Bootcamp de Desarrollo Web

Sprint 4

Temario

Temario

- Problema – Flujo de datos en React
- Redux.
  - *Store.*
  - *Actions.*
  - *Reducers.*
- Redux Toolkit.

Problema – Flujo de datos en React

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

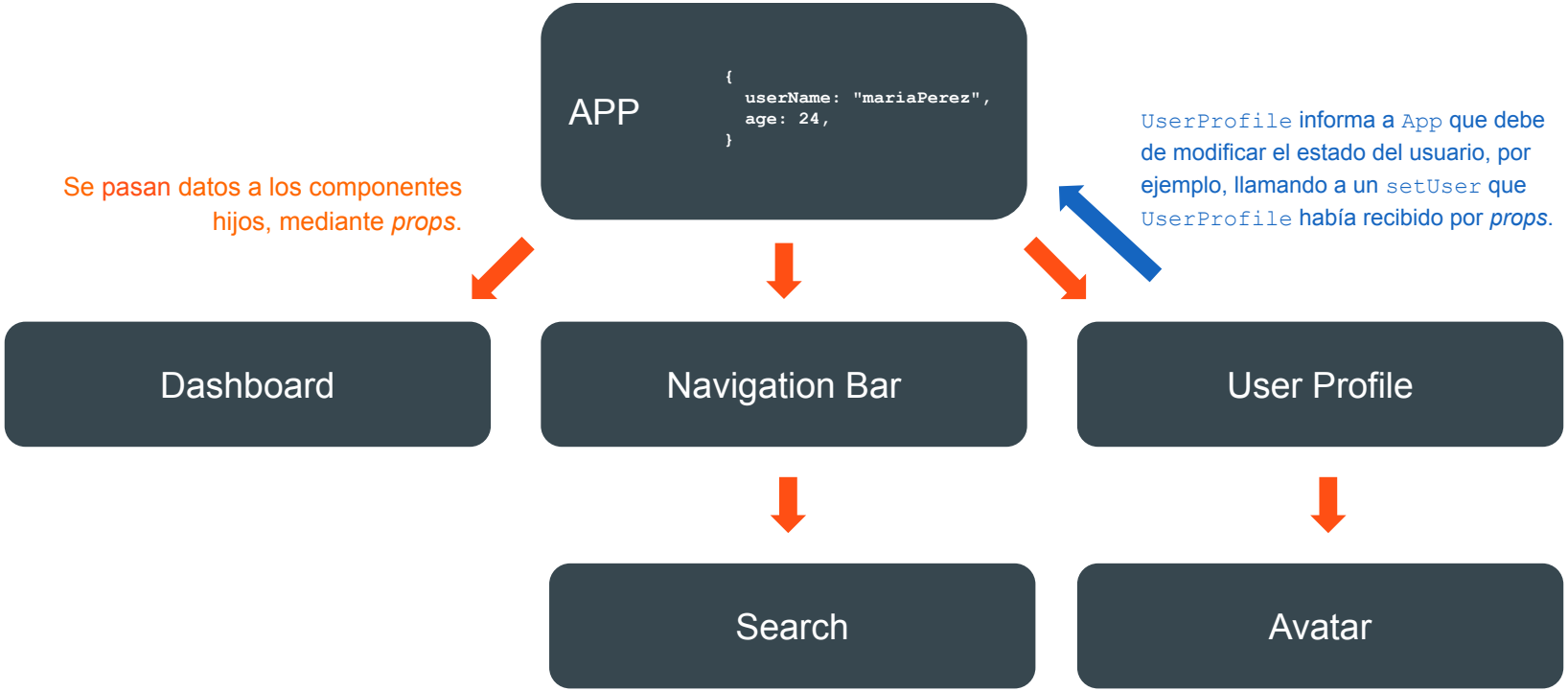
Los componentes de React tienen dos formas de manejar los "datos" que se muestran al usuario: las *props* (que reciben del componente padre) o el *state* propio de cada componente (privado).

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 (1/7)

Redux es una pequeña librería (2kB) para gestionar el 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.

