Tema 5: Frameworks JS en el cliente

parte III: Gestión del estado

1. Introducción. El estado de una *app*

¿Qué es el estado de una *app* en *frontend*?

- Datos que muestra la *app* y que vienen del servidor
- Datos que ha introducido el usuario y que habrá que sincronizar con el servidor
- Información global de la app como el usuario autentificado, las preferencias, ...

"Como los requisitos en aplicaciones JavaScript de una sola página se están volviendo cada vez más complicados, nuestro código, mas que nunca, debe manejar el estado. ***Este estado puede incluir respuestas del servidor y datos cacheados, así como datos creados localmente que todavía no fueron guardados en el servidor. El estado de las UI también se volvió más complejo****, al necesitar mantener la ruta activa, el tab seleccionado, si mostrar o no un spinner...*

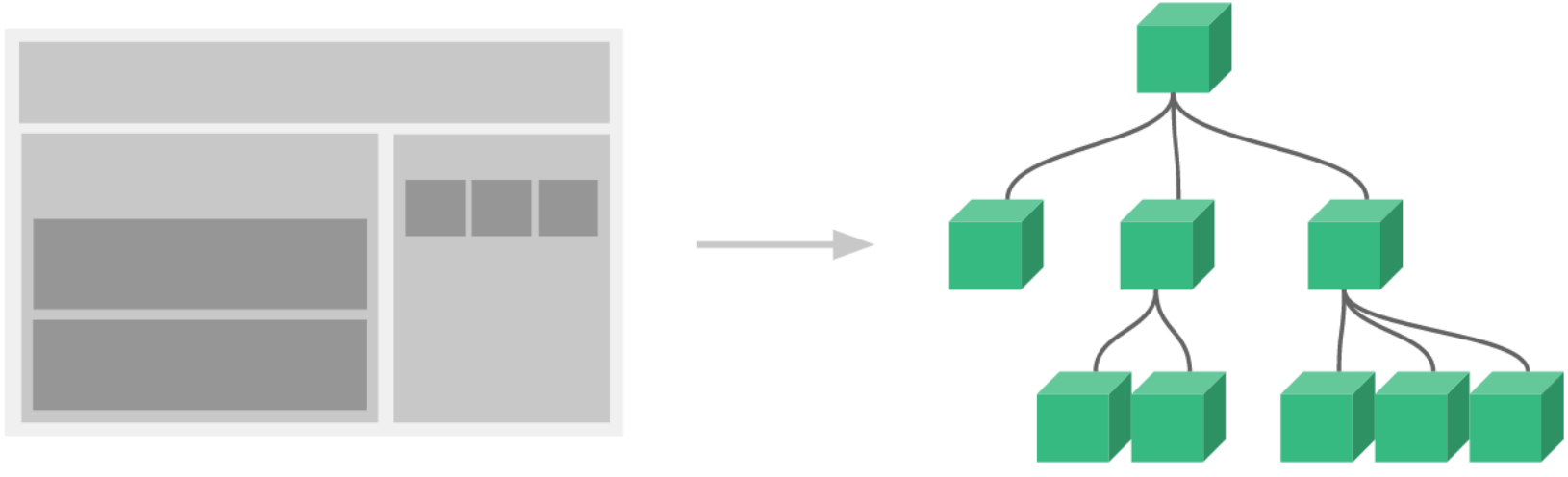
Controlar ese cambiante estado es difícil*. Si un modelo puede actualizar otro modelo, entonces una vista puede actualizar un modelo, el cual actualiza otro modelo, y esto causa que otra vista se actualice.* ***En cierto punto, ya no se entiende que esta pasando en la aplicación ya que perdiste control sobre el cuándo, el por qué y el cómo de su estado****.*

De la documentación de Redux: "Motivación"

> *If you don't think managing state is tricky, consider the fact that 80% of all problems in all complex systems are fixed by rebooting.*
>
> *— stuarthalloway (@stuarthalloway) June 1, 2019*

2. Estado local/distribuido

Recordemos que las aplicaciones Vue, React, Angular... están formadas de **componentes organizados jerárquicamente**

