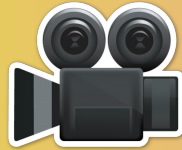




Clase 10. Vue JS

# ***Consumo de API RestFul***

***RECORDÁ PONER A GRABAR LA CLASE***





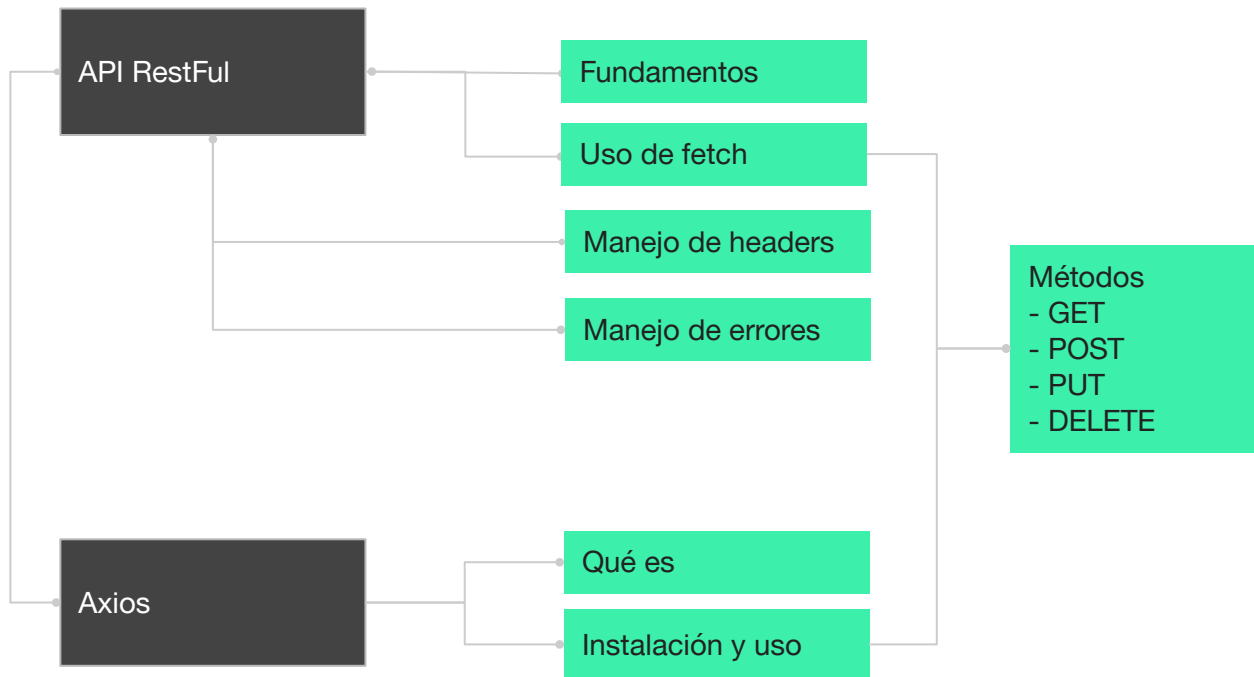
## ***OBJETIVOS DE LA CLASE***

- Consumir una API RESTful mediante la función fetch.
- Integrar la librería Axios.
- Manejar la información de headers y la gestión de errores.

# ***MAPA DE CONCEPTOS***

# MAPA DE CONCEPTOS CLASE 10

¡Para  
recordar!



# ***CRONOGRAMA DEL CURSO***

## Clase 09



### **Router y Lifecycle Hooks**



SINGLE PAGE APPLICATION



EJEMPLO EN VIVO



APLICAR LIFECYCLE HOOKS

## Clase 10



### **Consumo de API RestFul**



PETICIÓN REMOTA VÍA  
FETCH



EJEMPLO EN VIVO



PETICIONES REMOTAS VÍA  
AXIOS



SEGUNDA ENTREGA  
PROYECTO FINAL

## Clase 11



### **Introducción a Vuex**



VUEX EN VUE CLI 2



EJEMPLO EN VIVO

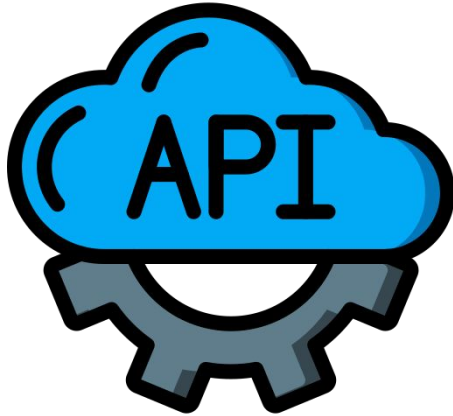
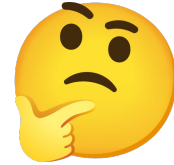


ESTRUCTURA VUEX

# ***Consumo de API REST***

# API

Primero, definamos... ¿qué es una API?



