MANUAL VIXIA ALGAS

FRONT-END

F. Javier García González

Este manual está diseñado para ayudar a los desarrolladores a entender y trabajar con el proyecto frontend-vite. Aquí encontrarás información sobre la estructura del proyecto, componentes clave, prácticas recomendadas y cómo contribuir al desarrollo.

Índice

Sección 1: Introducción

1.1 Propósito del proyecto y del Frontend

Este proyecto presenta un sistema integral de monitorización y gestión de fotobiorreactores de microalgas. Su objetivo principal es proporcionar una plataforma robusta y en tiempo real para la visualización y análisis de datos críticos recolectados por sensores conectados a un Arduino, que mide parámetros clave para el crecimiento y bienestar de las microalgas.

El frontend de esta aplicación es una Single Page Application (SPA) intuitiva y dinámica, diseñada para ofrecer una experiencia de usuario fluida. Su propósito central es:

- Visualizar datos en tiempo real: Mostrar de forma instantánea y legible las mediciones de los sensores (temperatura, pH, longitud de onda del crecimiento, color del agua, número de células) directamente desde el Arduino.
- Permitir la interacción y control: Ofrecer funcionalidades para que el usuario pueda interactuar con el sistema, como controlar el estado del Arduino, solicitar tomas de muestras manuales, y configurar parámetros del sistema (ej. estado de las luces).
- Facilitar el análisis histórico: Proporcionar herramientas para consultar y comparar datos de mediciones pasadas, permitiendo un seguimiento del rendimiento del fotobiorreactor a lo largo del tiempo.
- Mejorar la toma de decisiones: Convertir datos crudos en información útil y actionable para los investigadores o técnicos que gestionan los cultivos de microalgas.

En resumen, el frontend es la ventana interactiva que permite a los usuarios monitorizar, analizar y gestionar el entorno de las microalgas, transformando datos complejos en una interfaz accesible y funcional

1.2 Tecnología Principales Utilizadas en el Frontend

El desarrollo del frontend de este proyecto se ha llevado a cabo utilizando un stack de tecnologías eficientes, seleccionadas por su rendimiento y escalabilidad:

- **Vite:** Un *bundler* de JavaScript de última generación que proporciona un entorno de desarrollo extremadamente rápido, optimizando la experiencia de desarrollo.
- **React:** Una librería de JavaScript declarativa y basada en componentes para construir interfaces de usuario. React facilita la creación de UIs complejas y reutilizables, lo que contribuye a la modularidad y mantenibilidad del código.
- **TypeScript:** Un supertype de JavaScript que añade tipado estático. Su uso en este proyecto mejora la calidad del código, facilita la detección temprana de errores, y proporciona una mayor claridad y documentación implícita sobre las estructuras de datos y las interfaces.
- **Nivo.rocks:** Una colección rica y configurable de componentes de gráficos para React. Nivo se utiliza para la representación visual de los datos de los sensores (ej., el gráfico de longitud de onda), ofreciendo gráficos interactivos y estéticamente agradables.
- **React-Toastify:** Una librería popular para mostrar notificaciones (o "toasts") en la interfaz de usuario. Se utiliza para proporcionar información al usuario sobre el estado de las operaciones (éxito, error, advertencia) y la conexión con el servidor.

1.3 Interacción con el backend

El frontend interactúa con un servidor backend desarrollado en Python con Flask, estableciendo una comunicación bidireccional y eficiente a través de dos mecanismos complementarios:

- **Peticiones HTTP:** Se utilizan para operaciones estándar de solicitud-respuesta, como la obtención de datos históricos de los sensores, la recuperación y el envío de configuraciones, y la gestión de la autenticación de usuarios. Estas peticiones se realizan de forma asíncrona para no bloquear la interfaz de usuario.
- **WebSockets (mediante Socket.IO):** Para la comunicación en tiempo real, la aplicación establece una conexión WebSocket persistente. Esto permite que el backend envíe actualizaciones instantáneas de los datos de los sensores (temperatura, pH, estado de las luces, etc.) al frontend tan pronto como están disponibles, sin necesidad de que el cliente realice solicitudes constantes. Esta funcionalidad es crucial para la monitorización en vivo del fotobiorreactor.

Esta combinación de HTTP y WebSockets asegura que la aplicación pueda manejar tanto la transferencia de datos bajo demanda como la actualización en tiempo real, optimizando el rendimiento y la capacidad de respuesta del sistema.

