender** y también de **usar**.

2. Mobx **requiere de menos código** para escribir el mismo programa. Mientras que en Redux se deben actualizar al menos cuatro artefactos, en MobX, solo es necesario actualizar dos: la store y el diseño de la interfaz.

3. **Soporte completo para la programación orientada a objetos.**

4. Con Mobx, el **tratamiento con datos anidados es más sencillo** que en Redux.

/ 13. Bibliografía

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

13.1. Webgrafía

Mobx. Recuperado de: https://mobx.js.org/observable-state.html