Podríamos decir que es un conjunto de protocolos y definiciones que sirven para la comunicación de datos entre aplicaciones de iguales o diferentes tecnologías.



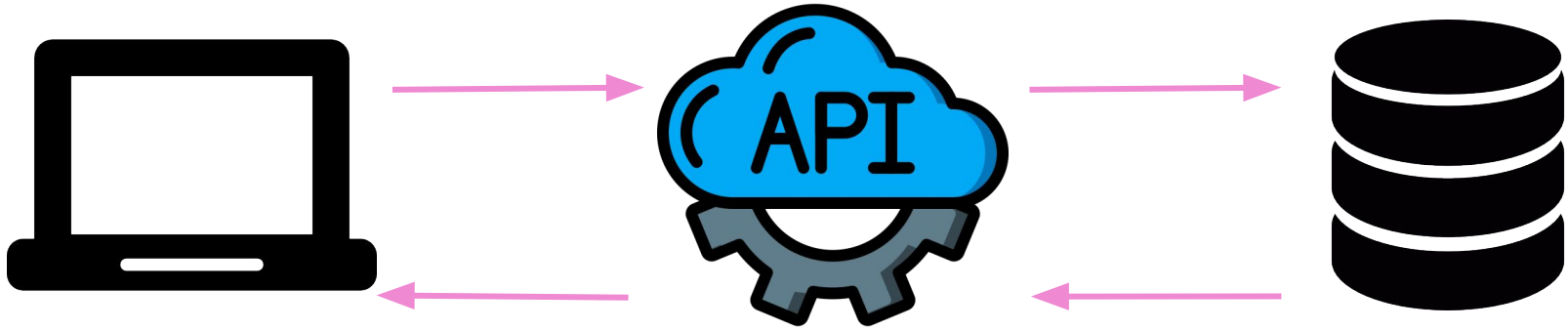
# ***API-REST***

**REST** representa un conjunto de restricciones que sirven para que las solicitudes HTTP cumplan con las directivas definidas en su arquitectura

👉 API Rest nació para unificar la forma en la que se entrega la información, más allá de qué tecnología se usa de fondo para crear la API o qué tecnología se usa en el frontend para pedir dicha información.

👉 Básicamente se trata de interactuar con los métodos HTTP

# ***API-REST***



Cliente

API-REST

Base de datos

# ***Mockapi***

👉 Durante nuestra anterior After class aprendimos que Mockapi es una aplicación del tipo SaaS que nos permite configurar una APIRestFul y definir endpoints para que una aplicación pueda acceder y trabajar con datos remotos.

A su vez, creamos nuestro API Endpoint.

Veamos a continuación cómo consumir una API RESTful mediante la función fetch.

# ***FETCH API***

***¿Qué es?***

# ***Fetch API***



Es una interfaz que nació con ES6, la cual permite trabajar con recursos remotos tal como lo haces con **Ajax**. Podemos definirlo como una evolución de este último.



# ***Fetch API***

A diferencia de las tecnologías Ajax originales, **fetch trabaja de forma asincrónica** y se basa en **Promesas JS** lo cual lo hace más efectivo, además de simplificar notablemente su código.



***¡Veamos su evolución!***



***CODER HOUSE***



# Fetch API



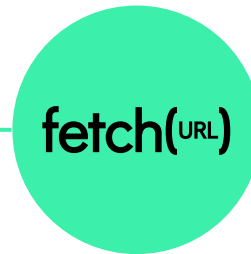
2002

**XMLHttpRequest** fue el objeto JS que introdujo el paradigma Ajax, permitiendo consumir datos remotos en un formato estándar.



2011

**Jquery** lo tomó y mejoró, haciéndolo mucho más ágil y amigable en cuanto a sintaxis, e integrando control de errores y headers simplificados.



2015

En ES6 llegó **fetch API**, simplificando la complejidad media de sus ancestros y basando su operativa en Promesas y asincronismo.