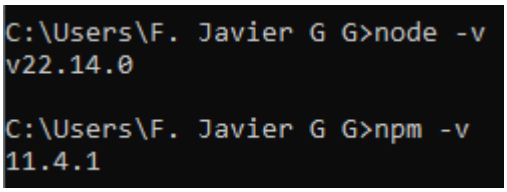
Sección 2: Configuración del Entorno de Desarrollo

Esta sección detalla los pasos necesarios para configurar el entorno de desarrollo y poner en marcha el frontend de la aplicación. Se cubren los requisitos previos, la instalación de las dependencias del proyecto y la configuración de las variables de entorno.

2.1. Requisitos Previos

Antes de poder iniciar el desarrollo, asegúrate de tener instaladas las siguientes herramientas en tu sistema:

- Node.js y npm (Node Package Manager):
  - Versión recomendada: Node.js v18.x (LTS - Long Term Support) o superior. La usada durante el desarrollo fue la v22.14.0
  - Verificación: Puedes comprobar si Node.js y npm están instalados y sus versiones abriendo una terminal o línea de comandos y ejecutando:



Instalar en caso de no tenerlo.

Sección 3: Estructura del Proyecto

Esta sección describe la organización de los directorios y archivos del frontend de la aplicación. Comprender esta estructura es esencial para navegar por el código, localizar funcionalidades específicas y contribuir al proyecto de manera eficiente.

El proyecto frontend-vite sigue una estructura estándar para aplicaciones React con Vite y TypeScript, con una clara separación de responsabilidades:

```
frontend-vite/
├── dist/
├── node_modules/
├── public/
├── src/
│   ├── assets/
│   ├── components/
│   ├── hooks/
│   ├── interface/
│   ├── pages/
│   ├── styles/
│   ├── App.tsx
│   ├── main.tsx
│   └── vite-env.d.ts
├── .env
├── .gitignore
├── eslint.config.js
├── index.html
├── nohup.out
├── package-lock.json
├── package.json
├── README.md
├── tsconfig.app.json
├── tsconfig.json
├── tsconfig.node.json
└── vite.config.ts
```

3.1. Resumen de Directorios y Archivos Clave

A continuación, se detalla el propósito de cada directorio y archivo principal:

- **frontend-vite/ (Directorio Raíz del Proyecto):**
  - Contiene todos los archivos y directorios del proyecto frontend.
- **dist/:**
  - Este directorio se genera automáticamente cuando se compila el proyecto para producción (npm run build). Contiene la versión optimizada lista para desplegar de la aplicación.
- **node_modules/:**
  - Contiene todas las dependencias y librerías de Node.js instaladas por npm, no debe modificarse manualmente.
- **public/:**
  - Almacena archivos estáticos que se sirven directamente en la raíz del servidor web (ej., icono web de la aplicación).
- **src/ (Source - Código Fuente):**
  - **Este es el directorio principal que contiene todo el código fuente de la aplicación React.**
  - **src/assets/:**

        - Contiene archivos de recursos como imágenes.
    - **src/components/:**
        - Aloja todos los componentes reutilizables de React. Se organiza en subdirectorios para agrupar componentes relacionados por funcionalidad o tipo (ej. ArduinoController).
    - **src/hooks/:**
        - Contiene los *custom hooks* de React. Estos hooks encapsulan lógicas con estado reutilizables y son fundamentales para compartir funcionalidades entre componentes sin duplicar código (ej., useWebSocketLastData).
    - **src/interface/:**
        - Almacena las definiciones de interfaces y tipos de TypeScript. Esto es crucial para mantener un código fuertemente tipado, proporcionando claridad sobre las estructuras de datos, especialmente las que se intercambian con el backend (Global_Interface.ts).
    - **src/pages/:**
        - Contiene los componentes de nivel superior que representan las diferentes "páginas" o "vistas" de la aplicación (ej., Index.tsx).
    - **src/styles/:**
        - Contiene los archivos de hojas de estilo (ej., index.css). Aquí se definen los estilos globales de la aplicación, incluyendo las directivas y configuraciones para *Tailwind* CSS.
    - **src/App.tsx:**
        - El componente raíz principal de la aplicación. Define la estructura general de la interfaz de usuario y gestiona el enrutamiento a las diferentes páginas. Es donde se inicializa el hook de WebSocket y se distribuyen los datos globales a los componentes hijos.
    - **src/main.tsx:**
        - El punto de entrada principal de la aplicación React. Aquí se importa el componente App.tsx y se renderiza en el elemento root del index.html.
    - **src/vite-env.d.ts:**
        - Archivo de declaración de tipos generada automáticamente por Vite.
- **.env:**
    - Archivo de configuración de las variables de entorno del proyecto (ej., URLs del backend, claves de API). Es vital que este archivo **no sea versionado** (excluido por .gitignore) para proteger información sensible y permitir configuraciones específicas para cada entorno de despliegue.
- **.gitignore:**
    - Un archivo de texto que indica a *Git* qué archivos y directorios debe ignorar y no incluir en el control de versiones (ej. .env).
- **eslint.config.js:**
    - Archivo de configuración para ESLint, una herramienta de *linting* que ayuda a mantener la calidad y consistencia del código JavaScript/TypeScript, identificando problemas y errores de estilo.
- **index.html:**
    - La plantilla HTML principal de la aplicación. Es el único archivo HTML servido directamente por el navegador y contiene el donde se monta la aplicación React.
- **nohup.out:**
    - Este archivo es un *log* de salida que se genera cuando se ejecuta un proceso en Linux/Unix usando nohup, lo que permite que el proceso continúe ejecutándose incluso si el usuario cierra la sesión de la terminal. Indica que el backend (o quizás un proceso de construcción/servidor) puede estar siendo ejecutado de esta manera.

