EXERCISES QUE TRABAJAREMOS EN EL CUE:

- EXERCISE 1: INTRODUCCIÓN A AJAX Y SU IMPLEMENTACIÓN EN UN HTML.
- EXERCISE 2: INTRODUCCIÓN A FETCH.
- EXERCISE 3: MANIPULANDO INFORMACIÓN CON FETCH.

EXERCISE 1: INTRODUCCIÓN A AJAX Y SU IMPLEMENTACIÓN EN UN HTML

En la lectura establecimos que jQuery tiene una serie de métodos para lograr implementar funcionalidades AJAX a una página web. Los que están disponibles son los que corresponden a los verbos HTTP `Get` y `Post`.

Estos dos verbos son unos de los métodos más comunes para establecer comunicación entre un cliente y un servidor. De manera breve, repasaremos los objetivos de cada uno de estos métodos de consulta.

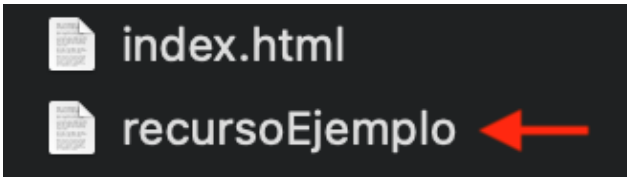
El método `Get` es utilizado para solicitar información desde un recurso especificado, y su objetivo es obtener información desde un servidor. Mientras que el método `Post` es usado para enviar información desde el cliente hacia un servidor o recurso especificado. Con jQuery podemos reflejar la funcionalidad de estos dos verbos HTTP con los métodos: `$.get()` y `$.post()`.

El método `$.get()` solicita información desde el servidor con una solicitud HTTP `Get`. Su sintaxis es la siguiente:

```
1 $.get(URL, callback);
```

En este método, el parámetro `URL` es obligatorio, dado que es la dirección especifica a la que queremos consultar, y si no está delimitada, la consulta no se llevaría a cabo. El segundo parámetro es el `Callback`, y como hemos visto anteriormente, esta es una función que se realiza una vez que se finalice la acción del método. En este caso, el Callback se ejecuta una vez que se realice la consulta.

Realicemos una prueba sencilla. Vamos a plantear un HTML que solo contiene un botón que lleva a cabo la consulta. Además, dentro de nuestro script, establecemos un método `Get` que hace una consulta a un archivo de prueba, llamado **recursoEjemplo.txt** y que está ubicado dentro de la carpeta donde se encuentra nuestro proyecto:



El archivo **recursoEjemplo.txt** contiene la siguiente información:

```
Este es un documento de texto usado como ejemplo para demostrar las capacidades de jQuery con las funcionalidades AJAX.
```

```html
<!DOCTYPE html>
<html lang="en">

<head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>Ejercicio JQuery</title>
        <!-- jQuery -->
        <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
        <!-- Bootstrap -->
        <link rel="stylesheet" href="/bootstrap-5.0.0-beta2-dist/css/bootstrap.min.css">
</head>

<body>
        <div class="container">
                <button class="w-100 btn btn-lg btn-primary" style="margin-top: 45%;">Realiza consulta</button>
        </div>
</body>
<script>
$(function() {
        $("button").click(function() {
                $.get("recursoEjemplo.txt", function(data, status) {
                        alert(`Contenido: ${data} \nEstado: ${status}`);
                })
        })
});
</script>

</html>
```

Si nos fijamos en nuestro método `$.get()`, notaremos que el Callback contiene dos parámetros importantes: `data` y `status`. `Data` es un parámetro especial que nos permite almacenar la información contenida en el recurso consultado. En `status` se almacena el estado HTTP de nuestra consulta, como:

`"success"` – Éxito.

`"error"` – Error.

`"abort"` – Aborta consulta.

Si tenemos nuestro archivo **recursoEjemplo** dentro de la misma carpeta que nuestro archivo HTML, podremos ver el siguiente resultado en nuestro navegador:

Cómo se puede observar, la consulta realizada mediante el método `$.get()` nos muestra el contenido de nuestro archivo de consulta: **recursoEjemplo.txt**.