# ***VENTAJAS DE Fetch API***

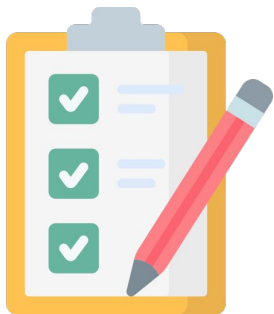
Entre sus múltiples ventajas, podemos destacar 🙌



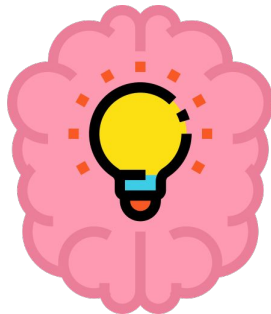
- Integra objetos **Request** y **Response** de forma genérica
- Proporciona definiciones para **CORS** y encabezados HTTP de origen
- Está disponible tanto en window como en **WorkerGlobalScope**



# ***VENTAJAS DE Fetch API***



- Cuenta con múltiples métodos para convertir el contenido del cuerpo
- Ante respuestas inesperadas, permite cancelar una petición previo su finalización, utilizando objeto **JS AbortController**



# ***FETCH API: SINTAXIS***

# SINTAXIS

Ejemplo  
en vivo



Fetch resuelve en muy pocas líneas de código: la petición de datos remotos, el estado del servidor (**status**), la conversión de la respuesta (**response**) a un formato válido (**JSON**), para finalmente interactuar con los mismos de acuerdo a nuestra necesidad.

```
fetch(URL)
  .then((response) => response.json())
  .then((json) => {
    json.forEach(element => { console.table(element) })
  })
```

**CODER HOUSE**

# SINTAXIS

Ejemplo  
en vivo



Gracias a su estructura basada en **JavaScript Promises**, podemos integrar también un **control de errores**, además de la ejecución de sentencias satélites, en la misma línea de control de la petición, evitando así “desarmar” la estructura de este código con un **try-catch-finally**.

```
...  
.then((json) => {  
    json.forEach(element => { console.table(element) })  
})  
.catch(err => console.error(`Ocurrió un error ${err}`))  
.finally(ok => console.log(`Esto se ejecuta obtengamos o no los datos`))
```

**CODER HOUSE**

# ***SINTAXIS: CONTROL DE ERRORES***

Ejemplo  
en vivo



Si queremos limitar, por ejemplo, que una petición responda en máximo 2 segundos, integramos el objeto **AbortController** a la llamada **fetch()**.

```
const controller = new AbortController()
setTimeout(() => {
  controller.abort()
}, 2000);
```

# SINTAXIS: CONTROL DE ERRORES

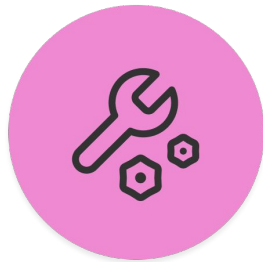
Ejemplo  
en vivo



Fetch sabe cómo trabajar con este, escuchando sus eventos a través de la propiedad **signal**. Así podremos manejarlo, desde **catch()**, según el tipo de error informado.

```
const response = await fetch(URL, {signal: controller.signal})
  .then(...) //manejo de datos de fetch()
  .catch(err => err.name == 'AbortError'
    ? console.log('Manejo el error...') //Función o manejo de error
    : console.log(`${err}`) //Veo el error real
  )
```





# ***PETICIÓN DE DATOS REMOTOS CON FETCH***

Crea una petición fetch a un endpoint remoto.

**Tiempo estimado:** 20 minutos



# ***PETICIÓN DE DATOS REMOTOS CON FETCH***

Utiliza algún endpoint en la nube, gratuito, como [JSONPlaceholder](#) o [RandomUsers](#), para peticionar datos remotos utilizando fetch.

Lista el “*response*” en la consola JS mediante `console.table()` e integra un control de errores para cambiar, a posteriori, la URL original del servicio forzándolo así a fallar y que se ejecute dicho control.

**Tiempo estimado:** 20 minutos



***BREAK***

**¡5/10 MINUTOS Y VOLVEMOS!**

# ***FETCH EN VUE***

# ***Fetch EN VUE***




Veamos cómo integrar a **fetch** en nuestras aplicaciones  
Vue.

Vamos a elaborar un proyecto que nos permita consumir  
los datos que generamos en nuestro perfil de **Mockapi**.

# Fetch EN VUE



Tenemos nuestro backend resuelto .