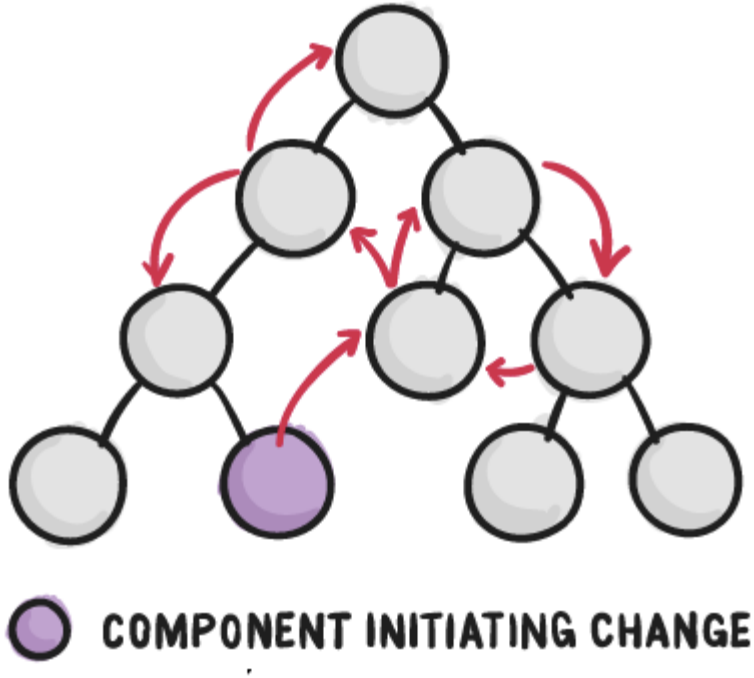


- Una idea *natural* es que **cada componente almacene localmente su estado**

```
<template>
  <li :class="estado ? 'tachado' : ''" @click="cambiarEstado">
    {{ nombre }}
  </li>
</template>

<script setup>
  import { ref } from 'vue';
  const props = defineProps(["nombre","comprado", "id"])
  const estado = ref(props.comprado);
  const cambiarEstado = () => { estado.value = !estado.value};
</script>

<style scoped>
  .tachado {
    text-decoration: line-through;
  }
</style>
```

- Problema: si hacemos esto necesitaremos **pasar estado entre componentes**: ejemplo:
  - para mostrar items ordenados por diferentes criterios (nombre, comprado o no,...)
  - para mostrar el número de las cosas que quedan por comprar

Queremos reducir/organizar al máximo el paso de datos entre componentes, para evitar un flujo de datos complicado



1. Reducir el número de componentes con estado
2. Mantener un flujo unidireccional de la información
3. Estandarizar comunicación entre componentes no relacionados

1. Reducir el número de componentes con estado

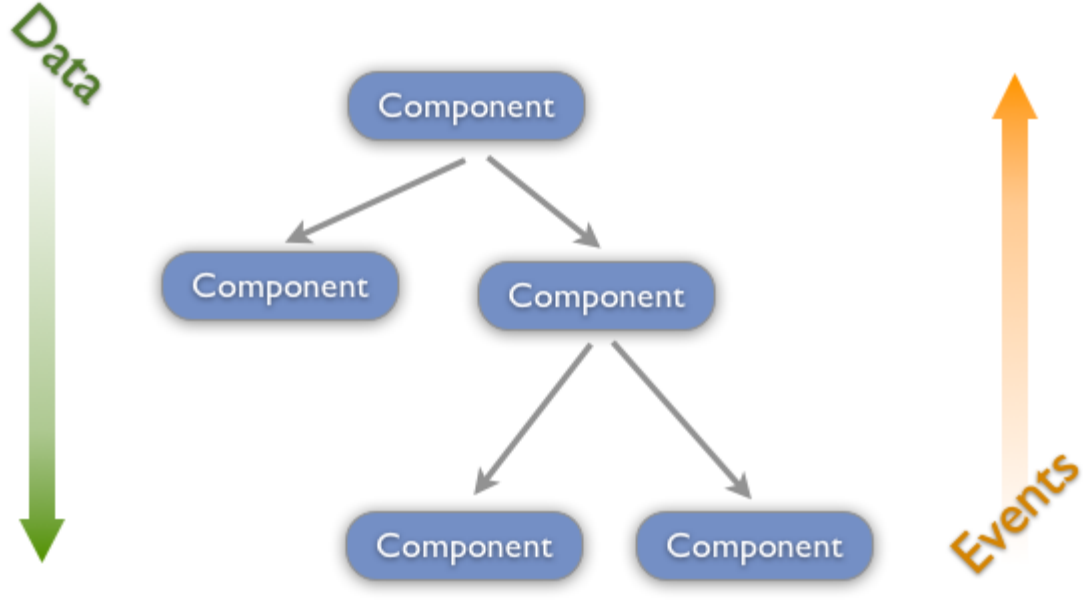
> *Práctica recomendada: almacenar el **estado** solo en el componente de **nivel superior***

Beneficio: si un componente no tiene estado podemos considerar la vista como una **función pura de sus `props`**.

