DESENVOLVEMENTO DE INTERFACES WEB 22_23_Ord

2. API REST con Vue.js

1. Introducción

En este ejercicio vamos avanzar un poco más en Vue.js. Concretamente en las operaciones **CRUD (Crear, Consultar, Actualizar y Borrar).** Para ello usaremos **API REST.**

REST significa ***Representational State Transfer,*** y define lo estándares de la arquitectura de servicios web, básicamente cliente-servidor. Hay que entender que es una idea no una herramienta que determina la metodología a seguir para el desarrollo de los servicios de una aplicación:

- ***GET:*** para obtener datos del servidor
- ***POST:*** para insertar un recurso nuevo o dato en el servidor
- ***PUT:*** para actualizar un recurso del servidor
- ***PATCH:*** para actualizaciones parciales, por ejemplo, la dirección de un cliente
- ***DELETE***; para eliminar un recurso

En web se usan el término **RESTful,** es la implementación propiamente dicha para hacer las aplicaciones web más accesibles e intuitivas al usuario.

API que significa ***Application Programming Interface*** son las funcionalidades o recursos para que un sistema pueda interactuar con otros independientemente de la tecnología, lenguaje de programación u otra herramienta. Las pasarelas de pago son un buen ejemplo ya que con nuestro móvil accedemos a nuestra cuenta para realizar un pago en una tienda online. Una API web consta de dos acciones básica que son la **petición** al servidor y la **respuesta** de este.

Por último, hablemos de **JSON,** que es un formato de texto para intercambio de datos (ver fichero ***clients.json)***.

En resumen, una **API** que usa la arquitectura REST se le llama RESTful, la cual emplea ficheros en formato JSON.

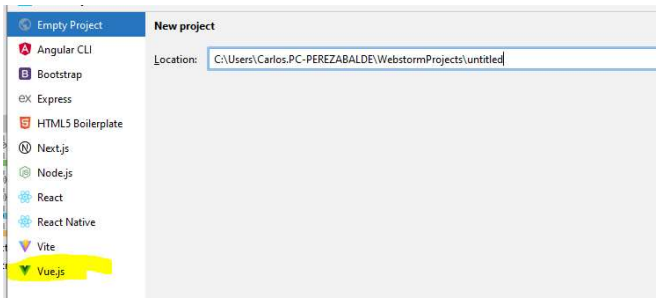
El objetivo de la práctica siguiente es conectarnos a una API REST, para llevar a cabo los servicos de la aplicación antes mencionados GET, POST, PUT y DELETE.

2. Instalación y configuración básica

Si no hemos instalado antes VUE los pasos son:

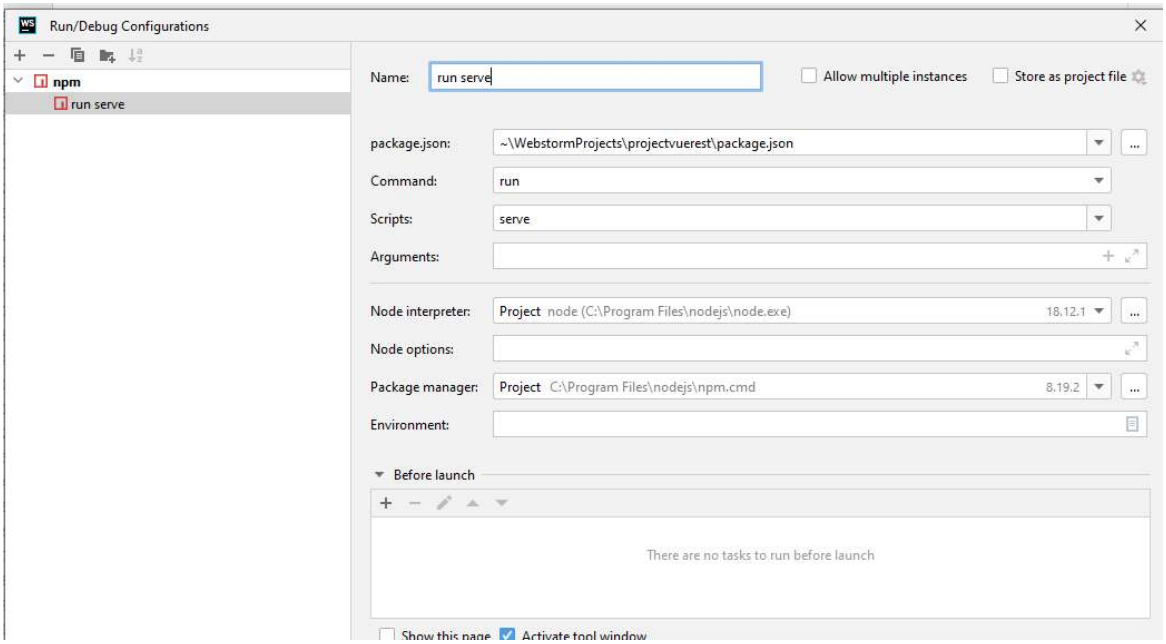
```
# Instalación con NPM
npm i -g @vue/cli @vue/cli-service-global
```