**Armamos la Vista HTML**, incluyendo en éste **un botón** el cual será el disparador de la petición de datos remotos, y **un contenedor** del tipo `<ul>` donde listaremos los mismos una vez recibidos.

***HTML***

...

```
<body>
```

```
<div id="app">
```

```
  <h1>Nuestros Cursos</h1>
```

```
  <div center>
```

```
    <button>CURSOS</button>
```

```
  </div>
```

```
<br>
```

```
<div>
```

```
  <ul>
```

```
    <!--aquí, los datos :)-->
```

```
  </ul>
```

```
</div>
```

```
</div>
```

```
</body>
```

# Fetch EN VUE

Ejemplo  
en vivo



En la estructura **<body>** de la Vista HTML, definimos una etiqueta del tipo título, un elemento **<button>** y la etiqueta **<ul>**.

No olvides englobar todos ellos dentro de un **<div>** cuyo atributo **id** se referencie en Vue.

**CODER HOUSE**



***JS / VUE***

# Fetch EN VUE

Ejemplo  
en vivo



```
const app = new Vue(  
  {  
    el: "#app",  
    data: {  
  
    },  
    methods: {  
  
    }  
  }  
)
```

Ahora armamos la estructura base de Vue. Definiremos en ésta **un array vacío**, la URL con el endpoint, y **un método** que llamaremos **getUsers()**, que se ocupará de invocar a fetch para obtener los datos.

Por último, cargamos la respuesta recibida en el array vacío, y listamos su resultado en la **Consola JS**.



# Fetch EN VUE

```
const app = new Vue(  
  {  
    el: "#app",  
    data: {  
      URL: "https://6c.../cursos",  
      cursos: []  
    },  
    methods: {  
      async getCursos(){  
        //Aquí, fetch :)  
      }  
    }  
  }  
)
```

Una vez definido el **array** y la **URL** en el apartado **data: { }** además del método asincrónico que hará la petición de datos, solo nos queda armar la estructura de fetch.

**¡Resolvámoslo a continuación!**

# Fetch EN VUE

Ejemplo  
en vivo



```
await fetch(this.URL)
  .then(response => response.json())
  .then(data => {this.cursos = data})
  .then(data => {console.table(this.cursos)})
  .catch(err => console.error(`${err}`))
  .finally(() => {
    console.log("Finalizó la petición de datos remotos...")
  })
```

Acompañamos la petición fetch con la sentencia **await**, para controlar el asincronismo. Convertimos los datos que llegan en **response** al formato JSON, los guardamos en el array **cursos**, y finalmente los mostramos con **console.table()**.

**CODER HOUSE**

# ***VUE en HTML***

# Fetch EN VUE

Ejemplo  
en vivo



Asociemos el método **getCursos()** al elemento HTML **<button>**,  
utilizando la directiva **@click** como disparador del mismo.

```
...  
<button @click="getCursos">CURSOS</button>  
...
```



# Fetch EN VUE

Ejecutemos nuestro proyecto en el servidor web, y pulsemos el botón **Cursos**, para ver qué ocurre en la **Consola JS**. 🙌

vue-devtools Detected Vue v2.6.14 backend.js:2237

vuefetch.js:12

(índice)	id	nombre	creado	duracion	__ob...	val...	dep	vmCount
0	1	'JavaScript'	'2014-05-15'	16	Obse...			
1	2	'React'	'2017-08-30'	18	Obse...			
2	3	'Angular'	'2021-09-01'	18	Obse...			
3	4	'Vue'	'2021-09-01'	16	Obse...			
4	5	'Node JS'	'2020-07-14'	24	Obse...			
__ob__						Arr...	Dep	0

► Array(5)

Finalizó la petición de datos remotos... vuefetch.js:15

¡Conseguimos acceder a los datos remotos! 😎

# Fetch EN VUE

Ejemplo  
en vivo



Creemos a continuación un elemento HTML `<li>` el cual iteraremos mediante un renderizado de lista **v-for**.

```
...
<div class="container">
  <ul>
    <li v-for="curso in cursos" :key="curso.id">
      {{ curso.nombre }}
    </li>
  </ul>
</div>
...
```

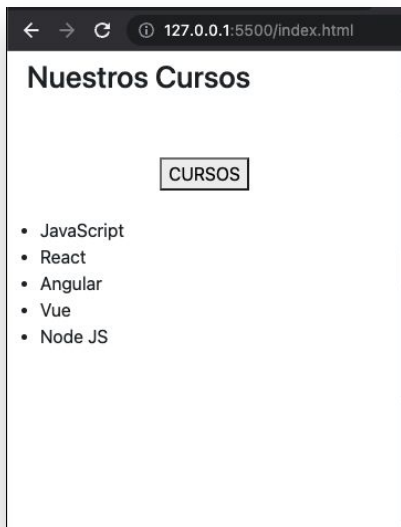
index.html X

Mostraremos en pantalla solo el nombre del curso.