Simplifica el *testing* y el razonamiento sobre el componente

2. Mantener un flujo unidireccional de información

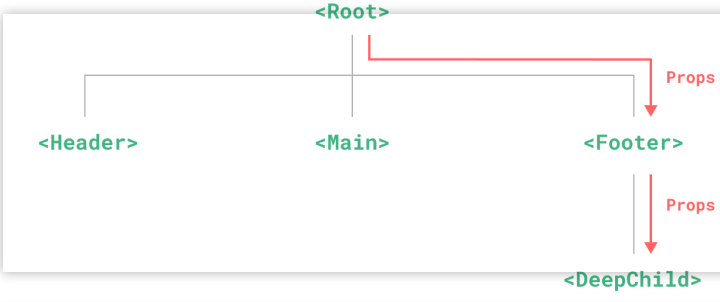
- Si cambia el estado: comunicación de "padres" a "hijos" con las *props*
- Si hay que cambiarlo : de "hijos" a "padres" mediante eventos



Más fácil seguir la pista de los cambios si hay algún *bug*

Provide/Inject en Vue

- Permite compartir datos de un componente a sus descendientes




```
//componente en un nivel superior
<script setup>
  import { provide } from 'vue'
  provide('mensaje', 'Hey!')
</script>
```

```
//descendiente (hijo, nieto, ...)
<script setup>
  import { inject } from 'vue'
  const mensaje = inject('mensaje')
</script>
```

- En otros *frameworks* existen funcionalidades similares, por ejemplo **Context** en React

3. Organizar la comunicación entre componentes no relacionados

Posibilidad: **event bus**

- Es simplemente un objeto global que permite **publicar eventos y suscribirse a ellos**. Los eventos serán los mensajes entre componentes.
- En Javascript este patrón suele llamarse *event bus* o *event emitter*. Hay multitud de librerías que implementan esta idea

Event Bus en Vue

Tenemos que usar alguna librería externa (aquí usamos mitt)

```
//Esto debería ser global a todos los componentes
import mitt from "mitt"
const emitter = mitt()

//para emitir un evento:
emitter.emit("nombre-evento", {dato1:"hola", dato2:1})

//para suscribirse a un evento:
emitter.on("nombre-evento", function(payload) { console.log(payload.dato1)})
```

