

Projecte Redux JS

DAW2 M06 UF2

Curs 2021-22

Raúl Heredia i Ana Menéndez



Índex

Què és Redux?	3
Avantatges del store de Redux.....	3
Com funciona Redux?	4
Accions en Redux.....	4
Reducers en Redux	5
Store en Redux.....	5
Exemple bàsic de funcionament.	6
El nostre projecte	7
APIS Utilitzades	7
Web Storage	7
IndexedDB	8
Drag And Drop	8
FileReader.....	8
Lògica de la aplicació	9
Emmagatzematge de dades	10
POO amb JS.....	11
Funcions amb JS	12
Arrays amb JS	14
Mètodes d'array utilitzats.....	14

Què és Redux?

Redux és una llibreria JavaScript de codi obert per al maneig de l'estat de les aplicacions. És comunament utilitzada amb altres llibreries com React o Angular per a la construcció d'interfícies d'usuari.

Al 2015 per Dan Abramov i Andrew Clark es van inspirar en una altra llibreria de Facebook, Flux per crear Redux. Abramov es trobava impressionat per la similitud del patró Flux amb la funció de reduir.

"Estava pensant sobre Flux com una operació reduir... el teu magatzem, com acumulen un estat en resposta a unes accions. Vaig pensar a anar més enllà. Què si el teu magatzem Flux no fos un magatzem sinó una funció reduir?"

Abramov va contactar amb Clark per col·laborar junts. Gràcies a Clark hi ha les eines que fan possible l'ecosistema Redux, va ajudar a definir un API coherent i implementar la possibilitat d'extensió mitjançant middleware i store enhancers.

Redux és una petita llibreria amb una API simple i limitada que està dissenyada per ser un contenidor predictable de l'estat de l'aplicació.

El propòsit de Redux és fer predictibles els canvis d'estat, imposant certes restriccions sobre com i quan es poden produir les actualitzacions. Redux aconsegueix que la teva gestió d'estat sigui transparent i determinista, cosa que entre altres coses aporta:

- Millor comprensió de l'evolució de l'estat en un moment donat.
- Facilitat per incorporar noves característiques a l'app.
- Nou ventall d'eines de debugging.
- Capacitat de reproduir un bug.
- Millores en el procés de desenvolupament i poden reiniciar l'execució a partir d'un estat concret.

Avantatges del store de Redux

- Model de dades més consistent i segur
- Gestió senzilla de múltiples fonts de dades:
- Dades accessibles en temps real:

Com funciona Redux?

La manera com funciona Redux és senzilla. Hi ha una “Store” central que conté tot l'estat de l'aplicació. Cada component pot accedir a l'estat emmagatzemat sense haver d'enviar accessoris d'un component a un altre, això resulta molt útil en aplicacions de Angular o React que tenen diferents components.

Principalment, Redux es divideix en tres parts: **accions**, **Store** i **reducers**. Ara, veurem que fan cadascuna.

Accions en Redux

En poques paraules, les accions són esdeveniments. Són l'única manera d'enviar dades de la teva sol·licitud a la nostra Store de Redux. Les dades poden provenir d'interaccions d'usuari, trucades d'API o fins i tot d'enviaments de formularis.

Les accions s'envien mitjançant el mètode `store.dispatch()`. Les accions són objectes JavaScript simples, i han de tenir una propietat tipus per indicar el tipus d'acció a dur a terme. També han de tenir una càrrega útil que contingui la informació que ha de ser treballada per l'acció. Les accions es creen a través d'un creador d'accions.

Per exemple:



```
1 addUserBtn.addEventListener('click', () => {  
2   store.dispatch(  
3     { type: "ADD_USER", username: userInput.value }  
4   );  
5 })
```

En aquest cas, quan cliquem el botó afegir usuari, enviem una acció de tipus “ADD_USER” amb la propietat “username” i el valor agafat d’un input.

Reducers en Redux

Els reducers són funcions pures que prenen l'estat actual d'una aplicació, realitzen una acció i retornen un nou estat. Aquests estats s'emmagatzemen com a objectes i especifiquen com canvia l'estat d'una aplicació en resposta a una acció enviada a la store.

Es basa en la funció de reducció en JavaScript, on un sol valor es calcula a partir de múltiples valors després de realitzar una funció de resposta.

