Manejo y uso de peticiones asíncronas con Fetch.

La API Fetch proporciona una interfaz JavaScript para acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global fetch() que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de XMLHttpRequest. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías como Service Workers. Fetch también aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

El objeto Promise devuelto desde fetch() no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo será rechazado ante un fallo de red o si algo impidió completar la solicitud.

La forma de realizar una petición es bastante simple, a partir de su uso dentro de una función asíncrona simplemente se utiliza la siguiente sintaxis:

```
let response = await fetch(“url”, {
          method: 'GET',
          headers: {
              'Content-Type': 'application/json',
              'Accept': 'application/json'
          }
      });
const dataJSON =  await response.json();
```

El método puede ser del tipo GET, PUT, DELETE, POST por lo que realizar este tipo de peticiones es bastante similar, excepto que en el caso de la petición POST y PUT se adjunta lo que serian los datos por medio de la propiedad body.

```
let response = await fetch((“url”, {
          method: 'POST',
          headers: {
              'Content-Type': 'application/json',
              'Accept': 'application/json'
          },
          body: JSON.stringify(Datos)
 });
const dataJSON =  await response.json();
```

Implementación

En muchas ocasiones querremos que nuestras peticiones sean manejadas con control de errores y es probable que sea necesario que la data esperada por esta sea conservada en el cliente de la aplicación para evitar problemas con APIs deprecadas, servers que por razones desconocidas o fuera de nuestro control están fuera de línea o servicios que tienen sus

propios horarios de funcionamiento, o incluso errores inesperados bajo ciertas circunstancias. Un ejemplo claro podría ser que nuestra app esta siendo desarrollada como una PWA (Progresive Web App) y esta requerirá tener funcionamiento fuera de línea. Por ende, esta propuesta incluye una función que intenta manejar estos escenarios valiéndose del API LocalStorage de JS.

La implementación se hará por medio de una clase llamada WAjaxTools dentro de WComponentsTools, esta clase constará de 3 métodos principales y 2 métodos de ayuda.

```
class WAjaxTools {
    static Request = async (Url, typeRequest, Data = {}, typeHeader) => {···
    }
    static PostRequest = async (Url, Data = {}, typeHeader) => {···
    }
    static GetRequest = async (Url) => {···
    }
    static ProcessRequest = async (response, Url) => {···
    }
    static LocalData = (Url) => {···
    }
}
```

El primer método estático llamado Request, este recibirá el tipo de petición que ejecutara además de un parámetro Data y el tipo de encabezado que enviara, estos parámetros tendrán valores por defecto para evitar dificultades en su implementación.

```
static Request = async (Url, typeRequest = "GET", Data = {}, typeHeader) => {
        try {
            let ContentType = "application/json; charset=utf-8";
            let Accept = "*/*";
            if (typeHeader == "form") {
                ContentType = "application/x-www-form-urlencoded; charset=UTF-8";
                Accept = "*/*";
            }
            let dataRequest =  {
                method: typeRequest,
                headers: {
                    'Content-Type': ContentType,
                    'Accept': Accept
                }
            }
            if (Data != {}) {
                dataRequest.body = JSON.stringify(Data);
            }
            let response = await fetch(Url,);
            const ProcessRequest = await this.ProcessRequest(response, Url);
            return ProcessRequest;
        } catch (error) {
            if (error == "TypeError: Failed to fetch") {
                return this.LocalData(Url);
            }
        }
    }
```

