EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: ESCUCHANDO EVENTOS.
- EXERCISE 2: MANEJANDO EVENTOS DESDE MÉTODO.
- EXERCISE 3: MODIFICADOR PREVENT.
- EXERCISE 4: MODIFICADOR STOP.
- EXERCISE 5: MODIFICADOR CAPTURE.
- EXERCISE 6: MODIFICADOR SELF.

EXERCISE 1: ESCUCHANDO EVENTOS

Para comenzar, vamos a crear un proyecto con `vue/cli`, tal como lo hemos hecho en los CUE anteriores. Creamos una carpeta llamada “cue4”, y dentro de ella, instalaremos desde la consola, el proyecto con el comando “vue create cue4”.

En el siguiente ejercicio, aplicaremos la directiva `v-on`, para capturar el evento clic cuando un botón sea presionado por el usuario. Para esto, crearemos un componente llamado `“ListenComponent.vue”` en la carpeta components, el cual estará encargado de la funcionalidad.

CREANDO COMPONENTE:

Una vez creado, vamos a vincular el evento clic con el dato `“counter”`. En este ejemplo, no lo vincularemos a un método, si no que haremos la lógica de sumar en el mismo evento, y por cada clic sumaremos 1.

```
<template>

  <div>
    <button v-on:click="counter +=1">Sumar 1</button>
    <p>El boton de arriba ha sido clickeado {{counter}}</p>
  </div>
</template>

<script>
```

```
export default {
  data: function(){
    return {
      counter:0,
    }
  }
}
</script>
```

ENTENDIENDO COMPORTAMIENTO

Lo primero que debemos hacer es crear un dato llamado `counter`, el cual será un número.

```
data: function(){
    return {
      counter:0,
    }
  }
```

Una vez teniendo el dato creado, ya podemos renderizarlo en la vista.

```
<p>El botón de arriba ha sido clickeado {{counter}}</p>
```

Al partir, nuestro valor será 0.

LIGANDO EVENTO:

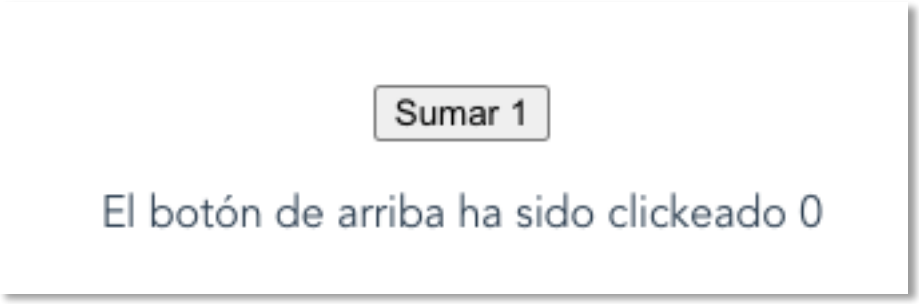
Para ligar eventos, podemos utilizar la palabra `v-on`, o su atajo `@`. En este caso, se ligará el evento clic, al dato counter.

```
<button v-on:click="counter +=1">Sumar 1</button>
```

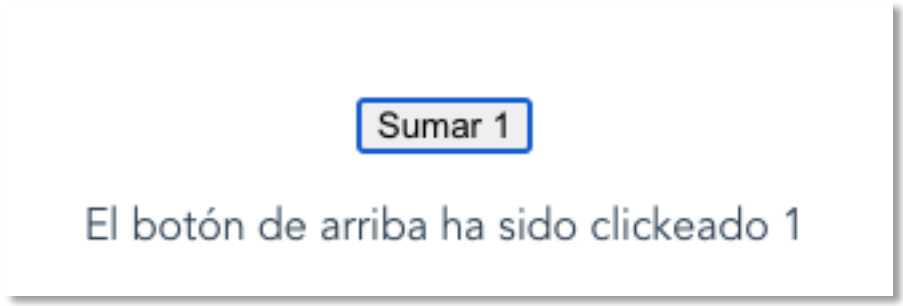
REACTIVIDAD VUE:

Como sabemos, al declarar un dato dentro de **`'data'`**, Vue les asigna un **`watcher`** (método para observa cambios). Si se provoca algún cambio en la data declarada, dicho dato se vuelve a renderizar, y es por esto que podemos ver cada cambio de valor, simplemente renderizándolo.

```
1 <p>El botón de arriba ha sido clickeado {{counter}}</p>
```



Por cada clic, podremos observar el cambio en el dato "**counter**".