Per exemple:

```
1 const reducer = (state = [], action) => {  
2   console.log("reducer", state, action);  
3   if (action.type === "ADD_USER") {  
4     return [...state, action.username];  
5   }  
6   return state
```

En aquest cas, veiem que el reducer està compost per l'estat i la acció, cada vegada que fem un dispatch, la acció arriba al reducer. Com podem veure, si la petició es de tipus "ADD_USER", retornarà l'estat més el nom d'usuari que li acabem de passar, en cas de que la acció no sigui de tipus "ADD_USER", simplement retornarà l'estat sense fer canvis

Store en Redux

La Store, té l'estat de l'aplicació. És molt recomanable mantenir només una Store en qualsevol aplicació Redux. Amb la Store, podem accedir a l'estat emmagatzemat, actualitzar l'estat i registrar o deixar de registrar listeners.

Els mètodes més utilitzats son Store.subscribe() i Store.getState(). Subscribe es un mètode que es truca cada vegada que canvia l'estat de la nostra aplicació, getState serveix per obtenir l'estat de la nostra aplicació.

Per exemple:

```
1 store.subscribe(() => {  
2   list.innerHTML = '';  
3   userInput.value = '';  
4   store.getState().forEach(user => {  
5     const li = document.createElement('li');  
6     li.className = 'list-group-item'  
7     li.textContent = user;  
8     list.appendChild(li);  
9   });
```

Com podem veure, fem un subscribe amb una funció anònima, això es el que farà cada vegada que s'actualitzi l'estat. En aquest cas el que fem es eliminar el contingut d'una llista HTML i el value del input d'usuari, seguidament fem un getState i iterem mitjançant un forEach, per cada usuari que hi hagi a l'estat crearem un element 'li', el qual tindrà una classe de bootstrap i el text que

contindrà serà l'usuari, finalment fem un `appendChild` del element 'li' per afegir-lo a la llista.

Exemple bàsic de funcionament.

```
1  const reducer = (state = [], action) => {
2    console.log("reducer", state, action);
3    if (action.type === "ADD_USER") {
4      return [...state, action.username];
5    }
6    return state
7  };
8  const store = Redux.createStore(reducer);
9  const list = document.getElementById('list');
10 const addUserBtn = document.getElementById('addUser');
11 const userInput = document.getElementById('userInput');
12
13 store.subscribe(() => {
14   list.innerHTML = '';
15   userInput.value = '';
16   store.getState().forEach(user => {
17     const li = document.createElement('li');
18     li.className = 'list-group-item'
19     li.textContent = user;
20     list.appendChild(li);
21   });
22 });
23
24
25 addUserBtn.addEventListener('click', () => {
26   store.dispatch(
27     { type: "ADD_USER", username: userInput.value }
28   );
29 })
```

En aquest cas, podem veure un codi sencer, amb el reducer, la Store i la acció que em vist anteriorment.

Cada vegada que introduïm un usuari i fem clic al botó afegir, s'envia una acció al reducer mitjançant un dispatch i canvia l'estat, llavors amb el subscribe cada vegada que aquest canvia, actualitza la llista:

Raúl

Ana

Inspector

Consola

Depurador

Red

>>

Filtrar salida

Errores

Advertencias

Registros

Información

Depurar

CSS

XHR

Peticiones

reducer ▶ Array [] ▶ Object { type: "@@redux/INIT3.0.x.c.g.e" } index.js:6:17

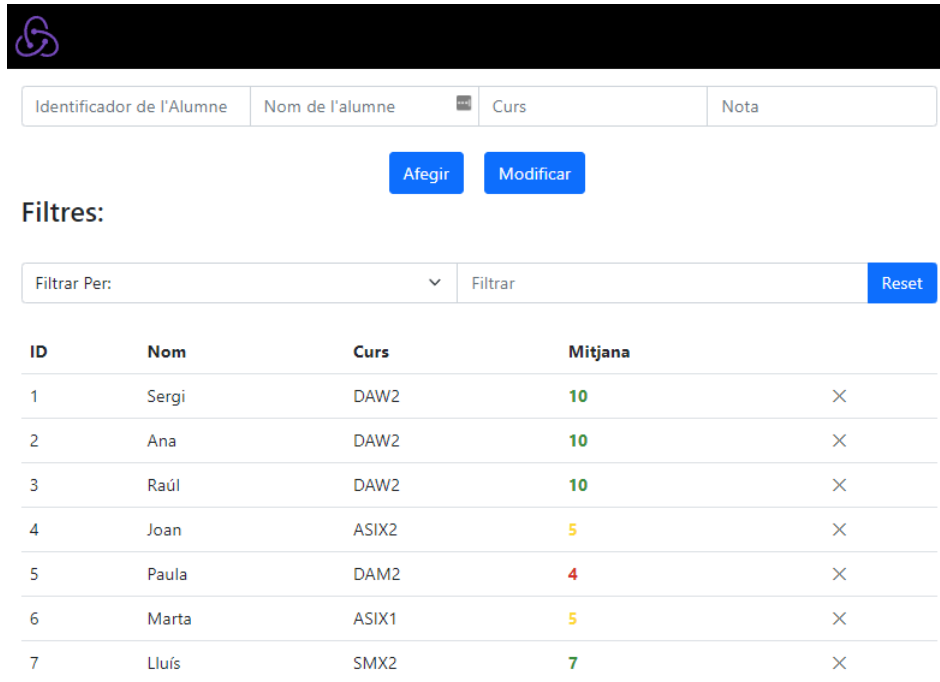
reducer ▶ Array [] ▶ Object { type: "ADD_USER", username: "Raúl" } index.js:6:17

reducer ▶ Array ["Raúl"] ▶ Object { type: "ADD_USER", username: "Ana" } index.js:6:17

>>

El nostre projecte

El projecte es una aplicació web que permet afegir alumnes, amb un identificador, nom curs i nota. En cas de voler modificar-ne algun, simplement posem el identificador, la dada que volem modificar i fem clic al botó de modificar. Podem filtrar els alumnes per aprovats i suspesos, a més de poder buscar-los per nom o curs, escrivint al input anomenat filtrar.



The screenshot shows a web application interface. At the top, there is a dark header with a logo. Below it, there is a form with four input fields: 'Identificador de l'Alumne', 'Nom de l'alumne', 'Curs', and 'Nota'. There are two buttons: 'Afegir' (Add) and 'Modificar' (Modify). Below the form, there is a 'Filtres:' section with a dropdown menu 'Filtrar Per:' and a 'Filtrar' button. To the right of the dropdown is a 'Reset' button. Below the filter section is a table with the following data:

ID	Nom	Curs	Mitjana	
1	Sergi	DAW2	10	×
2	Ana	DAW2	10	×
3	Raúl	DAW2	10	×
4	Joan	ASIX2	5	×
5	Paula	DAM2	4	×
6	Marta	ASIX1	5	×
7	Lluís	SMX2	7	×

APIS Utilitzades

Web Storage

Hem utilitzat Web Storage per emmagatzemar únicament el ID de l'alumne i el seu nom. Quan carrega la pàgina obté les dades de IndexedDB, cada vegada que afegim un usuari, es guarda el ID i el nom, quan eliminem un usuari, s'elimina també, es guarda mitjançant Local storage i no Session Storage. A més la utilitzem per a mostrar les dades per consola mitjançant una funció dinàmica que es veurà més endavant.

Key	Value
5	Paula
2	Ana
6	Marta
4	Joan
7	Lluís
1	Sergi
3	Raúl

IndexedDB

Hem utilitzat IndexedDB per a guardar els objectes dels alumnes. La key que utilitza es el id d'usuari. Al entrar a la aplicació, IndexedDB envia cada usuari al reducer de la api Redux i d'aquesta forma els carreguem en el estat, que es mostrat en la taula d'alumnes

#	Key (Key path: "id")	Value
0	1	{id: 1, nom: 'Sergi', curs: 'DAW2', nota: 10}
1	2	{id: 2, nom: 'Ana', curs: 'DAW2', nota: 10}
2	3	{id: 3, nom: 'Raül', curs: 'DAW2', nota: 10}
3	4	{id: 4, nom: 'Joan', curs: 'ASIX2', nota: 5}
4	5	{id: 5, nom: 'Paula', curs: 'DAM2', nota: 8}
5	6	{id: 6, nom: 'Marta', curs: 'ASIX1', nota: 5}
6	7	{id: 7, nom: 'Lluís', curs: 'DAW1', nota: 7}

Drag And Drop

Hem utilitzat la API de drag and drop per a poder arrossegar arxius JSON, amb els quals podem afegir alumnes de forma massiva.

