Coding Bootcamp

Sprint 4

Temario

Temario

- Redux.
- React Redux.

Problema – Flujo de datos en React

Problema – Flujo de datos en React (1/2)

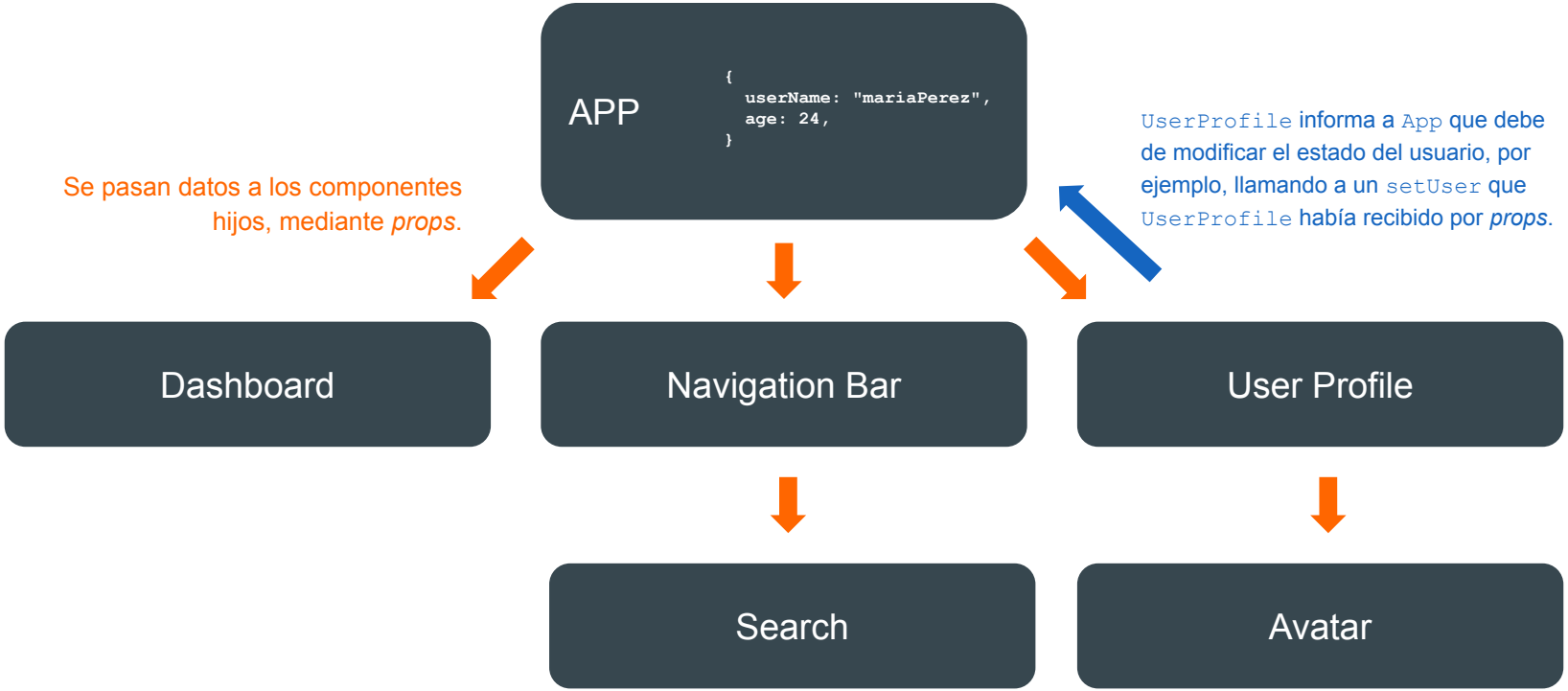
Los componentes de React tienen dos formas de manejar los "datos" que se muestran al usuario: las *props* (que reciben del componente padre) o el *state* propio de cada componente (privado).

A priori, la única forma de pasar datos entre componentes es mediante `props`, desde un componente "padre" a un componente "hijo". Esto es particularmente complicado cuando se quieren pasar datos entre componentes "hermanos".

El problema es aún peor cuando la jerarquía crece y se necesita pasar datos entre componentes que están "lejos" entre sí. Se genera un "pasamanos" de `props`, es decir, hay componentes que reciben `props` sólo para luego pasarlos a sus hijos. Este problema se conoce como "prop drilling".

Ver el siguiente diagrama.

Problema – Flujo de datos en React (2/2)



Redux

Redux (1/6)

Redux es una pequeña librería (2kB) para gestionar de estado en una aplicación JavaScript. Si bien suele asociarse con React es importante destacar que es *framework agnostic*. Nace como una implementación de Flux.

El estado de una aplicación es toda la información almacenada en la misma, en un instante dado. En aplicaciones web modernas, el estado es cada vez más dinámico y difícil de gestionar.

