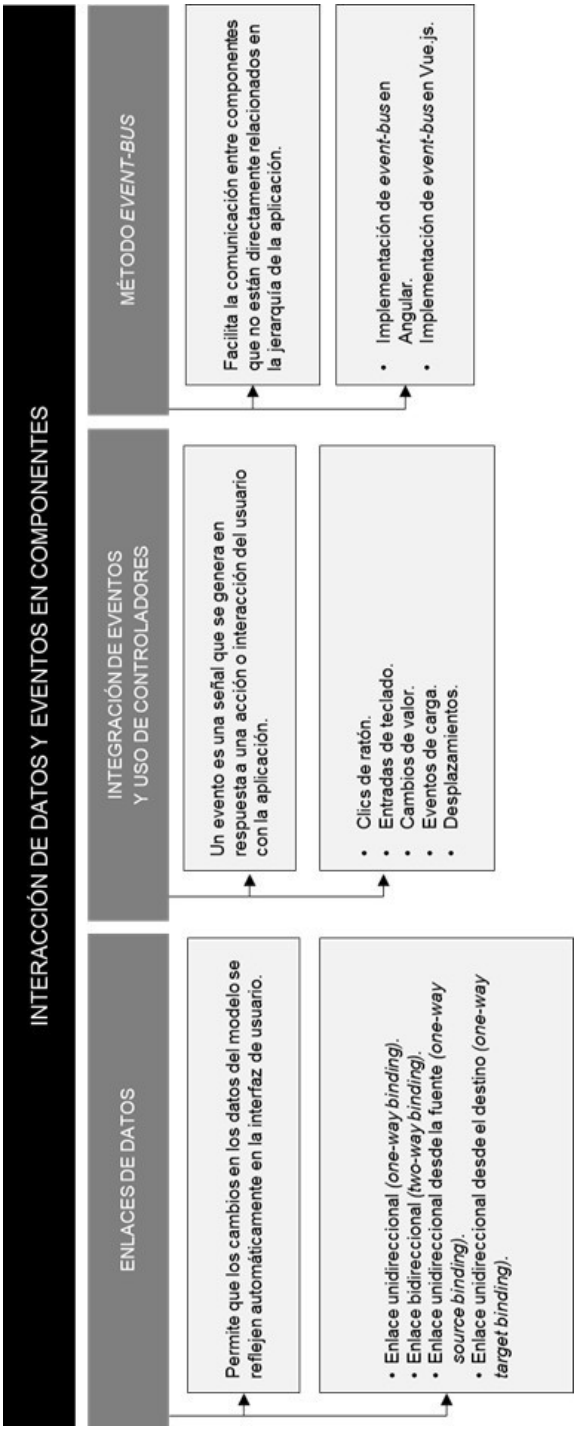
Desarrollo de Interfaces

Tema 4. Interacción de datos y eventos en componentes

Índice


INTERACCIÓN DE DATOS Y EVENTOS EN COMPONENTES
ENLACES DE DATOS
Permite que los cambios en los datos del modelo se reflejen automáticamente en la interfaz de usuario.
• Enlace unidireccional (one-way binding).
• Enlace bidireccional (two-way binding).
• Enlace unidireccional desde la fuente (one-way source binding).
• Enlace unidireccional desde el destino (one-way target binding).
INTEGRACIÓN DE EVENTOS Y USO DE CONTROLADORES
Un evento es una señal que se genera en respuesta a una acción o interacción del usuario con la aplicación.
• Clics de ratón.
• Entradas de teclado.
• Cambios de valor.
• Eventos de carga.
• Desplazamientos.
MÉTODO EVENT-BUS
Facilita la comunicación entre componentes que no están directamente relacionados en la jerarquía de la aplicación.
• Implementación de event-bus en Angular.
• Implementación de event-bus en Vue.js.

4.1. Introducción y objetivos

En el desarrollo moderno de aplicaciones, especialmente en *frameworks* como Vue.js y Angular, la **comunicación efectiva entre componentes** es esencial para crear interfaces de usuario dinámicas y reactivas. A medida que las aplicaciones crecen en tamaño y complejidad, gestionar la interacción entre componentes puede volverse un desafío. Es en este contexto donde el **método *event-bus*** emerge como una solución poderosa y flexible.

El ***event-bus*** es un **patrón de diseño** que facilita la **comunicación desacoplada** entre componentes que no están directamente relacionados en la jerarquía de la aplicación. A diferencia de la comunicación directa entre componentes padre e hijo mediante la transmisión de propiedades y emisión de eventos, el *event-bus* actúa como un **intermediario centralizado.** Esto permite que cualquier componente de la aplicación pueda emitir eventos y que otros componentes, que quizás no tengan una relación jerárquica directa, puedan escuchar y reaccionar a esos eventos.

Este patrón es, especialmente, útil en **aplicaciones grandes** y **complejas,** donde la necesidad de una comunicación global, pero controlada, es fundamental. Por ejemplo, imagina una aplicación donde diferentes partes del sistema necesitan reaccionar a un cambio de configuración o a un evento global como el inicio de sesión de un usuario. Con el *event-bus,* un componente puede emitir un evento cuando un usuario inicia sesión, y todos los componentes interesados en ese evento pueden actualizar su estado en consecuencia, sin necesidad de interacciones directas o relaciones complejas.

Uno de los principales **beneficios** del *event-bus* es su capacidad para **reducir** el **acoplamiento entre componentes,** lo que a su vez mejora la mantenibilidad y escalabilidad del código. Al centralizar la gestión de eventos, se facilita la adición de nuevas funcionalidades y se minimizan las dependencias innecesarias entre componentes. Sin embargo, es importante usar el *event-bus* con precaución, ya que su abuso puede llevar a un código difícil de depurar y mantener, especialmente en aplicaciones muy grandes.

E l *event-bus* se puede implementar de diferentes maneras, dependiendo del *framework* utilizado. En Vue.js, por ejemplo, es común utilizar un objeto Vue vacío como *event-bus,* mientras que en Angular, el *event-bus* se suele implementar utilizando servicios y la biblioteca RxJS para manejar la emisión y suscripción a eventos. Este enfoque flexible permite adaptar el uso del *event-bus* a las necesidades específicas de cada proyecto.

Los **objetivos** específicos que se pretenden alcanzar con este tema son:

- **Comprender el concepto de *event-bus:*** adquirir una comprensión sólida de lo que es un *event-bus* y cómo funciona como un intermediario centralizado para la comunicación entre componentes.
- **Aprender a implementar *event-bus* en diferentes *frameworks:*** desarrollar la habilidad para implementar un *event-bus* en *frameworks* como Vue.js y Angular, utilizando las herramientas y técnicas adecuadas para cada entorno.
- **Facilitar la comunicación desacoplada:** aprender a utilizar el *event-bus* para facilitar la comunicación entre componentes que no están directamente relacionados, mejorando la flexibilidad y escalabilidad de la aplicación.
- **Evaluar el uso adecuado del *evento-bus:*** reconocer cuándo es apropiado utilizar un *event-bus* y cuándo podría ser mejor utilizar otros patrones o herramientas de comunicación entre componentes.