FileReader

Hem utilitzat la API FileReader per a processar les dades del fitxer.

```
1 function dropJSON(targetEL, callback) {
2     // disable default drag & drop functionality
3     targetEL.addEventListener('dragenter', function (e) { e.preventDefault(); });
4     targetEL.addEventListener('dragover', function (e) { e.preventDefault(); });
5
6     targetEL.addEventListener('drop', function (event) {
7
8         var reader = new FileReader();
9         reader.onloadend = function () {
10             var data = JSON.parse(this.result);
11             callback(data);
12         };
13
14         reader.readAsText(event.dataTransfer.files[0]);
15         event.preventDefault();
16     });
17 }
18
19 dropJSON(
20     document.getElementById("dropTarget"),
21     function (data) {
22         for (let i in data) {
23             afegirUsuari(data[i].id, data[i].nom, data[i].curs, data[i].nota)
24             //console.log("for", data[i].nom);
25         }
26     }
27 )
```

Aquí podem veure la funció DropJSON, a la qual passem el element en el qual arrossegarem i deixarem l'arxiu i retorna un callback, a l'hora de trucar, fem una funció anònima en la qual iterem cada usuari de l'arxiu i cridem la funció afegirUsuari passant cada propietat de l'usuari.

Lògica de la aplicació

La lògica en un projecte Redux es senzilla, enviem accions al reductor, el qual canvia l'estat i quan aquest canvia, la Store realitza algunes accions, a mode d'exemple veurem el procés de afegir un alumne:

```
1 addUserBtn.addEventListener('click', () => {
2   if (idInput.value !== "" || userInput.value !== "" || cursInput.value !== "" || notaInput.value !== "") {
3     // Si els camps no estan buits truquem a la funció afegirUsuari
4     afegirUsuari(idInput.value, userInput.value, cursInput.value, notaInput.value)
5   } else { // Si falta algun camp fem un alert
6     alert("Error, falten camps per omplir.");
7   }
8 }
```

En el moment de fer clic al botó d'afegir, comprovem que tots els camps han estat omplerts, en el cas de que faltés algun camp, apareixeria una alerta, en cas de que tots els camps hagin estat omplerts, cridem a la funció afegirUsuari passant els paràmetres necessaris.

```
1 function afegirUsuari(id, nom, curs, nota) {
2   let objectPos = store.getState().map((x) => { return x.id; }).indexOf(parseInt(id));
3   // Busquem si l'usuari ja existeix
4   if (objectPos === -1) { // Si no existeix l'afegim
5     Emmagatzematge.desar(parseInt(id), nom, curs, parseInt(nota));
6     WebStorageEmmagatzematge.desar(parseInt(id), nom);
7     store.dispatch({
8       type: "ADD_USER", id: parseInt(id), nom: nom, curs: curs, nota: parseInt(nota)
9     });
10  } else { // Si existeix no l'afegim i enviem una alerta
11    alert('Error, ja existeix un alumne amb el identificador ${id}.');
12    netejaCamps()
13  }
14 }
```

Ara, creem la variable objectPos i apliquem el mètode map() a l'estat, per a que ens retorni el índex de l'usuari que acabem d'introduir, en cas de que l'índex no sigui -1, alertem a l'usuari de que l'alumne ja existeix i netejem els camps, si l'índex es -1, vol dir que l'alumne no existeix i guardarem l'usuari a IndexedDB i WebStorage, finalment farem un dispatch amb la acció "ADD_USER" per a que passi al reducer de Redux, el qual té una condició if per a les accions de tipus ADD_USER, la qual es igual que la que hem vist a l'exemple d'ús, però en lloc d'afegir un nom a l'array, afegeix un objecte. Finalment la Store detecta el canvi d'estat i genera la taula un altre cop.

Emmagatzematge de dades

S'ha utilitzat IndexedDB per a guardar els alumnes, les dades de IndexedDB son llegides i enviades al reducer de Redux a l'iniciar l'aplicació, amb una ordre "ADD_USER", concretament quan s'estableix la connexió amb indexedDB.

```
1  peticioObertura.onsuccess = function (event) {
2    db = event.target.result;
3    let magatzemObjetsAlumnes = db.transaction("alumnes", "readwrite").objectStore("alumnes");
4    magatzemObjetsAlumnes.openCursor().onsuccess = (event) => {
5      let cursor = event.target.result;
6      if (cursor) {
7        //console.log(cursor.key + " es " + cursor.value.nom);
8        store.dispatch({ // Afegim els usuaris de indexeddb enviant un dispatch de ADD_USER
9          type: "ADD_USER", id: parseInt(cursor.key), nom: cursor.value.nom, curs: cursor.value.curs, nota: parseInt(cursor.value.nota)
10        });
11        WebStorageEmmagatzematge.desar(parseInt(cursor.key), cursor.value.nom) // Guardem el key i el nom a local storage
12        cursor.continue();
13      }
14    };
15  }
```

S'ha fet així, per a assegurar-nos de que iniciava indexedDB abans de que s'executés l'ordre, ja que si no donava un error ja que deia que la transacció tenia un valor undefined. Com podem veure, quan inicia guarda la Key i el nom al webstorage també.