Ya que hemos logrado consultar un recurso externo, guardado de manera local, ahora utilizaremos el mismo método para consultar una API. Para efectos de este ejercicio, vamos a consultar la API pública del Sistema de Información Nacional de Calidad del Aire de Chile: https://sinca.mma.gob.cl/index.php/json

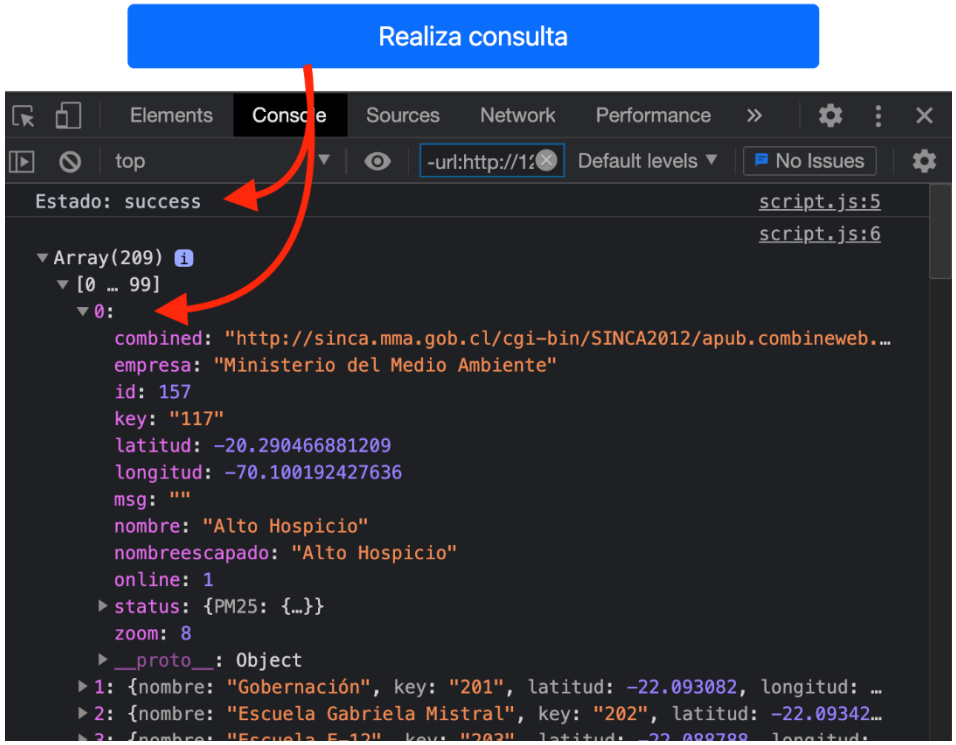
La ruta que queremos consultar muestra un arreglo de objetos JSON. El siguiente es un ejemplo de los objetos JSON que deseamos consumir:

```
[
  {
    "nombre": "Alto Hospicio",
    "key": "117",
    "latitud": -20.290466881209,
    "longitud": -70.100192427636,
    "zoom": 8,
    "nombreescapado": "Alto Hospicio",
    "empresa": "Ministerio del Medio Ambiente",
    "id": 157,
    "online": 1,
    "status": {
      "PM25": {
        "parname": "MP-2,5",
        "date": "2021-04-26",
        "hour": "10:00",
        "level": 1,
        "movil": 6.21,
        "index": 12,
        "value": 0,
        "unit": "&amp;micro;g/m&lt;sup&gt;3&lt;/sup&gt;",
        "umovil": 6.21,
        "uvalue": 0,
        "uunit": "&amp;micro;g/m&lt;sup&gt;3&lt;/sup&gt;"
      }
    },
    "combined": "http://sinca.mma.gob.cl/cgi-bin/SINCA2012/apub.combineweb.cgi?page=pageComb&;amp;ins=SIVICA",
    "msg": ""
  },
```


Para llegar a consumir esta API, y retornar el JSON que aparece en la ruta, vamos a implementar la ruta nueva a nuestro método `$.get()`, mostrándola mediante la consola:

```
$(function() {
        $("button").click(function() {
                $.get("https://sinca.mma.gob.cl/index.php/json",
function(data, status) {
                        console.log(`Estado: ${status}`);
                        console.log(data);
                })
        })
});
```

El resultado de nuestra consulta HTTP `Get`, es la siguiente:



Como podemos ver mediante la consola, este método de jQuery nos permite consultar incluso los valores en formato JSON. Ahora, revisaremos cómo utilizar otro método de jQuery para realizar una petición de tipo Post.