Para correr el ejemplo anterior, debemos vincular el component **`"ListenComponente.vue"`** a **`App.vue`**, como se ha hecho en los CUE's anteriores.

Archivo App.vue

```vue
<template>
  <div id="app">
    <ListenComponent></ListenComponent>
  </div>
</template>

<script>
import ListenComponent from '@/components/ListenComponent.vue'
export default {
  name: 'App',
  components: {
    ListenComponent
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

EXERCISE 2: MANEJANDO EVENTOS DESDE MÉTODO.

El siguiente ejemplo, utilizaremos el mismo proyecto “cue4” para la realización del ejercicio. El componente que utilizaremos será: **“ListenComponent.vue”**. Este archivo lo modificaremos, por lo que se recomienda utilizar Git para el control de versiones, y guardar los cambios del Ejercicio 1.

Para contextualizar, en algunos casos es necesario generar cierta lógica cuando un evento es gatillado, por ejemplo: validar un formulario, enviar datos a un servidor, entre otros. En estos, no podemos hacer lógica dentro del **template**, por lo que es necesario hacerlo en un método.

En CUE’s anteriores, ya hemos utilizado métodos, pero es necesario estudiarlos nuevamente para consolidar nuestro conocimiento.

Lo que haremos en este ejercicio, será trasladar la lógica que desarrollamos en el Ejercicio 1, a un método, para que el template quede ordenado y podamos incluir más lógica si es que así lo necesitamos.

```
<template>
  <div>
    <button v-on:click="contador">Sumar 1</button>
    <p>El botón de arriba ha sido clickeado {{counter}}</p>
  </div>
</template>
```

CREANDO MÉTODO:

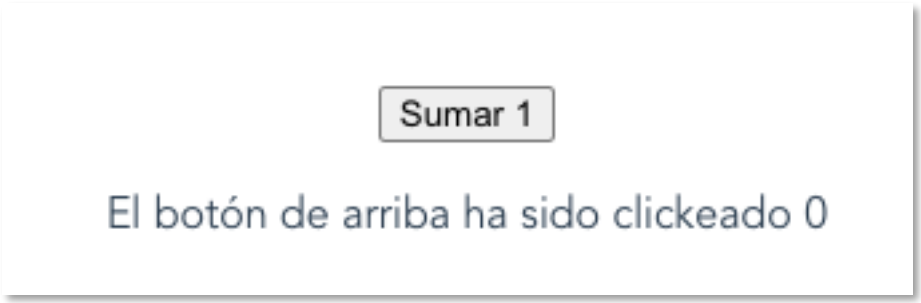
Si observamos, el evento **“v-on:click=”** está ligado a un método llamado contador, y para que esto funcione, debemos crear el método en el objeto **methods**.

```
<script>
export default {
  data: function(){
    return {
      counter:0,
    }
  },
  methods:{
    contador: function(){
      this.counter += 1;
```

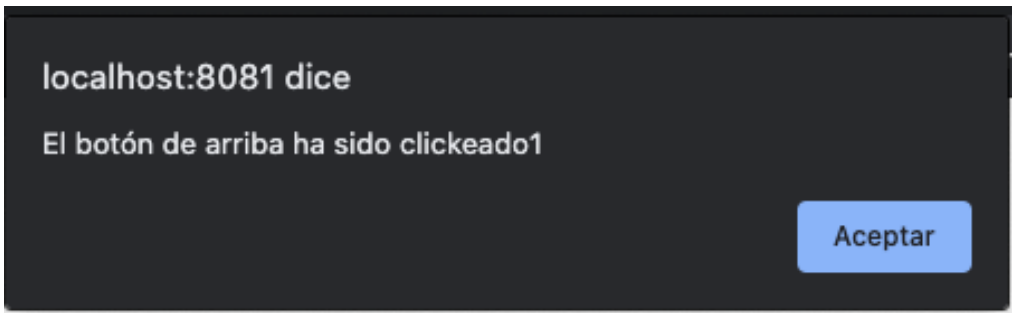
```
11          alert("El botón de arriba ha sido clickeado" +
12 this.counter);
13      }
14    }
15 }
16 </script>
```

En el ejemplo anterior, el método está sumando 1 al dato counter, y posteriormente lanza una alerta. El resultado es el mismo del ejercicio 1.

Si guardamos nuestros cambios, y nos dirigimos al navegador, se podrá observar así:



Al hacer clic, ahora también nos mostrará una alerta.



EXERCISE 3: MODIFICADOR PREVENT

Para comenzar, vamos a continuar en el proyecto “cue4”, pero crearemos un componente nuevo llamado “**EventModifier.vue**”, dentro de la carpeta components.

Para que nuestro componente pueda ser visualizado, lo importaremos a App.vue.