4.2. Tipos de enlaces de datos

El **enlace de datos,** también conocido como *data binding,* es un concepto esencial en el desarrollo de interfaces de usuario (UI), especialmente en aplicaciones modernas que requieren una sincronización continua entre los datos y la vista. Este mecanismo permite que los cambios en los datos del modelo se reflejen automáticamente en la interfaz de usuario y, en algunos casos, que los cambios en la interfaz de usuario se reflejen de vuelta en el modelo de datos. Existen varios tipos de enlace de datos que los desarrolladores pueden utilizar según las necesidades de la aplicación y el comportamiento deseado. A continuación, se describen en detalle los principales tipos de enlace de datos, acompañados de ejemplos claros para ilustrar su uso.

Enlace unidireccional (one-way binding)

El **enlace unidireccional** es el tipo más básico de *data binding* y es comúnmente utilizado cuando se necesita mostrar datos en la interfaz de usuario que provienen de un modelo de datos, pero sin la necesidad de que la interfaz de usuario retroalimente o modifique esos datos. En este escenario, los datos **fluyen en una sola dirección,** desde el modelo hasta la vista, asegurando que la UI siempre refleje el **estado actual del modelo,** pero los cambios en la UI no afectan el modelo.

Imaginemos una aplicación que muestra el nombre de un usuario en la pantalla. El nombre del usuario se almacena en un modelo de datos y se muestra en la UI de manera estática. En *frameworks* como Angular, este tipo de enlace se puede lograr mediante la **interpolación:**

```
<p>Nombre del usuario: {{ nombreUsuario }}</p>
```

Figura 1. Interpolación del nombre de usuario. Fuente: elaboración propia.

En este ejemplo, `nombreUsuario` es una propiedad del modelo que **se enlaza unidireccionalmente** con la UI. Si el valor de `nombreUsuario` cambia en el modelo, el cambio se reflejará automáticamente en la interfaz de usuario. Sin embargo, si el usuario modifica el contenido en la UI (por ejemplo, utilizando el navegador para editar el HTML), ese cambio no afectará el valor original de `nombreUsuario` en el modelo.

Este tipo de enlace es útil en casos donde **los datos se originan en el modelo** y solo necesitan ser **presentados en la interfaz** de usuario, como en reportes, *dashboards* o cualquier otro contexto donde la UI sirve, principalmente, como una vista de datos.

Enlace bidireccional (two-way binding)

El **enlace bidireccional** es un mecanismo más avanzado y flexible que permite una **sincronización automática en ambas direcciones** entre el **modelo** y la **vista.** Es decir, cualquier cambio en los datos del modelo se refleja en la interfaz de usuario, y cualquier cambio que el usuario realice en la UI se actualiza automáticamente en el modelo. Este tipo de enlace es especialmente útil en **formularios** o cualquier interfaz donde el usuario interactúa directamente con los datos y se espera que sus entradas modifiquen el estado del modelo.

Consideremos un formulario de entrada de texto en una aplicación de Angular donde el usuario introduce su nombre. Utilizando el enlace bidireccional, podemos enlazar el campo de texto a una propiedad del modelo de datos de la siguiente manera:

```
<input [(ngModel)]="nombreUsuario" placeholder="Introduce tu nombre">
```

Figura 2. Enlace del campo de texto a una propiedad del modelo de datos. Fuente: elaboración propia.

En este caso, la propiedad `nombreUsuario` del modelo está enlazada **bidireccionalmente** al campo de texto. Si el usuario escribe su nombre en el campo de texto, el valor de `nombreUsuario` se actualiza automáticamente. Del mismo modo, si `nombreUsuario` cambia por alguna otra razón (por ejemplo, si se actualiza desde otro componente o a través de un servicio), el cambio se reflejará automáticamente en el campo de texto.

El enlace bidireccional es extremadamente útil en **aplicaciones interactivas** donde los datos de la UI y el modelo deben estar siempre sincronizados, como en formularios de registro, configuraciones de usuario o interfaces de administración.

Enlace unidireccional desde la fuente (one-way source binding)

El **enlace unidireccional** desde la fuente es un tipo de *data binding* que permite actualizar el modelo de datos a partir de los cambios en la UI, pero sin que los cambios en el modelo afecten a la interfaz de usuario. Es lo opuesto al enlace unidireccional tradicional, donde la UI solo refleja los cambios del modelo.

Este tipo de enlace se utiliza menos frecuentemente que otros, pero puede ser útil en escenarios donde la UI está diseñada para **capturar datos de entrada** que no necesariamente necesitan reflejar cambios en tiempo real en la pantalla. Por ejemplo, en una aplicación React, podríamos tener una función que actualiza el estado de un componente en respuesta a un evento de la UI, como un cambio en un campo de entrada:

```
function handleChange(event) {
  setNombreUsuario(event.target.value);
}

return (
  <input type="text" onChange={handleChange} />
);
```

Figura 3. Ejemplo en React que actualiza estado de un componente. Fuente: elaboración propia.

En este ejemplo, cada vez que el usuario escribe en el campo de texto, la función `handleChange` se activa y actualiza el estado `nombreUsuario`. Sin embargo, si el estado `nombreUsuario` cambia desde otra parte del código, no hay un mecanismo automático para que el campo de texto refleje ese cambio, ya que la dirección del enlace es solo desde la UI hacia el modelo.

Este tipo de enlace es útil en escenarios donde se desea un **control** más explícito sobre **cómo** y **cuándo** se **actualizan** los **datos** en el modelo en función de las interacciones del usuario.

Enlace unidireccional desde el destino (one-way target binding)

El **enlace unidireccional desde el destino** es un enfoque donde **solo la UI es actualizada** en respuesta a cambios en el modelo de datos, sin que los cambios en la UI tengan un efecto en el modelo. Este tipo de enlace es adecuado cuando se necesita asegurar que la UI siempre refleja **el estado más reciente del modelo,** pero donde la UI no debe influir en el estado del modelo.

Un ejemplo clásico de enlace unidireccional desde el destino se puede ver en aplicaciones donde los datos son presentados a los usuarios sin que estos tengan la posibilidad de editarlos. Imaginemos una aplicación de Vue.js que muestra los detalles de un producto:

```
<p>Nombre del producto: {{ producto.nombre }}</p>
<p>Precio: {{ producto.precio }} €</p>
```

Figura 4. Ejemplo enlace unidireccional. Fuente: elaboración propia.

En este caso, las propiedades producto.nombre y producto.precio están vinculadas unidireccionalmente a la UI. Si los valores de producto.nombre o producto.precio cambian en el modelo, la interfaz de usuario se actualizará automáticamente para reflejar estos cambios. Sin embargo, cualquier intento de modificar estos valores en la UI (por ejemplo, a través de la edición manual del HTML en un navegador) no afectará el modelo original.