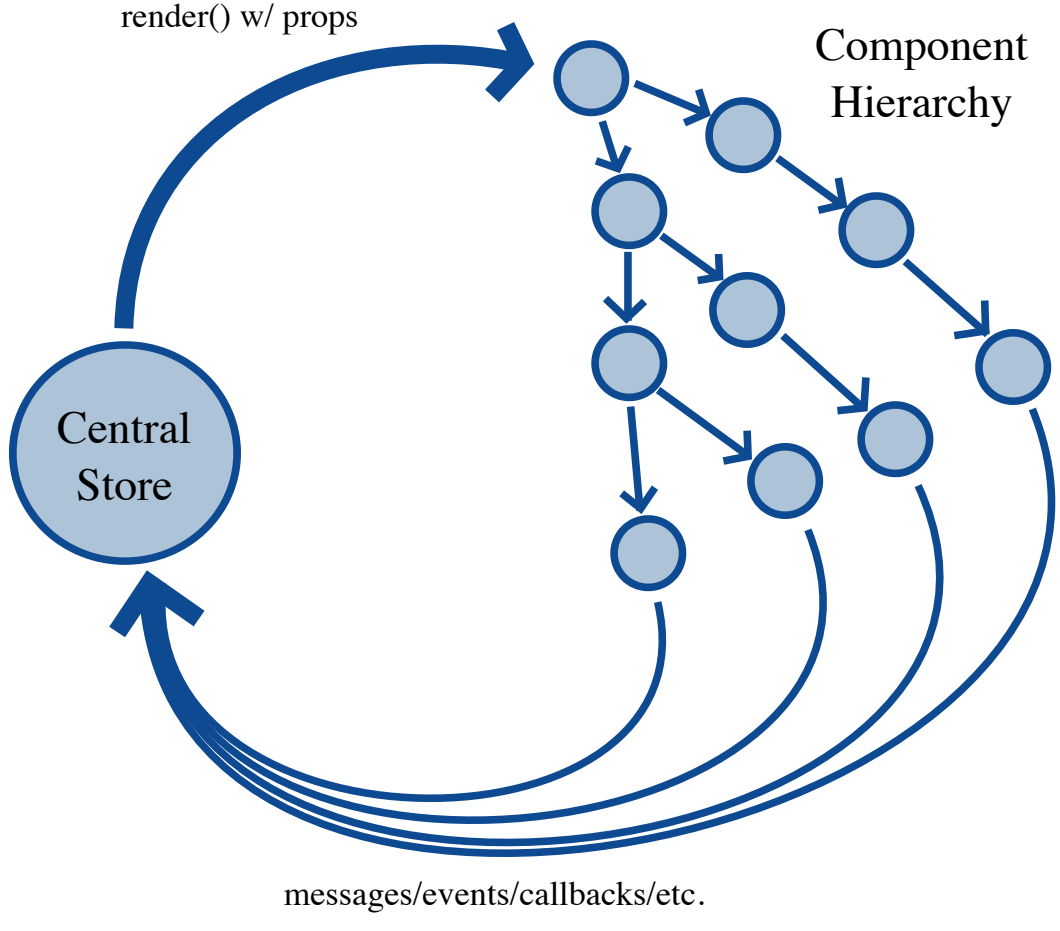
Ejemplo completo

Por desgracia, el *event bus* rompe la idea de *flujo unidireccional*.

¿Cómo podemos seguir manteniendo un flujo unidireccional de información en toda la aplicación?

2. Estado centralizado. El patrón *Store*

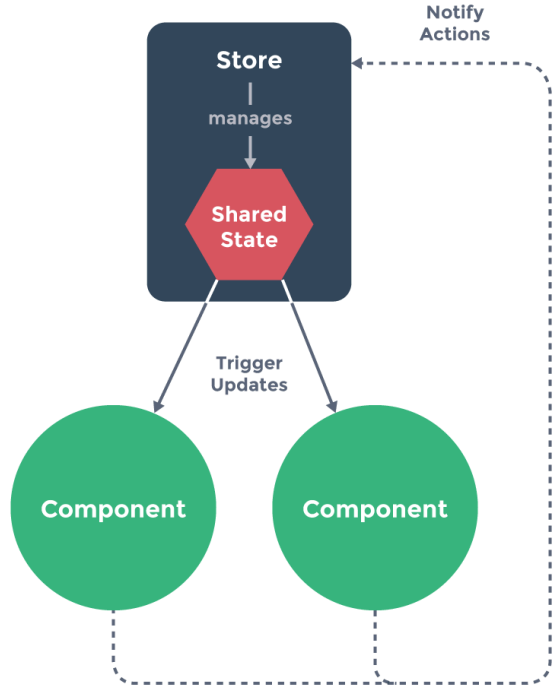
Idea: ¿por qué no sacamos el estado fuera de todos los componentes y nos lo llevamos a un "almacén centralizado"?



De ese modo **todos los componentes** se convertirían en funcionales

El "patrón" *store*

- *store*: almacén centralizado con el estado de la *app*
- Los componentes **no modifican directamente** el estado, las modificaciones se hacen siempre a través de **métodos del *store***




```
var store = {
  state: reactive({
    message: 'Hello!'
  }),
  setMessage (newValue) {
    this.state.message = newValue
  },
  clearMessage () {
    this.state.message = ''
  }
}
```

Ejemplo completo

3. Estado centralizado en Vue: Pinia/Vuex

Pinia/Vuex 5 es el *framework* "oficial" de Vue para la gestión centralizada del estado. Es una implementación del "patrón *store*" (algo más sofisticada que lo que vimos antes)

Aunque es particular de Vue se basa en los mismos principios básicos que se aplican habitualmente en el resto de *frameworks* Javascript: React (Redux), Angular (NgRedux), Svelte (Stores), ...