Quan afegim un alumne, aquest es afegit, quan el modifiquem es modificat i quan l'eliminem, es eliminat de indexedDB, tot això es controla segons les accions enviades al Reducer. En el cas de webstorage, quan afegim un usuari s'afegeix, en cas de modificar un alumne només es modifica si modifiquem el nom i en cas d'eliminar un alumne s'elimina.

Les dades de WebStorage son llegides via consola a l'arrancar la aplicació mitjançant una funció dinàmica:

```
Dades Local Storage, ID: 5, nom: Paula
Dades Local Storage, ID: 2, nom: Ana
Dades Local Storage, ID: 6, nom: Marta
Dades Local Storage, ID: 4, nom: Joan
Dades Local Storage, ID: 7, nom: Lluís
Dades Local Storage, ID: 1, nom: Sergi
Dades Local Storage, ID: 3, nom: Raúl
```

POO amb JS

Hem creat una classe emmagatzematge que conté tres mètodes estàtics. El mètode `desar`, el mètode `esborrarAlumne` i el mètode `modificarAlumne`.

```
1 class Emmagatzematge {
2   static desar(id, nom, curs, nota) {
3     let magatzemObjsAlumnes = db.transaction("alumnes", "readwrite").objectStore("alumnes");
4     let alumne = {
5       'id': parseInt(id), 'nom': nom, 'curs': curs, 'nota': parseInt(nota)
6     };
7     magatzemObjsAlumnes.add(alumne);
8   }
9   static esborrarAlumne(id) {
10    let magatzemObjsAlumnes = db.transaction("alumnes", "readwrite").objectStore("alumnes");
11    magatzemObjsAlumnes.delete(parseInt(id));
12  }
13  static modificarAlumne(id, nom, curs, nota) {
14    let magatzemObjsAlumnes = db.transaction("alumnes", "readwrite").objectStore("alumnes");
15    let alumne = {
16      'id': parseInt(id), 'nom': nom, 'curs': curs, 'nota': parseInt(nota)
17    };
18    magatzemObjsAlumnes.put(alumne);
19  }
20 }
21
```

També, hem creat una classe anomenada `WebStorageEmmagatzematge` que hereta de `Emmagatzematge`, aquesta compta amb un `override` dels mètodes `desar` i `esborrarAlumne`, ja que per a modificar simplement utilitzem el mètode `desar` i sobreescriu el nom.

```
1 class WebStorageEmmagatzematge extends Emmagatzematge {
2   static desar(id, nom) {
3     localStorage.setItem(parseInt(id), nom);
4   }
5   static esborrarAlumne(id) {
6     localStorage.removeItem(parseInt(id));
7   }
8 }

```

Funcions amb JS

Hem utilitzat funcions, arrow functions i una funció dinàmica. Les funcions que necessitàvem executar a diferents parts del codi son funcions normals, la majoria de funcions anònimes s'han utilitzat arrow functions i la funció dinàmica s'ha utilitzat a l'hora de llegir les dades del Web Storage.

Exemple funcions:

```
1 function netejaCamps() { // Netejem els camps de input
2   idInput.value = '';
3   userInput.value = '';
4   cursInput.value = '';
5   notaInput.value = '';
6 }
7 function esborraTaula() { // Esborrem Les row de la taula a partir de la número 1, per a no esborrar el header
8   while (taula.rows.length > 1) {
9     taula.deleteRow(1);
10  }
11 }
```

A mode d'exemple, tenim aquestes dues petites funcions, la funció netejaCamps, principalment es executada després de afegir o modificar un alumne, ja que ens interessa que els camps de introduir dades tornin a estar buits.

La funció esborraTaula, s'encarrega d'esborrar la taula abans de generar una nova quan hi ha un canvi d'estat.