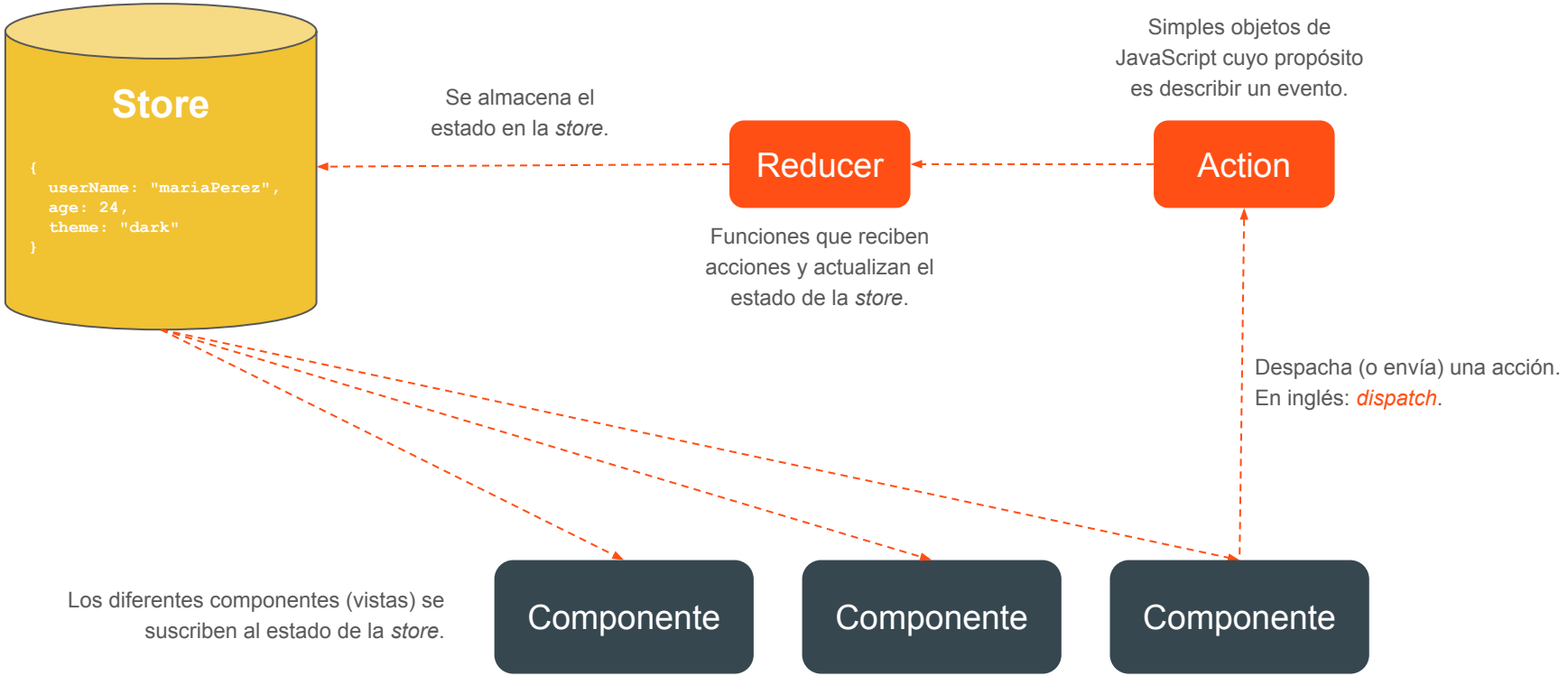
Redux proporciona ciertas pautas o guías a la hora de gestionar ese estado. Por ejemplo, propone guardar todo el estado de la aplicación de forma global, en un objeto JavaScript llamado *store*.