Sin embargo, para simplificar los llamados y el número de parámetros que se deben declarar crearemos funciones específicas para peticiones GET y POST.

```
static PostRequest = async (Url, Data = {}, typeHeader) => {
        try {
            let ContentType = "application/json; charset=utf-8";
            let Accept = "*/*";
            if (typeHeader == "form") {
                ContentType = "application/x-www-form-urlencoded; charset=UTF-8";
                Accept = "*/*";
            }
            let response = await fetch(Url, {
                method: 'POST',
                headers: {
                    'Content-Type': ContentType,
                    'Accept': Accept
                },
                body: JSON.stringify(Data)
            });
            const ProcessRequest = await this.ProcessRequest(response, Url);
            return ProcessRequest;
        } catch (error) {
            if (error == "TypeError: Failed to fetch") {
                return this.LocalData(Url);
            }
        }
    }
    static GetRequest = async (Url) => {
        try {
            let response = await fetch(Url, {
                method: 'GET',
                headers: {
                    'Content-Type': 'application/json',
                    'Accept': 'application/json'
                }
            });
            const ProcessRequest = await this.ProcessRequest(response, Url);
            return ProcessRequest;
        } catch (error) {
            console.log(error)
            if (error == "TypeError: Failed to fetch") {
                return this.LocalData(Url);
            }
        }
    }
```

Los métodos ProcessRequest se usará para verificar y procesar el resultado del response y determinaran si ocurrió un error 404 o 500 y en dependencia del resultado retornara datos locales usando la función LocalData o un valor por defecto.

```
    static ProcessRequest = async (response, Url) => {
        if (response.status == 404 || response.status == 500) {
            console.log("ocurrio un error: " + response.status);
            if (typeof response !== "undefined" && typeof response !== "null" && response != "") {
                return this.LocalData(Url);
            } else {
                return [];
            }
        } else {
            response = await response.json();
            localStorage.setItem(Url, JSON.stringify(response));
            return response;
        }
    }
    static LocalData = (Url) => {
        let responseLocal = localStorage.getItem(Url);
        return JSON.parse(responseLocal);
    }
```

Peticiones GET

Para hacer uso de estas funciones, podemos hacer una exportación de la clase dentro del WComponetsTools, por lo que la declaración quedaría de esta forma.

```
export { WAjaxTools, WRender }
```

para realizar una petición GET sería tan fácil como ejecutar esta línea de código

```
const myFun = ()=>{
    WAjaxTools.GetRequest("url");
}
```

Si deseáramos almacenar algún valor resultante de la petición se ejecutaría dentro de una función asíncrona utilizando la palabra clave await.

```
const myFun = async ()=>{
    const response = await WAjaxTools.GetRequest("url");
    //hacer algo con response
}
```

En el caso que no se pudiese utilizar una función asíncrona o no se deseara hacer uso de estas, se puede utilizar la función reservada then, que permite ejecutar acciones después que la promesa es resuelta

```
const myFun =  ()=>{
    WAjaxTools.GetRequest("url").then((response)=>{
        console.log(response)
       //hacer algo con response
    });
}
```

Peticiones POST

El caso de las peticiones POST, su uso es bastante similar, en este caso podremos adjuntar un arreglo de datos si fuese requerido.

```
const myFun = ()=>{
    WAjaxTools.PostRequest("url", {datos:"url"});
}
```

Almacenando el retorno en una variable.

```
const myFun = async ()=>{
    const response = await WAjaxTools.PostRequest("url", {datos:"url"});
    //hacer algo con response
}
```

Uso en funciones no asíncronas

```
const myFun =  ()=>{
    WAjaxTools.PostRequest("url", {datos:"url"}).then((response)=>{
        console.log(response)
       //hacer algo con response
    });
}
```

En este caso debemos de tener en cuenta que es probable que la aplicación envié los datos en formato de formulario y no en formato JSON, por lo que deberemos especificar como parámetro que se enviara una petición con datos en form-part.
En el caso de recuperar los datos a partir de un formulario en html:

```
const myFun = async ()=>{
    const formData = new FormData(document.getElementById('formulario'));
    const res = await WAjaxTools.PostRequest("url", formData, "form");
    console.log(res);
}
```

Para crear datos puramente en JS.

```
const myFun = async ()=>{
    const formData = new FormData();
    formData.append('dato1', 'valor');
    formData.append('dato2', 'valor');
    const res = await WAjaxTools.PostRequest("url", formData, "form");
    console.log(res);
}
```