# Fetch EN VUE



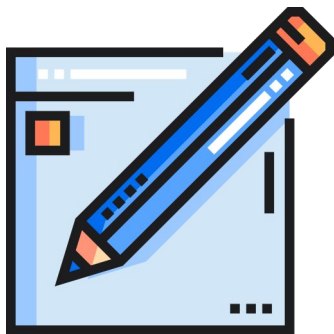
👉 Obtendremos así el total de cursos almacenados en **mockapi.io**, visualizados a través de la lista desordenada.

Te invitamos a “*vitaminizar*” este proyecto integrando el componente HTML [list-group with badges](#) de BootStrap, para mejorar notoriamente su estética y mostrar más de la información obtenida mediante fetch. 👉



# ***Método POST***

# ***MÉTODO POST***



Veamos cómo agregar contenido a través del mismo endpoint de nuestro backend. Para ello, utilizaremos el método **POST** en lugar de **GET**.

Pero, antes de atender el código en Vue, probemos la funcionalidad del endpoint a través de la extensión **Thunder Client**.

# MÉTODO POST

Ejemplo  
en vivo



1

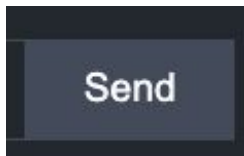
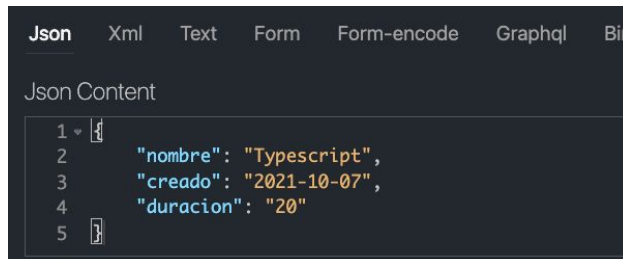
2

3



Ingresamos la URL de nuestro endpoint en un nuevo request, cambiando el método **GET**, por **POST**.

Debajo, seleccionamos **JSON** como contenido a enviar, y cargamos el curso obviando su **ID**.



Pulsamos el botón **SEND**.

# MÉTODO POST

Ejemplo  
en vivo



Como resultado, obtendremos una respuesta del servidor, con el **response** de la información enviada, junto al **ID** asignado al nuevo curso.

Es el servidor quien resuelve el ID que debe obtener cada nuevo registro que agreguemos en el backend.

```
Status: 201 Created   Size: 70 Bytes   Time: 1.57 s

Response   Headers 11   Cookies   Test Results

1  {
2    "nombre": "Typescript",
3    "creado": "2021-10-07",
4    "duracion": "20",
5    "id": "6"
6  }
```

Refresca la App de Vue para confirmar que se visualice el nuevo registro.

# ***POST en Vue mediante Fetch***



# POST EN VUE

```
async postCurso() {  
  const cursoData = {  
    "nombre": "Deno",  
    "creado": "2021-10-25",  
    "duracion": "24"  
  }  
  
  const encabezado = {  
    method: "POST",  
    headers: {  
      "Content-Type": "application/json;",  
    },  
    body: JSON.stringify(this.cursoData)  
  }  
  
  await fetch(this.URL, encabezado)  
    .then((response) => response.json())  
    .then((json) => console.log(json))  
    .catch((err) => console.log(err))  
}
```

- Armas un nuevo método asincrónico para realizar el POST
- Defines una estructura JSON con los datos a enviar al endpoint
- Defines un encabezado con parámetro, **method**, y **content-type**, con el tipo de información a enviar
- Agregas el parámetro **body** donde envías, en formato string, la estructura JSON de la información a guardar