A continuación, vamos a utilizar el método AJAX `$.post()`, el cual tiene la siguiente sintaxis:

```
$.post(URL, data, function(data, status, xhr)
```

`URL` – Dirección por consultar (este es el único atributo obligatorio, todos los demás son opcionales).

`Data` – Información que queremos enviar.

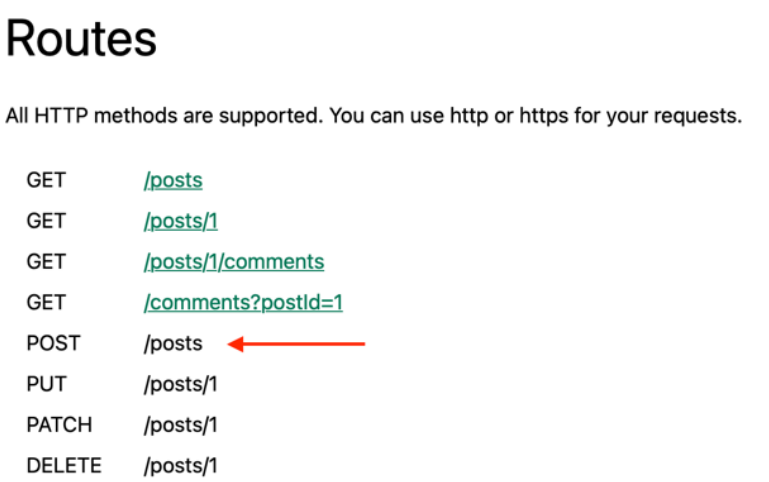
DENTRO DE LA FUNCIÓN:

`Data` - La información que se recibió en la URL.

`Status` – El estado de la consulta.

`XHR` – Objeto que se utiliza para interactuar con los servidores.

Este método nos permite realizar peticiones HTTP de tipo Post hacia una API. En este caso, vamos a consultar **JsonPlaceHolder** (https://jsonplaceholder.typicode.com), que es una API gratuita hecha para probar nuestros desarrollos. Realizaremos las consultas a la siguiente ruta:



La función post que plantearemos es la siguiente:

```
$(function() {
        $("button").click(function() {
                $.post("https://jsonplaceholder.typicode.com/posts", {
                        "userId": 4,
                        "id": 7,
                        "title": "Esto es un Post realizado con jQuery",
                        "body": "AJAX es muy util."
                }, function(data, status) {
                        console.log(data);
                        console.log(status);
                })
        })
});
```

En nuestra consola podremos ver que el post se realiza de manera exitosa, dado su estado: `success` (“éxito”) y también, vemos el objeto que fue enviado a la API.

De esta forma queda demostrado cómo utilizar los métodos AJAX de jQuery, para realizar peticiones HTTP de tipo `Get` y `Post`. En el siguiente ejercicio continuaremos profundizando acerca de esta temática.

EXERCISE 2: INTRODUCCIÓN A FETCH

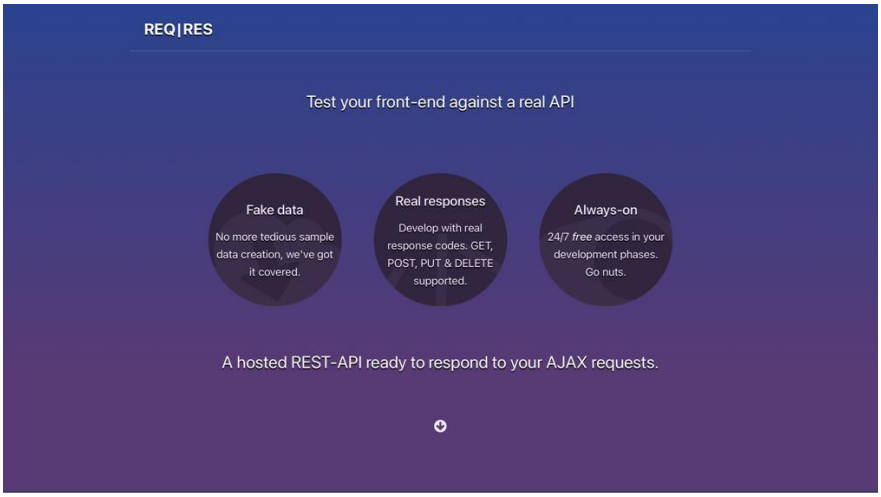
El método `fetch` proporciona una interfaz para buscar recursos (incluso a través de la red), y permite que un navegador realice solicitudes HTTP a servidores web utilizando JavaScript puro. La siguiente es su sintaxis:

```
fetch(recurso, [propiedades]);
```