```vue
<template>
  <div id="app">
    <EventModifier></EventModifier>
  </div>
</template>

<script>

import EventModifier from '@/components/EventModifier.vue'
export default {
  name: 'App',
  components: {
    EventModifier
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

MODIFICADOR PREVENT:

En algunos casos, los elementos HTML se conforman de cierta forma predefinida. Por ejemplo: al ser enviado un formulario (`<form>`), su comportamiento es recargar la página posterior a ello.

En el siguiente ejercicio, se explica cómo poder evitar este problema a través de modificadores de eventos, en este caso, el **modificador prevent**, el cual detendrá el comportamiento de recargar página.

COMPONENTE EVENTMODIFIER.VUE:

Como se explicó en las indicaciones, el que editaremos será el recién creado `"EventModifier"`, y partiremos por la sección de `"template"`. En éste vamos a crear un formulario.

```
<template>
  <div>
    <form>
      <label for="">Nombre</label>
      <input type="text" v-model="form.name">
      <label for="">Apellido</label>
      <input type="text" v-model="form.lastName">
      <input type="submit" @click.prevent="sendData">
    </form>
  </div>
</template>
```

La clave de este ejercicio es el input `"submit"`, el cual está asociado a un evento "click.prevent". El prevent no permitirá recargar la página.

En la sección de script, crearemos los datos que ligamos a través de `v-model`, y un método llamado `sendData`

```
<script>
export default {
  data: function(){
    return {
      form:{
        name: "",
        lastName: "",
      }
    }
```

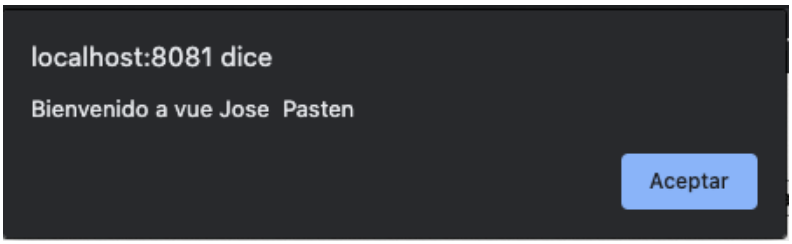
```
  },
  methods:{
    sendData: function(){
      alert("Bienvenido a vue " + this.form.name +" " +
this.form.lastName);
    }
  }
}
</script>
```

Para prevenir el recargo de la página, se ha utilizado el modificador prevent, luego del evento click.

```
<input type="submit" @click.prevent="sendData">
```

Si guardamos y nos dirigimos a `localhost` desde el navegador, veremos:

Si escribimos nuestro nombre en el formulario, y presionamos enviar, el método `sendData` lanzará un evento.



EXERCISE 4: MODIFICADOR STOP

Vamos a continuar en el proyecto “cue4”, y crearemos un nuevo componente llamado: **“EventModifierStop.vue”**.

Lo importaremos al componente principal **App.vue**, como lo hemos hecho en los ejercicios anteriores.

MODIFICADOR STOP:

En algunas ocasiones, podemos tener problemas de propagación de eventos, cuando tenemos elementos HTML anidados. Al hacer clic sobre uno de estos elementos, dicho evento se propaga a elementos ancestros, que pueden provocar comportamientos no deseados. Es para esto, que se crea el modificador Stop.

En el componente **“EventModifierStop.vue”**, crearemos 3 **div** que están anidados, para provocar una propagación de evento, y poder estudiar su comportamiento.

