UD12 - Axios

1. Introducción

El framework *Vue* sólo se ocupa de la capa de vista de la aplcación pero su "ecosistema" como sus creadores le llaman, incluye multitud de herramientas para todo lo que podamos necesitar a la hora de realizar grandes proyectos.

Una de las librerías más utilizadas es la que permite realizar de forma sencilla peticiones Ajax a un servidor. Existen múltiples librerías para ello y la más utilizada es **axios**.

Podríamos hacer peticiones Ajax como vimos en Javascript (con promesas o *fetch*) pero es más sencillo con *axios*. Axios ya devuelve los datos transformados a JSON en una propiedad llamada *data*.

2. Instalación y uso

Como esta librería vamos a usarla en producción la instalaremos como dependencia del proyecto:

```
npm install axios -S
```

2.1 Usar *axios*

En el componente en que vayamos a usarla la importaremos:

```
import axios from 'axios'
```

Como es una dependencia incluida en el *package.json* no se indica su ruta (se buscará en **node-modules**).

Ya podemos hacer peticiones Ajax en el componente. Para ello axios incluye los métodos:

- **.get(url)**: realiza una petición GET a la url pasada como parámetro que supondrá una consulta SELECT a la base de datos
- **.post(url, objeto)**: realiza una petición POST a la url pasada como parámetro que posiblemente realizará un INSERT del objeto pasado como segundo parámetro
- **.put(url, objeto)**: realiza una petición PUT a la url pasada como parámetro que posiblemente realizará un UPDATE sobre el registro indicado en la url que será actualizao con los datos del objeto pasado como segundo parámetro
- **.delete(url)**: realiza una petición DELETE a la url pasada como parámetro que supondrá una consulta DELETE a la base de datos para borrar el registro indicado en la url

Estos métodos devuelven una promesa por lo queal hacer la petición indicaremos con el método **.then** la función que se ejecutará cuando responda el servidor si la petición se resuelve correctamente y con el método **.catch** la función que se ejecutará cuando responda el servidor si ocurre algún error.

La respuesta del servidor tiene, entre otras, las propiedades:

- **data**: aquí tendremos los datos devueltos por el servidor
- status: obtendremos el código de la respuesta del servidor (200, 404, ...)
- statusText: el texto de la respuesta del servidor ('Ok', 'Not found', ...)
- message: mensaje del servidor en caso de producirse un error
- headers: las cabeceras HTTP de la respuesta
- ...

La sintaxis de una petición GET a axios sería algo como:

```
axios.get(url)
  .then(response => ...realiza lo que sea con response.data ...)
  .catch(response => ... trata el error ...)
```

2.2 Aplicación de ejemplo

Vamos a seguir con la aplicación de la lista de tareas pero ahora los datos no serán un array estático sino que estarán en un servidor. Usaremos como servidor para probar la aplicación **json-server** por lo que las peticiones serán a la URL 'localhost:3000' que es el servidor web de json-server.

Los cambios que debemos hacer en nuestra aplicación son:

1. El componente principal (TodoList) pide todos los datos al cargarse
2. Al borrar un elemento haremos una petición al servidor para que lo borre de allí y cuando sepamos que se ha borrado lo borramos del array (o recargamos los datos)
3. Lo mismo al insertar un nuevo elemento
4. Al marcar/desmarcar un elemento lo modificaremos en la base de datos
5. Para borrarlos todos haremos peticiones DELETE al servidor

Vamos a modificar los diferentes componentes para implementar os cambios requeridos:

2.2.1 Pedir los datos al cargarse

Modificamos el fichero **Todo-List.vue** para añadir en su sección *script*:

- Antes del objeto vue:

```
import axios from 'axios'

const url='http://localhost:3000'
```