Este tipo de enlace es ideal en situaciones donde la consistencia de los datos mostrados en la UI es crítica, pero donde **no se permite que los usuarios modifiquen esos datos** directamente a través de la interfaz, como en sistemas de consulta, reportes financieros o vistas de detalles de productos.

4.3. Configuración y ejemplos de uso

El **proceso de configuración del enlace de datos** es fundamental para asegurar que la interfaz de usuario y el modelo de datos interactúen de manera fluida y eficiente. Configurar correctamente el enlace de datos en un proyecto no solo implica elegir el tipo de enlace más adecuado para cada situación, sino también asegurarse de que los componentes de la aplicación estén diseñados y organizados de manera que faciliten esta interacción. A continuación, se detalla cómo configurar el enlace de datos en diferentes *frameworks* populares, junto con ejemplos prácticos que ilustran su uso en escenarios comunes de desarrollo de interfaces de usuario.

Configuración del enlace unidireccional en Angular

Angular es un *framework* ampliamente utilizado para la creación de aplicaciones web robustas y dinámicas. Uno de los aspectos clave de Angular es su capacidad para gestionar el enlace de datos de manera efectiva, ya sea unidireccional o bidireccional, lo que permite a los desarrolladores mantener la sincronización entre el modelo y la vista con facilidad.

El **enlace unidireccional** en Angular se configura, principalmente, utilizando la **interpolación** en las **plantillas HTML.** Esto permite que los datos del modelo se reflejen en la UI de manera directa.

Supongamos que estamos construyendo una aplicación que muestra información del perfil de un usuario, como el nombre y la dirección de correo electrónico. La configuración del enlace unidireccional en Angular sería algo como esto:

```
<div>
  <h1>Perfil del Usuario</h1>
  <p>Nombre: {{ usuario.nombre }}</p>
  <p>Email: {{ usuario.email }}</p>
</div>
```

Figura 5. Configuración unidireccional en Angular. Fuente: elaboración propia.

En este ejemplo, usuario.nombre y usuario.email son propiedades del modelo que se enlazan a la vista mediante interpolación. La configuración es simple y directa, y garantiza que cualquier cambio en el modelo (por ejemplo, si el nombre del usuario se actualiza desde una llamada API *[application programming interface]*) se refleje automáticamente en la UI.

Configuración del enlace bidireccional en Angular

Para escenarios donde se necesita que la UI también pueda **actualizar el modelo de datos,** Angular ofrece el **enlace bidireccional** utilizando el **atributo** ngModel **.**

Imaginemos un formulario donde el usuario puede actualizar su información de contacto. Para permitir que las entradas del usuario actualicen el modelo y que cualquier cambio en el modelo se refleje en la UI, configuramos el enlace bidireccional de la siguiente manera:

```
<form>
  <label for="nombre">Nombre:</label>
  <input id="nombre" [(ngModel)]="usuario.nombre" name="nombre">

  <label for="email">Email:</label>
  <input id="email" [(ngModel)]="usuario.email" name="email">

  <button type="submit">Actualizar</button>
</form>
```

Figura 6. Enlace bidireccional. Fuente: elaboración propia.

En este formulario, el uso de [(ngModel)] asegura que cualquier cambio en los campos de entrada actualice las propiedades correspondientes del modelo usuario. Al mismo tiempo, si el modelo se actualiza por otros medios, esos cambios se verán reflejados en los campos de entrada.

Este enfoque es, particularmente, útil en formularios donde los datos del usuario deben ser enviados al servidor para su almacenamiento, ya que garantiza que la **información más reciente** del usuario esté **siempre sincronizada** entre la UI y el modelo de datos.

Configuración del enlace unidireccional en React

En React, el enlace unidireccional se configura mediante el uso de propiedades *(props)* y estado *(state).* El **estado** se utiliza para almacenar datos dentro de un componente, mientras que las **propiedades** se pasan desde los componentes padres a los componentes hijos.

Consideremos una lista de tareas en la que queremos mostrar los elementos pendientes. Cada tarea se almacena en el estado del componente y se pasa a la UI a través de las propiedades:

```
class ListaDeTareas extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      tareas: [
        { id: 1, texto: 'Aprender React' },
        { id: 2, texto: 'Escribir código' },
        { id: 3, texto: 'Leer documentación' }
      ]
    };
  }

  render() {
    return (
      <ul>
        {this.state.tareas.map((tarea) => (
          <li key={tarea.id}>{tarea.texto}</li>
        ))}
      </ul>
    );
  }
}
```

Figura 7. Paso de propiedades a la UI. Fuente: elaboración propia.

En este ejemplo, la lista de tareas se almacena en el estado del componente ListaDeTareas . Luego, cada tarea se representa en la UI mediante el método *render,* asegurando que la UI **siempre esté sincronizada** con el estado del componente.

Configuración de interactividad mediante el estado en React

Aunque React no implementa el enlace bidireccional de manera automática, se puede lograr un comportamiento similar gestionando manualmente el estado en respuesta a los eventos de la UI.

Supongamos que queremos permitir al usuario añadir una nueva tarea a la lista mediante un campo de entrada y un botón:

```
class ListaDeTareas extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      tareas: [],
      nuevaTarea: ''
    };
  }

  manejarCambio = (event) => {
    this.setState({ nuevaTarea: event.target.value });
  }

  agregarTarea = () => {
    const nuevaTarea = {
      id: Date.now(),
      texto: this.state.nuevaTarea
    };
    this.setState((prevState) => ({
      tareas: [...prevState.tareas, nuevaTarea],
      nuevaTarea: ''
    }));
  }
  render() {
    return (
      <div>
        <input
          type="text"
          value={this.state.nuevaTarea}
          onChange={this.manejarCambio}
        />
        <button onClick={this.agregarTarea}>Agregar Tarea</button>
        <ul>
          {this.state.tareas.map((tarea) => (
            <li key={tarea.id}>{tarea.texto}</li>
          ))}
        </ul>
      </div>
    );
  }
}
```

Figura 8. Añadir nueva tarea. Fuente: elaboración propia.

Aquí, el campo de entrada está enlazado al estado `nuevaTarea`. Cada vez que el usuario escribe en el campo de entrada, se actualiza el estado mediante el método `manejarCambio`. Cuando el usuario hace clic en «Agregar Tarea», el estado se actualiza nuevamente para incluir la nueva tarea en la lista de tareas, y la UI se actualiza automáticamente para reflejar esta adición.

Este enfoque permite una gran **flexibilidad,** ya que los desarrolladores tienen control total sobre **cómo** y **cuándo** se **actualizan** los **datos** y la UI en respuesta a las interacciones del usuario.

