UD11 - Comunicación entre componentes

1. Introducción

Cada componente tiene sus propios datos que son **datos de nivel de componente**, pero hay ocasiones en que varios componentes necesitan acceder a los mismos datos. Es lo que nos sucede en nuestra aplicación de los ejercicios de repaso donde varios componentes necesitan acceder a la lista de tareas (variable *todos*) para mostrarla (*todo-list*), añadir items (*todo-add*) o borrarla (*todo-del-all*).

Estos datos se consideran **datos de nivel de aplicación** y hay varias formas de tratarlos.

Ya hemos visto que podemos pasar información a un componente hijo mediante *props*. Esto permite la comunicación de padres a hijos, pero queda por resolver cómo comunicarse los hijos con sus padres para informarles de cambios o eventos producidos y cómo comunicarse otros componentes entre sí.

Nos podemos encontrar las siguientes situaciones:

- Comunicación de padres a hijos: paso de parámetros (***props***)
- Comunicación de hijos a padres: emitir eventos (***emit***)
- Comunicación entre otros componentes: usar el patrón ***store pattern***

- Comunicación más compleja: ***Pinia***

2. Props (de padre a hijo)

Ya hemos visto que podemos pasar parámetros del padre al componente hijo. Si el valor del parámetro cambia en el padre automáticamente se reflejan esos cambios en el hijo.

NOTA: Cualquier parámetro que pasemos sin *v-bind* se considera texto. Si queremos pasar un número, booleano, array u objeto hemos de pasarlo con *v-bind* igual que hacemos con las variables para que no se considere texto:

```
<ul>
  <todo-item todo="Aprender Vue" :done="false" ></todo-item>
</ul>
```

Si queremos pasar varios parámetros a un componente hijo podemos pasarle un objeto en un atributo *v-bind* sin nombre y lo que recibirá el componente hijo son sus propiedades:

```
<template>
  <ul>
    <todo-item v-bind="propsObject" ></todo-item>
  </ul>
</template>

<script>
  ...
  data() {
    return {
      propsObject: {
        todo: 'Aprender Vue',
        done: false
      }
    }
  }
  ...
</script>
```

y en el componente se reciben sus parámetros separadamente:

```
app.component('todo-item', {
  props: ['todo', 'done'],
  ...
})
```

También es posible que el nombre de un parámetro que queramos pasar sea una variable:

```
<child-component :[paramName]="valorAPasar" ></child-component>
```

| Para prácticar, puedes realizar el ejercicio del tutorial de Vue.js

2.1 Nunca cambiar el valor de una prop

Al pasar un parámetro mediante una *prop* su valor se mantendrá actualizado en el hijo si su valor cambiara en el padre, pero no al revés por lo que no debemos cambiar su valor en el componente hijo (de hecho Vue3 no nos lo permite).

Si tenemos que cambiar su valor porque lo que nos pasan es sólo un valor inicial podemos crear una variable local a la que le asignamos como valor inicial el parámetro pasado:

```
props: ['initialValue'],
data(): {
  return {
    myValue: this.initialValue
  }
}
```

Y en el componente usaremos la nueva variable *myValue*.

Si no necesitamos cambiarla sino sólo darle determinado formato a la variable pasada lo haremos creando una nueva variable (en este caso mejor una *computed*), que es con la que trabajaremos:

```
props: ['cadenaSinFormato'],
computed(): {
  cadenaFormateada() {
    return this.cadenaSinFormato.trim().toLowerCase();
  }
}
```

OJO: Si el parámetro es un objeto o un array éste se pasa por referencia por lo que si lo cambiamos en el componente hijo **sí** se cambiará en el padre, cosa que debemos evitar.

2.2 Validación de props

Al recibir los parámetros podemos usar *sintaxis de objeto* en lugar de *sintaxis de array* y en ese caso podemos indicar algunas cosas como:

- **type**: su tipo (String, Number, Boolean, Array, Object, Date, Function, Symbol o una clase propia).