- Dentro del objeto añadimos el *hook* **mounted** para hacer la petición Ajax al montar el componente (recordad que esa función se ejecuta automáticamente cuando se acaba de *renderizar* el componente):

```
...
  mounted() {
    axios.get(url+'/todos')
      .then(response => this.todos=response.data)
      .catch(response => {
        if (!response.status) {// Si el servidor no responde 'response' no es un objeto sino texto
          alert('Error: el servidor no responde');
```

```
          console.log(response);
        } else {
          alert('Error '+response.status+': '+response.message);

        }
        this.todos=[];
      })
  },
...
```

2.2.2 Borrar un todo

Modificamos el método *delTodo* del fichero **Todo-List.vue**:

```
    delTodo(index){
      var id=this.todos[index].id;
      axios.delete(url+'/todos/'+id)
        .then(response => this.todos.splice(index,1) )
        .catch(response => alert('Error: no se ha borrado el registro.
'+response.message))
    },
```

2.2.3 Añadir un todo

Modificamos el método *addTodo* del fichero **Todo-List.vue**:

```
    addTodo(title) {
      axios.post(url+'/todos', {title: title, done: false})
        .then(response => this.todos.push(response.data)
        )
        .catch(response => alert('Error: no se ha añadido el registro.
'+response.message))
    },
```

Al servidor hay que pasarle como parámetro el objeto a añadir. E el caso del json-server devolverá en el **response.data** el nuevo objeto añadido al completo. Otros servidores devuelven sólo la *id* del nuevo registro o pueden no devolver nada.

2.2.4 Actualizar el campo *done*

Ahora ya no nos es útil el índice de la tarea a actualizar sino que necesitamos su id, su título y su estado así que modificamos el *template* del fichero **Todo-List.vue** para pasar el elemento entero a la función:

```
      <todo-item
        v-for="(item,index) in todos"
        :key="item.id"
        :todo="item"
        @delItem="delTodo(index)"
        @doneChanged="toogleDone(item)">
       </todo-item>
```

A continuación modificamos el método *changeTodo* del fichero **Todo-List.vue**:

```
    toogleDone(todo) {
      axios.put(url+'/todos/'+todo.id, {
          id: todo.id,
          title: todo.title,
          done: !todo.done
        })
        .then(response => todo.done=response.data.done)
        .catch(response => alert('Error: no se ha modificado el registro. '+response.message))
    },
```

Lo que hay que pasar en el objeto y qué se devuelve en la respuesta depende del servidor API-REST usado. En el caso de json-server los campos que no le pasemos en el objeto los eliminará por lo que debemos pasar también al campo *title* (otros servidores dejan como están los campos no incluidos en el objeto por lo que no haría falta pasárselo). Y lo que devuelve en **response.data** es el registro completo modificado.

2.2.5 Borrar todas las tareas

Modificamos el método *delTodos* del fichero **Todo-List.vue**. Como el servidor no tiene una llamada para borrar todos los datos podemos recorrer el array *todos* y borrar cada tarea usando el método **delTodo** que ya tenemos hecho:

```
    delTodos() {
      this.todos.forEach((todo, index) => this.delTodo(index));
    }
```

Si lo probáis con muchos registros es posible que no se borren todos correctamente (en realidad sí se borran de la base de datos pero no del array). ¿Sabes por qué?. ¿Cómo lo podemos arreglar? (PISTA: el índice cambia según los elementos que haya y las peticiones asíncronas pueden no ejecutarse en el orden que esperamos).

2.3. Organizar las peticiones

Para mejorar la legibilidad del código vamos a crear un fichero que será donde estén las peticiones a axios de forma que nuestros componentes queden más limpios. Podríamos llamar al fichero *APIService.js* y allí creamos las funciones que laman a la API:

```
import axios from 'axios';
const baseURL = 'http://localhost:3000';

export default {
  getTodos() {
    return axios.get(baseURL+'/todos')
  },

  delTodo(id){
    return axios.delete(baseURL+'/todos/'+id)
  },

  addTodo(newTodo) {
    return axios.post(baseURL+'/todos', newTodo)
  },

  toogleDone(todo) {
    return axios.put(baseURL+'/todos/'+todo.id, {
      id: todo.id,
      title: todo.title,
      done: !todo.done
    })
  },
}
```