# POST EN VUE

```
async postCurso() {  
  const cursoData = {  
    "nombre": "Deno",  
    "creado": "2021-10-25",  
    "duracion": "24"  
  }  
  
  const encabezado = {  
    method: "POST",  
    headers: {  
      "Content-Type": "application/json;",  
    },  
    body: JSON.stringify(this.cursoData)  
  }  
  
  await fetch(this.URL, encabezado)  
    .then((response) => response.json())  
    .then((json) => console.log(json))  
    .catch((err) => console.log(err))  
}
```

- Finalmente, invocas la petición **fetch()**, sumándole el parámetro **encabezado**, que tiene la información competente para el servidor
- Puedes invocar nuevamente a la promesa **then()**, para recibir los datos del servidor, junto con el **ID** asignado al contenido enviado, y **catch()** para trabajar algún error que surja



# POST EN VUE

Ejemplo  
en vivo



```
const encabezado = {  
  method: "POST",  
  mode: 'cors',  
  cache: "no-cache",  
  credentials: 'same-origin',  
  redirect: 'follow',  
  referrerPolicy: 'no-referrer',  
  headers: {  
    "Content-Type": "application/json;",  
  },  
  body: {data}  
}
```

El encabezado de todo **método HTTP** puede variar, e incluir muchas más información en éste, como ser:

- Definir el uso de CORS
- Cachear o no la información enviada
- Información de identificación de usuario ante el servidor
- entre otras cosas más

El orden es indistinto, como también el envío de información extra. El endpoint se ocupa de usar solo la información que necesite.

# ***Método PUT***

# MÉTODO PUT

Ejemplo  
en vivo



**PUT** varía ligeramente de **POST**. Los endpoint pueden interpretar -según cómo hayan sido programados- de dos formas diferentes la URL a la cual petitionamos pasándole un parámetro adicional.

En algunos casos, se resuelve mediante **/datoAbuscar**.

```
https://615ceedeCH21033001773636d.mockapi.io/cursos/6
```

Mientras que, en otros, debemos informarlo mediante **?campo = valor**.

```
https://615ceedeCH21033001773636d.mockapi.io/cursos/?id=6
```

# MÉTODO PUT

Ejemplo  
en vivo



1

PUT <https://615ceedeCH21033001773636d.mockapi.io/cursos/6>

Volvamos a probar el funcionamiento del endpoint modificando su método y agregando a su URL, de acuerdo a la documentación oficial de **Mockapi**, la **/** más el **ID** del registro que deseamos modificar.

2

Hagamos algún cambio mínimo de la información del registro seleccionado...

Json Content

```
1 {  
2   "nombre": "TypeScript",  
3   "creado": "2021-10-10",  
4   "duracion": "20"  
5 }
```

3

Send

Y pulsemos finalmente el botón **SEND**.

# ***Método DELETE***

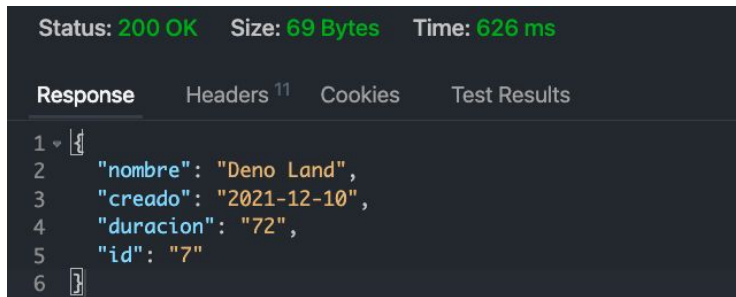
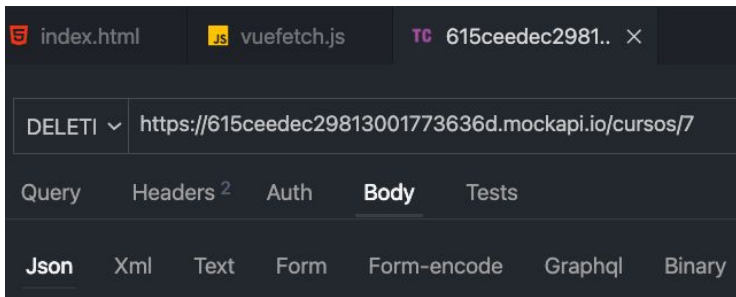
# MÉTODO DELETE

Ejemplo  
en vivo



**DELETE** puede tener la misma definición que encontramos en **PUT**: o definimos el registro a eliminar mediante **/id**, o lo parametrizamos en la URL del endpoint mediante **?id=7**.

Usemos **Thunder Client** para eliminar algún registro existente. Es probable que el mismo endpoint nos devuelva la información eliminada en el formato de datos que manejamos.