- **package.json:**
  - El manifiesto del proyecto. Contiene metadatos (nombre, versión), lista todas las dependencias del proyecto y define los *scripts* de npm
- **package-lock.json:**
  - (Generado automáticamente por npm) Registra las versiones exactas y las dependencias anidadas de todos los paquetes instalados. Esto asegura que cualquier persona que instale el proyecto obtenga exactamente las mismas versiones de las librerías, garantizando la consistencia del entorno.
- **README.md:**
  - El archivo de documentación principal del proyecto. Generalmente contiene una descripción breve, instrucciones de instalación y uso rápido.
- **vite.config.ts:**
  - El archivo de configuración principal para Vite. Aquí se configuran los *plugins* de React, las opciones de *bundling*, los alias de rutas, el servidor de desarrollo, y otras configuraciones específicas de Vite.

Sección 4: Componentes Principales

Esta sección describe los componentes clave del frontend, explicando su propósito y función dentro de la aplicación. Estos componentes, ubicados principalmente en src/pages y src/components, son los pilares de la interfaz de usuario y la interacción con el sistema.

4.1 Componentes de Gráficos (Nivo)

- **Ubicación:** src/components/nivo
- **Propósito:** Estos componentes son responsables de la **visualización interactiva de los datos** históricos y en tiempo real utilizando la librería Nivo.
- **Props Clave:** Reciben los datos a graficar, que tendrán una estructura diferente según el componente que se va a usar
- **Interacción con el Backend:** No interactúan directamente. Los datos provienen de peticiones HTTP (para datos históricos) o de WebSockets (para datos en vivo), que son pasados a estos componentes como props.

4.1.2 Ejemplo de uso

¿Lo primero es que componente vas a usar? ¿Un NivoLine? ¿Un ScatterPlot? Esto es crucial ya que cada uno puede tener una estructura de datos diferente. Puedes visitar https://nivo.rocks/components/ y ver en la documentación de cada componente su estructura, pero no solo eso también puedes cambiar cómo se comporta colores y estilos.

Vamos hacer un ejemplo con la gráfica usada en la vista principal. Lo primero sería la instalación del componente, necesitas bajarte su librería, esto solo es necesario hacerlo una vez por componente

npm i @nivo/line

Una vez lo tengas desde la propia página de nivo puedes configurar el componente, que colores usará y todo eso. Lo importante es usar el import y retornear el componente

También debes tener claro la estructura q vas a usar con el componente. En este caso es;

```
Array<{
  id:   string | number
  data: Array<{
    x: number | string | Date
    y: number | string | Date
  }>
}>
```

Es decir, es un array de un id y un array de datos x, y. Nivo puede mostrar un solo grafico, pero debe ser un array el dato pasado en su parámetro *data*

```
import { ResponsiveLine } from '@nivo/line'

return (<div className="w-full h-full p-4"> {/* altura fija */}
      <ResponsiveLine
        markers={}// esto es la línea roja que cruza la gráfica, se usa para marcar el punto
        data={serie} // el array de datos
        margin={{ top: 30, right: 25, bottom: 25, left: 35 }}
        xScale={{ type: 'linear', min: 'auto', max: 'auto' }}
        yScale={{ type: 'linear', min: 'auto', max: 'auto' }}
        curve="monotoneX"
        colors={['#008b1e']}
        axisTop={null}
        axisRight={null}
        enablePoints={false}
        axisBottom={{
          tickSize: 5,
          tickPadding: 5,
          tickRotation: 0,
          legend: 'Rango Espectro',
          legendOffset: 36,
          legendPosition: 'middle',
        // esto vale para modificar el eje x, en vez mostar números del 1 a 280(numero de posiciones del
        array, muestra el valor de la longitud de onda
          format: (value) => {
            if (data?.x && data.x[Number(value) - 1] !== undefined) {
              return data.x[Number(value) - 1];
            }
            return '';
          },
        }}
        axisLeft={{
          tickSize: 5,
          tickPadding: 5,
          tickRotation: 0,
          legend: 'Valor',
          legendOffset: -40,
          legendPosition: 'start',
        }}
        pointSize={6}
        useMesh={true}
        enableGridX={true}
        enableGridY={true}
        tooltip={({ point }) => (

        //Aquí solo se cambia la tooltip para personalizarla!

      />
    </div>)
```