En cada componente que tenga que hacer una llamada a la API se importa este fichero y se llama a sus funciones:

```
import APIService from '../APIService';

export default {
```

```
  ...
  methods: {
    getData() {
      getTodos()
      .then(response=>response.data.forEach(todo=>this.todos.push(todo)))
      .catch(error=>console.error('Error: '+(error.statusText ||
error.message || error))
    },
    ...
  },
  mounted() {
    this.getData();
  },
```

2.3.1 Api con varias tablas

Si nuestra APIService tiene que acceder a varias tablas el código va haciéndose más largo. Podemos escribir lo mismo de antes pero de forma más concisa:

```
import axios from 'axios';
const baseURL = 'http://localhost:3000';

const todos = {
    getAll: () => axios.get(`${baseURL}/todos`),
    getOne: (id) => axios.get(`${baseURL}/todos/${id}`),
    create: (item) => axios.post(`${baseURL}/todos`, item),
    modify: (item) => axios.put(`${baseURL}/todos/${item.id}`, item),
    delete: (id) => axios.delete(`${baseURL}/todos/${id}`),
    toogleDone: (item) => axios.put(`${baseURL}/categories/${item.id}`, {
      id: item.id,
      title: item.title,
      done: !item.done
    }),
};

const categories = {
    getAll: () => axios.get(`${baseURL}/categories`),
    getOne: (id) => axios.get(`${baseURL}/categories/${id}`),
    create: (item) => axios.post(`${baseURL}/categories`, item),
    modify: (item) => axios.put(`${baseURL}/categories/${item.id}`,
item),
    delete: (id) => axios.delete(`${baseURL}/categories/${id}`),
```

```
};


export default {
    todos,
    categories,
};
```

2.3.2 Api como clase

También podemos usar programación orientada a objetos para hacer nuestra ApiService y construir una clase que se ocupe de las peticiones a la API:

```
import axios from 'axios';
const baseURL = 'http://localhost:3000';

export class APIService{
  constructor(){
  }
  getTodos() {
    return axios.get(baseURL+'/todos')
  }
  delTodo(id){
    return axios.delete(baseURL+'/todos/'+id)
  },
  addTodo(newTodo) {
    return axios.post(baseURL+'/todos', newTodo)
  },
  toogleDone(todo) {
    return axios.put(baseURL+'/todos/'+todo.id, {
      id: todo.id,
      title: todo.title,
      done: !todo.done
    })
  },
}
```

Y en los componentes donde queramos usarlo importamos la clase y creamos una instancia de la misma:

```
import { APIService } from '../APIService';

const apiService=new APIService();
```

```
4
5  export default {
6    ...
7    methods: {
8      getData() {
9        apiService.getTodos()
10       .then(response=>response.data.forEach(todo=>this.todos.push(todo)))
11       .catch(error=>console.error('Error: '+(error.statusText ||
   error.message || error))
12     },
13     ...
14   },
15   mounted() {
16     this.getData();
17   },
```

2.3.3 El fichero *.env*

Se trata de un fichero donde guardar las configuraciones de la aplicación y la ruta del servidor es una constante que estaría mejor en este fichero que en el código como hemos hecho nosotros.

Vue puede acceder a todas las variables de *.env* que comiencen por VUE_APP_ por medio del objeto `process.env` por lo que en nuestro código en vez de darle el valor a *baseURL* podríamos haber puesto:

```
1  const baseURL = process.env.VUE_APP_RUTA_API;
```

Y en el fichero ,*env* ponemos

```
1  VUE_APP_RUTA_API=http://localhost:3000
```

El fichero *.env* por defecto se sube al repositorio por lo que no debemos poner información sensible (como usuarios o contraseñas). Para ello tenemos un fichero ***.env.local*** que no se sube, o bien debemos añadir al *.gitignore* dicho fichero. En cualquier caso, si el fichero con la configuración no lo subimos al repositorio es conveniente tener un fichero *.env.exemple*, que sí se sube, con valores predeterminados para las distintas variables que deberán cambiarse por los valores adecuados en producción. Además del .env y el .env.local también hay distintos ficheros que son usados en desarrollo (*.env.development*) y en producción (*.env.production*) y que pueden tener distintos datos según el entorno en que nos encontramos. Por ejemplo en el de desarrollo el valor de VUE_APP_RUTA_API podría ser "http://localhost:3000" si usamos *json-server* mientras que en el de producción tendríamos la ruta del servidor de producción de la API.