**TIP:** creemos un registro cualquiera mediante **POST**, para luego eliminarlo.

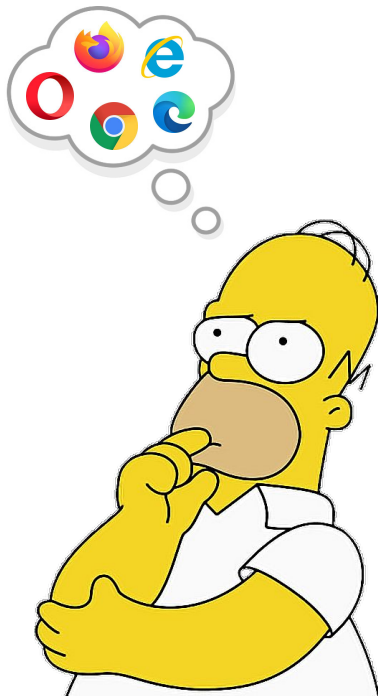
# ***AXIOS en Vue***

# AXIOS EN VUE

**Fetch()** es parte de **Vanilla JS** y puede **usarse con Vue**, aunque no está integrado a su núcleo. Vue quiere su Core lo más pequeño posible dejando “*el trabajo pesado*” a librerías externas.

Ahí es donde llega [Axios](#), para cumplir un papel fundamental:

**Simplificar el uso de fetch;** (sí, aún más simple), y asegurar **compatibilidad con todos los web browsers...** sí, todos.





# AXIOS EN VUE

Ejemplo  
en vivo



Para integrar Axios en tus proyectos, puedes hacerlo de dos formas:

- 1) Sumarlo vía CDN directamente en el HTML

```
<script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

- 2) Integrarlo utilizando la herramienta NPM

```
:$> npm install axios
```

- 3) Recuerda instanciarlo luego, previo a usarlo

```
const axios = require('axios')
```

# ***Petición GET con AXIOS***

# PETICIÓN GET CON AXIOS

Ejemplo  
en vivo



Su estructura para peticionar datos casi no varía respecto a fetch. Lo que sí nos evita Axios es convertir a JSON la respuesta del endpoint, dado que ya nos devuelve un array de datos en dicho formato.

```
axios.get(this.URL)
  .then((response) => {console.table(response.data)})
  .catch((err) => {console.error(`${err}`)})
```

El resultado:

(índice)	id	nombre	creado	duracion
0	1	'JavaScript'	'2014-05-15'	16
1	2	'React'	'2017-08-30'	18
2	3	'Angular'	'2021-09-01'	18
3	4	'Vue'	'2021-09-01'	16
4	5	'Node JS'	'2020-07-14'	24

► Array(5)

# ***POST con AXIOS***

# POST CON AXIOS

Ejemplo  
en vivo



POST también se simplifica con Axios. Si no necesitamos pasarle **headers** al endpoint, directamente podemos pasar el JSON con la información a guardar.

Axios asigna el mismo al **body** y resuelve el encabezado faltante.

```
const nuevoCurso = {  
  "nombre": "Deno",  
  "creado": "2020-12-01",  
  "duracion": 25  
}  
axios.post(this.URL, nuevoCurso)  
  .then((response) =>  
    {console.table(response.data)})  
  .catch((err) => {console.error(`${err}`)})
```



(índice)	Valor
nombre	'Deno'
creado	'2020-12-01'
duracion	25
id	'7'

► Object

# ***PUT y DELETE en AXIOS***

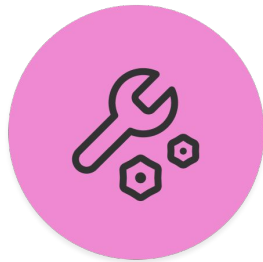


# ***PUT y DELETE EN AXIOS***

También disponemos de los métodos **PUT** y **DELETE**, los cuales operan sin variar mucho a lo que vimos hasta ahora. Por supuesto que ambos aceptan también un **header** predefinido a través de un segundo parámetro dentro de sus métodos.