Redux (2/7)

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 (3/7) – Flujo del estado



Redux (4/7) – *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 },
};
```

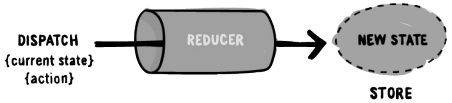
OPCIONAL

Redux (5/7) – *Action Creators*

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 (6/7) – Reducers



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

Todo *reducer* tiene las siguientes características:

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

2. Retorna un 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”. Ver explicación.

   Nota: No es necesario hacer un *deep clone* del *state* cada vez que se retorna un nuevo estado. Ver esta explicación, esta otra explicación y este tweet de Dan Abramov.

En archivo: `tasksReducer.js`

```
function tasksReducer(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*) y se le asigna un valor por defecto (opcional) que es un array vacío.

Redux (7/7) – Reducers (cont)

Dado que los *reducers* son funciones “puras”, dentro de un *reducer* no se debería:

- Mutar los argumentos recibidos por el *reducer* (`state` y `action`).
- Ejecutar “efectos secundarios” como llamar a una API.
- Invocar a funciones “no puras” como `Date.now()` o `Math.random()`.

Redux Toolkit

Críticas comunes a Redux 😢

- La configuración es muy complicada.
- Se necesita instalar varios paquetes extra para lograr hacer algo útil.
- Se necesita escribir mucho *boilerplate* (código repetitivo).

👉 ¿Qué se puede hacer al respecto?

Redux Toolkit (1/2)

Al ver los problemas mencionados anteriormente, los creadores de Redux crearon Redux Toolkit (RTK), un paquete que tiene como objetivo estandarizar la forma en que se escribe la lógica de Redux. Según ellos, es la forma recomendada de utilizar Redux.

En cierta manera, RTK cumple el mismo objetivo que cumple Vite o CRA a la hora de hacer el *setup* de un proyecto de React.

👉 Ver documentación de RTK.

Redux Toolkit (2/2)

A diferencia de Redux “puro”, RTK es opinionado. Es menos flexible, pero tiene ventajas 😀 como:

- Simplifica la instalación y configuración de Redux.
- Reduce el *boilerplate*.
- Sigue buenas prácticas.

Instalación de React Toolkit en un proyecto de React existente:

```
npm i @reduxjs/toolkit
```

React Redux

React Redux

React Redux es una librería que proporciona *bindings* para facilitar el uso de Redux con React.

Recordar que Redux es una librería independiente de React.
React Redux es quien “une” a ambas partes.

Instalación:

```
npm i react-redux
```

Creación de la *store*

Creación de la *store* (1/2)

En Redux “puro” existe una función `createStore` que se utiliza para crear la *store*. Sin embargo, en RTK existe se utiliza una versión “mejorada” de esta función llamada `configureStore`. Ejemplo de uso:

En archivo: `index.js`

```
import { configureStore } from "@reduxjs/toolkit";
import tasksReducer from "./tasksReducer";


const store = configureStore({
  reducer: { tasks: tasksReducer },
});
```

En este ejemplo, hay un sólo *reducer*, pero en caso de ser necesario, se podrían agregar más. La función `configureStore` de RTK se encarga de “combinarlos”. En caso de no usar RTK, habría que combinar los *reducers* usando la función `combineReducers` de Redux.

Creación de la *store* (2/2)

Luego de crear la *store*, es necesario “inyectarla” en todo el “árbol” de componentes de la *app* (para quedar disponible en todos los componentes). Esto se hace gracias al componente `Provider` de React Redux.

En archivo: `main.js`:

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

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Notar que la inyección debe hacerse en lo más alto de la jerarquía.

Acceder a la *store*

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 tasks = useSelector((state) => state.tasks);
```

- `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 se está retornando la parte de la *store* que contiene la lista de tareas.

Despachar una acción

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 – 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.

Redux – Combinar *reducers*

Redux – Combinar *reducers*

Si bien RTK se encarga de combinar los *reducers*, en algunos casos especiales podría llegar a ser necesario combinar los *reducers* antes de definir la *store*.

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

Al *reducer* resultante se le suele llamar `rootReducer`.

```
import { combineReducers } from "redux";
import users from "./userReducer";
import tasks from "./tasksReducer";
import tweets from "./tweetsReducer";

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