Puede ser un array con varios tipos: `type: [Boolean, Number]`

- **default**: su valor por defecto si no se pasa ese parámetro
- **required**: si es o no obligatorio
- **validator**: una función que recibe como parámetro el valor del parámetro y devolverá true o false en función de si el valor es o no válido

Ejemplos:

```
props: {
  nombre: String,
  apellidos: {
    type: String,
    required: true
  },
  idPropietario: {
    type: [Boolean, Number],
    default: false
  },
  products: {
    type: Object,
    default(): {
      return {id:0, units: 0}
    }  // Si es un objeto o array _default_ debe ser una función que devuelva el valor
  },
  nifGestor: {
    type: String,
    required: true,
    validator(value): {
      return /^[0-9]{8}[A-Z]$/.test(value)   // Si devuelve *true* será válido
    }
  }
```

2.3 Pasar otros atributos de padre a hijo

Además de los parámetros, que se reciben en *props*, el componente padre puede poner cualquier otro atributo en la etiqueta del hijo, quien lo recibirá y se aplicará a su elemento raíz. A esos atributos se puede acceder a través de `$attr`. Por ejemplo:

```
<!-- componente padre -->
<date-picker id="now" data-status="activated" class="fecha"></date-picker>
```

```
// Componente hijo
app.component('date-picker', {
  template: `
    <div class="date-picker">
      <input type="datetime" />
    </div>
  `,
  methods: {
    showAttributes() {
      console.log('Id: ' + this.$attrs.id + ', Data: ' +
this.$attrs['data-status'])
    }
  }
})
```

El subcomponente se renderizará como:

```
<div class="fecha date-picker" id="now" data-status="activated">
  <input type="datetime" />
</div>
```

y al ejecutar el método *showAttributes* mostrará en la consola `Id: now, Data: activated`.

A veces no queremos que esos atributos se apliquen al elemento raíz del subcomponente sino a alguno interno (habitual si le pasamos escuchadores de eventos). En ese caso podemos deshabilitar la herencia de parámetros definiendo el atributo del componente `inheritAttrs` a *false* y aplicándolos nosotros manualmente:

```
<!-- componente padre -->
<date-picker id="now" data-status="activated" @input="dataChanged"></date-
picker>
```

```
// Componente hijo
app.component('date-picker', {
  inheritAttrs: false,
  template: `
    <div class="date-picker">
      <input type="datetime" v-bind="$attrs" />
    </div>
  `,
})
```

En este caso se renderizará como:

```
<div class="date-picker">
  <input type="datetime" class="fecha" id="now" data-status="activated" @input="dataChanged" />
</div>
```

El componente padre está escuchando el evento *input* sobre el <INPUT> del componente hijo.

En Vue3, si el componente hijo tiene varios elementos raíz deberemos *bindear* los *attrs* a uno de ellos como acabamos de ver.

3. Emitir eventos (de hijo a padre)

Si un componente hijo debe pasarle un dato a su padre o informarle de algo puede emitir un evento que el padre capturará y tratará convenientemente. Para emitir el evento el hijo hace:

```
  this.$emit('nombre-evento', parametro);
```

El padre debe capturar el evento como cualquier otro. En su HTML hará:

```
<my-component @nombre-evento="fnManejadora" ... />
```

y en su JS tendrá la función para manejar ese evento:

```
  methods: {
    fnManejadora(param) {
      ...
    },
  }
  ...
```

El componente hijo puede emitir cualquiera de los eventos estándar de JS ('click', 'change', ...) o un evento personalizado ('cambiado', ...).

Ejemplo: continuando con la aplicación de tareas que dividimos en componentes, en el componente ***todo-item*** en lugar de hacer un alert emitiremos un evento al padre:

```
delTodo() {
  this.$emit('del-item');
},
```

y en el componente ***todo-list*** lo escuchamos y llamamos al método que borre el item:

```
<template>
    <div>
      <h2>{{ title }}</h2>
```

```
      <ul>
       <todo-item
         v-for="(item, index) in todos"
         :key="item.id"
         :todo="item"
         @del-item="delTodo(index)">
       </todo-item>
      </ul>
      <add-item></add-item>
      <br>
      <del-all></del-all>
    </div>
</template>

<script>
  ...
  methods: {
    delTodo(index){
      this.todos.splice(index,1);
    },
  }
</script>
```

NOTA: En componentes y *props* se hace la conversión automáticamente entre los nombres en Javascript escritos en camelCase y los usados en HTML en kebab-case pero esto no sucede con los eventos, por lo que en el código deberemos nombrarlos también en kebab-case.