Ahora la estructura de datos el archivo; Global_Interface.ts es donde definimos nuestras estructuras de datos. Por ejemplo, el backend, cada vez que envía datos de las mediciones.

```
export interface SampleData {
   datetime: string,
   colors: {
      blue: boolean;
      red: boolean;
      white: boolean;
   };
   data: {
      ph: number;
      temperature: number;
   };
   rgb: {
      b: number;
      g: number;
      r: number;
   };
   wave_length: number[];
   x: number[]
   nc: number
}
```

Esto quiere decir que los datos deben tener esa estructura y los nombres son exactamente iguales a los del json recibido. Un json se puede dividir entre objetos y arrays los objetos, que puede tener propiedades y se encapsulan con {} y los arrays con [] con esto se pueden generar estructuras como la siguiente

```
{  ejemplo de datos recibidos del back

   "datetime": "2025-06-12 10:37:05", un string para la fecha

   "colors": null, ya no se usa

   "rgb": { un objeto compuesto de 3 propiedades tipo number

      "r": 1.523121387283237,

      "g": 95.10345191597172,

      "b": 13.389541088580577

   },

   "data": {

      "temperature": 24.1795,

      "ph": 5.7165

   },

   "wave_length": [ un array con la longitud de honda (number)

   ],

   "nc": 9182630 un number

}
```

Una vez ya se tiene la estrechura se puede empezar a pasar los datos desde *PageIndex.tsx*

```
import Nivo_ResponsiveLine from '../components/nivo/Nivo_ResponsiveLine_Index';

import type { SampleData } from '../../interface/Global_Interface';
```

debemos importar el archivo en el que está el componente

```
interface IndexProps {
   data: SampleData | null;
   isManual: boolean | null;
   isWake: boolean | null;
   lightsState: LightsState | null;
   isConnected: boolean;
}
```

Esto es una interfaz interna, se usará para los datos que vengan de props. Nos centraremos en data.

```
export default function PageIndex({ data, isManual, isWake, lightsState, isConnected }: IndexProps) {
   //Como se puede ver, los datos de la interfaz son los que se usan para las props y finalmente le pasamos el tipado de la interfaz

// mucho html

            <Nivo_ResponsiveLine data={data} /> // se le pasan las props a la grafica

// más html

}
```

Ahora ya desde la grafica se hace lo mismo

```
import type { SampleData } from '../../interface/Global_Interface';
interface NivoLineProps {
   data?: SampleData | null;
}

export default function NivoLine({  data }: NivoLineProps) {

// Aquí ahora se hace un paso extra, para ahorra uso de interfaces extra etc se combierte la estructura de datos en una serie
const serie = []
   if (data != null) {
      const last_data = {
         id: 'Last Data',
         data: data?.wave_length?.map((value, index) => ({ x: index +1, y: value })) || [],
      }

      serie.push(last_data)
   }
//esto básicamente lo que hace es crear un array con los datos recibidos, poniendo un id y los valores x, y

//también es buena practica, esto asegura que si no hay datos por lo que sea muestre un mensaje
if (serie.length == 0) {
      return (<div className="w-full h-full p-4"> {/* altura fija */}
         <div className="flex flex-col items-center justify-center h-full w-full">
            <p className="text-center text-red-500">No hay datos para mostrar</p>
         </div>
      </div>)
   }

return <ResponsiveLine
         data={serie}
/>


} Y nada más.
```

4.2. PanelInfo.tsx

- **Ubicación:** src/components/PanelInfo.tsx
- **Propósito:** Este componente es el encargado de mostrar los **datos recibidos de las mediciones, Ph, temp, número de células y el rgb**.
- **Props Clave: titulo**, para que sea más personalizable y **sampleData**, que es la estructura vista anteriormente
- **Interacción con el Backend:** No realiza peticiones directas, sino que consume los datos de los WebSockets y peticiones HTTP que le llegan a través de sus props.
- **Relación con otros componentes:** Es un componente hijo de index y compare
- **Ejemplo de uso;** `<PanelInfo sampleData={data} titulo='Ultimos Datos'></PanelInfo>`