Configuración del enlace unidireccional en Vue.js

Vue.js es conocido por su simplicidad y flexibilidad, proporcionando soporte tanto para **enlace unidireccional** como **bidireccional.** La configuración del enlace de datos en Vue.js es muy intuitiva, lo que lo convierte en una opción popular para desarrolladores que buscan un *framework* ligero pero potente.

Al igual que en Angular, Vue.js permite el **enlace unidireccional** mediante la **interpolación** en las **plantillas.**

Imaginemos que estamos creando una aplicación para mostrar detalles de productos. Los datos del producto se muestran en la UI utilizando la siguiente configuración en Vue.js:

```
<div id="app">
  <h2>{{ producto.nombre }}</h2>
  <p>Precio: {{ producto.precio }} €</p>
</div>
```

Figura 9. Detalle de productos. Fuente: elaboración propia.

```
new Vue({
  el: '#app',
  data: {
    producto: {
      nombre: 'Camiseta Vue.js',
      precio: 19.99
    }
  }
});
```

Figura 10. Detalle de productos Vue.js. Fuente: elaboración propia.

En este ejemplo, producto.nombre y producto.precio están enlazados a la UI mediante interpolación. Si el modelo producto se actualiza, la UI reflejará automáticamente estos cambios.

Configuración del enlace bidireccional en Vue.js

El **enlace bidireccional** en Vue.js se logra utilizando la **directiva *v-model,*** que simplifica la vinculación de los campos de formulario a los datos en el modelo.

Supongamos que estamos desarrollando un formulario donde el usuario puede introducir su nombre. El formulario se configuraría de la siguiente manera:

```
<div id="app">
  <form>
    <label for="nombre">Nombre:</label>
    <input v-model="nombre" id="nombre">
    <p>Tu nombre es: {{ nombre }}</p>
  </form>
</div>
```

Figura 11. Formulario inserción nombre. Fuente: elaboración propia.

```
new Vue({
  el: '#app',
  data: {
    nombre: ''
  }
});
```

Figura 12. Inserción nombre Vue.js. Fuente: elaboración propia.

Aquí, la directiva *v-model* enlaza el campo de entrada directamente con la propiedad «nombre» en el modelo de datos. Esto asegura que cualquier entrada del usuario se refleje inmediatamente en la UI, y cualquier cambio en el modelo «nombre» se actualice en él.

4.4. Integración de eventos y uso de controladores

La integración de eventos y el uso de controladores son componentes fundamentales en el desarrollo de interfaces de usuario dinámicas e interactivas. Los **eventos** permiten que una **aplicación responda** a las **acciones del usuario,** como clics de botón, entradas de teclado, movimientos del ratón, entre otros. Los **controladores,** por su parte, son **funciones** o **métodos** que **gestionan** la **lógica** necesaria para **responder** a esos **eventos,** actualizando la interfaz de usuario o el modelo de datos según sea necesario. Este proceso es esencial para crear aplicaciones reactivas y fáciles de usar.

En el contexto de las interfaces de usuario, un **evento** es una **señal** que se genera en **respuesta** a una **acción** o **interacción** del **usuario** con la **aplicación.** Estos eventos pueden ser originados por varias fuentes, como componentes de la interfaz (botones, formularios, etc.), el sistema operativo o, incluso, por la propia lógica de la aplicación.

Algunos de los **eventos más comunes** en el desarrollo de interfaces de usuario incluyen:

- **Clics de ratón:** ocurre cuando el usuario presiona y suelta un botón del ratón sobre un elemento interactivo, como un botón o un enlace.
- **Entradas de teclado:** se generan cuando el usuario presiona o suelta una tecla mientras un campo de entrada o un componente similar está enfocado.
- **Cambios de valor:** asociados a campos de entrada, estos eventos se disparan cuando el contenido de un campo, como un cuadro de texto, cambia.
- **Eventos de carga:** ocurren cuando un documento, imagen u otro recurso se carga completamente en la interfaz de usuario.

- **Desplazamientos:** se generan cuando el usuario desplaza una vista, como una página web o una lista.

Estos eventos se manejan utilizando **controladores de eventos,** que son funciones específicas diseñadas para responder a la ocurrencia de estos eventos.

Integración de eventos en JavaScript

Los **controladores de eventos** son **funciones** o **métodos** que contienen la lógica que debe ejecutarse cuando ocurre un evento específico. En la mayoría de los *frameworks* de desarrollo de interfaces de usuario, los controladores de eventos **se asocian** directamente con los elementos de la interfaz a través de **atributos específicos** o **mediante programación.**

En JavaScript, los eventos se pueden manejar de manera sencilla utilizando métodos como addEventListener o asignando directamente un controlador de eventos a un elemento HTML.

Supongamos que tenemos un botón en una página HTML y queremos mostrar un mensaje en la consola cuando el usuario haga clic en el botón. El código JavaScript para manejar este evento sería:

```
<button id="miBoton">Haz clic aquí</button>

<script>
  // Seleccionamos el botón por su ID
  var boton = document.getElementById('miBoton');

  // Asignamos un controlador de eventos al evento 'click'
  boton.addEventListener('click', function() {
    console.log('El botón ha sido clicado');
  });
</script>
```

Figura 13. Código JavaScript mostrar datos por consola. Fuente: elaboración propia.

En este ejemplo, cuando el usuario hace clic en el botón con el ID «miBoton», se ejecuta el controlador de eventos que registra el mensaje «El botón ha sido clicado» en la consola.

Este enfoque es básico y funciona bien para aplicaciones pequeñas o donde se necesita un manejo directo de eventos.

Integración de eventos en Angular

Angular, como *framework* estructurado, maneja los eventos de una manera ligeramente diferente, proporcionando un **enfoque declarativo.** Los eventos se manejan directamente en la **plantilla HTML** utilizando la sintaxis (evento)="método()" , donde *evento* es el nombre del evento que se va a manejar y método es el controlador definido en el componente.

Consideremos un botón en una aplicación Angular que incrementa un contador cada vez que se hace clic en él:

```
<button (click)="incrementarContador()">Incrementar</button>
<p>Contador: {{ contador }}</p>
```

Figura 14. Botón Angular para aumentar contador. Fuente: elaboración propia.

El controlador de eventos incrementarContador se define en el componente TypeScript de la siguiente manera:

```
export class MiComponente {
  contador: number = 0;

  incrementarContador() {
    this.contador++;
  }
}
```

Figura 15. Componente TypeScript para incrementar contador. Fuente: elaboración propia.

Aquí, cada vez que el usuario hace clic en el botón, el método incrementarContador se ejecuta, incrementando el valor del contador. Este valor se muestra automáticamente en la interfaz de usuario gracias al enlace de datos unidireccional.

Angular facilita la gestión de eventos al permitir una clara **separación** entre la **lógica** del **controlador** y la **estructura** de la **interfaz** de usuario. Además, la sintaxis declarativa hace que sea más fácil de leer y mantener.