2.3.4. Añadir cabeceras a la petición

Muchas veces necesitamos añadir cabeceras a una petición *axios*, por ejemplo para enviar un token que nos autentifique ante una API. Axios permite pasar como tercer parámetro un objeto con una configuración personalizada, que puede incluir esas cabeceras:

```
import axios from 'axios';
const baseURL = 'http://localhost:3000';

const config = {
  headers: {
    'Authorization' = 'Bearer ' + localStorage.token;
  }
}

const data = ...

axios.post(baseURL, data, config)
.then(...)
```

Si queremos añadir una cabecera a todas las peticiones POST que realicemos podemos hacerlo con:

```
axios.defaults.headers.post['header1'] = 'value'
```

Y si queremos añadir una cabecera a TODAS las peticiones de cualquier tipo:

```
axios.defaults.headers.common['header1'] = 'value'
```

4. Axios interceptors

Podemos hacer que se ejecute código antes de cualquier petición a axios o tras recibir la respuesta del servidor usando los *interceptores* de axios. Es otra forma de enviar un token que nos autentifique ante una API sin tener que ponerlo en el código de cada petición, pero también nos permite hacer cualquier cosa que necesitemos.

Para interceptar las peticiones usaremos `axios.interceptors.request.use( (config) => fnAEjecutar, (error) => fnAEjecutar)` y para las respuestas `axios.interceptors.response.use( (response) => fnAEjecutar, (error) => fnAEjecutar)`. Se les pasa como parámetro la función a ejecutar si todo es correcto y la que se ejecutará si ha habido algún error. El interceptor de peticiones recibe como parámetro un objeto con toda la configuración de la petición (incluyendo sus cabeceras) y el interceptor de respuestas recibe la respuesta del servidor.

Veamos un ejemplo en que queremos enviar en las cabeceras de cada petición el token que tenemos almacenado en el LocalStorage y queremos mostrar un alert siempre que el servidor devuelva en su respuesta un error que no sea de tipo 400. Además mostraremos por consola las peticiones y las respuestas si activamos el modo DEBUG:

```
import axios from 'axios';
const baseURL = 'http://localhost:3000';
const DEBUG = true;

axios.interceptors.request.use((config) => {
    if (DEBUG) {
        console.info('Request: ', config);
    }

    const token = localStorage.token;
    if (token) {
        config.headers['Authorization'] = 'Bearer ' + localStorage.token;
    }
    return config;
}, (error) => {
    if (DEBUG) {
        console.error('Request error: ', error);
    }
    return Promise.reject(error);
});

axios.interceptors.response.use((response) => {
    if (DEBUG) {
        console.info('Response: ', response);
    }
    return response;
}, (error) => {
    if (error.response && error.response.status !== 400) {
        alert('Response error ' + error.response.status + '(' +
error.response.statusText + ')'
    }
    if (DEBUG) {
        console.info('Response error: ', error);
    }
    return Promise.reject(error);
});

```

```
37 const categories = {
38     getAll: () => axios.get(`${baseURL}/categories`),
39     getOne: (id) => axios.get(`${baseURL}/categories/${id}`),
40     create: (item) => axios.post(`${baseURL}/categories`, item),
41     modify: (item) => axios.put(`${baseURL}/categories/${item.id}`,
   item),
42     delete: (id) => axios.delete(`${baseURL}/categories/${id}`),
43 };
44
45 export default {
46     categories,
47 };
```