En éste, el primer argumento que pasa es el `endpoint` de una API, o la ubicación de un recurso. El segundo parámetro es un conjunto de propiedades que podemos configurar, como por ejemplo, especificar si vamos a hacer una solicitud de tipo `POST`, en lugar de un `GET`. Sea cual sea la configuración que necesitamos implementar, es aquí donde se puede detallar esa información.

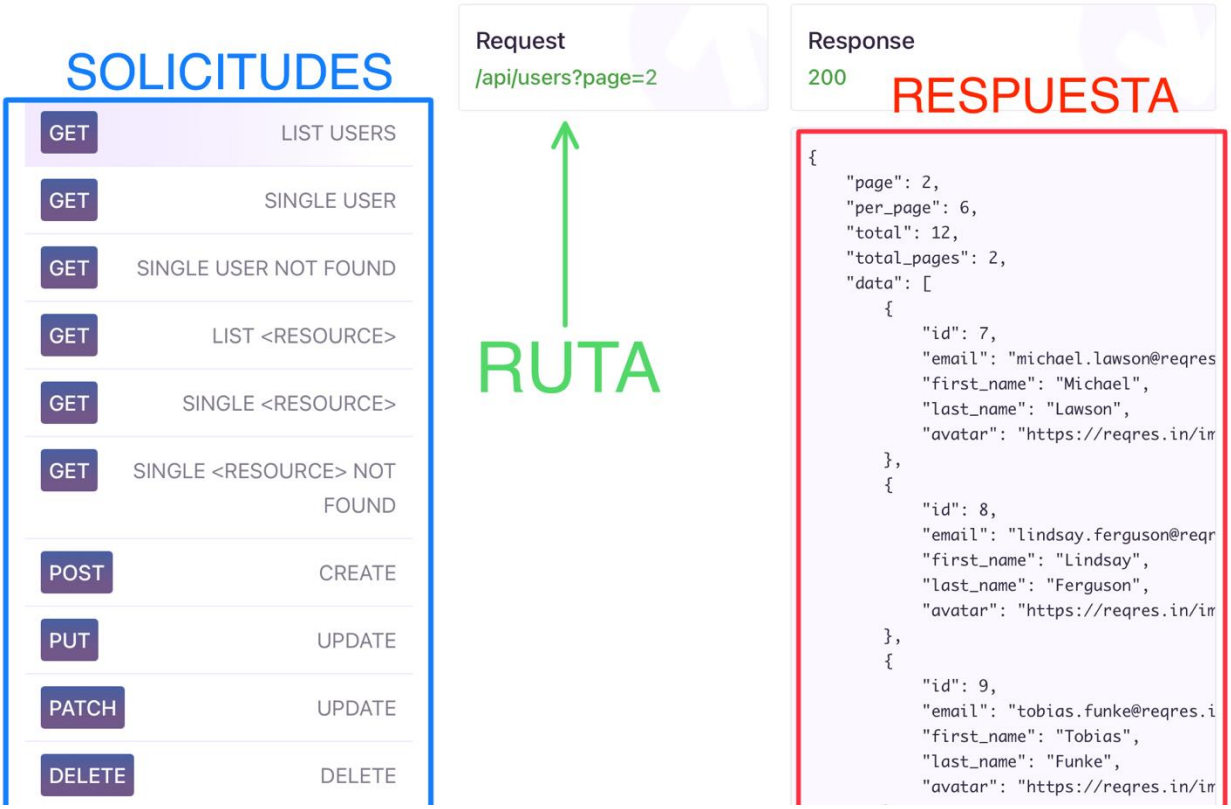
Para consumir una API utilizando `fetch`, usaremos una gratuita y sin clave llamada reqres.in (este es su sitio web oficial: https://reqres.in).