4.3. ArduinoController

- **Ubicación:** src/components/ArduinoController.tsx
- **Propósito:** Este componente es el encargado de **manipular estados del Arduino, tomar muestras manuales, modificar los parámetros de medición etc.**
- **Props Clave:** dadas por el WebSocket, principalmente para saber si hay conexión, si está tomando una medida, si esta *Wake* o no
- **Interacción con el Backend:** Se comunica tanto por HTTP para las acciones de los botones, como por websocket para interacción con usuario, si hay conexión, si está midiendo etc.
- **Relación con otros componentes:** Es un componente hijo de Index.tsx
- **Ejemplo de uso;** `<ArduinoController isManual={isManual} isWake={isWake} datetime={data?.datetime ?? null} isConnected={isConnected} />`

4.4 Heather

- **Ubicación:** src/components/Heather.tsx
- **Propósito:** Ser el heather de la app**.**
- **Props Clave:** nav con botón de acceso a cada vista controlada por una bool.
- **Relación con otros componentes:** Es un componente hijo deApp.tsx
- **Ejemplo de uso;**

```
let headerProps = {
    texto: 'Principal',
    showCompare: true,
    showSensores: true,
    showPrediciones: true,
    showBack: true,
    isConnected: isConnected
  };
  switch (location.pathname) {
    case '/':
      headerProps = {
        texto: 'Principal',
        showCompare: true,
        showSensores: true,
        showPrediciones: true,
        showBack: false,
        isConnected: isConnected
      };
      break;
// otras paginas
<Heather {...headerProps}></Heather>
```

4.3. Calendar

- **Ubicación:** src/components/Callendar.tsx
- **Propósito:** Este componente es el encargado de solicitar los datos seleccionados y borrarlos**.**
- **Props Clave: setDatos**, que pasa de hijo a padre
- **Interacción con el Backend:** Se comunica tanto por HTTP.
- **Relación con otros componentes:** Es un componente hijo de compare.tsx
- **Ejemplo de uso;** `<DatePickerWithData setDatos={setDatos} setData={setData} />`

4.4 Diversos UI

Se utilizan diversos ui para informar del tema de la aplicación, el switch de encender apagar, un elemento de carga de las peticiones etc uno para cerrar el formulario modal.

Sección 5: Comunicación con el Backend

El frontend de la aplicación se comunica con el servidor backend (Python Flask) a través de dos mecanismos principales:

- **Peticiones HTTP:** Utilizadas para operaciones de solicitud/respuesta, como obtener datos históricos, configuraciones, exportar mediciones, etc.
- **WebSockets:** Empleados para la comunicación en tiempo real, permitiendo la actualización instantánea de los datos de los sensores y el estado de las mediciones sin necesidad de recargar la página.

  El socket usado es el de la librería socket.io, y en el caso del frontend se usará el *socket.io-cliet.*

A continuación, se detalla la implementación de la comunicación mediante WebSockets y algunos tutoriales básicos de uso y nuevas implementaciones. Las peticiones HTTP se cubrirán en la siguiente subsección.

5.1. Comunicación en Tiempo Real (WebSockets)

La comunicación en tiempo real con el backend se gestiona a través de un *custom hook* de React llamado `useWebSocketLastData`. Este hook encapsula toda la lógica de conexión y manejo de eventos del WebSocket, haciendo que su consumo en los componentes sea sencillo y declarativo.

5.1.1. Configuración de la URL del WebSocket

La URL del servidor WebSocket se define en el archivo de variables de entorno *.env* para facilitar su configuración en diferentes entornos (desarrollo, producción).

```
PUBLIC_API_HOST=000.000.000.000 <- Ip del back
PUBLIC_API_PORT=5000
VITE_API_URL=http://${PUBLIC_API_HOST}:${PUBLIC_API_PORT}
```

5.1.2. El useWebSocketLastData Custom Hook

Este hook (ubicado en src/hooks/WebSockect_lasData.ts) es el encargado de establecer y mantener la conexión WebSocket, así como de procesar los mensajes recibidos del servidor.