Exemple Arrow Function:

```
1 store.subscribe(() => {
2   generaTaula();
3 });
4
```

Un exemple podria ser el Store.subscribe, que te una arrow function, cada vegada que l'estat canvia, crida la funció generaTaula, tot i que podríem trucar directament a generaTaula, no ho hem fet ja que podríem fer moltes mes coses a part de generar la taula.

Un altre Exemple d'Arrow Function:

```
1   resetFilter.addEventListener('click', () => {
2       selectMitjana.getElementsByTagName('option')[0].selected = 'selected';
3       // resetejem la opció seleccionada de selectMitjana
4       inputFiltrar.value = ""; // Resetejem el value de inputFiltrar
5       store.dispatch({
6           // Enviam un canvi d'estat al reducer de tipus RESET_FILTER per a que recarregui l'estat i ens retorni
7           // la taula original sense filtres
8           type: "RESET_FILTER"
9       });
10  });
```

En aquest cas tenim un listener que l'únic que fa és que quan cliquem el botó de resetejar filtres, reseteja el value del input de filtrar i selecciona la opció 0 del select list, finalment envia una acció de tipus "RESET_FILTER".

Exemple funció dinàmica:

```
1   let valor = 1, b;
2   if (valor == 1) {
3       b = 'return x + y + z ';
4   } else {
5       b = 'return x*y*z';
6   }
7   let funcDinamica = new Function('x', 'y', 'z', b)
8   for (let key = 0; key < localStorage.length; key++) {
9       console.log("Dades Local Storage, ID:", funcDinamica(localStorage.key(key), ", nom: ",
10       localStorage.getItem(localStorage.key(key))))
11   }
```

La funció en si, depenent de la variable valor, farà una cosa o una altra, en aquest cas el que fa és sumar retornar la suma de x, y i z, si el valor fos diferent a 1 faria la multiplicació, finalment creem la funció anomenada funcDinàmica amb un new Function, on definim els paràmetres i el que retorna. Tenint en compte que s'executa a l'inici quan ja s'han carregat tots els valors a web storage, fem un for i mostrem per consola "Dades Local Storage ID: ", ara truquem a la funció dinàmica i passem:

Com a valor X: localStorage.key(key).

Com a valor Y: ", nom: ".

Com a valor Z: localStorage.getItem(localStorage.key(key))

Finalment, obtenim per consola:

```
Dades Local Storage, ID: 5, nom: Paula
Dades Local Storage, ID: 2, nom: Ana
Dades Local Storage, ID: 6, nom: Marta
Dades Local Storage, ID: 4, nom: Joan
Dades Local Storage, ID: 7, nom: Lluís
Dades Local Storage, ID: 1, nom: Sergi
Dades Local Storage, ID: 3, nom: Raúl
```

Arrays amb JS

Nomes hem utilitzat un array d'objectes que es el array de l'estat del Reducer. A part d'algun array temporal utilitzat per a filtrar l'estat, etc.

Mètodes d'array utilitzats

map()

Hem utilitzat el mètode map(), per a comprovar si els usuaris existien o no abans d'afegir-los, modificar-los o eliminar-los. Exemple:

```
1      if (action.type === 'DELETE_USER') { // Si la acció es DELETE_USER
2          let objectPos = state.map((x) => { return x.id; }).indexOf(action.id); // Busquem l'usuari a l'array d'estat
3          if (objectPos !== -1) { // si existeix
4              state.splice(objectPos, 1); // fem un splice per a esborrar-lo i que no deixi una posició buida a l'array
5              Emmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
6              WebStorageEmmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
7          } else { // Si no existeix donem un error
8              alert('Error, no existeix cap alumne amb el identificador ${action.id}.');
9          }
10     }
```

Com podem observar a la imatge, en la condició "DELETE_USER" del reducer, tenim la variable objectPos, la qual utilitzem per a comprovar si existeix l'usuari abans d'eliminar-lo, en cas de que la posició sigui diferent a -1, l'esborrem amb un splice i l'esborrem tant de indexeddb com de webstorage.

filter()