Si vas a la API en tu navegador, podrás ver la siguiente página:



Give it a try

Si bajas en la página principal, podrás ver una clara documentación visual sobre cómo se consume esta API.



Saber cómo funciona la API, es un primer paso importante para usar cualquiera de ellas, ya que cada una funciona bajo los mismos principios, pero tienen su propio conjunto de reglas o requisitos. En el caso de esta, toda la información que necesitamos se puede desglosar en lo siguiente:

- **SOLICITUDES:** esto especifica el tipo de solicitudes HTTP que podemos realizar, y lo que devuelven.
- **RUTA:** esta es la ruta específica de la API a la que tenemos que apuntar o, en términos más específicos: el endpoint al que debemos enviar nuestras solicitudes.
- **RESPUESTA:** este es el objeto que recibimos del servidor como respuesta a nuestra solicitud. En el caso de la imagen que se muestra, tenemos una matriz de objetos JSON. En esta API, todas las respuestas son objetos JSON.

Ahora que tenemos toda esta información considerada, comencemos a consumir una API con el método `fetch()`. Para empezar, solo la pasaremos sin más opciones, y la encapsularemos en un `console.log()`.

```
1 console.log(fetch('https://reqres.in/'))
```

Si ahora nos dirigimos a la consola de nuestro navegador, notaremos el siguiente resultado:

Como podemos ver, `fetch()` devuelve un objeto de respuesta que es una Promesa, por lo que se usan las palabras clave `async/await`, o expresiones `.then()`. Para visualizar la respuesta, vamos a utilizar una declaración de .then() que muestre el objeto respuesta “res” en la consola.

```
1 fetch('https://reqres.in/api/users')
2     .then(res => console.log(res))
```

Al dirigirnos a nuestro navegador, veremos el siguiente (`response`).

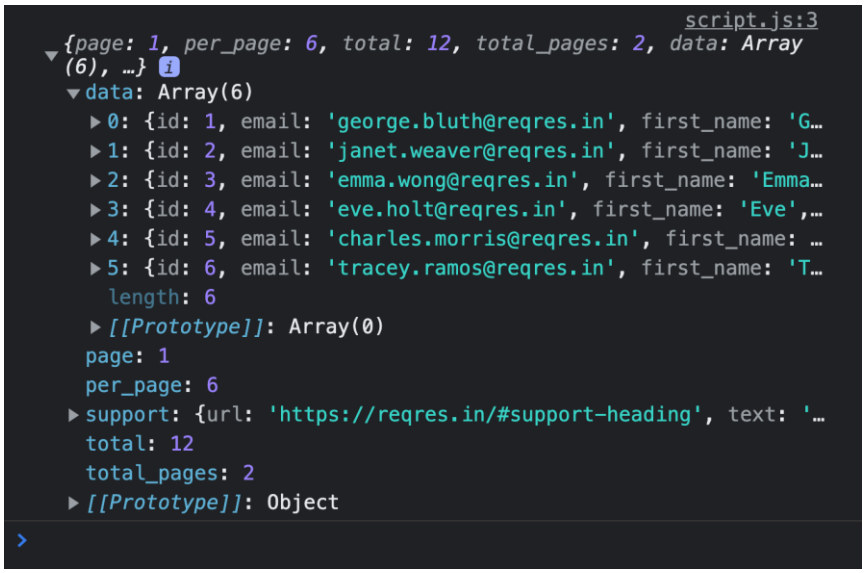
El objeto de respuesta nos otorga información acerca de la solicitud, así como también información pertinente a la respuesta, como los valores `ok:true` y `status:200` que indican que fue exitosa. Ahora bien, si nos fijamos en la respuesta, vemos que en el atributo `body` no se logra ver información, solo se muestra (…) indicando que hay datos, pero no los puede desplegar. Esto se debe a que esta API devuelve información en formato JSON, por lo que podemos modificar la respuesta esperada usando `.json()`.

```
1 fetch('https://reqres.in/api/users')
2 .then(res => res.json())
```