Funcionalidad Principal:

- **Conexión y Reconexión:** Utiliza la librería socket.io-client para establecer la conexión. Incluye configuración de `timeout` y `reconnectionDelay` para manejar la estabilidad de la conexión.
- **Gestión del Estado de la Conexión:** Mantiene un estado `isConnected` que indica si el frontend está actualmente conectado al servidor WebSocket.
- **Recepción de Datos en Tiempo Real:** Escucha diversos eventos emitidos por el servidor, actualizando los estados locales con la información más reciente.
- **Notificaciones al Usuario:** Utiliza `react-toastify` para mostrar mensajes informativos (éxito, error, advertencia) sobre el estado de la conexión y la recepción de datos.
- **Control de Conexión/Desconexión:** Expone funciones *connect()* y *disconnect()* para gestionar manualmente el ciclo de vida del socket, aunque por defecto se conecta al montar el componente que lo usa y se desconecta al desmontar.

Eventos Escuchados desde el Servidor:

- **connect**: Se dispara cuando la conexión al servidor se establece exitosamente.
- **connect_error**: Se dispara si ocurre un error durante el intento de conexión.
- **connect_timeout**: Se dispara si la conexión excede el tiempo de espera configurado.
- **disconnect**: Se dispara cuando la conexión se cierra, indicando el motivo.
- **arduino_data**: Recibe los últimos datos recolectados por el Arduino (pH, temperatura, longitud de onda, etc.).
- **lights_state**: Recibe el estado actual de las luces del Arduino.
- **manual_mode**: Informa si el Arduino está en modo de toma de muestra manual.
- **wake_up_state**: Informa si el Arduino está en modo "despierto" (guardando y recibiendo datos) o "dormido".
- **onreboot**: Evento que indica un reinicio del sistema (usado para fines de depuración o información).

Si quisiéramos añadir nuevos eventos, sería muy fácil. Primero hay que plantear la estructura de los datos. ¿Voy trabajar con un booleano? ¿Con un int? ¿Algo más complejo como un JSON?

```
export interface UseWebSocketLastDataResult {
   lightsState: LightsState | null;
   data: SampleData | null;
   isManual: boolean | null;
   isWake: boolean | null; //<- Nuevo evento!
   socket: Socket | null;
   isConnected: boolean;
   error: Error | null;
   connect: () => void;
   disconnect: () => void;
}
```

Como estamos trabajando con TypeScript los datos deben ser tipados. Supongamos que *isWake* es nuestro nuevo evento para añadir, debemos añadirlo en su interfaz, la del ***useWebSocketLastData*.** Lo siguiente es añadir los estados, que será los que devolvamos a los componentes. En caso de ser algo mas complejo se debería usar una interfaz y exportarla

```
const [isWake, setWake] = useState<boolean | null>(null);
```

y ahora solo debemos añadir el método, utilizando la instancia del objeto

```
newSocket.on('wake_up_state', (wake: boolean) => {
            console.log('Wake up state:', wake);
            setWake(wake);
          });
```

El primer parámetro debe ser igual al configurado en el back, ahora solo queda devolverlo en el return final añadimos nuestro el usestate de nuestro nuevo evento, concretamente el get. Y ya estará listo.

```
 return {

      lightsState,
      data,
      isManual,
      isWake, //<- Nuevo evento!
      socket,
      isConnected, error, connect, disconnect,};
```

5.1.3. Uso del WebSocket en la Aplicación (App.tsx)

El hook *useWebSocketLastData* se inicializa y se consume en el componente raíz de la aplicación, **App.tsx**. Esto asegura que la conexión WebSocket se establezca tan pronto como la aplicación se carga y que los datos en tiempo real estén disponibles globalmente para los componentes hijos que los necesiten.

Ejemplo de uso en **App.tsx**

```
const { data, isManual, isWake, lightsState, isConnected } = WebSocket import.meta.env.VITE_API_URL
```

Esta declaración cuenta con; eventos sacados del websocket, declaración del import, variable de entorno del .env

Luego estos parámetros se pasan a las páginas que vayan a hacer uso de esta información

```
<Route path='/' element={
      <Index data={data} isManual={isManual} isWake={isWake} lightsState={lightsState}
isConnected={isConnected} />
```

Por ejemplo, a la página principal pasa *data*, que corresponde con la última medición de la base de datos, *isConnected* que nos ayuda a saber si hay conexión con el servidor, de este modo se puede bloquear controles si no hay conexión. Y otros.

5.2. Peticiones HTTP

Esta sección describe cómo el frontend interactúa con el servidor backend Flask para realizar operaciones de solicitud/respuesta que no requieren comunicación en tiempo real. Estas incluyen la obtención de datos históricos, la gestión de configuraciones, autenticación de usuarios, exportación de datos, etc.