Integración de eventos en React

En React, la gestión de eventos es similar a la de JavaScript puro, pero con algunas diferencias clave debido al **enfoque basado en componentes** y al **uso del estado.** En lugar de utilizar addEventListener, React utiliza **atributos JSX** que se asemejan a los eventos HTML estándar, pero que están escritos en *camelCase.*

Supongamos que queremos manejar un evento de clic en un botón para cambiar el estado de un componente React:

```
class BotonContador extends React.Component {
  constructor(props) {
    super(props);
    this.state = { contador: 0 };
  }

  manejarClick = () => {
    this.setState({ contador: this.state.contador + 1 });
  }

  render() {
    return (
      <div>
        <button onClick={this.manejarClick}>Haz clic aquí</button>
        <p>Contador: {this.state.contador}</p>
      </div>
    );
  }
}
```

Figura 16. Cambio de estado de un componente React. Fuente: elaboración propia.

En este ejemplo, `onClick` es el atributo JSX que se utiliza para gestionar el evento de clic. El método `manejarClick` se define como un controlador de eventos, que actualiza el estado del componente utilizando `setState`. Cada vez que se actualiza el estado, React vuelve a renderizar el componente, lo que garantiza que la interfaz de usuario esté siempre sincronizada con el estado actual.

React es conocido por su modelo de **flujo de datos unidireccional,** lo que significa que el manejo de eventos se realiza de manera explícita y controlada, facilitando el seguimiento y la depuración del flujo de datos en la aplicación.

Integración de eventos en Vue.js

Vue.js ofrece una forma muy simple e intuitiva de manejar eventos directamente en las plantillas HTML mediante la **directiva *v-on,*** que también se puede escribir como *@.* Esto permite a los desarrolladores asignar **controladores de eventos a cualquier elemento HTML.**

Imaginemos que queremos manejar un evento de clic en Vue.js para mostrar un mensaje de alerta:

```
<div id="app">
  <button @click="mostrarAlerta">Haz clic aquí</button>
</div>
```

Figura 17. Evento clic en Vue.js. Fuente: elaboración propia.

```
new Vue({
  el: '#app',
  methods: {
    mostrarAlerta() {
      alert('¡El botón ha sido clicado!');
    }
  }
});
```

En este ejemplo, cuando el usuario hace clic en el botón, se dispara el evento `click`, y el método `mostrarAlerta` muestra una ventana de alerta. Vue.js simplifica la integración de eventos al hacer que el **código** sea muy **legible** y **fácil de mantener.**

4.5. Método event-bus para la comunicación entre componentes

El método *event-bus* es un **patrón de diseño** utilizado en el desarrollo de *software* para facilitar la **comunicación** entre **componentes** que no están directamente relacionados en la jerarquía de la aplicación. Este patrón es especialmente útil en **aplicaciones** de **gran tamaño** o **complejidad,** donde la necesidad de intercambiar información entre componentes que no comparten una relación padre-hijo se vuelve esencial. En este contexto, el *event-bus* actúa como un **intermediario** que permite que los componentes emitan y escuchen eventos de manera centralizada, evitando la necesidad de pasar datos a través de múltiples capas de la jerarquía de componentes.

Un *event-bus* es, en esencia, un **objeto centralizado** que facilita la **emisión** y **recepción** de **eventos** dentro de una aplicación. Los componentes pueden suscribirse a eventos específicos en el bus, y cuando esos eventos son emitidos por cualquier otro componente, los suscriptores son notificados y pueden responder en consecuencia. Esto permite una comunicación flexible y **desacoplada** entre diferentes partes de la aplicación.

El *event-bus* se puede implementar utilizando diferentes enfoques y herramientas dependiendo del *framework* o la biblioteca que se esté utilizando. A menudo, se implementa utilizando **sistemas de suscripción/publicación** *(pub/sub),* donde un componente se suscribe a ciertos eventos y otro los publica cuando es necesario.

Implementación del event-bus en Vue.js

Uno de los casos de uso común en Vue es la necesidad de **comunicar componentes** que **no** están **directamente relacionados.** En versiones anteriores de Vue.js, era común utilizar un objeto Vue vacío como *event-bus,* aunque en las versiones más recientes y con la llegada de Vue 3, se recomienda utilizar sistemas de gestión de estado como Vuex para estos casos más complejos. Sin embargo, el patrón de *event-bus* sigue siendo relevante para casos simples o específicos.

Supongamos que tenemos una aplicación en Vue.js donde un componente de botón emite un evento cuando se hace clic, y otro componente, en una parte diferente de la aplicación, necesita reaccionar a este clic.

Primero, podemos crear un nuevo objeto Vue que actuará como nuestro *event-bus:*

```
const eventBus = new Vue();
```

Figura 19. Nuevo objeto Vue que actúa como *event-bus*. Fuente: elaboración propia.

Luego, en el componente que emite el evento:

```
export default {
  methods: {
    emitirEvento() {
      eventBus.$emit('botonClicado', { mensaje: 'El botón fue clicado' });
    }
  },
  template: `
    <button @click="emitirEvento">Haz clic aquí</button>
  `
};
```

Figura 20. Componente que emite el evento. Fuente: elaboración propia.

En este ejemplo, cuando se hace clic en el botón, se emite un evento llamado botonClicado a través del *event-bus,* junto con un objeto que contiene un mensaje. Por otro lado, en un componente diferente que necesita escuchar este evento:

```
export default {
  created() {
    eventBus.$on('botonClicado', (data) => {
      console.log(data.mensaje);
    });
  },
  template: `
    <div>
      <p>Estoy esperando que el botón sea clicado...</p>
    </div>
  `
};
```

Figura 21. Componente que escucha el evento. Fuente: elaboración propia.

En este componente, utilizamos el ciclo de vida created para suscribirnos al evento botonClicado . Cuando el evento se emite, el mensaje se imprime en la consola.

Este enfoque permite que los dos componentes se comuniquen sin necesidad de una relación directa, lo que es, particularmente, útil en aplicaciones grandes o en aquellas donde la jerarquía de componentes es compleja.

Implementación del event-bus en Angular

Aunque Angular no tiene un mecanismo de *event-bus* integrado como Vue.js, se puede lograr una funcionalidad similar utilizando **servicios.** En Angular, los servicios se utilizan para compartir datos y lógica entre diferentes componentes, y un servicio puede actuar como un *event-bus* al utilizar RxJS, una biblioteca para la programación reactiva.