Primero, vamos a codificar la sección de **template**.

```
<template>
  <div>
    <div id="abuelo"  @click="saludo('hola Soy abuelo')">
      Div Abuelo
      <div id="padre" @click="saludo('hola Soy padre')">
        Div Padre
        <div id="hijo"  @click="saludo('hola Soy hijo')">
          Div Hijo
        </div>
      </div>
    </div>
  </div>
</template>
```

Ahora, nos dirigimos a la sección de script, en la que vamos a crear un método llamado “saludo”, el cual recibirá como argumento un String, que será mostrado dentro de una alerta como mensaje. En el ejemplo, cada div tiene un mensaje distinto, para poder darnos cuenta cuando cada uno de ellos se ejecuta.

```
<script>
export default {
  methods:{
    saludo: function(str){
      alert(str)
    }
  }
}
</script>
```

En la sección de style, agregaremos los siguientes estilos CSS.

```
#abuelo{
    background:grey;
    width: 300px;
    height: 300px;
    font-size:25px;
    margin: 0 auto;
  }
  #padre{
    background: rgb(158, 147, 84);
    width: 200px;
    height: 200px;
    margin: 0 auto;
    position:relative;
    top:10%;
    font-size:25px;
  }
  #hijo{
    background: rgb(88, 88, 124);
    width:100px;
    height: 100px;
    margin:0 auto;
    position:relative;
    top:10%;
    font-size:20px;
  }
```

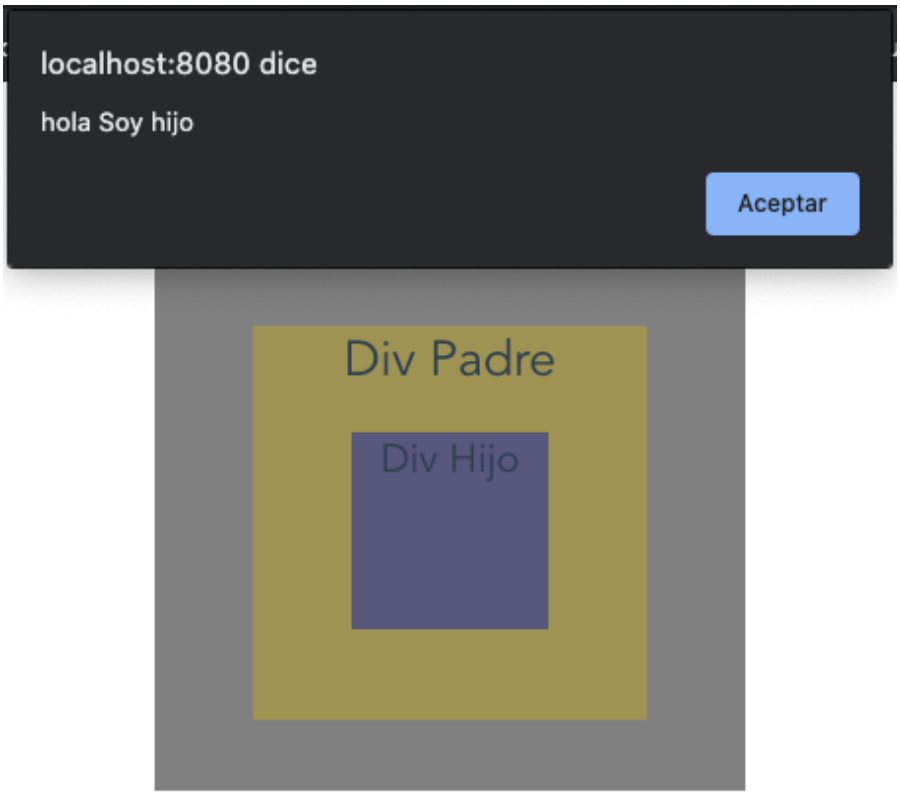
Si guardamos y vamos a localhost en el navegador, veremos lo siguiente:

Si hiciéramos clic en el div hijo, éste se propagaría hacia los demás elementos, ya que está dentro de ellos. Con propagar, nos referimos a que los métodos en los divs ancestros igual se ejecutarán, difundiendo el evento clic a todos los elementos que contengan el div inicial del evento. Para evitar esto, solo hace falta agregar “stop” después de clic.