1. Instalar **Vue CLI**



2. Crear un proyecto e**n WebStorm o Visual Code**

A continuación configuramos el **Run**



Para no dilatar el proyecto echaremos mano del **framework** **Bootstrap** para los estilos. Para eso en el **public/index.html** del proyecto insertamos:

```
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" />
```

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.5.2/css/bootstrap.min.css" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

En primer lugar, modificamos el fichero **App.Vue** añadiendo el código siguiente:

App.vue

```vue
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12 mt-2">
        <h1>Usuarios Clientes</h1>
      </div>
    </div>
  </div>
</template>

<script>
2 usages  jcarlos
export default {
  name: 'app',
}
</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>
```

Fichero App.vue

index.html

```html
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scal
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.co
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

Fichero index.html

Básicamente lo que hacemos es que **src/index.html** cargue la **aplicación app.**

3. Codificación de los métodos.

Vamos, tal como se comentó, a aceptar peticiones GET, POST, PUT y DELETE, utilizando ficheros JSON, que albergaremos en nuestro servidor. Para ello, creamos un **array** que almacene los datos de los clientes, tal como se muestra en la figura siguiente:

```
<script>
2 usages  jcarlos *
export default {
  name: 'app',
  data() {
    return {
      usuarios: [],
    }
  },
}

</script>
```

Y estos son los métodos aún sin codificar:

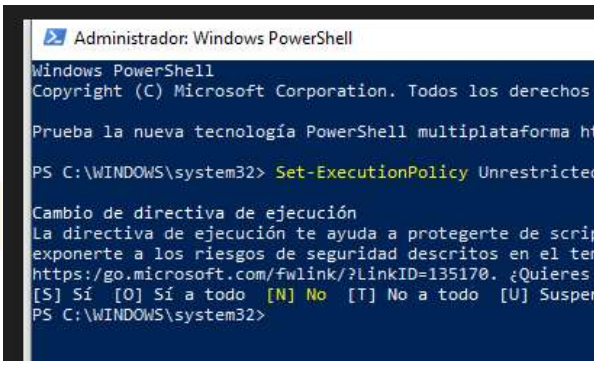
```
  methods: {
    getUsuarios() {
      // Método para obtener la lista de usuarios
    },
    postUsuario() {
      // Método para crear un usuario
    },
    putUsuario() {
      // Método para actualizar un usuario
    },
    deleteUsuario() {
      // Método para borrar un usuario
    },
  },
  mounted() {
    this.getUsuarios();
  }
},
```

Si nos fijamos existe la función **mounted().** Esta directiva se cargará cuando el servidor se lance llamando al listado de usuarios.

Llegados a este punto y, antes de continuar, vamos a preparar el acceso al fichero **clientes.json,** al que pretendemos acceder y cuya configuración se resume en la siguiente tabla:

```
npm install -g json-server
```

1. Instalación del servidor json

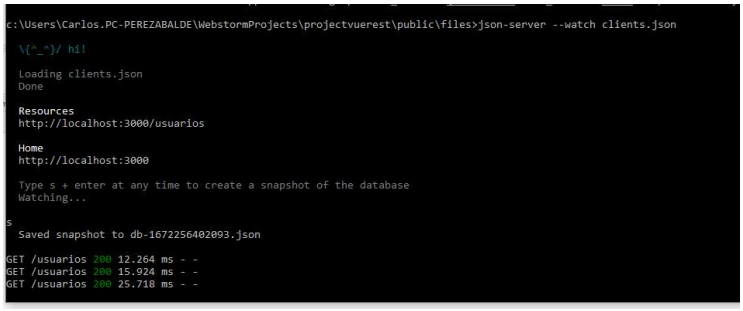


2. Si tenemos problemas con los permisos para ejecutar json-s
A continuación ejectuamos **Set-ExecutionPolicy Unrestricted**