Supongamos que queremos implementar un *event-bus* simple en Angular para que varios componentes puedan reaccionar a un evento de cambio de configuración. Primero, creamos un servicio que actuará como nuestro *event-bus:*

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class EventBusService {
  private subject = new Subject<any>();

  emitirEvento(data: any) {
    this.subject.next(data);
  }

  obtenerEventos() {
    return this.subject.asObservable();
  }
}
```

Figura 22. Servicio que actúa como *event-bus*. Fuente: elaboración propia.

En este servicio, Subject es un tipo especial de observable de RxJS que permite emitir eventos y suscribirse a ellos.

Luego, en el componente que emite el evento:

```
import { Component } from '@angular/core';
import { EventBusService } from './event-bus.service';

@Component({
  selector: 'app-emisor',
  template: `<button (click)="cambiarConfiguracion()">Cambiar Configuración</button>`
})
export class EmisorComponent {
  constructor(private eventBusService: EventBusService) {}

  cambiarConfiguracion() {
    this.eventBusService.emitirEvento({ config: 'nuevaConfiguracion' });
  }
}
```

Figura 23. Componente que emite el evento. Fuente: elaboración propia.

Este componente emite un evento utilizando el servicio cuando se hace clic en el botón.

En un componente diferente que escucha este evento:

```
import { Component, OnInit } from '@angular/core';
import { EventBusService } from './event-bus.service';

@Component({
  selector: 'app-receptor',
  template: `<p>Configuración actual: {{ configuracion }}</p>`
})
export class ReceptorComponent implements OnInit {
  configuracion: string;

  constructor(private eventBusService: EventBusService) {}

  ngOnInit() {
    this.eventBusService.obtenerEventos().subscribe((data) => {
      this.configuracion = data.config;
    });
  }
}
```

Figura 24. Otro componente que escucha el evento. Fuente: elaboración propia.

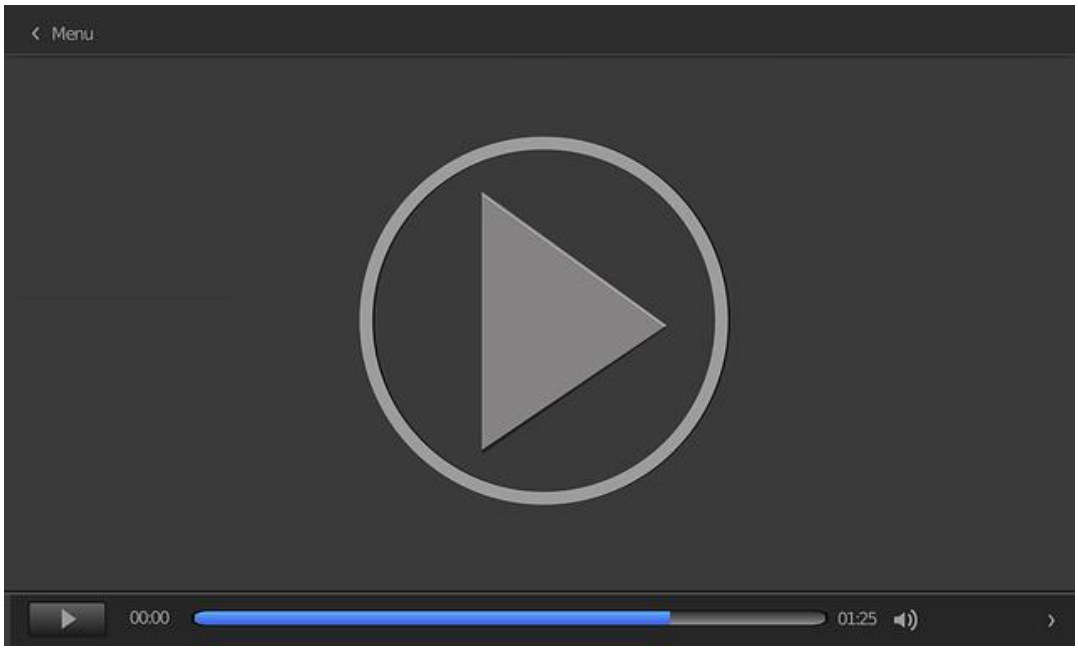
Este componente se suscribe al observable en el servicio y actualiza su vista cuando recibe un nuevo evento.

Este patrón es muy potente en Angular, ya que aprovecha la capacidad de RxJS para gestionar flujos de datos reactivos, lo que permite una gestión más avanzada y escalable de la comunicación entre componentes.

Uso de data binding en Angular

OpenWebinars. (2021, febrero 20). *Uso de Data-Binding en Angular* [Vídeo]. YouTube. https://www.youtube.com/watch?v=WSZGFcEb-q4

Este vídeo explica el *Data-Binding* de Angular, una de las características más importantes de este *framework.*



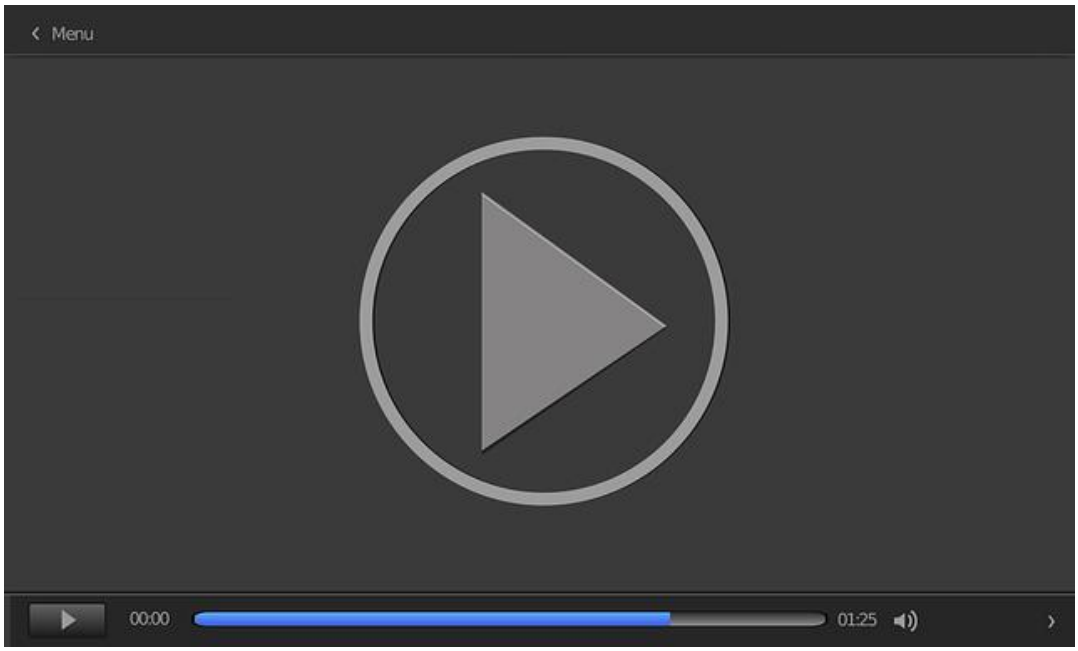
Accede al vídeo:

https://www.youtube.com/embed/WSZGFcEb-q4

Tutorial DOM JavaScript. Eventos con addEventListener

Code Hive. (2021, marzo 20). *Tutorial DOM Javascript | Eventos con addEventListener* [Vídeo]. YouTube. https://www.youtube.com/watch?v=TzhpyFmSioY

Este vídeo explica cómo utilizar los eventos en JavaScript para ejecutar funciones cuando sucede algo en el DOM.



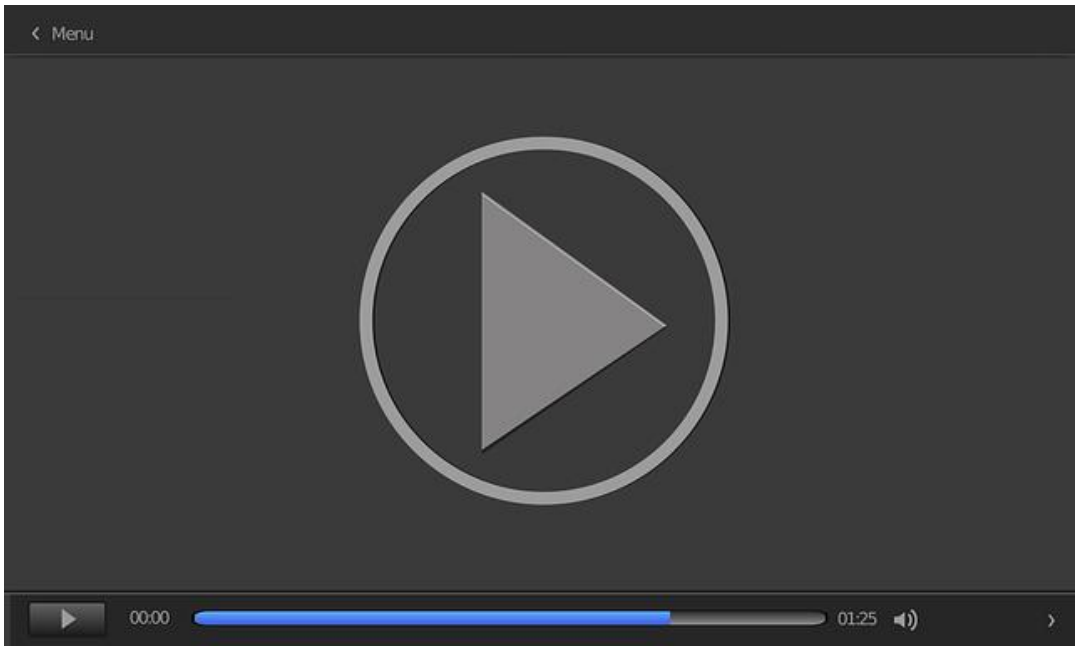
Accede al vídeo:

https://www.youtube.com/embed/TzhpyFmSioY

Cómo hacer un event-bus en Vue.js

Desarrollo web. (2021, julio 27). *Como hacer un EventBus en Vuejs* [Vídeo]. YouTube. https://www.youtube.com/watch?v=HoKSTDnHWCA

Este vídeo se explica cómo realizar un *event-bus* para mejorar la comunicación entre componentes.



Accede al vídeo:

https://www.youtube.com/embed/HoKSTDnHWCA

Entrenamiento 1

- Planteamiento del ejercicio

Crea una pequeña aplicación en Vue.js con dos componentes independientes: un componente de botón que emite un evento cuando se hace clic, y un componente de texto que muestra un mensaje cuando recibe el evento.

- Desarrollo paso a paso
- Crea un nuevo proyecto Vue.js o utilizar un proyecto existente.
- **Define el *event-bus:*** crea un archivo event-bus.js donde definirás un objeto Vue vacío que actuará como el *event-bus.*
- **Crea el primer componente («Botón»):** define un componente que emita un evento `botonClicado` cuando se haga clic en un botón.
- **Crea el segundo componente(«Texto»):** define otro componente que escuche el evento `botonClicado` y muestre un mensaje en la pantalla cuando se reciba.
- **Integra ambos componentes en una aplicación principal:** importa los componentes y el *event-bus* en la aplicación principal, y prueba la funcionalidad

- Solución