El método `res.json()` devuelve una Promesa, por lo que debemos aplicarle otro método `.then()` para desplegar la información de la respuesta en formato JSON. Es por eso que en el último `.then()` desplegaremos la respuesta en JSON, usando la variable `data` (en este caso usamos el nombre `"data"` porque coincide con el nombre de la propiedad en respuesta que muestra la información, pero ten en cuenta que podemos usar cualquier nombre). Dada la URL que estamos utilizando, esta ruta nos debería devolver una serie de usuarios.

```
1 fetch('https://reqres.in/api/users')
2 .then(res => res.json())
3 .then (data => console.log(data))
```

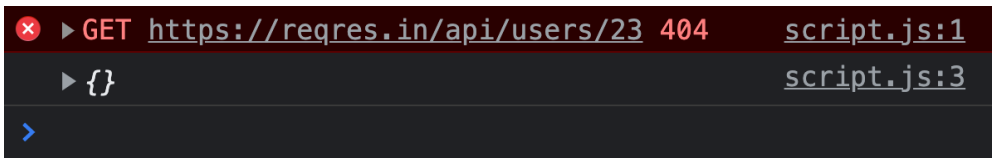
Si volvemos a nuestra consola, podremos ver bajo la propiedad `data` que efectivamente recibimos un Array de objetos de tipo “usuarios”, con seis usuarios distintos.



Hasta este momento, todas nuestras solicitudes han sido exitosas, pero ¿qué sucede cuando éstas fallan? Para ver un error, intentemos obtener un usuario aleatorio. Si buscamos acceder al usuario 23 - que no existe - deberíamos ver un código de error 404 *"no encontrado"*, seguido por la caída de nuestro código. Intentémoslo con el siguiente código:

```
fetch('https://reqres.in/api/users/23')
    .then(res => res.json())
    .then(data => console.log(data))
```

Esto da como resultado un código 404, pero el programa no ha caído porque nuestro segundo código `.then()` es exitoso.



Aunque 404 es un código de error, esto no significa que `fetch()` falló. Lo que realmente falló fue la solicitud GET. El método en sí puede fallar debido a factores externos, como si se cae la red, o si el

navegador tiene problemas para conectarse al Internet; sin embargo, otro punto débil que puede fallar en el código es nuestra respuesta. Para verificar si una respuesta está bien, podemos usar el siguiente método if:

```
fetch('https://reqres.in/api/users/23')
    .then(res => {
        if (res.ok) {
            console.log('Éxito')
            return res.json()
        } else {
            console.log("Falló")
        }
    })
    .then(data => console.log(data))
```

Si ejecutamos esto, obtendremos un mensaje “Falló”, porque es un 404, pero si ejecutamos la URL que usamos antes:


```
fetch('https://reqres.in/api/users')
    .then(res => {
        if (res.ok) {
            console.log('Éxito')
            return res.json()
        } else {
            console.log("Falló")
        }
    })
    .then(data => console.log(data))
```

Recibiremos el mensaje “Éxito”, pues la URL usada es válida.

```
Éxito                                                    script.js:4
                                                         script.js:10
▶ {page: 1, per_page: 6, total: 12, total_pages: 2, data: Array
  (6), …}
>
```

De esta forma hemos logrado consumir una API exitosamente con `fetch()`. En el próximo Exercise aprenderemos a utilizarlo con otros métodos HTTP.

EXERCISE 3: MANIPULANDO INFORMACIÓN CON FETCH

Ahora no solo queremos obtener información de un servidor, también queremos enviar información nueva, actualizarla o eliminarla. Para hacer todo esto, necesitamos usar el segundo parámetro del método de búsqueda: *las opciones*.

Lo primero que debemos hacer en las propiedades es definir un `method` o "método". Aquí podemos establecer si deseamos efectuar un "POST", "DELETE", o cualquiera de los métodos HTTP. En este caso, usaremos “POST”, ya que vamos a crear un nuevo usuario en esta API. Luego, necesitamos pasar la información necesaria para crear un nuevo usuario, que se debe colocar bajo la propiedad `body` que puede contener en sí mismo otro conjunto de propiedades como un objeto JSON.