Igual que un componente declara las *props* que recibe, también puede declarar los eventos que emite. Esto es opcional pero proporciona mayor claridad al código:

```
// TodoItem.vue
...
props: {
  todo: Object
},
emits: ['del-item'],
...
```

| Haz el ejercicio del tutorial de Vue.js

3.1 Capturar el evento en el padre

En ocasiones (como en este caso) el componente hijo no hace nada más que informar al padre de que se ha producido un evento sobre él. En estos casos podemos hacer que el evento se capture directamente en el padre en lugar de en el hijo:

Componente ***todo-list.vue***

```
<template>
    <div>
      <h2>{{ title }}</h2>
      <ul>
       <todo-item
         v-for="(item, index) in todos"
         :key="item.id"
         :todo="item"
         @dblclick="delTodo(index)">
        </todo-item>
    ...
</template>
```

Le estamos indicando a Vue que el evento *dblclick* se capture en *todo-list* directamente por lo que el componente *todo-item* no tiene que capturarlo ni hacer nada:

Componente ***todo-item.vue***

```
<template>
    <li>
      <label>
    ...
</template>
```

3.2 Definir y validar eventos

Como hemos dicho, los eventos que emite un componente pueden (y se recomienda por claridad) definirse en la opción *emits*:

```
app.component('todo-item', {
  emits: ['toogle-done', 'dblclick'],
  props: ['todo'],
  ...
```

Es recomendable definir los argumentos que emite usando sintaxis de objeto en vez de array, similar a como hacemos con las *props*. Para ello el evento se asigna a una función que recibe como parámetro los parámetros del evento y devuelve *true* si es válido o *false* si no lo es:

`custom-form.vue`

```
<script>
  emits: {
    // No validation
    click: null,
    // Validate submit event
    submit: ({ email, password }) => {
      if (email && password) {
        return true
      } else {
        console.warn('Invalid submit event payload!')
        return false
      }
    }
  },
</script>
```

En este ejemplo el componente emite *click* que no se valida y *submit* donde se valida que reciba 2 parámetros.

4. Compartir datos

Una forma más sencilla de modificar datos de un componente desde otros es compartiendo los datos entre ellos. Definimos en un fichero *.js* aparte un objeto que contendrá todos los datos a compartir entre componentes, lo importamos y lo registramos en el *data* de cada componente que tenga que acceder a él. Ejemplo:

Fichero `/src/store/index.js`

```
import { reactive } from 'vue';

export const store = reactive({
  message: '',
  newData: {},
  ...
})
```

NOTA: En Vue3 para que la variable store sea reactiva (que la vista reaccione a los cambios que se produzcan en ella) hay que declararla con `reactive` si es un objeto o con `ref` si es un tipo primitivo

(*string*, *number*, ...).

Fijaos que se declara el objeto *store* como una constante porque NO puedo cambiar su valor para que pueda ser usado por todos los componentes, pero sí el de sus propiedades.

Componente `compA.vue`

```
<template>
  <p>Mensaje: { { message}} </p>
</template>

<script>
import { store } from '../store/'
  ...
  data() {
    return {
      store,
      // y a continuación el resto de data del componente
      ...
    }
  },
  ...
</script>
```

Componente `compB.vue`

```
<template>
  <button @click="delMessage">Borrar mensaje</button>
</template>

<script>
import { store } from '../store/'
  ...
  data() {
    return {
      store,
      // y a continuación el resto de data del componente
      ...
    }
  },
  methods: {
    delMessage() {
      this.store.message='';
    }
```

```
  },
  ...
})
</script>
```

Desde cualquier componente podemos modificar el contenido de **store** y esos cambios se reflejarán automáticamente tanto en la vista de todos ellos.

Esta forma de trabajar tiene un grave inconveniente: como el valor de cualquier dato puede ser modificado desde cualquier parte de la aplicación es difícilmente mantenible y se convierte en una pesadilla depurar el código y encontrar errores.

Para evitarlo podemos usar un patrón de almacén (*store pattern*) que veremos en el siguiente apartado.

4.1 $root y $parent

Todos los componentes tienen acceso además de a sus propios datos declarados en `data()`, a los datos y métodos definidos en la instancia de Vue (donde hacemos el `new Vue`). Por ejemplo:

```
new Vue({
  el: '#app',
  data: {
    message: 'Hola',
  },
  methods: {
    getInfo() {
  ...
}
```

Desde cualquier componente podemos hacer cosas como:

```
console.log(this.$root.message);
this.$root.message='Adios';
this.$root.getInfo();
```

También es posible acceder a los datos y métodos del componente padre del actual usando `$parent` en lugar de `$root`.

De esta manera podríamos acceder directamente a datos del padre o usar la instancia de Vue como almacén (evitando crear el objeto **store** para compartir datos). Sin embargo, aunque esto puede ser útil en aplicaciones pequeñas, es difícil de mantener cuando nuestra aplicación crece por lo que se recomienda usar un ***Store pattern*** como veremos a continuación o **Pinia** si nuestra aplicación va a ser grande.