```
// event-bus.js

import Vue from 'vue';

export const eventBus = new Vue();
```

```
// BotonComponent.vue

<template>

  <button @click="emitirEvento">Haz clic aquí</button>

</template>



<script>

import { eventBus } from './event-bus';



export default {

  methods: {

    emitirEvento() {

      eventBus.$emit('botonClicado', '¡Botón clicado!');

    }

  }

};

</script>



// TextoComponent.vue

<template>

  <p>{{ mensaje }}</p>
```

```
</template>


<script>
import { eventBus } from './event-bus';


export default {
  data() {
    return {
      mensaje: ''
    };
  },
  created() {
    eventBus.$on('botonClicado', (mensaje) => {
      this.mensaje = mensaje;
    });
  }
};
</script>


// App.vue
```

```
<template>
  <div>
    <BotonComponent />
    <TextoComponent />
  </div>
</template>

<script>
import BotonComponent from './BotonComponent.vue';
import TextoComponent from './TextoComponent.vue';

export default {
  components: {
    BotonComponent,
    TextoComponent
  }
};
</script>
```

Entrenamiento 2

- Planteamiento del ejercicio

Crea una aplicación Vue.js con dos botones en componentes separados: uno para incrementar un contador y otro para disminuirlo. Muestra el valor del contador en un tercer componente.

- Desarrollo paso a paso

  - **Define el *event-bus:*** crea un archivo event-bus.js con un objeto Vue vacío.
  - **Crea el componente de «Incremento»:** define un componente que emita un evento incrementar cuando se haga clic en un botón.
  - **Crea el componente de «Decremento»:** define otro componente que emita un evento decrementar cuando se haga clic en un botón.
  - **Crea el componente de «Visualización del Contador»:** define un componente que escuche los eventos incrementar y decrementar y actualice el valor del contador en la UI.
  - Integrar los tres componentes en la aplicación principal.

- Solución

```
// event-bus.js

import Vue from 'vue';

export const eventBus = new Vue();
```

```
// IncrementoComponent.vue

<template>

  <button @click="incrementar">Incrementar</button>

</template>



<script>

import { eventBus } from './event-bus';



export default {

  methods: {

    incrementar() {

      eventBus.$emit('incrementar');

    }

  }

};

</script>



// DecrementoComponent.vue

<template>

  <button @click="decrementar">Decrementar</button>
```

```
</template>


<script>
import { eventBus } from './event-bus';


export default {
  methods: {
    decrementar() {
      eventBus.$emit('decrementar');
    }
  }
};
</script>


// ContadorComponent.vue
<template>
  <p>Contador: {{ contador }}</p>
</template>


<script>
```

```
import { eventBus } from './event-bus';


export default {
  data() {
    return {
      contador: 0
    };
  },
  created() {
    eventBus.$on('incrementar', () => {
      this.contador++;
    });
    eventBus.$on('decrementar', () => {
      this.contador--;
    });
  }
};
</script>