Bajo la propiedad `body`, vamos a incorporar la propiedad `name`, y le colocaremos el nombre “Usuario 1”. Al incorporar todos estos cambios, tenemos el siguiente código:

```
1  fetch('https://reqres.in/api/users', {
2      method: 'POST',
3      body: {
4          name: 'Usuario 1'
5      }
6  }).then(res => {
7      if (res.ok) {
8          console.log('Éxito')
9          return res.json()
10     } else {
11         console.log("Falló")
12     }
13 }).then(data => console.log(data))
```

Después de ejecutarlo, la información que muestra nuestra consola puede darnos la impresión de que se ejecutó el código correctamente, pero si abrimos el objeto que se envió, podremos ver que no hay una propiedad `name`, lo cual indica que no se creó un usuario nuevo. ¿Por qué no resultó la solicitud? Esto se debe a que el `body` no se envió como corresponde, ya que `fetch()` debe enviar información en formato JSON, por lo que debemos modificarlo y agregar `JSON.stringify()` para enviar toda la información de un objeto Usuario de manera correcta.

Otra razón por la que no resultó nuestra solicitud es que nos hizo falta agregar los "encabezados" o `headers` necesarios para enviar toda la información correctamente. Para agregarlos solo necesitamos ubicar la propiedad `headers` debajo del `method`. El valor de dicha propiedad debe estar entre corchetes; y al incluir aquí la propiedad `"Content-type":"application/json"`, le estaremos diciendo a `fetch()` que estamos enviando un objeto JSON. Luego de agregar estas modificaciones, nuestro código se ve así:

```
fetch('https://reqres.in/api/users', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json'
    },
    body: JSON.stringify({
        name: 'Usuario 1'
    })
}).then(res => {
    if (res.ok) {
        console.log('Éxito')
        return res.json()
    } else {
        console.log("Falló")
    }
}).then(data => console.log(data))
```

Si ahora ejecutamos este código, veremos que hemos creado correctamente un nuevo usuario, ya que se notará claramente su propiedad de nombre `name`, y la fecha en que se creó en la propiedad `createdAt`.

```
Éxito                                                    script.js:11
                                                         script.js:16
▼{name: 'Usuario 1', id: '204', createdAt: '2021-09-24T23:34:31.727Z'} i
    createdAt: "2021-09-24T23:34:31.727Z"
    id: "204"
    name: "Usuario 1"
  ▶ [[Prototype]]: Object
>
```

De esta manera, hemos creado con éxito un nuevo usuario en nuestra API, utilizando el método **`fetch()`**, pero ¿qué tendríamos que hacer para eliminar un objeto Usuario de la API? Para ello, primero debemos identificar a dónde enviaremos nuestra solicitud HTTP. En este caso, la documentación de reqres.in establece que podemos usar un método HTTP "DELETE" en la URL https://reqres.in/api/users/2 , por lo que esa URL será el endpoint que usaremos como nuestro primer parámetro. En las opciones de **`fetch()`** solo especificaremos nuestro **`method`**, que es DELETE. A continuación, se muestra el código completo que vamos a emplear para ejecutar el método DELETE:

```
1  fetch('https://reqres.in/api/users/2', {
2      method: 'DELETE',
3  }).then(res => {
4      if (res.ok) {
5          console.log('Éxito')
6          console.log(res.status)
7      } else {
8          console.log("Falló")
9          console.log(res.status)
10     }
11 })
```

En el código que acabamos de ver, es importante destacar que lo único nuevo es **`res.status`**, el cual consiste en llamar a la propiedad **`status`** del objeto de respuesta **`res`**. Si ejecutamos nuestro código, y nos dirigimos a la consola del navegador, podremos ver que tenemos un mensaje de éxito y al mismo tiempo tenemos nuestro código de estado 204 que indica que la solicitud fue exitosa:

```
Éxito
204
>
```

Otra forma en la que podemos utilizar el método `fetch()`, es para consumir archivos. Esto se puede hacer usando la ubicación de un archivo en el lugar de la URL de la API, es decir, la ubicación del archivo que queremos consumir debe ser nuestro primer parámetro.