4.2 Store pattern

Es una mejora sobre lo que hemos visto de compartir datos. Para evitar que cualquier componente pueda modificar los datos compartidos en el almacén, las acciones que modifican dichos datos están incluidas dentro del propio almacén, lo que facilita su seguimiento:

Fichero `/src/store/index.js`

```
import { reactive } from 'vue';

export const store={
  debug: true,
  state: reactive({
    message: '',
    ...
  }),
  setMessageAction (newValue) {
    if (this.debug) console.log('setMessageAction triggered with ', newValue)
    this.state.message = newValue
  },
  clearMessageAction () {
    if (this.debug) console.log('clearMessageAction triggered')
    this.state.message = ''
  }
}
```

Componente `compA.vue`

```
<script>
import { store } from '/src/datos.js'
  ...
  data() {
    return {
      sharedData: store.state,
      // o aún mejor declaramos sólo las variables que necesitemos, ej
      // message: store.state.message,
      ...
    }
  },
  ...
</script>
```

Componente `compB.vue`

```
<script>
import { store } from '/src/datos.js'
  ...
  methods: {
    delMessage() {
      store.clearMessageAction();
    }
  },
  ...
</script>
```

IMPORTANTE: no podemos machacar ninguna variable del *state* si es un objeto o un array, por ejemplo para borrar los datos de un array no podemos hacer

```
// Esto está MAL
clearProductsAction () {
 this.state.products = [];
}
```

porque entonces el array *products* dejaría de ser reactivo (lo estamos machacando con otro). Debemos usar métodos como *push*, *splice*, ...

```
// Esto está BIEN
clearProductsAction () {
 this.state.products.splice(0, this.state.products.length);
}
```

Esto se soluciona usando librerías de gestión de estado como *Pinia* o *Vuex*.

5. Pinia

Es una librería para gestionar los estados en una aplicación Vue. Ofrece un almacenamiento centralizado para todos los componentes con unas reglas para asegurar que un estado sólo cambia de determinada manera. Es el método a utilizar en aplicaciones medias y grandes y le dedicaremos todo un tema más adelante. En Vue2 y anteriores la librería que se usaba es *Vuex*.

En realidad es un *store pattern* que ya tiene muchas cosas hechas y que se integra perfectamente con las *DevTools*.

Lo veremos en detalla en la unidad dedicada a esta librería.

6. Slots

Otra forma en que un componente hijo puede mostrar información del padre es usando *slots*. Un *slot* es una ranura en un componente que, al renderizarse, se rellena con lo que le pasa el padre en el innerHTML de la etiqueta del componente. El *slot* tiene acceso al contexto del componente padre, no al del componente donde se renderiza. Los *slots* son una herramienta muy potente. Podemos obtener toda la información en la documentación de Vue.

Ejemplo:
Tenemos un componente llamado *my-component* con un slot:

```
app.component('my-component', {
  template:
    `<div>
      <h3>Componente con un slot</h3>
      <slot><p>Esto se verá si no se pasa nada al slot</p></slot>
    </div>`
})
```

Si llamamos al componente con:

```
<my-component>
  <p>Texto del slot</p>
</my-component>
```

se renderizará como:

```
<div>
  <h3>Componente con un slot</h3>
  <p>Texto del slot</p>
</div>
```

Pero si lo llamamos con:

```
<my-component>
</my-component>
```

se renderizará como:

```
<div>
  <h3>Componente con un slot</h3>
  <p>Esto se verá si no se pasa nada al slot</p>
</div>
```

| Haz el ejercicio del tutorial de Vue.js

6.1 Slots con nombre

A veces nos interesa tener más de 1 slot en un componente. Para saber qué contenido debe ir a cada slot se les da un nombre.

Vamos a ver un ejemplo de un componente llamado *base-layout* con 3 *slots*, uno para la cabecera, otro para el pie y otro principal:

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

A la hora de llamar al componente hacemos:

```
<base-layout>
  <template slot="header">
    <h1>Here might be a page title</h1>
  </template>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template slot="footer">
    <p>Here's some contact info</p>
  </template>
</base-layout>
```

Lo que está dentro de un *template* con atributo *slot* irá al_slot_ del componente con ese nombre. El resto del innerHTML irá al *slot* por defecto (el que no tiene nombre).

El atributo *slot* podemos ponérselo a cualquier etiqueta (no tiene que ser <template>):

```
<base-layout>
  <h1 slot="header">Here might be a page title</h1>

  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <p slot="footer">Here's some contact info</p>
</base-layout>
```