Antecedentes de Pinia

- **La arquitectura Elm** (2012): Elm es un lenguaje específico para clientes web que transpila a JS
- **Flux** (2014): arquitectura propuesta por Facebook para estructurar aplicaciones con su framework React
- **Redux** (2015): la variante de Flux de mayor éxito, normalmente usada en React pero portada luego a frameworks como Angular o Vue
- Versiones anteriores de **Vuex**

El patrón *store* básico en Pinia

- El *store* es un objeto que contiene las propiedades
  - `state`: el "árbol global" con el estado de la *app*
  - `getters`: variables calculadas en función del estado
  - `actions`: métodos para modificar el estado

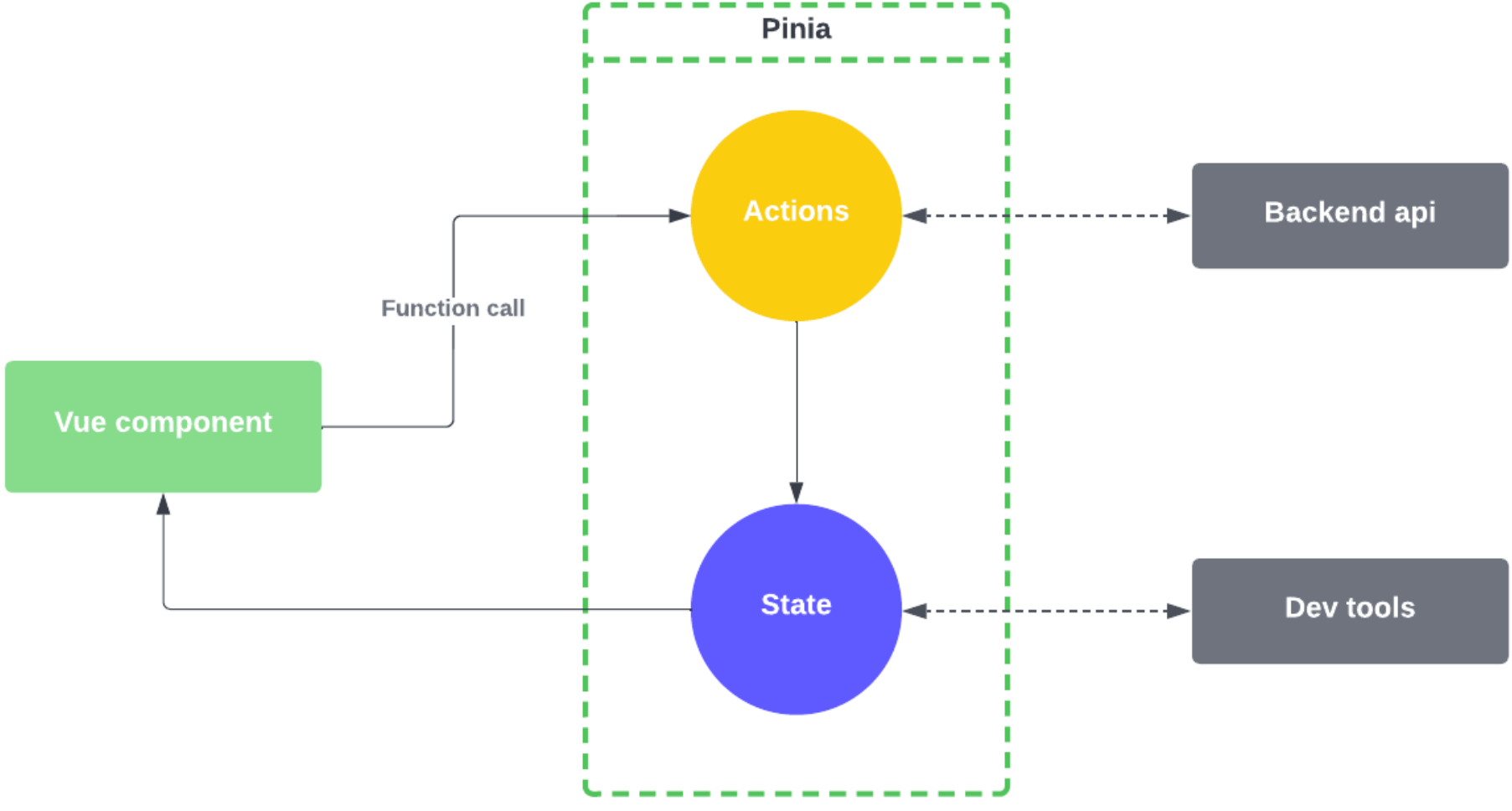
```
import { defineStore } from "pinia";

export const useContadorStore = defineStore("contador", {
  state: () => ({ valor: 0 }),  //función que devuelve el estado inicial
  getters: { //reciben el estado como parámetro
    valorDoble: (state) => state.valor * 2
  },
  actions: {  //acceden al estado con this
    incrementar() {
      this.valor++;
    }
  }
});
```