Iremos a la sección de “template”, y en el div `id=”hijo”`, agregaremos el modificador stop.

```
<template>
  <div>
    <div id="abuelo"  @click="saludo('hola Soy abuelo')">
      Div Abuelo
      <div id="padre" @click="saludo('hola Soy padre')">
        Div Padre
        <div id="hijo" @click.stop="saludo('hola Soy hijo')">
          Div Hijo
        </div>
      </div>
    </div>
  </div>
</template>
```

De esta forma, al hacer clic en el div `id="hijo"`, éste no se propagará hacia los ancestros directos.



EXERCISE 5: MODIFICADOR CAPTURE

INSTRUCCIONES:

Continuando en el proyecto "cue4", crearemos un componente llamado **"EventModifierCapture.vue"**, el cual nos permitirá estudiar el comportamiento de este modificador.

Para que el nuevo componente que acabamos de crear pueda ser visualizado en el navegador, debemos vincularlo en el archivo **App.vue**. Para ello, se recomienda comentar el componente creado en el ejercicio anterior.

Para entender qué hace **capture,** en primer lugar debemos asociar que, cuando hacemos clic en algún elemento HTML, el evento se procesa en 2 fases:

1. Capturing.
2. Bubbling.

CAPTURE:

En el componente **"EventModifierCaptue.vue"**, agregaremos la siguiente codificación:

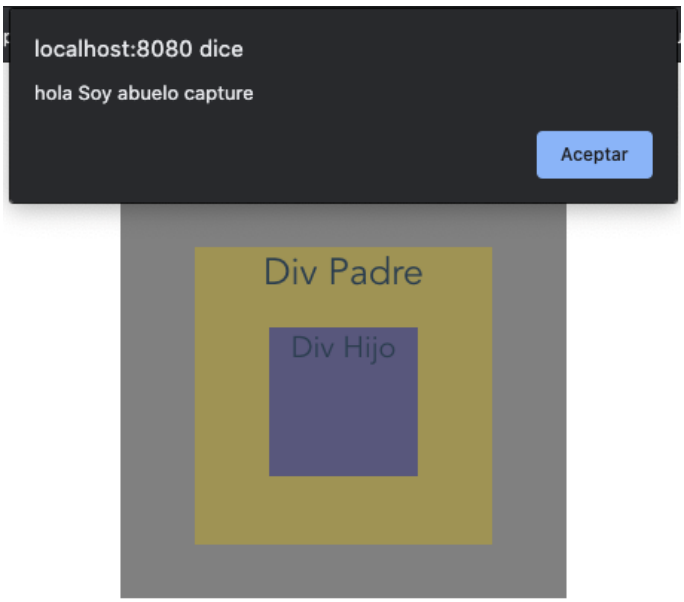
```
<template>
  <div>
    <div
      id="abuelo"
      @click.capture="saludo('hola Soy abuelo capture')"
      @click="saludo('hola Soy abuelo bubbling')"
      >
      Div Abuelo
      <div
        id="padre"
        @click.capture="saludo('hola Soy padre capture')"
        @click="saludo('hola Soy padre bubbling')"
        >
        Div Padre
        <div
          id="hijo"
          @click.capture="saludo('hola Soy hijo capture')"
          @click="saludo('hola Soy hijo bubbling')"
        >
          Div Hijo
        </div>
      </div>
```