5.2.1. Definición de Endpoints API

Todas las URLs de los *endpoints* HTTP del backend se definen y centralizan en el archivo de variables de entorno .env de Vite. Esto proporciona una configuración flexible y fácil de gestionar para diferentes entornos de despliegue. Aquí algunos ejemplos

```
PUBLIC_API_HOST=000.000.000.000 <- Ip del back
PUBLIC_API_PORT=5000
VITE_API_URL=http://${PUBLIC_API_HOST}:${PUBLIC_API_PORT}

VITE_LOGIN_URL=${VITE_API_URL}/login
VITE_REGISTER_URL=${VITE_API_URL}/register
VITE_DATA=${VITE_API_URL}/data
```

Si quisiéramos añadir nuevas rutas HTTP existentes en back solo habría que hacer esto

```
VITE_GET_HOURS=${VITE_API_URL}/get_hours
VITE_GET_COMPARASION=${VITE_API_URL}/get_comparation
```

Es importante que el nombre empieza por VITE_* de este modo se podrá acceder a la variable de entorno, el nombre de la variable que contiene la Ip host y el nombre de la ruta, que debe ser igual a la del back

*5.2.2. Implementación de las Peticiones con **fetch***

En el frontend, las peticiones HTTP se realizan utilizando la API nativa fetch de JavaScript. Este enfoque permite una comunicación eficiente con el backend para realizar operaciones como obtener datos, enviar configuraciones, o activar acciones específicas en el servidor.
A continuación, se explica cómo se implementan las peticiones fetch en el proyecto, con ejemplos prácticos y explicaciones detalladas.

Manejo General de Peticiones **fetch**

1. **Definición de URLs**:
   - Las URLs de los endpoints se definen en las variables de entorno de Vite (import.meta.env.VITE_NOMBRE_ENDPOINT). Esto permite una configuración flexible para diferentes entornos (desarrollo, producción, etc.).
2. **Métodos HTTP**:
   - Se especifica explícitamente el método HTTP (GET, POST, DELETE, etc.) según la operación que se desea realizar.
3. **Cabeceras (Headers)**:
   - Se establece la cabecera Content-Type: application/json para asegurar que el backend interprete correctamente los datos enviados en formato JSON.
4. **Cuerpo de la Petición (**body**)**:
   - Para métodos que envían datos (como POST), el cuerpo de la petición se serializa a JSON utilizando JSON.stringify().
5. **Manejo de Respuestas**:
   - Las respuestas se procesan en dos etapas:
     1. Verificar si la respuesta fue exitosa utilizando response.ok.
     2. Parsear el cuerpo de la respuesta a JSON con response.json().
6. **Gestión de Errores y Notificaciones**:
   - Se capturan errores de red (en el bloque catch) y respuestas HTTP no exitosas (response.ok === false).
   - Se utiliza react-toastify para mostrar notificaciones al usuario, mejorando la experiencia de usuario.
   - Los errores también se registran en la consola (console.error) para facilitar la depuración.

Ejemplo 1: Petición GET para Activar/Desactivar el Arduino

En este ejemplo, se implementa una función que envía una petición GET al backend para activar o desactivar el Arduino. La lógica incluye la actualización del texto de un botón según el estado actual (isWake).

```
const handleOnOff = () => {
    console.log("¡Botón 'Recibir Muestras' clickeado!");

    const boton = document.getElementById('recibirDatos') as HTMLButtonElement | null;
    if (boton) {
      boton.textContent = isWake ? "Apagar" : "Encender";
    }
    fetch(import.meta.env.VITE_WAKE_UP, { method: 'GET' })
      .then()
      .catch()
};
```

Explicación del Código:

1. **Cambio del texto del botón**:
   - Antes de enviar la petición, se actualiza el texto del botón para reflejar el estado actual (Encender o Apagar).
2. **Petición** fetch:
   - Se utiliza el método GET para enviar la solicitud al endpoint definido en VITE_WAKE_UP.
3. **Manejo de la respuesta**:
   - Si la respuesta no es exitosa (!response.ok), se lanza un error con un mensaje descriptivo.
4. **Gestión de errores**:
   - Los errores se capturan en el bloque catch y se notifican al usuario mediante react-toastify.