Usar el *store* en un componente

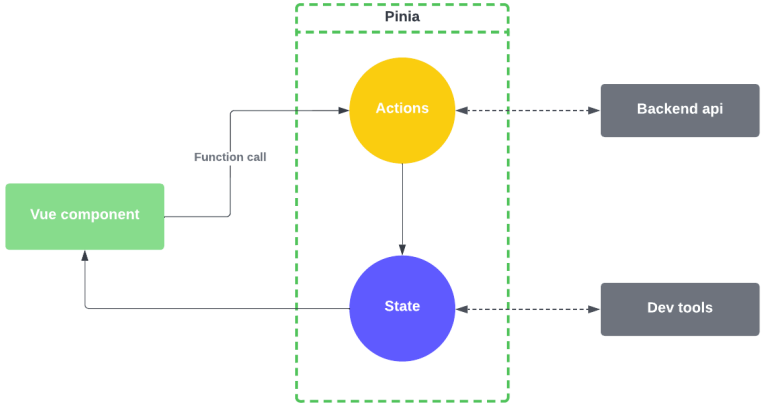
```
<script setup>
  import { useContadorStore } from './store.js';

  const store = useContadorStore();
</script>

<template>
  <div>
    <h1>{{ store.valor }}</h1>
    <button @click="store.incrementar()">Incrementar</button>
  </div>
</template>
```

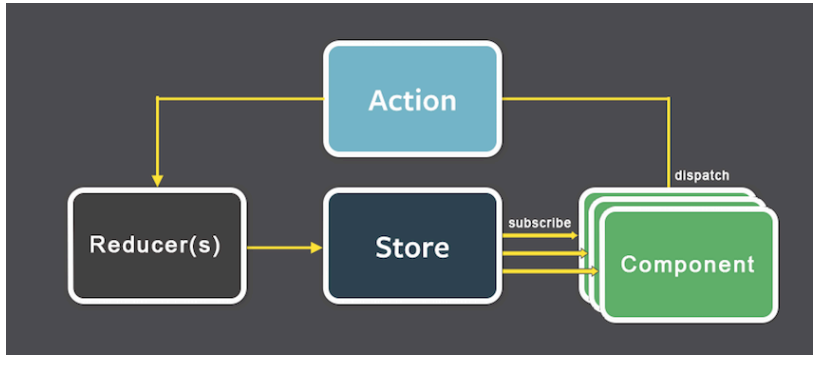
El ejemplo *online*

Flujo unidireccional en Pinia



Pinia vs. Redux





Ejemplo en Svelte 3

```
//store.js (definir el store)
import { writable } from 'svelte/store';
export let count = writable(0);
```

```
//App.svelte (usar el store)
<script>
     import {count} from './store.js'
     function incrementar(params) {
             $count++
          //tambien valdria
          //count.update(n=>n+1)
     }
</script>
<h1>{$count}</h1>
<button onclick={incrementar}>Incrementar</button>
```

Ejemplo online

Os recomiendo echarle un vistazo a la "complex state management guide" de la documentación de SolidJS para ver cómo se usan los *stores* en este framework

Para **aplicaciones pequeñas, Pinia/Redux... no son necesarios**

> ***People often choose Redux before they need it***. *"What if our app doesn't scale without it?"* ***Later, developers frown at the indirection Redux introduced to their code***. *"Why do I have to touch three files to get a simple feature working?" Why indeed! People blame Redux, React, functional programming, immutability, and many other things for their woes, and I understand them. It is natural to compare Redux to an approach that doesn't require "boilerplate" code to update the state, and to conclude that Redux is just complicated*

Dan Abramov, You Might Not Need Redux"

Time travel debugging

Si hacemos un *log* de los cambios de estado, avanzando y retrocediendo por él podemos **reproducir el estado de la aplicación en cualquier momento**