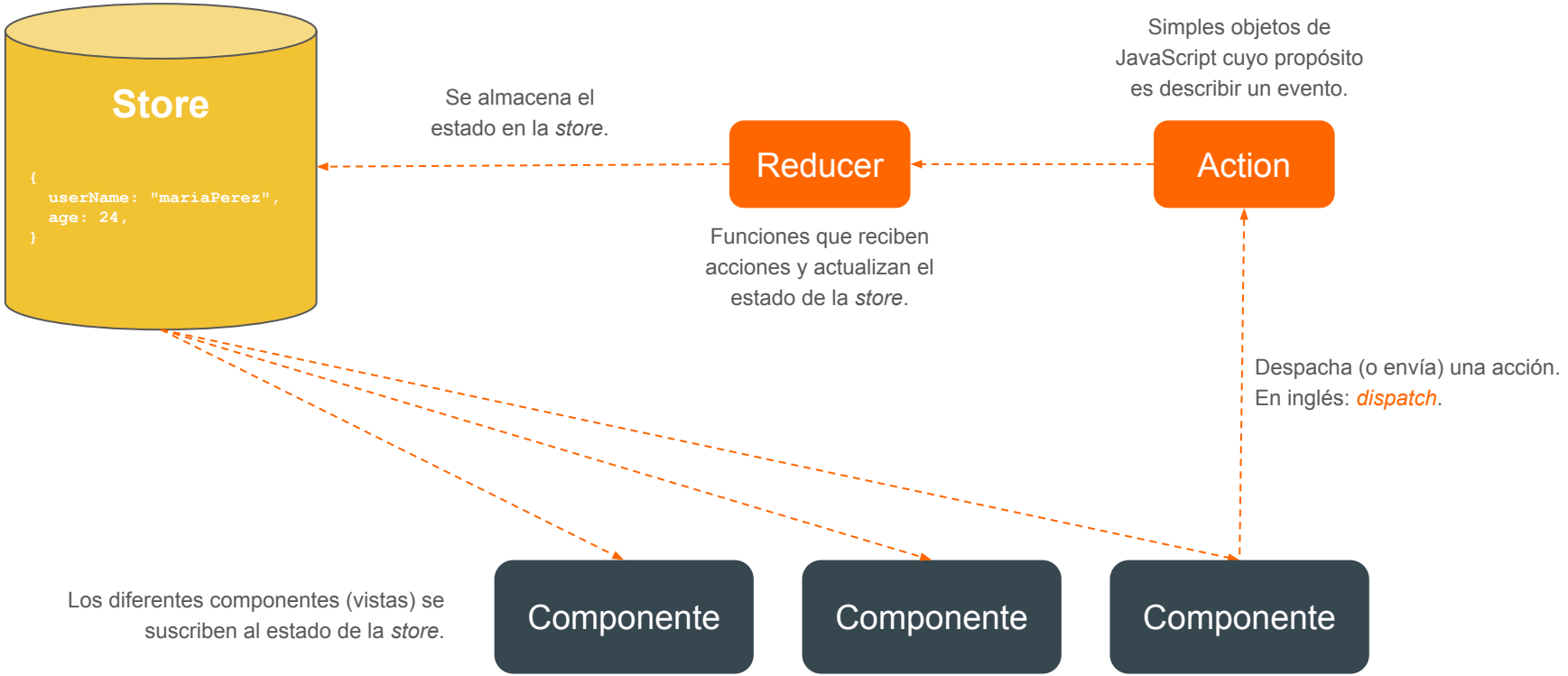
Redux proporciona ciertas pautas o guías a la hora de gestionar ese estado. Por ejemplo, propone guardar todo el estado de la aplicación de forma global, en un objeto JavaScript llamado *store*.

Redux (2/6)

Redux define 3 conceptos importantes:

- Store: Objeto JavaScript donde se guarda el estado (global) de la aplicación. Luego, los componentes se conectan a la *store*, con el fin de consumir y/o editar dicho estado. La *store* sólo existe en tiempo de ejecución. Al recargar la página, se destruye y vuelve a crear. Es *read-only.* La *store* sería el “modelo” en MVC.
- Actions: Objetos JavaScript que describen eventos que pueden “suceder” en una aplicación como “agregar un usuario”, “incrementar un contador” o “mostrar un modal”. Una acción suele provocar cambios en el estado.
- Reducers: Funciones JavaScript que modifican el estado en la *store*.
  Dado que la *store* es *read-only*, sólo se debe modificar vía *reducers*.
  Los *reducers* están a la “escucha” de acciones que ocurren en la aplicación.

Redux (3/6) – Flujo del estado



Redux (4/6) – Actions

Las acciones (***actions***) son objetos JavaScript que describen eventos que pueden “suceder” en una aplicación como “agregar un usuario”, “incrementar un contador” o “mostrar un modal”.