```
npm install -g json-server
```

1. Instalación del servidor json

2. Si tenemos problemas con los permisos para ejecutar json-s
A continuación ejectuamos **Set-ExecutionPolicy Unrestricted**



4. A medida que modificamos datos el servidor va resgistrandc

3. El paso siguiente es lanzar el servidor json-server para que el fichero de datos **clients.json sea accesible y modificable.**
json-serve --watch clients.json

- **Captura de usuarios. Método GET**

En primer lugar vamos a listar los usuarios contenidos en el fichero **clients.json.**
Para ello, usaremos los **métodos asíncronos** que se conectarán con la API**.** Por simplificar, usaremos **la API Fetch** que incorpora JavaScript de forma nativa**.** La **API Fetch** mejoró el sistema **AJAX (Asynchronous JavaScript And XML.)** reduciendo la complejidad de las peticiones asíncronas al servidor. Hoy por hoy, junto con **Axios** son las más utilizadas. Su estructura es la siguiente:

```
async metodoAsincrono() {
  try {
    // Obtenemos los datos usando await
    const response = await fetch('url');

    // Respuesta en formato JSON
    const data = await response.json();

    // Aquí procesamos los datos
  } catch (error) {
    // Ejecución en caso de error
  }
}
```

La función **fetch** es la encargada de realizar la solicitud y la función **await** está a la espera de la respuesta de la petición realizada por fetch de los datos que se remitan del servidor.

Si lo aplicamos a la obtención de usuarios nos quedaría así:

```
methods: {
  async getUsuarios() {
    try {
      const response = await fetch( input: 'http://localhost:3000/usuarios');
      this.usuarios = await response.json();
    } catch (error) {
      console.error(error);
    }
  },
```

- **Alta usuario. Método POST.**

Si nos fijamos en el código tenemos dos métodos. El método **JSON.stringify** para transformar el a**rray de usuarios a formato JSON**. El operador **spread** de propagación para unir el array de usuarios con el objeto que hemos insertado**.**

```
  //alta un usuario cliente
  async postUsuario(usuario) {
    try {
      const response = await fetch( input: 'http://localhost:3000/usuarios', init: {
        method: 'POST',
        body: JSON.stringify(usuario),
        headers: { 'Content-type': 'application/json; charset=UTF-8' },
      });

      const usuarioCreado = await response.json();
      this.usuarios = [...this.usuarios, usuarioCreado];
    } catch (error) {
      console.error(error);
    }
  },
```

- **Actualizar el usuario. Método PUT**

En este caso, necesitamos el **id del usuario** para su actualización.

```
  //modifica un usuario cliente
  async putUsuario(usuario) {
    try {
      const response = await fetch( input: `http://localhost:3000/usuarios/${usuario.id}`, init: {
        method: 'PUT',
        body: JSON.stringify(usuario),
        headers: { 'Content-type': 'application/json; charset=UTF-8' },
      });

      const usuarioActualizado = await response.json();
      this.usuarios = this.usuarios.map(u => (u.id === usuario.id ? usuarioActualizado : u));
    } catch (error) {
      console.error(error);
    }
  },
```

- **Borrar el usuario. Método DEL**

Junto con el de GET es el método más sencillo. Por supuesto, precisa de la **id del usuario.**

```
  //elimina cliente
  async deleteUsuario(usuario) {
    try {
      await fetch( input: `http://localhost:3000/usuarios/${usuario.id}`, init: {
        method: "DELETE"
      });

      this.usuarios= this.usuarios.filter(u => u.id !== usuario.id);
    } catch (error) {
      console.error(error);
    }
  },
},
```

4. Codificación de los componentes. Visualización de los datos.

Ya hemos construido **el núcleo de la aplicación**. Nos quedan las vistas, es decir, los **componentes.**

Vamos a mostrar a los clientes en una tabla, para ello creamos nuestro primer componente **TablaUsuarios.vue,** en el directorio **src/components,** con el siguiente código:

p.vue TablaUsuarios.vue

```
<template>
  <div id="tabla-usuarios">
    <div v-if="!usuarios.length" class="alert alert-info" role="alert">
      No existen usuarios
    </div>
    <table class="table">
      <thead>
      <tr>
        <th>Nombre</th>
        <th>Email</th>
        <th>Ciudad</th>
      </tr>
      </thead>
      <tbody>
      <tr v-for="usuario in usuarios" :key="usuario.id">
        <td>{{ usuario.name }}</td>
        <td>{{ usuario.email }}</td>
        <td>{{ usuario.address.city }}</td>
      </tr>
      </tbody>
    </table>
  </div>
</template>