Ejemplo 2: Petición POST para Enviar Configuración

En este ejemplo, se implementa una función que envía una configuración al backend utilizando el método POST. La configuración se envía desde un formulario modal.

```
        const handleOnSubmit = (evento: React.FormEvent) => {

    event.preventDefault();
    const white = Number(config?.light_white);
    const blue = Number(config?.light_blue);
    const red = Number(config?.light_red);

    if ((white + blue + red) > 100) {
      toast.warn("La suma total de las luces no puede superar el 100%.");
      return;
    }

    console.log("check!");
    setShowModal(false);
    toast.promise(
      fetch(import.meta.env.VITE_CHANGE_CONFIG, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(config),
      })
        .then(response => {
          if (!response.ok) {
            console.error(`Error al guardar la configuración: ${response.status}`);
            throw new Error('Error al guardar la configuración');
          }
          console.log("Configuración guardada exitosamente");
        })
        .catch(error => {
          console.error("Error al realizar la petición para guardar la configuración:", error);
          throw error;
        })
        .finally(() => {
        }),
      {
        pending: 'Cambiando configuracion del arduino...',
        success: 'Configuracion modificada',
        error: 'Algo salió mal... 💀'
      }
    );
  };
```

Explicación del Código:

1. **Validación previa**:
    - Antes de enviar la petición, se valida que la suma de los valores de las luces no supere el 100%. Si la validación falla, se muestra una advertencia al usuario con toast.warn.
2. **Petición** fetch:
    - Se utiliza el método POST para enviar la configuración al backend.
    - La configuración se serializa a JSON utilizando JSON.stringify(config).
3. **Gestión de la respuesta**:
    - Si la respuesta no es exitosa (!response.ok), se lanza un error con un mensaje descriptivo.
4. **Notificaciones al usuario**:
    - Se utiliza toast.promise para mostrar notificaciones durante el proceso:
        - pending: Mientras se envía la configuración.
        - success: Si la configuración se guarda exitosamente.
        - error: Si ocurre un error durante el proceso.

Sección 6: Librerías Clave y su Uso

Esta sección detalla las principales librerías de terceros utilizadas en el frontend, describiendo su función y cómo se aplican en el proyecto.

6.1. Librerías Principales

- **React:** Biblioteca principal para construir la interfaz de usuario. Uso: Componentes funcionales, hooks (useState, useEffect, etc.).
- **Vite:** Herramienta de construcción rápida para proyectos frontend. Uso: Configuración del entorno de desarrollo y producción.
- **TypeScript:** Superset de JavaScript que añade tipado estático. Uso: Definición de interfaces y tipos para garantizar un código más robusto.

6.2. Librerías de Estilo

- **TailwindCSS:** Framework de utilidades CSS para diseño rápido. Uso: Clases de estilo en los componentes.

6.3. Librerías de Gráficos

- **@nivo/line:** Librería para renderizar gráficos de líneas. Uso: Visualización de datos como longitud de onda y gráficos comparativos.
- **@nivo/scatterplot:** Librería para gráficos de dispersión. Uso: Análisis de datos específicos en gráficos de dispersión.

6.4. Librerías de Notificaciones

- **React-Toastify:** Librería para mostrar notificaciones (o "toasts") en la interfaz de usuario. Uso: Mostrar mensajes de éxito, error o advertencia al usuario, y proporcionar retroalimentación sobre el estado de las operaciones y la conexión con el servidor.

6.5. Librerías para WebSockets

- **socket.io-client:** Cliente para manejar conexiones WebSocket con el backend. Uso: Comunicación en tiempo real con el servidor Flask para actualizaciones instantáneas de datos y estado.

6.6. Librerías de Configuración y Plugins

- **@vitejs/plugin-react-swc:** Plugin para integrar React con Vite. Uso: Optimización del rendimiento en el entorno de desarrollo.
- **@tailwindcss/vite:** Plugin para integrar TailwindCSS con Vite. Uso: Configuración y procesamiento de estilos en el proyecto.

6.7. Librerías de Desarrollo

- **ESLint:** Herramienta para analizar y mantener la calidad del código. Uso: Configuración de reglas para evitar errores comunes y asegurar la consistencia del código.
- **Prettier** (posiblemente): Herramienta para formatear el código. Uso: Asegurar un estilo de código consistente y automático.

6.8. Librerías de Utilidades

- **date-fns** (posiblemente): Biblioteca para manipulación de fechas. Uso: Formateo y manejo de fechas en el frontend.
- **Sección 7: Scripts y Ejecución**
- Esta sección describe los comandos principales utilizados para gestionar el proyecto frontend, desde la instalación de dependencias hasta el inicio del servidor de desarrollo y la construcción para producción.

Sección 7: Comandos

Comando	**Descripción**
npm install	Instala todas las dependencias del proyecto definidas en package.json.
npm run dev	Inicia el servidor de desarrollo local de Vite, con recarga en caliente y optimizaciones para el desarrollo.
npm run build	Genera una versión optimizada y minificada de la aplicación en el directorio dist/, lista para producción.
npm run preview	Previsualiza la aplicación construida en el directorio dist/, simulando un entorno de producción localmente.