El mètode filter, ha estat usat per a poder filtrar les dades de la taula, s'ha hagut de fer fora del reducer, ja que si es filtrava dintre, per molt que es fes creant una nova variable, el filter modificava l'estat original i no es podien recuperar els usuaris que s'havien filtrat (és a dir, si demanava els usuaris suspesos, no podia recuperar els aprovats després). Exemple:

```
1      inputFiltrar.addEventListener('keyup', () => { // cada vegada que aixequem una tecla
2          let filterText = store.getState().filter((a) => a.nom.toLowerCase().startsWith(inputFiltrar.value.toLowerCase()) || a.curs.toLowerCase().startsWith(inputFiltrar.value.toLowerCase()))
3          // Genereu un array en el qual el nom o el curs comenci per el value de inputFiltrar
4          generaTaulaFiltre(filterText); // generem la taula i le passem el array filtrat
5      })
6      selectMitjana.addEventListener('change', (event) => {
7          mitjana = event.target.value;
8          let filtre;
9          switch (mitjana) { // fem un array segons si està aprovat o no
10             case "aprovats":
11                 filtre = store.getState().filter((a) => a.nota >= 5)
12                 break;
13             case "suspesos":
14                 filtre = store.getState().filter((a) => a.nota < 5)
15                 break;
16             }
17             generaTaulaFiltre(filtre); // generem la taula passant el array filtrat
18         });
```

Per a filtrar amb el input text, tenim un listener keyup, cada vegada que introduïm una lletra, creem una variable anomenada filterText, aquesta obté el estat, i el

filtrem on el nom comença per el valor que conté inputFiltrar o el curs comença pel valor que te inputFiltrar, un cop filtrat truquem a la funció generaTaulaFiltre i li passem el array ja filtrat.

Em el cas de voler filtrar per la mitjana, com es un select list tenim un listener change, cada cop canviï el valor, si el valor es “aprovats” filtrarà per els alumnes aprovats, nota ≥ 5 , en cas de ser “suspesos” filtrarà per els alumnes supesos, nota < 5 . Un cop hagi fet el filtre truca a la funció generaTaulaFiltre passant el array filtrat.

sort()

El mètode sort s'utilitza al llarg del reductor per a ordenar el estat per ID, tal i com podem veure a la última línia del reductor:

```
1 return state.sort((a, b) => { return a.id - b.id });
```

splice()

el mètode splice, s'utilitza per a esborrar els usuaris:

```
1 if (action.type === "DELETE_USER") { // Si la acció es DELETE_USER
2     let objectPos = state.map((x) => { return x.id; }).indexOf(action.id); // Busquem l'usuari a l'array d'estat
3     if (objectPos !== -1) { // si existeix
4         state.splice(objectPos, 1); // fem un splice per a esborrar-lo i que no deixi una posició buida a l'array
5         Emmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
6         WebStorageEmmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
7     } else { // Si no existeix donem un error
8         alert('Error, no existeix cap alumne amb el identificador ${action.id}.');
9     }
10 }
```

Com podem observar, si compleix amb la condició de que existeix (`objectPos !== -1`), fem un splice del estat passant-li el seu index, al posar un 1, només s'elimina l'usuari seleccionat.

forEach()

El mètode `forEach` s'ha utilitzat per a iterar tant l'estat com els arrays filtrats de l'estat:

```
1 function generaTaula() {
2   netejaCamps() // Netejem els camps
3   esborraTaula() // Esborrem la taula
4   store.getState().forEach(user => { // Obtenim el estat amb store.getState i iterem per cada objecte que conté
5     let fila = taula.insertRow(-1); // amb insertRow(-1) afegim cada línia al final
6     fila.insertCell(0).innerHTML = user.id; // A la cel·la 0 afegim el id
7     fila.insertCell(1).innerHTML = user.nom; // A la cel·la 1 afegim el nom
8     fila.insertCell(2).innerHTML = user.curs; // A la cel·la 2 afegim el curs
9     let nota = fila.insertCell(3); // Fem una variable per a la cel·la 3
10    if (user.nota < 5) {
11      nota.className = 'suspes'; // Si la nota es menor a 5 afegim la classe suspes a la cel·la
12    } else if (user.nota == 5) {
13      nota.className = 'suficient'; // Si la nota es igual a 5 afegim la classe suficient a la cel·la
14    } else if (user.nota > 5) {
15      nota.className = 'aprovat'; // Si la nota es major a 5 afegim la classe aprovat a la cel·la
16    }
17    nota.innerHTML = user.nota; // Afegim la nota a la cel·la
18    let esborrar = fila.insertCell(4); // Creem una variable per a la cel·la d'esborrar
19    esborrar.innerHTML = '<i class="bi bi-x-lg"></i>'; // Afegim el html per a la icona d'esborrar
20    esborrar.className = "esborrar"; // Afegim la classe esborrar a la cel·la
21    esborrar.setAttribute('id', user.id); // Afegim l'id de usuari a la cel·la
22  })
}
```