Para entender esto más claramente haremos un ejemplo. Para ello, crearemos un archivo de texto en la raíz de nuestro proyecto llamado *lorem.txt*. Dentro de éste pondremos el siguiente texto ficticio:

"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi quis orci id risus rhoncus mollis. Sed volutpat posuere metus, mattis tempus ex efficitur sed. Quisque in luctus nulla. Mauris cursus tempor nisl, sed ornare leo sollicitudin ac. Maecenas placerat elit vel dolor blandit, a accumsan nulla viverra. Suspendisse velit felis, ullamcorper ac dapibus sit amet, efficitur eget eros. Quisque leo leo, fermentum eget dignissim id, tincidunt et velit. Praesent fermentum velit mi, sit amet feugiat nibh eleifend at. Nam justo elit, vulputate quis tempor vel, ornare vitae mauris. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin ut finibus nulla. Duis gravida ante nisi, et faucibus massa dictum sed. Pellentesque eleifend enim in laoreet finibus.

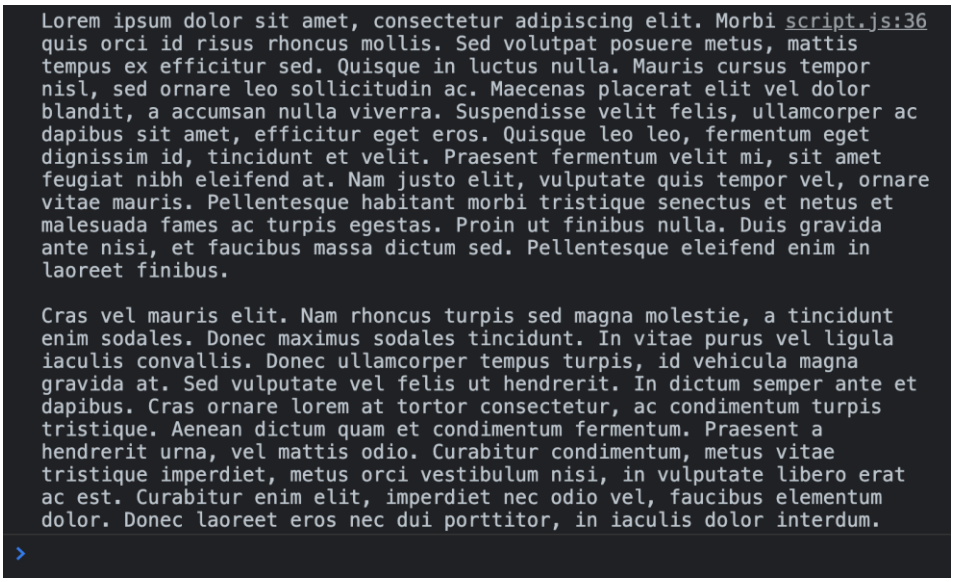
Cras vel mauris elit. Nam rhoncus turpis sed magna molestie, a tincidunt enim sodales. Donec maximus sodales tincidunt. In vitae purus vel ligula iaculis convallis. Donec ullamcorper tempus turpis, id vehicula magna gravida at. Sed vulputate vel felis ut hendrerit. In dictum semper ante et dapibus. Cras ornare lorem at tortor consectetur, ac condimentum turpis tristique. Aenean dictum quam et condimentum fermentum. Praesent a hendrerit urna, vel mattis odio. Curabitur condimentum, metus vitae tristique imperdiet, metus orci vestibulum nisi, in vulputate libero erat ac est. Curabitur enim elit, imperdiet nec odio vel, faucibus elementum dolor. Donec laoreet eros nec dui porttitor, in iaculis dolor interdum."

Ahora que tenemos nuestro archivo de texto listo, podemos continuar con el código. Dado que `fetch()` se basa en Promesas, podemos usar `async` y `await` para trabajar con esta función. Para ver esto en acción, crearemos una función `async`, que llamará a nuestra búsqueda usando la palabra clave `await`. El siguiente es el código que vamos a utilizar:

```
async function leerArchivo() {
    var x = await fetch('lorem.txt');
    var y = await x.text()
    console.log(y)
}

leerArchivo();
```

Después de guardar el resultado de `fetch()` en la variable `x`, extrapolamos el valor de texto de `x` en nuestra variable `y`, mostrándolas en nuestra consola. El resultado de todo este proceso, es el siguiente:



De esta forma, hemos adquirido las herramientas necesarias para poder interactuar con cualquier API o archivo, utilizando el método `fetch()`.