// App.vue
```

```
<template>
  <div>
    <IncrementoComponent />
    <DecrementoComponent />
    <ContadorComponent />
  </div>
</template>

<script>
import IncrementoComponent from './IncrementoComponent.vue';
import DecrementoComponent from './DecrementoComponent.vue';
import ContadorComponent from './ContadorComponent.vue';

export default {
  components: {
    IncrementoComponent,
    DecrementoComponent,
    ContadorComponent
  }
};
```

```
</script>
```

Entrenamiento 3

- Planteamiento del ejercicio

Crea una aplicación Vue.js donde un componente emite notificaciones (como mensajes de éxito o error), y otro componente muestra estas notificaciones globalmente en la interfaz.

- Desarrollo paso a paso

  - **Define el *event-bus:*** crea un archivo event-bus.js.
  - **Crea el componente «Emisor de Notificaciones»:** define un componente con botones para emitir diferentes tipos de notificaciones.
  - **Crea el componente «Receptor de Notificaciones»:** define un componente que escuche los eventos y muestre las notificaciones en la interfaz.
  - Integra ambos componentes en la aplicación principal.

- Solución

```
// event-bus.js

import Vue from 'vue';

export const eventBus = new Vue();


// IncrementoComponent.vue

<template>
```

```
  <button @click="incrementar">Incrementar</button>
</template>

<script>
import { eventBus } from './event-bus';

export default {
  methods: {
    incrementar() {
      eventBus.$emit('incrementar');
    }
  }
};
</script>

// DecrementoComponent.vue
<template>
  <button @click="decrementar">Decrementar</button>
</template>
```

```
<script>
import { eventBus } from './event-bus';

export default {
  methods: {
    decrementar() {
      eventBus.$emit('decrementar');
    }
  }
};
</script>

// ContadorComponent.vue
<template>
  <p>Contador: {{ contador }}</p>
</template>

<script>
import { eventBus } from './event-bus';
```

```
export default {
  data() {
    return {
      contador: 0
    };
  },
  created() {
    eventBus.$on('incrementar', () => {
      this.contador++;
    });
    eventBus.$on('decrementar', () => {
      this.contador--;
    });
  }
};
</script>

// App.vue
<template>
  <div>
```

```
    <IncrementoComponent />
    <DecrementoComponent />
    <ContadorComponent />
  </div>
</template>

<script>
import IncrementoComponent from './IncrementoComponent.vue';
import DecrementoComponent from './DecrementoComponent.vue';
import ContadorComponent from './ContadorComponent.vue';

export default {
  components: {
    IncrementoComponent,
    DecrementoComponent,
    ContadorComponent
  }
};
</script>
```

Entrenamiento 4

- Planteamiento del ejercicio

Crea una aplicación Vue.js donde un componente de formulario permite al usuario introducir datos y enviarlos, y otro componente muestra los datos enviados en tiempo real.

- Desarrollo paso a paso

  - **Define el *event-bus:*** crea un archivo event-bus.js.
  - **Crea el componente de «Formulario»:** define un formulario donde el usuario pueda introducir un nombre y una dirección.
  - **Crea el componente de «Visualización de Datos»:** define un componente que escuche los datos enviados y los muestre en la interfaz.
  - Integrar ambos componentes en la aplicación principal.

- Solución

```
// event-bus.js

import Vue from 'vue';

export const eventBus = new Vue();


// FormularioComponent.vue

<template>
```

```
  <form @submit.prevent="enviarDatos">
    <input v-model="nombre" placeholder="Nombre">
    <input v-model="direccion" placeholder="Dirección">
    <button type="submit">Enviar</button>
  </form>
</template>

<script>
import { eventBus } from './event-bus';

export default {
  data() {
    return {
      nombre: '',
      direccion: ''
    };
  },
  methods: {
    enviarDatos() {
      eventBus.$emit('datosEnviados', {
```

```
        nombre: this.nombre,
        direccion: this.direccion
      });
    }
  }
};
</script>

// DatosComponent.vue
<template>
  <div>
    <p>Nombre: {{ nombre }}</p>
    <p>Dirección: {{ direccion }}</p>
  </div>
</template>

<script>
import { eventBus } from './event-bus';

export default {
```

```
  data() {
    return {
      nombre: '',
      direccion: ''
    };
  },
  created() {
    eventBus.$on('datosEnviados', (data) => {
      this.nombre = data.nombre;
      this.direccion = data.direccion;
    });
  }
};
</script>

// App.vue
<template>
  <div>
    <FormularioComponent />
    <DatosComponent />
```

```
  </div>
</template>


<script>
import FormularioComponent from './FormularioComponent.vue';
import DatosComponent from './DatosComponent.vue';


export default {
  components: {
    FormularioComponent,
    DatosComponent
  }
};
</script>
```

Entrenamiento 5

- Planteamiento del ejercicio

Crea una aplicación Vue.js donde un componente simula un proceso de inicio de sesión y otro componente muestra el estado de autenticación (conectado o desconectado) en la interfaz.

- Desarrollo paso a paso

  - Crea un archivo event-bus.js.
  - **Crea el componente de «Inicio de Sesión»:** define un componente con botones para iniciar y cerrar sesión, emitiendo eventos correspondientes.
  - **Crea el componente de «Estado de Autenticación»:** define un componente que escuche los eventos de inicio y cierre de sesión y muestre el estado actual.
  - Integra ambos componentes en la aplicación principal.

- Solución

```
// event-bus.js

import Vue from 'vue';

export const eventBus = new Vue();


// LoginComponent.vue

<template>
```

```
  <div>
    <button @click="iniciarSesion">Iniciar Sesión</button>
    <button @click="cerrarSesion">Cerrar Sesión</button>
  </div>
</template>

<script>
import { eventBus } from './event-bus';

export default {
  methods: {
    iniciarSesion() {
      eventBus.$emit('estadoAutenticacion', true);
    },
    cerrarSesion() {
      eventBus.$emit('estadoAutenticacion', false);
    }
  }
};
</script>
```

```
// EstadoAutenticacionComponent.vue

<template>

  <p>{{ estadoAutenticacion ? 'Conectado' : 'Desconectado' }}</p>

</template>



<script>

import { eventBus } from './event-bus';



export default {

  data() {

    return {

      estadoAutenticacion: false

    };

  },

  created() {

    eventBus.$on('estadoAutenticacion', (estado) => {

      this.estadoAutenticacion = estado;

    });

  }
```

```
};

</script>


// App.vue

<template>

  <div>

    <LoginComponent />

    <EstadoAutenticacionComponent />

  </div>

</template>


<script>

import LoginComponent from './LoginComponent.vue';

import EstadoAutenticacionComponent from
'./EstadoAutenticacionComponent.vue';


export default {

  components: {

    LoginComponent,

    EstadoAutenticacionComponent

  }
```

```
};

</script>
```