Estos objetos suelen tener los siguientes atributos:

- `type` (obligatorio) [String]. Es para identificar la acción.
- `payload` (opcional) [String, Number, Object, Array, etc.]. Es para contener información adicional sobre la acción.

```
const removeTask = {
  type: "REMOVE_TASK",
  payload: "Aprender sobre jQuery",
};
```

```
const addUser = {
  type: "ADD_USER",
  payload: { username: "mariaPerez", age: 24 },
};
```

Redux (5/6) – Action Creators

OPCIONAL

Normalmente, los mismos tipos de acciones se utilizan en varias partes de una aplicación. Por eso, suele ser útil contar con Action Creators, funciones que retornan (construyen) una acción determinada a partir de los datos que se pasan en el *payload*.

```
const actionCreator = (value) => {
  return {
    type: "Type of the action",
    payload: { data: value },
  };
};
const action = actionCreator("information");
```

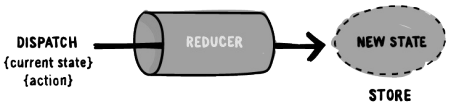
Redux (6/6) – Reducers

Los ***reducers*** son funciones “puras” que modifican el estado en la *store*.

Todo *reducer* tiene las siguientes características:

1. Recibe dos parámetros:
   El 1^{ero} es el estado previo de la *store*.
   El 2^{ndo} es una acción (*action*).

2. Retorna el nuevo estado. Esto debe ser un nuevo objeto. No se puede mutar el estado previo. De lo contrario el *reducer* no sería una función “pura”.



En archivo: `taskReducer.js`

```
function taskReducer(state = [], action) {
  switch (action.type) {
    case "ADD_TASK":
      // Agregar al array de tareas la tarea del action.payload.
      // Retornar el nuevo estado.
    case "REMOVE_TASK":
      // Remover del array de tareas la tarea del action.payload.
      // Retornar el nuevo estado.
    default:
      return state;
  }
}
```

Este ejemplo presupone que dentro de la *store* hay una array de tareas (*tasks*).

React Redux

React Redux (1/4)

[React Redux](#) es una librería que proporciona *bindings* para facilitar el uso de Redux con React.

Antes que nada es necesario instalar Redux:

```
npm i redux
```

Luego se instala React Redux:

```
npm i react-redux
```

React Redux (2/4)

La función `createStore` permite crear la *store*, que es donde se almacenará el estado global de la aplicación.

Esta función recibe dos parámetros: el 1^{ero} es el *reducer* y el 2^{ndo} es el estado inicial de la aplicación.

En el ejemplo de la derecha, la *store* es inicialmente un array de tareas vacío.

El componente `Provider` sirve para inyectar la *store* en todo el "árbol" de componentes de la app, y por eso se coloca en lo más alto de la jerarquía.

En archivo: `index.js`

```
import { createStore } from "redux";
import { Provider } from "react-redux";
import taskReducer from "./taskReducer";

const store = createStore(taskReducer, []);

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

React Redux (3/4) – Acceder a la *store*

Para acceder a la *store* dentro de un componente, es necesario utilizar un *hook* llamado `useSelector`, provisto por React Redux.

```
import { useSelector } from "react-redux";

// ...

const taskList = useSelector(state => state)
```

- `useSelector` recibe una función (*callback*) que se invoca con el estado actual de la *store*.
- Permite usar una parte o todo el estado en cualquier componente que implemente dicha función.
- En este caso retornamos todo la *store*, que es la lista de tareas.

React Redux (4/4) – Despachar una acción

Para despachar una acción dentro de un componente, es necesario utilizar un *hook* llamado `useDispatch`, provisto por React Redux.

```
import { useSelector, useDispatch } from "react-redux";
// ...
const dispatch = useDispatch();
// ...
dispatch({
  type: "ADD_TASK",
  payload: "Estudiar Redux",
});
```

Redux – Combinar *reducers*

Redux – Combinar reducers

Si la aplicación crece, es posible que queramos crear *reducers* separados para manejar distintas partes del estado.

Para esto, Redux cuenta con una función llamada `combineReducers` que recibe como parámetro varios *reducers* y los combina en un sólo.

```
import { combineReducers } from "redux";
import users from "./userReducer";
import tasks from "./taskReducer";
import tweets from "./tweetReducer";

const rootReducer = combineReducers({
  users,
  tasks,
  tweets,
});
```

Luego, para crear la *store* se hace:

```
const store = createStore(rootReducer, {});
```

Redux – Beneficios

Redux – Beneficios

- Comportamiento más predecible: hay una única forma de alterar el estado.
- Reproducir (o deshacer) cambios de estado.
- “Rehidratar” estados desde una representación serializada.
- *Tooling* avanzado:
  - Extensión para Chrome y Firefox: Redux Dev Tools.
  - Para usarlo, hay que agregar un 3er parámetro en el `createStore`:

```
const store = createStore(
  taskReducer, [],
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__()
);
```