Podem veure que fem un `Store.getState().forEach`, d'aquesta forma iterem cada alumne i els afegim a la taula.

reduce()

Com ja em esmentat prèviament, tot Redux es basa en la funció `reduce` de JS, a la qual li arriben accions i retorna un estat segons les condicions d'aquesta acció. El nostre Reducer de Redux es el següent:

```
1  const reducer = (state = {}, action) => {
2    console.log("reducer", state, action);
3    // Segons el tipus d'acció el reducer farà una cosa o una altra
4    if (action.type === "ADD_USER") { // Si l'acció es un ADD_USER afegirem l'usuari nou a l'estat actual, ordenant tot el array d'estat per ID ascendent
5      return [...state, { id: action.id, nom: action.nom, curs: action.curs, nota: action.nota }].sort((a, b) => { return a.id - b.id });
6      // Retornem l'estat amb el nou usuari
7    }
8    if (action.type === "MODIFY_USER") { // Si el tipus es MODIFY_USER
9      let objectPos = state.map((a) => { return a.id; }).indexOf(action.id); // Compruem si existeix l'usuari
10     if (objectPos !== -1) { // Si existeix l'usuari
11       if (action.nom !== "") { // Si el camp nom no està buit el canviem, si està buit no canviem.
12         state[objectPos].nom = action.nom; // canviem el nom que està guardat en la posició ObjectPos de l'estat
13       }
14       if (action.curs !== "") { // Si el camp curs no està buit el canviem, si està buit no canviem.
15         state[objectPos].curs = action.curs; // canviem el curs que està guardat en la posició ObjectPos de l'estat
16       }
17       if (!isNaN(action.nota)) { // Si el camp nota es un número el canviem, si no es un número no canviem.
18         state[objectPos].nota = action.nota; // canviem la nota que està guardada en la posició ObjectPos de l'estat
19       }
20       Emmagatzematge.modificarAlumne(parseInt(state[objectPos].id), state[objectPos].nom, state[objectPos].curs, state[objectPos].nota);
21       // Per modificar qualsevol dels camps després de modificar-los per a poder actualitzar tots els registres independentment de si s'han modificat o no.
22       WebStorageEmmagatzematge.donar(parseFloat(state[objectPos].id), state[objectPos].nom);
23       // En web storage no fa falta un mètode especial per modificar, simplement el tornem a afegir,
24       } else { // Si l'objecte no existeix donem un alert
25         alert("Error, no existeix cap alumne amb el identificador " + action.id);
26       }
27     }
28   }
29   if (action.type === "DELETE_USER") { // Si la acció es DELETE_USER
30     let objectPos = state.map((a) => { return a.id; }).indexOf(action.id); // Busquem l'usuari a l'array d'estat
31     if (objectPos !== -1) { // Si existeix
32       state.splice(objectPos, 1); // Fem un splice per a esborrar-lo i que no deixi una posició buida a l'array
33       Emmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
34       WebStorageEmmagatzematge.esborrarAlumne(parseInt(action.id)); // Esborrem l'usuari per ID
35     } else { // Si no existeix donem un error
36       alert("Error, no existeix cap alumne amb el identificador " + action.id);
37     }
38   }
39   if (action.type === "RESET_FILTER") { // Si arriba un RESET_FILTER
40     return state.sort((a, b) => { return a.id - b.id });
41     // retornem el array normal, no fa falta tenir-lo ja que en cas de que no existís quan arribés el RESET_FILTER seria retornat al return de la línia 37
42   }
43   return state.sort((a, b) => { return a.id - b.id }); // Si no compleix cap tipus d'acció retorna l'estat ordenat per id
44 }
```

Com podem veure segons les accions que arriben, realitza unes accions o unes altres.

for in

La estructura `for in` ha estat usada per iterar els usuaris que rebem del fitxer JSON quan aquest es arrossegat.

```
1  document.getElementById("dropTarget"),
2  function (data) {
3    for (let i in data) {
4      afegirUsuari(data[i].id, data[i].nom, data[i].curs, data[i].nota)
5      //console.Log("for", data[i].nom);
6    }
7  }
8  );
```