```
// Método PUT
const cambioCurso = {
  "id": 7,
  "nombre": "Deno JS",
  "creado": "2021-12-01",
  "duracion": 25
}
axios.put(this.URL, cambioCurso)
  .then((response) => {console.table(response.data)})
  .catch((err) => {console.error(`${err}`)})
```

```
// Método DELETE
const idToDelete = 8
const deleteURL = `${this.URL}/${idToDelete}`
axios.delete(deleteURL)
  .then((response) => {console.table(response.data)})
  .catch((err) => {console.error(`${err}`)})
```



# ***MODIFICAR DATOS REMOTOS CON AXIOS***

Crea peticiones Axios a tu endpoint.

Tiempo estimado: 15 minutos



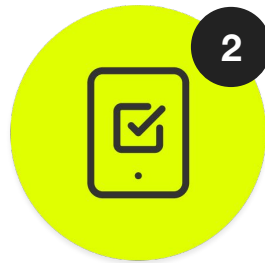


# ***MODIFICAR DATOS REMOTOS CON AXIOS***

Realiza una o más peticiones sobre el endpoint, utilizando Axios, y su representación gráfica, dentro de tu proyecto Vue.

Emplea las directivas apropiadas y ten presente de hacer, al menos, una petición sobre los datos remotos: (**PUT**, **POST**, o **DELETE**), tomando la información a parametrizar desde uno o más campos de formulario.

Cuentas con **15 minutos** para realizar esta actividad.



# ***SEGUNDA ENTREGA DEL PROYECTO FINAL***

Deberás agregar a **tu proyecto** las modificaciones que incorporen un backend.

# SEGUNDA ENTREGA DEL PROYECTO FINAL

**Formato:** Archivo en formato ZIP o link a Drive, Onedrive, Dropbox o Github con el proyecto nombrado de la siguiente forma: "Proyecto2 + Apellido".

**Sugerencia:** Si tienes comentarios que realizar, agrégalos en el **readme.md** de Github o en un archivo **leame.txt** dentro de la carpeta raíz del proyecto.

Proyecto  
Final



Debes continuar trabajando sobre la entrega intermedia 1, presentada anteriormente.

## >>Objetivos Generales:

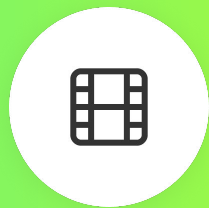
1. Sumar un backend a tu proyecto con login e interacción total sobre el endpoint.
2. Integrar Axios para trabajar contra los datos de tu endpoint.

## >>Objetivos Específicos:

1. Definir formularios de login y registro de usuario con sus respectivas validaciones.
2. Integrar Axios para consumir los recursos desde tu backend en mockapi.
3. Crear un Login y Registro de usuarios utilizando los métodos GET y POST.
4. Crear un recurso en el backend para listar productos o servicios, incorporando los métodos (GET, POST, PUT, DELETE).
5. Crear un último recurso que será el carrito, integrando GET y POST para realizar y revisar pedidos.
6. Crear una Vista de login de usuario con componentes de login y registro, hardcodeando el acceso (para corregir más fácilmente).
7. Crear Router de Vista que seleccione los componentes a representar, una vez logueado, según el perfil de usuario.  
Tendrás que crear un usuario ADMIN (*ABM de productos e Información general*), y un usuario CLIENTE (*listado de productos y carrito*)

## >>Se debe entregar:

Tu proyecto funcional, el cual será corregido por tu tutor.



***¿QUIERES SABER MÁS? TE DEJAMOS  
MATERIAL AMPLIADO DE LA CLASE***



- [Métodos HTTP](#) | **Mozilla Developer Network**
- [Documentación oficial de mockapi](#) | **Mockapi.io**
- [Web Oficial de JSONPlaceholder](#) | **JSONPlaceholder**
- [Documentación de Randomusers](#) | **Random Users**
- [Documentación de Axios](#) | **Node Package Manager**
- [Web Oficial de Axios](#) | **Axios HTTP**

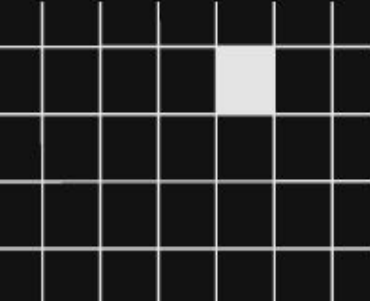
***¿PREGUNTAS?***





# ***¡MUCHAS GRACIAS!***

Resumen de lo visto en clase hoy:

- Fundamentos de API Restful
  - Uso e implementación de fetch
  - Uso e implementación de Axios
- 



***OPINA Y VALORA ESTA CLASE***



***#DEMOCRATIZANDO LA EDUCACIÓN***

***CODER HOUSE***