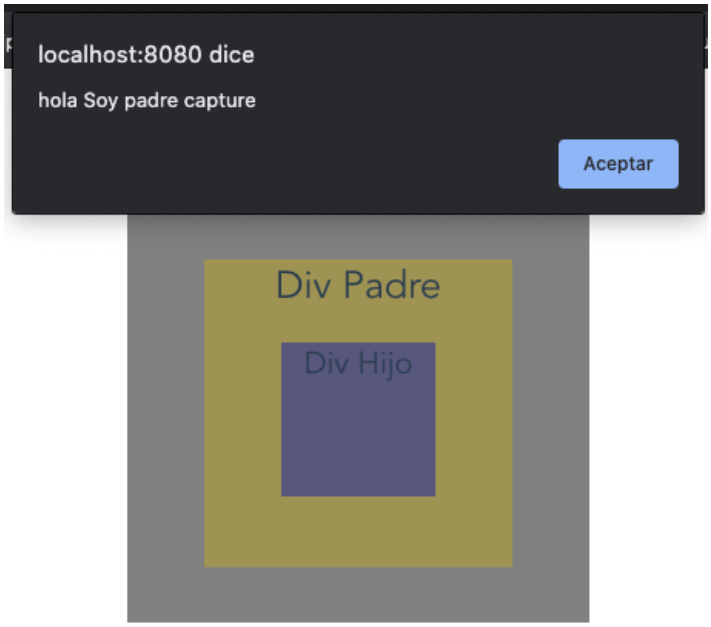
```
    </div>
  </div>
</template>
<script>
export default {
  methods:{
    saludo: function(str){
      alert(str)
    }
  }
}
</script>

<style scoped>
  #abuelo{
    background:grey;
    width: 300px;
    height: 300px;
    font-size:25px;
    margin: 0 auto;
  }
  #padre{
    background: rgb(158, 147, 84);
    width: 200px;
    height: 200px;
    margin: 0 auto;
    position:relative;
    top:10%;
    font-size:25px;
  }
  #hijo{
    background: rgb(88, 88, 124);
    width:100px;
    height: 100px;
    margin:0 auto;
    position:relative;
    top:10%;
    font-size:20px;
  }
</style>
```

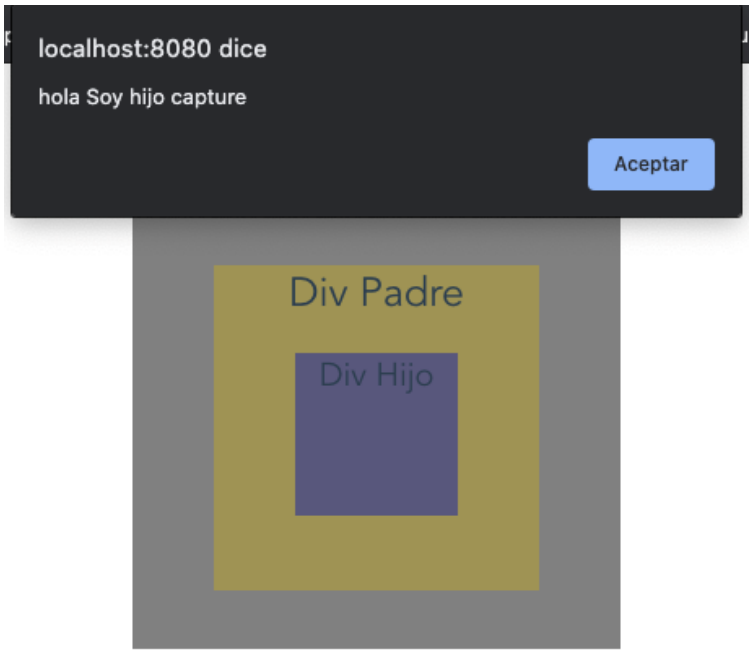
Si abrimos el navegador, observaremos un ejemplo similar al ejercicio 4.

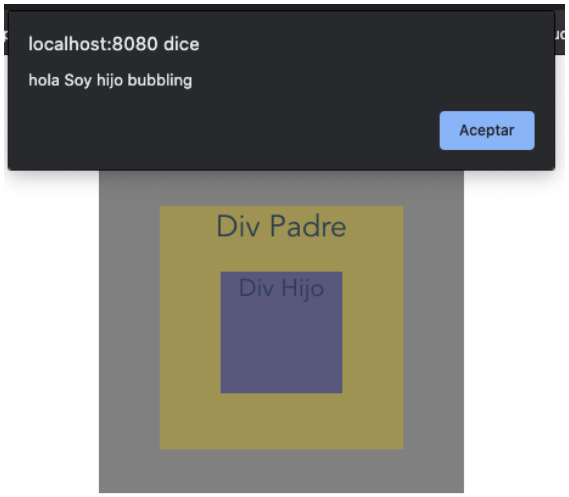
Al hacer clic sobre el div Hijo, este gatillará un evento, que puede ser utilizado para ancestros, antes de gatillar el propio.