<!-- zona javascript -->
<script>
2 usages  XoanCarlos
export default {
  name: 'tabla-usuarios',
  props: {
    usuarios: Array,
  },
}
</script>
```

Debemos fijarnos como se usa el **v-for** para recorrer el **array** de usuarios. Además, pddemos observar la condición **v-if,** que se ejecuta si el array está vacío. Finalmente en la sección de Javascript tenemos el objeto **props,** que define las propiedades que enviamos al componente cuando rendericemos o mostremos los datos.

Para que todo esto funciona necesitamos tres cosas más, pero esta vez en **App.vue.**

a. Importar el componente en la zona de javascript:

```
import TablaUsuarios from '@/components/TablaUsuarios.vue';
```

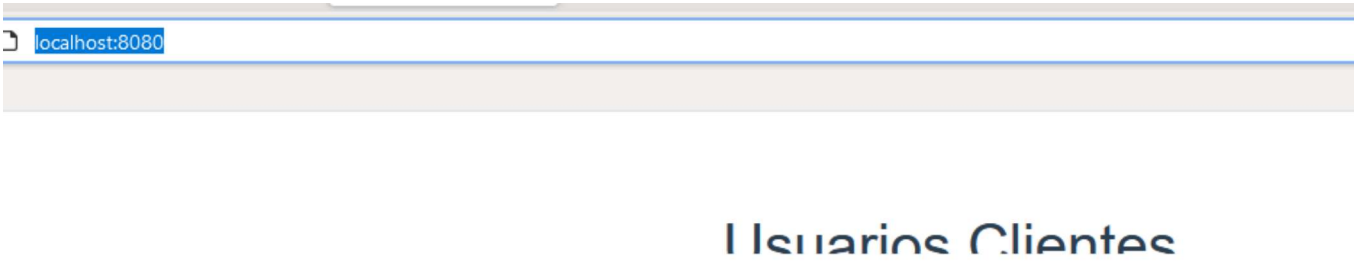
b. Agregarlo a la lista de componentes en la zona de javascript:

```
export default {
  // ...
  components: {
    TablaUsuarios,
  },
  // ...
}
```

c. Renderizar el componente pasando el array de usuarios como variable en el método **data.**

```
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12 mt-2">
        <h1>Usuarios</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <tabla-usuarios :usuarios="usuarios"/>
      </div>
    </div>
  </div>
</template>
```

Lanzamos el servidor y el resultado es:



Como venimos comentando la idea es crear una aplicación donde podamos **gestionar los datos.**

- **Eliminando DELETE**

Volvemos al componente **TablaUsuarios.vue** y añadimos el código que se indica en el recuadro rojo, para que el usuario pueda visualizar la acción:

Observamos que tenemos el atributo **@** seguido el **evento eliminar-usuario** cuya valor es que ejecute el **método deleteUsuario** presente en **App.vue.** Para elllo incluímos en **App.vue** la siguiente línea que **enlaza ambos elementos.**

Ahora nos vamos al código de **App.vue** y añadimos en la zona del script lo siguiente:

El resultado dela vista sería:

- **Actualizando. PUT**

Este método necesita algún código más. Como siempre empecemos por **TablaUsuarios.vue**

En primer lugar en la parte inferior añadimos el botón correspondiente a **Editar,** tal como hicimos con **Eliminar.**

Posteriormente observamos un **condicional v-if** que se ejecuta si se activa una variable llamada **editando**. Si se ejecuta en la vista se lanza un formulario en este caso con los **inputs** correspondiente y los **botones Guardar y Cancelar.**

Siguiendo en el fichero **TablaUsuarios.vue** modificamos la zona de javascript con el código siguiente dentro del método **data:**

Expliquemos este código, aunque es bastante claro. Si pulsamos el botón **Editar** la variable editando carga el valor **usuario.id,** lo que **activa el formulario de edición** (ver imagen anterior).
Posteriormente, si pulsamos **el evento guardarUsuario** llama al método correspondiente**,** de la misma forma **cancelarUsuario** y en ambos pone la variable **editando a null** para desactivar el formulario.

Solo nos queda conectar con **App.vue de putUsuario.**

Y en la zona del **script**

Una imagen de como funcionaría:

4. Formulario. Método POST.

Nos queda el último método, POST y la presencia de un **formulario** en la aplicación.
Para ello, vamos a crear un nuevo componente, así que vamos a crear el archivo **FormularioUsuario.vue** en la carpeta **src/components,** con el siguiente código.

Formulario

Parte 1 del template

Parte 2 del template

Parte 2 de javascript

Parte 1 de javascript

Nota.- Faltaría incluir la parte de ciudad. Queda como actividad.

Analicemos. Lo que hemos hecho es agregar un campo para el nombre del usuario y otro para el email. Mediante **el atributo v-model** enlazamos el valor de los campos con el de sus respectivas variables de estado.
Para poder validar los campos, asignaremos la **clase CSS is-invalid** a cada uno de los campos en caso de que estén vacíos cuando se envíe el formulario.
La clase se eliminará de los campos cuando el foco se sitúe sobre ellos o cuando se pulse una tecla, mediante los eventos `@focus` **y** `@keypress` respectivamente, que ejecutarán el método **resetEstado.**

Además, añadimos dos validadores**, nameValido y emailInvalido,** para comprobar que los campos name y email no están vacío, y a la clase **is-invalid.**

Ahora nos tenemos que ir al fichero central, **App.vue,** y al igual que antes en este caso importamos FormularioUsuario.

Y a la lista de componentes:

```
<template>
  <div id="app" class="container">
    <div class="row">
      <div class="col-md-12 mt-2">
        <h1>Usuarios Clientes</h1>
      </div>
    </div>
    <div class="row">
      <div class="col-md-12">
        <formulario-clientes @crear-usuario="postUsuario" />
        <tabla-usuarios :usuarios="usuarios" @eliminar-usuario="deleteUsuario" @actualizar-usuario="putUsuario" />
      </div>
    </div>
  </div>
</template>
```

Nos que da por último la parte del **script en App.vue de postUsuario**

Y el resultado sería:

Y ya está lista la aplicación **para pasar a producción.**

5. Build & Deploy de la aplicación

Para pasar a producción, primero paramos el servidor y luego ejecutamos:

Si queremos publicar y distribuir la aplicación en GitHub partiendo del hecho de que has creado el repositorio (y tienes una cuenta) **totalmente vacío:**

http://www.github.com/miusuario/projectvuerest

Seguimos los siguientes pasos**:**

- **Inicializar el repositorio de Git;**

- **Aunque no es necesario ya que al dar de alta el repositorio, ejecutamos la agregación del proyecto, cambiando el nombre de usuario.**

- **Haz un commit de los archivos de tu proyecto:**

- **Y finalmente se hace push desde el repositorio local hasta la rama main del repositorio en GitHub:**

Cuando compilas una aplicación Vue**, el directorio por defecto es el directorio /dist,** por lo que lo tendremos que enviar al repositorio. Para ello necesitaremos **crear una nueva rama** en la que agregaremos ciertos cambios a la configuración de Vue. Esto permite ver la aplicación a pleno rendimiento.

- **Creamos una nueva rama.**

- **Editamos el archivo .gitignore** y se elimina **la línea /dist**, de modo que se pueda enviar ese directori a GitHub

- Crear el archivo **vue.config.js en el directorio raíz del proyecto**. Establecer el **directorio público en el que estarán los archivos del proyecto**. El directorio, una vez esté el proyecto en GitHub Pages, **tendrá el mismo nombre que el repositorio asociad**o, por lo que debes agregar lo siguiente, reemplazando nombre-repositorio por el nombre de tu repositorio en GitHub:

- Hacer un **build** del proyecto para generar el directorio **/dist**

npm run build

- Agregar el directorio **/dist** al repositorio

- **Commit** para los cambios

- Y finalmente **push** para publicar y poner en producción la página.

Para ver el resultado y la página completamente operativa:

```
https://usuario.github.io/nombre-repositorio
```

Cambiando **usuario** por el **nombre de tu usuario** en GitHub y **nombre-repositorio** por el **nombre del repositorio** donde esté el proyecto.

Última modificación: Luns, 23 de Xaneiro de 2023, 19:44

◄ 1. Introducción a Vue.js

Ir a...

3. MEVN. MongoDB, Express, Vue y Node. Configuración del Servidor. (I) ►

DIW_22_23_Ord

Resumen da retención de datos
Obter a apli móbil