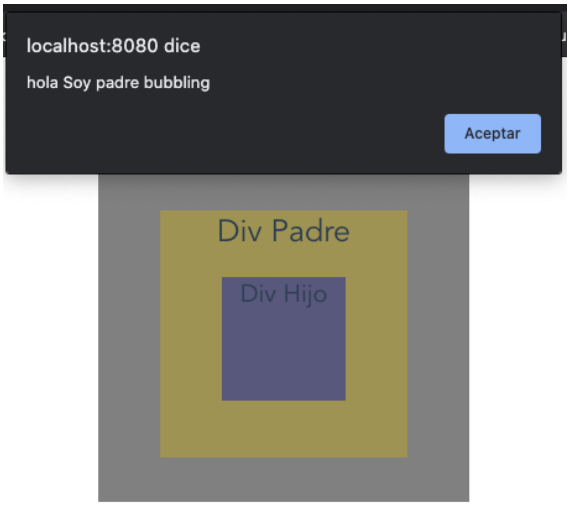
En el caso anterior, como cada div tiene un modificador capture, siempre se gatillará el más lejano.

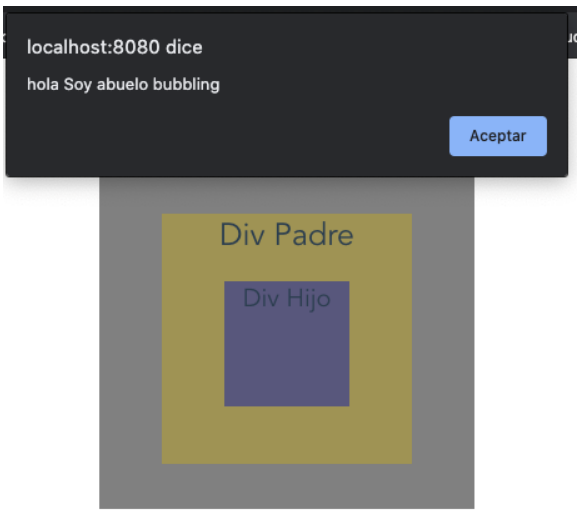



localhost:8080 dice
hola Soy padre capture
Aceptar
Div Padre
Div Hijo


localhost:8080 dice
hola Soy hijo capture
Aceptar
Div Padre
Div Hijo


localhost:8080 dice
hola Soy hijo bubbling
Aceptar
Div Padre
Div Hijo


localhost:8080 dice
hola Soy padre bubbling
Aceptar
Div Padre
Div Hijo


localhost:8080 dice
hola Soy abuelo bubbling
Aceptar
Div Padre
Div Hijo

EXERCISE 6: MODIFICADOR SELF

Continuaremos en el proyecto "cue4". Para el siguiente ejercicio, crearemos el componente "EventModifierSelf.vue", el cual debe ser incluido en el archivo principal App.vue. Es recomendable comentar el componente importado en el ejercicio anterior.

MODIFICADOR SELF:

Cuando lo aplicamos, le estamos diciendo al elemento que solo se active si es él quien provoca el evento, y no otro elemento HTML, es decir, un elemento hijo no podrá gatillarlo.

@CLICK.SELF:

Al hacer clic sobre el div hijo, este no gatillará al evento del div padre (`@click.self`), pero si al div abuelo. De esta forma, podemos ver cómo se comporta self.

```
<template>
  <div>
    <div id="abuelo"  @click="saludo('hola Soy abuelo')">
      Div Abuelo
      <div id="padre" @click.self="saludo('hola Soy padre')">
        Div Padre
        <div id="hijo" @click="saludo('hola Soy hijo')">
          Div Hijo
        </div>
      </div>
    </div>
  </div>

</template>

<script>
export default {
  methods:{
    saludo: function(str){
      alert(str)
    }
  }
}
</script>

<style scoped>
  #abuelo{
    background:grey;
    width: 300px;
```

```
    height: 300px;
    font-size:25px;
    margin: 0 auto;
  }
  #padre{
    background: rgb(158, 147, 84);
    width: 200px;
    height: 200px;
    margin: 0 auto;
    position:relative;
    top:10%;
    font-size:25px;
  }
  #hijo{
    background: rgb(88, 88, 124);
    width:100px;
    height: 100px;
    margin:0 auto;
    position:relative;
    top:10%;
    font-size:20px;
  }
</style>
```

Si abrimos el navegador, veremos lo siguiente: