

# AJAX - REST

---

**{ JSON }**

ES7 incorpora la interfaz **fetch()**

```
let promise = fetch(url);  
promise.then(response => ...do something... )
```

O la versión corta

```
fetch(url).then(response => ...do something... )
```

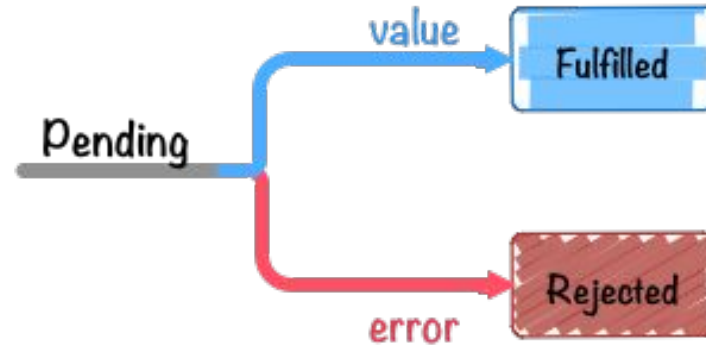
El uso más simple de `fetch()` toma un argumento (la ruta del recurso que se quiera traer) y **el resultado es una promesa** que contiene la respuesta (un objeto Response)

# Terminología

REPASO

Una promesa tiene 4 estados

- Cumplida (*fulfilled*)
- Rechazada (*rejected*)
- Pendiente (*pending*)
- Finalizada (*settled*)



Término **then**

Usando funciones

```
hazAlgo(exitoCallback, falloCallback);
```

Usando promesas

```
hazAlgo().then(exitoCallback, falloCallback);
```

```
hazAlgo().then(exitoCallback).catch(falloCallback);
```

# Cómo funciona fetch



REPASO

Ahora, queremos ver el contenido del archivo

```
fetch('/file.html')
  .then(function(r){
    return r.text()
  })
  .then(function(html) {
    console.log(html); // Contenido del archivo disponible
  })
  .catch(function(e) {
    console.log("Booo");
  })
```

Esperamos a que resuelva la promesa de Fetch pasando una función al método **then()**.

# fetch().then().then()



¿Qué está ocurriendo en cada llamado a la función **then()**?

```
fetch('/file.html')  
  .then(function(r){  
    return r.text()  
  })  
  .then(function(html) {  
    console.log(html);  
  })  
  .catch(function(e) {  
    console.log("Booo");  
  })
```

Respuesta de la solicitud fetch

Procesamiento de la respuesta  
(Nos da otra promesa)

Respuesta procesada

Error de conexión

# Con await/async

REPASO

```
async function load2(event) {
  event.preventDefault();

  let container = document.querySelector("#use-ajax");
  container.innerHTML = "<h1>Loading...</h1>";

  try {
    let response = await fetch(url);
    if (response.ok) {
      let t = await response.text();
      container.innerHTML = t;
    }
    else
      container.innerHTML = "<h1>Error - Failed URL!</h1>";
  }
  catch (response) {
    container.innerHTML = "<h1>Connection error</h1>";
  };
}
```

El valor de retorno de la promesa de Fetch es un **objeto Response** con información del request realizado

Cada respuesta puede tener datos en su cuerpo (HTML, texto, imagenes, JSON, etc)

Podemos especificar el tipo de contenido y cómo debe ser tratado, **todas estas operaciones dan otra promesa**

- `res.text()`
- `res.blob()`      `// Se usa para media: imagenes, audio, video`
- `res.json()`
- otros



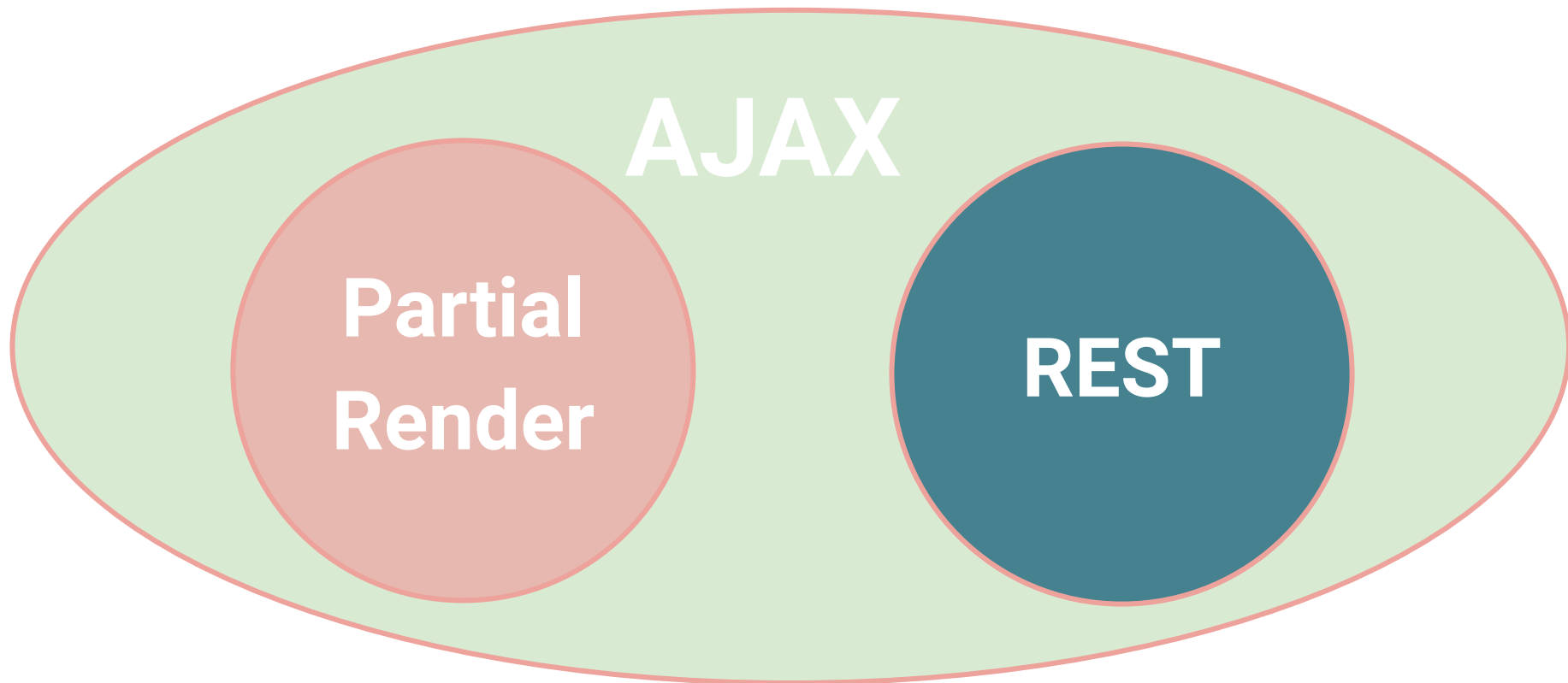
# AJAX REST

Información en formato JSON

# Estilos de AJAX

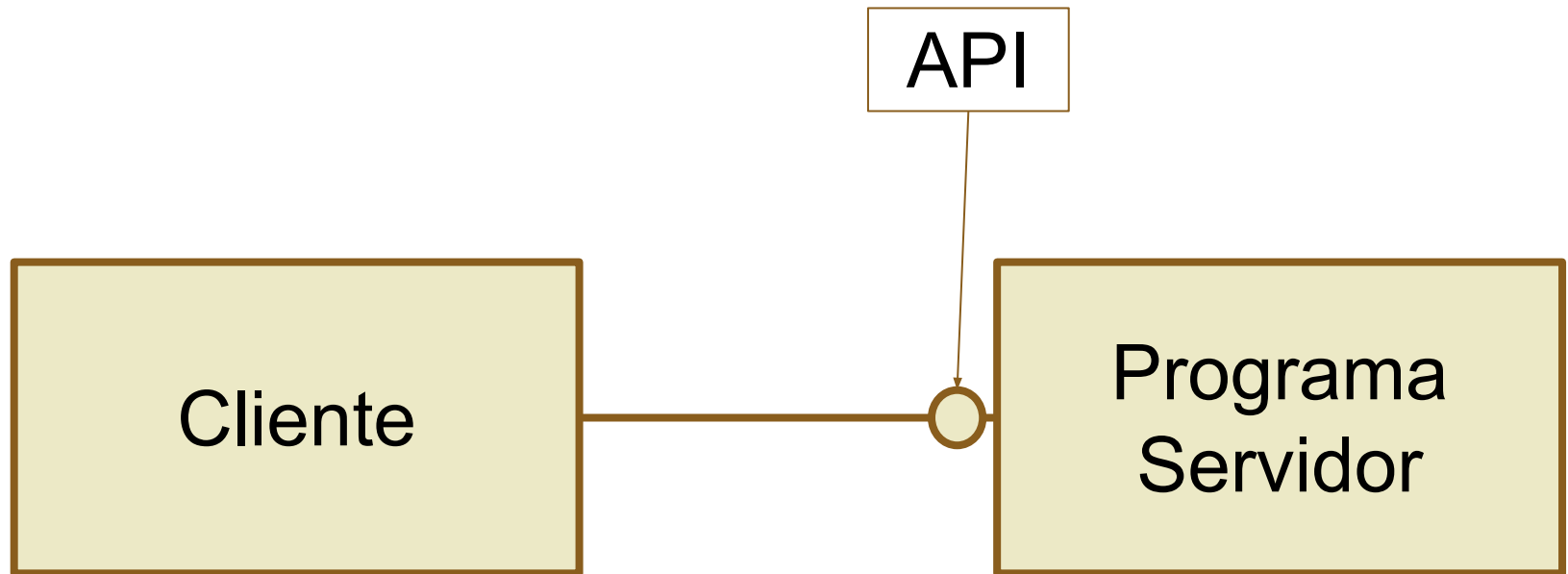
---

- Partial render de páginas
  - Cargar un fragmento de HTML y mostrarlo en un DIV.
- Servicio REST
  - Cargar un objeto JSON y procesarlo del lado del cliente con Javascript



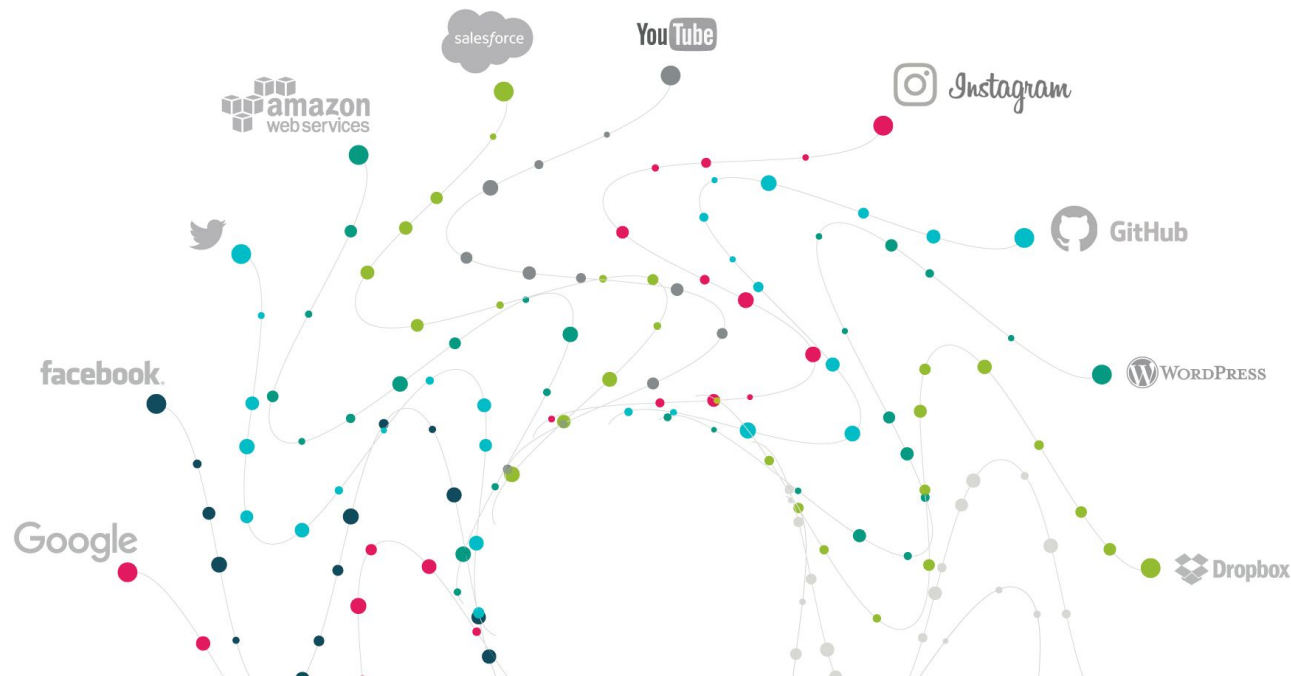
# API

- Una API es una interfaz que nos da una aplicación para comunicarnos con ella



# REST

- REST: REpresentational State Transfer, es un tipo de arquitectura de desarrollo web que se apoya totalmente en el estándar HTTP.
- Es el tipo de arquitectura más natural y estándar para crear APIs para servicios orientados a Internet.
- La mayoría de las APIs REST usan JSON para comunicarse.



# JSON (Repaso)

REPASO

## JavaScript Object Notation

Formato ligero para el intercambio de datos.

Alternativa a XML como representación de objetos.

```
let objeto =  
  {  
    "propiedad": valor,  
    "propiedad2": valor2,  
    "arreglo": [val1, val2]  
  }
```

# REST

---

- Se asocian URLs a recursos.
- Al que se puede acceder o modificar mediante los métodos del protocolo HTTP.
- Se basa en acciones (llamadas verbos) que manipulan los datos.
  - POST: Crear un recurso
  - GET: Obtener uno o muchos recursos
  - PUT: Actualizar uno o muchos recursos
  - DELETE: Borrar un recurso
- Se utilizan los errores del protocolo HTTP.
  - 200 ok, 404 not found, etc.

# API REST - EJEMPLO

---

- **GET** /facturas (en genérico /facturas)
  - Acceder al listado de facturas
- **POST** /facturas (en genérico /facturas)
  - Crear una factura nueva
- **GET** /facturas/123 (en genérico /facturas/:id\_fact)
  - Acceder al detalle de **una** factura
- **PUT** /facturas/123 (en genérico /facturas/:id\_fact)
  - Editar la factura, sustituyendo la **totalidad** de la información anterior por la nueva.
- **DELETE** /facturas/123 (en genérico /facturas/:id\_fact)
  - Eliminar la factura

# Manejo de errores en REST

---

## **Se pueden utilizar los errores del protocolo HTTP:**

- 200 OK Standard response for successful HTTP requests
- 201 Created
- 202 Accepted
- 301 Moved Permanently
- 400 Bad Request
- 401 Unauthorised
- 402 Payment Required
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 500 Internal Server Error
- 501 Not Implemented



## Servicio web-unicen

---

- Vamos a usar un servicio que creamos desde la cátedra.
  - URL: `web-unicen.herokuapp.com`
- Este servicio:
  - Guarda información con el siguiente formato:
    - Id: Es autogenerado (no se pasa al crearlo).
    - Thing: **Un objeto JSON.**
  - En la URL decimos nuestro grupo y que vamos a guardar
    - `/catedraweb/notas` para especificar que el grupo *catedraweb* guarda *notas*

## Usar ID de grupos únicos:

- número de grupo,
- inicial/es y apellido/s
- marca del sitio



**/api/groups/1/tpespecial**

**/api/groups/catedraweb/tpespecial**

**/api/groups/grupo01/tpespecial**

**/api/groups/javier/tpespecial**



**KEEP  
CALM  
AND  
LET'S  
CODE**

# AJAX REST

GET: Obteniendo datos

¿Cómo consulto la información?

- Consulta por la colección
  - Método: GET
  - URL:

`https://60aab45166f1d000177731ea.mockapi.io/api/:endpoint`

- Consulta por ID
  - Método: GET
  - URL:

`https://60aab45166f1d000177731ea.mockapi.io/api/:endpoint/:id`

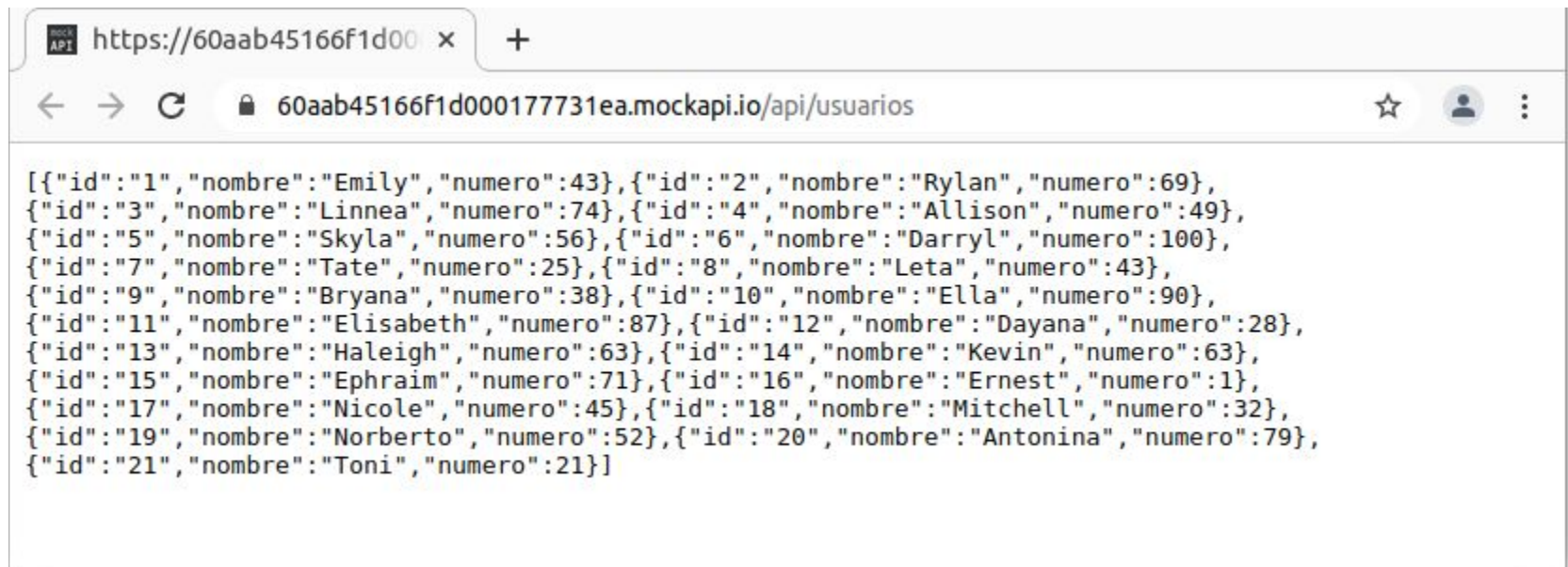
# Recibir JSON

```
try {  
  const url = 'https://60aab45166f1d000177731ea.mockapi.io/api/usuarios';  
  let res = await fetch(url);  
  let json = await res.json();  
  console.log(json);  
} catch (error) {  
  console.log(error);  
}
```

Al ejecutar `res.json()` se parsea (“compila”) a un objeto automáticamente.

# Ver la respuesta JSON

- El navegador por defecto siempre hace GET para bajar las páginas
- Si ponemos la URL en el navegador vemos directamente el JSON (aunque solo nos sirve para GET, no para otros métodos de HTTP)



A screenshot of a web browser window. The address bar shows the URL `https://60aab45166f1d00177731ea.mockapi.io/api/usuarios`. The page content displays a JSON array of 21 user objects. Each object has three fields: `id`, `nombre`, and `numero`. The data is as follows:

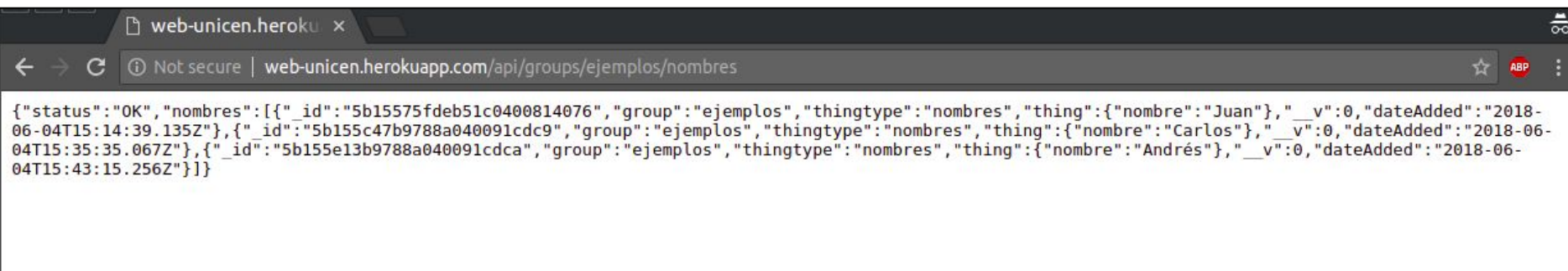
```
[{"id": "1", "nombre": "Emily", "numero": 43}, {"id": "2", "nombre": "Rylan", "numero": 69}, {"id": "3", "nombre": "Linnea", "numero": 74}, {"id": "4", "nombre": "Allison", "numero": 49}, {"id": "5", "nombre": "Skyla", "numero": 56}, {"id": "6", "nombre": "Darryl", "numero": 100}, {"id": "7", "nombre": "Tate", "numero": 25}, {"id": "8", "nombre": "Leta", "numero": 43}, {"id": "9", "nombre": "Bryana", "numero": 38}, {"id": "10", "nombre": "Ella", "numero": 90}, {"id": "11", "nombre": "Elisabeth", "numero": 87}, {"id": "12", "nombre": "Dayana", "numero": 28}, {"id": "13", "nombre": "Haleigh", "numero": 63}, {"id": "14", "nombre": "Kevin", "numero": 63}, {"id": "15", "nombre": "Ephraim", "numero": 71}, {"id": "16", "nombre": "Ernest", "numero": 1}, {"id": "17", "nombre": "Nicole", "numero": 45}, {"id": "18", "nombre": "Mitchell", "numero": 32}, {"id": "19", "nombre": "Norberto", "numero": 52}, {"id": "20", "nombre": "Antonina", "numero": 79}, {"id": "21", "nombre": "Toni", "numero": 21}]
```

**¿Cómo usamos la respuesta?**  
**¿Qué sabemos hacer y qué**  
**necesitamos?**



# JSON en el navegador

Vemos JSON formateado en el navegador?



A screenshot of a web browser window. The address bar shows the URL `web-unicen.herokuapp.com/api/groups/ejemplos/nombres`. The browser's developer console is open, displaying the raw JSON response from the API. The JSON is a single object with a `status` field set to `"OK"` and a `nombres` field containing an array of three objects. Each object in the array has fields for `_id`, `group`, `thingtype`, `thing` (which contains a `nombre` field), `__v`, and `dateAdded`.

```
{
  "status": "OK",
  "nombres": [
    {
      "_id": "5b15575fdeb51c0400814076",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Juan"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:14:39.135Z"
    },
    {
      "_id": "5b155c47b9788a040091cdc9",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Carlos"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:35:35.067Z"
    },
    {
      "_id": "5b155e13b9788a040091cdca",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Andrés"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:43:15.256Z"
    }
  ]
}
```

Extensión para Chrome:  
[JSON Formatter](#)



A screenshot of the same web browser window, but now the JSON data is displayed using the JSON Formatter extension. The JSON is visually structured with indentation, color-coding (strings in green, numbers in blue), and collapsible/expandable nodes indicated by arrows. The structure is identical to the raw JSON shown in the previous screenshot, but the formatting makes it much easier to read and understand the hierarchy of the data.

```
{
  "status": "OK",
  "nombres": [
    {
      "_id": "5b15575fdeb51c0400814076",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Juan"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:14:39.135Z"
    },
    {
      "_id": "5b155c47b9788a040091cdc9",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Carlos"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:35:35.067Z"
    },
    {
      "_id": "5b155e13b9788a040091cdca",
      "group": "ejemplos",
      "thingtype": "nombres",
      "thing": {
        "nombre": "Andrés"
      },
      "__v": 0,
      "dateAdded": "2018-06-04T15:43:15.256Z"
    }
  ]
}
```

# Ejemplo de respuesta

Analizamos la estructura de la respuesta para poder leerla

**GET** /api/usuarios

```
[  
  {  
    "id": "1",  
    "nombre": "Emily",  
    "numero": 43  
  },  
  {  
    "id": "2",  
    "nombre": "Rylan",  
    "numero": 69  
  },  
  ...  
]
```

Array de todos los registros



# Ejemplo - Consulta usuarios

```
const url =  
'https://60aab45166f1d000177731ea.mockapi.io/api/usuarios' ;  
const lista = document.querySelector("#lista_nombres");  
lista.innerHTML = "";  
try {  
    let res = await fetch(url); // GET url  
    let json = await res.json(); // texto json a objeto  
    console.log(json);  
    for (const usuario of json) {  
        let nombre = usuario.nombre;  
        lista.innerHTML += `<ul>${nombre}</ul>`;  
    }  
} catch (error) {  
    console.log(error);  
}
```

Llamada asincrónica mediante GET



Recibir info y editar el DOM

Loguear Error



<https://codepen.io/ndazeo/pen/yLMbjjR>

# Solución

---

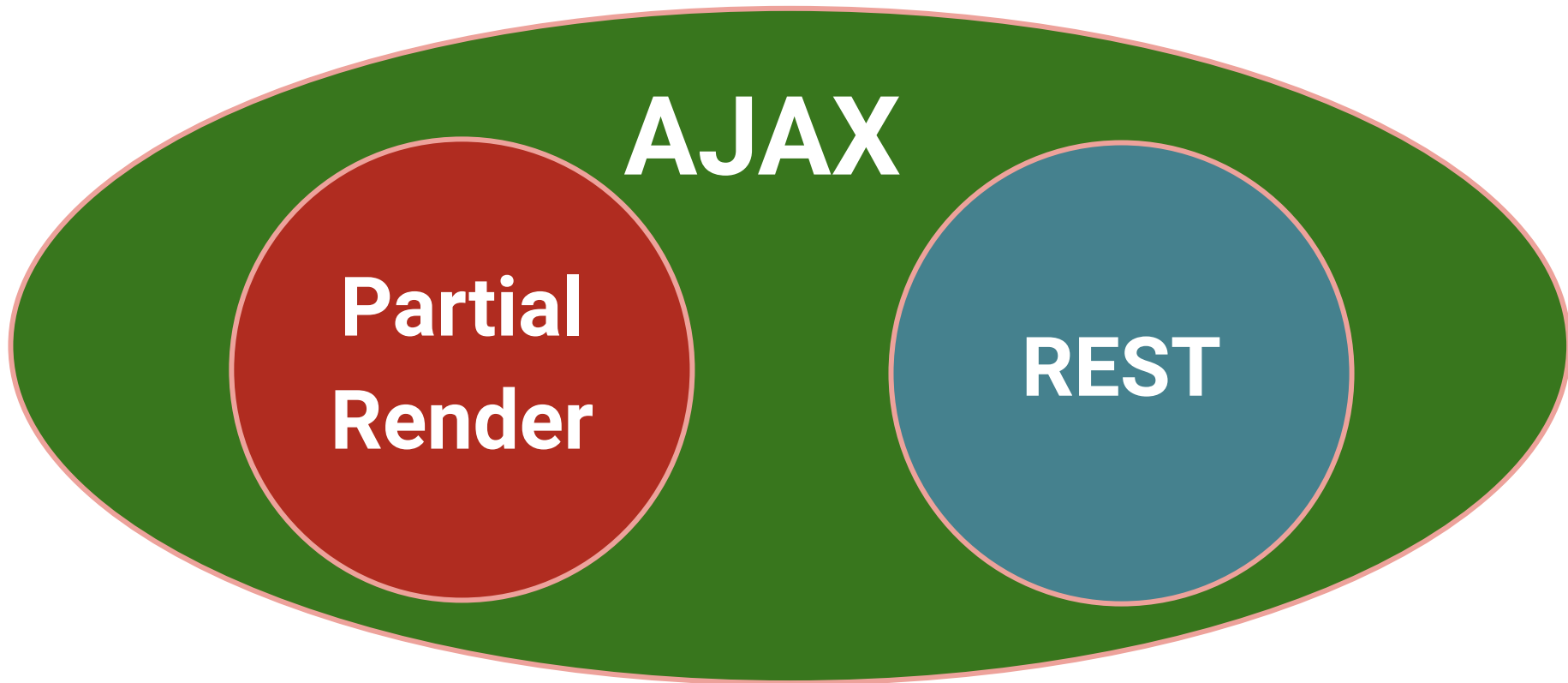
Solución ¿completa?

- <https://codepen.io/ndazeo/pen/yLMbjjR>

# Estilos de AJAX

---

- Partial render de páginas
  - Cargar un fragmento de HTML y mostrarlo en un DIV.
- Servicio REST
  - Cargar un objeto JSON y procesarlo del lado del cliente con Javascript



# Comparativa

## Partial Render

```
let urlHTML = ...;  
  
let r = await fetch(urlHTML);  
  
let html = await r.text();  
  
let div = document.querySe...  
  
contenedor.innerHTML = html;
```

## REST

```
let url = ".../api/...";  
  
let r = await fetch(url);  
  
let json = await r.json();  
  
let div = document.querySe...  
  
contenedor.innerHTML = ''  
  
for (let data of json.nombres) {  
  
    contenedor.innerHTML +=  
  
    "<p>"...  
  
}
```

# Comparativa

## Accede a un HTML

```
let urlHTML = ...;  
  
let r = await fetch(urlHTML);  
  
let html = await r.text();  
  
let div = document.querySelector(...)  
  
contenedor.innerHTML = html;
```

## Accede a una API

```
let url = ".../api/...";  
  
let r = await fetch(url);  
  
let json = await r.json();  
  
let div = document.querySelector(...)  
  
contenedor.innerHTML = ''  
  
for (let data of json.nombres) {  
  
    contenedor.innerHTML +=  
  
    "<p>" ...  
  
}
```

# Comparativa

## Partial Render

Lee un texto (que sabe que es un HTML)

```
let html = await r.text();  
  
let div = document.querySelector(...)  
  
contenedor.innerHTML = html;
```

## REST

Lee un JSON

```
let json = await r.json();  
  
let div = document.querySelector(...)  
  
contenedor.innerHTML = ''  
  
for (let data of json.nombres) {  
    contenedor.innerHTML +=  
  
    "<p>" ...  
  
}
```



# Comparativa

## Partial Render

```
let urlHTML = ...;
```

Inserta HTML en el DOM

```
let div = document.querySelector...
```

```
contenedor.innerHTML = html;
```

## REST

```
let url = ".../api/...";
```

Arma un HTML con el JSON  
y lo inserta en el DOM

```
let div = document.querySelector...
```

```
contenedor.innerHTML = ''
```

```
for (let data of json.nombres) {
```

```
    contenedor.innerHTML +=
```

```
    "<p>"...
```

```
}
```

# AJAX REST

POST: Envío de datos

¿Cómo subo la información?

- Envio una solicitud
  - Método: POST
  - URL:

`https://60aab45166f1d000177731ea.mockapi.io/api/:endpoint`

- La va a grabar en la colección

# Fetch: Opciones

---

Para enviar datos, necesitamos de fetch un segundo parámetro para indicar opciones

```
fetch(url, opciones).then(...)
```

Las opciones permiten definir:


- Verbos HTTP (method)
- Cuerpo del mensaje (body)
- Encabezados (headers)
- Otras opciones

# Opciones - Fetch POST


Usamos las opciones **method**, **headers** y **body**.

```
fetch('http://url...', {  
  'method': 'POST',  
  'headers': {  
    'Content-Type': 'application/json'  
  },  
  'body': '{ "nombre": "Juan" }'  
})  
.then(...
```


Indicamos el verbo  
de la solicitud



Content-Type dice el tipo de  
contenido del body enviado



El cuerpo debe ser una cadena  
(String)



## ¿Cómo guardar la información?

- Método: POST

- URL:

<https://60aab45166f1d000177731ea.mockapi.io/api/usuarios>

- Data: JSON Object

### ■ Ejemplo

```
{
  {
    "nombre": "Web Unicen!!",
    "numero": 1,
  }
}
```

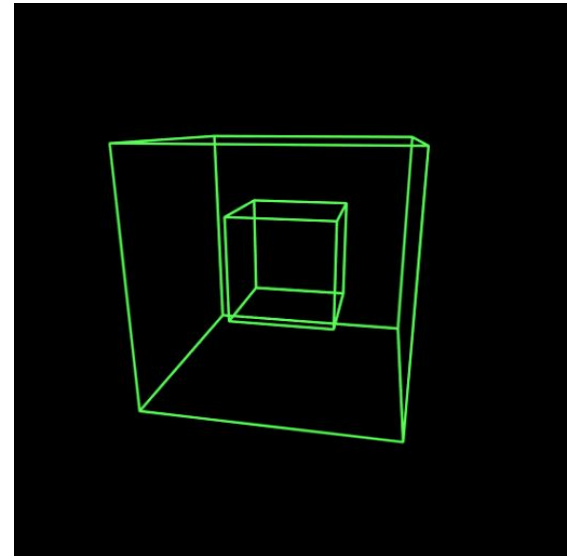
# Guardar datos

El objeto **thing** puede almacenar cualquier información que sea representada mediante JSON.

Ahí guardamos los datos que queremos en nuestro “producto”, “serie”, “noticia” o lo que sea.

Puedo guardar un objeto con subobjetos, un arreglo, o cualquier cosa!

```
let data = {  
  "thing":  
    {  
      "nombre": "Web Unicen!!"  
    }  
};
```

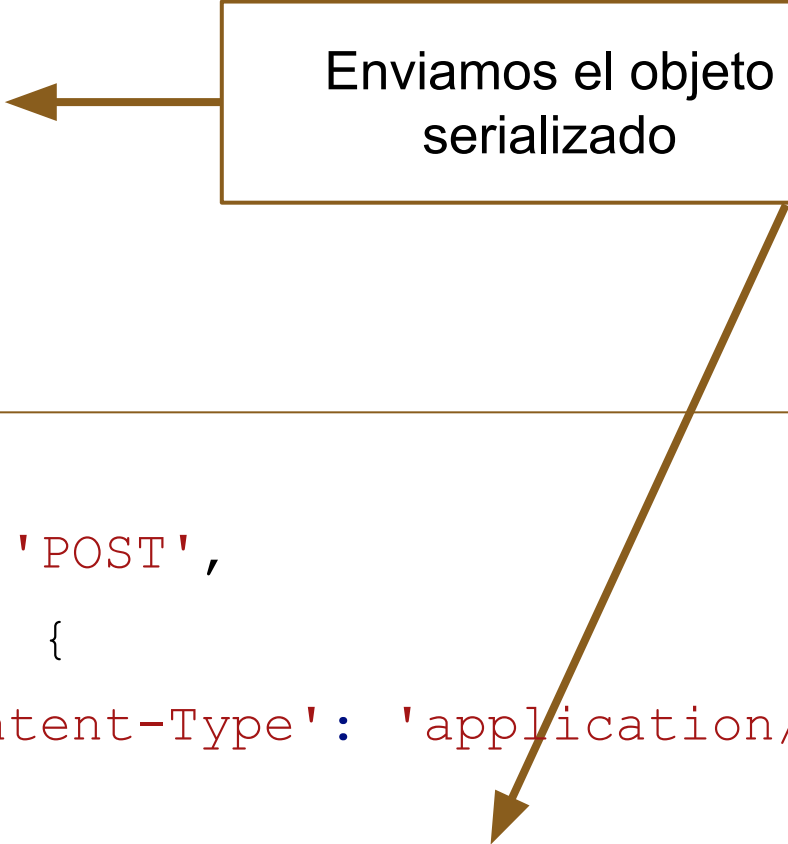


# Enviar JSON

Este servicio espera siempre un objeto en forma de texto, lo armamos y convertimos a string

```
let data = {  
  "nombre": "Web Unicen!!",  
  "numero": 1,  
};
```

Enviamos el objeto  
serializado



```
fetch(url, {  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify(data)  
}).then(...)
```



# Ejemplo - Crear Información

```
try {  
    let res = await fetch(url, {  
        "method": "POST",  
        "headers": { "Content-type": "application/json" },  
        "body": JSON.stringify(usuario)  
    });  
  
    if (res.status == 201) {  
        console.log("Creado!");  
    }  
} catch (error) {  
    console.log(error);  
}
```

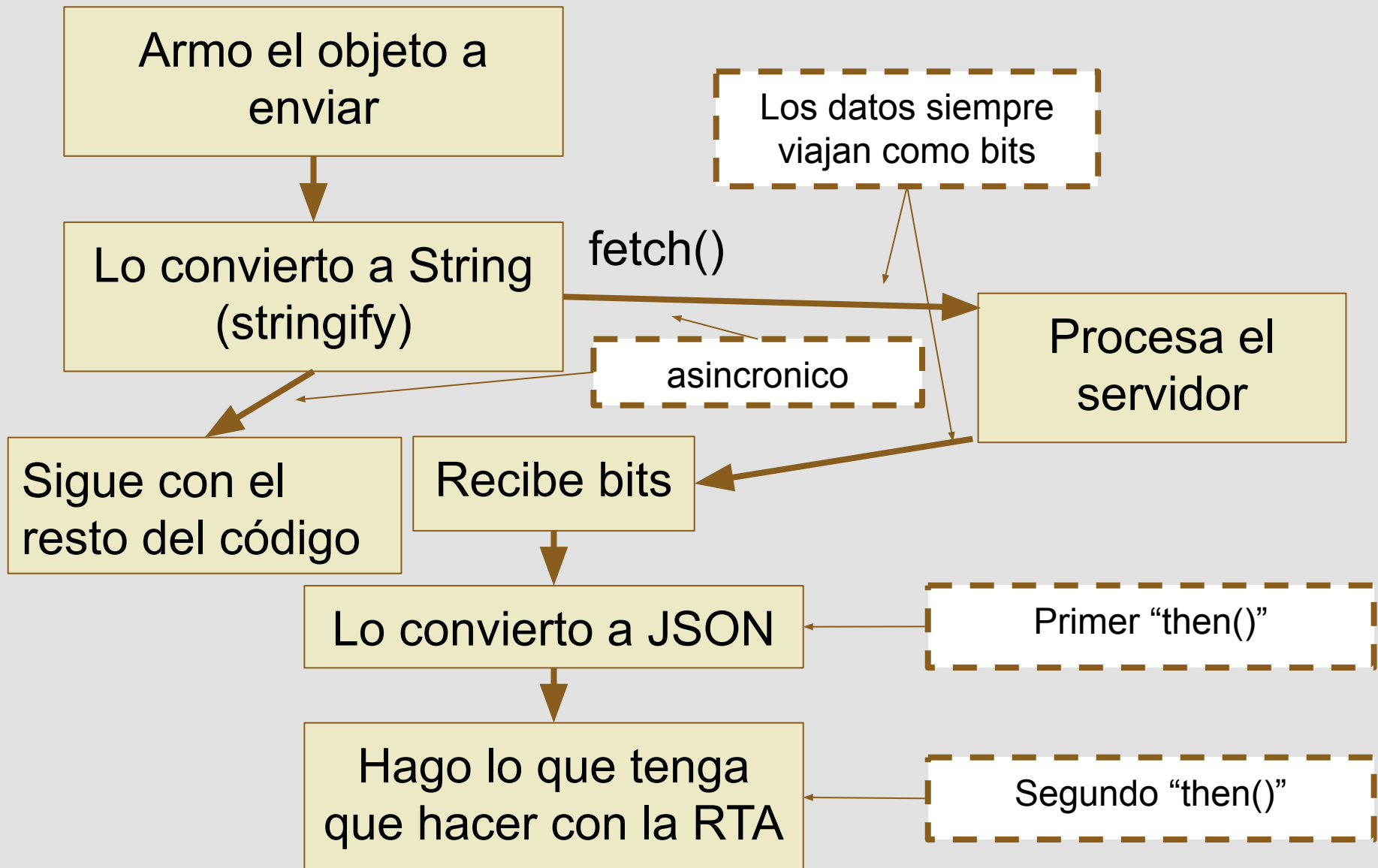
# CORS y Codepen

---

- Los navegadores/servidores tienen políticas de seguridad que en principio evitan que una página acceda a datos de otra
- Las políticas CORS permiten relajar estas restricciones de seguridad
- Por eso nuestros códigos de Codepen tienen opciones adicionales para permitir esta comunicación con otra página

```
fetch(url, {  
    method: 'GET',  
    mode: 'cors',  
}).then(...
```

# Resumen de cómo ejecuta



# AJAX REST

PUT: Modificación de datos

DELETE: Borrado de datos

# AJAX REST

PUT: Modificación de datos

**¿Qué estrategia tendríamos que adoptar para actualizar un único valor de un objeto?**

Ej: cambiar solo el numero del señor Juan

```
{  
  "nombre": "Juan",  
  "numero": 5  
},
```

## Respuesta

---

Este servicio sobrescribe todo el objeto entero, debemos enviar un objeto thing con el apellido también sin cambiar su valor.

# Fetch - Put

---

TBC

<https://codepen.io/webUnicen/pen/zarEeW>



# AJAX REST

DELETE: Borrado de datos

# Fetch - Delete

---

TBC

# Reflexionemos

---

**TPE: ¿Cómo harían para que esos resultados se muestren en una tabla?**

**[TBC en el TPE]**

# Repaso AJAX: Ensalada de tecnologías

---

Una mezcla donde cada tecnología aporta algo:

- Presentación en estándares HTML y CSS.
- Display e interacciones dinámicas via DOM.
- Intercambio de datos mediante XML (o JSON)
- Lectura de datos asincrónica mediante fetch
- Y JavaScript para unir todo.

# Consecuencias de asincrónico

---

Creo handlers para objetos que aún no existen.  
etc...

EXTRAS

# Ejemplo de carga asincrónica de una imagen



Podemos descargar una imagen de forma asincrónica y mostrarla en un tag **<img>**.

```
let miImagen = document.querySelector('.mi-imagen');  
fetch('https://picsum.photos/200/300.jpg')  
  .then(res => res.blob())  
  .then(res => {  
    var objectURL = URL.createObjectURL(res);  
    miImagen.src = objectURL;  
  });
```



se ejecuta el método **blob()** para que procese el tipo de archivo que esperamos (imagen).

Codepen: <https://codepen.io/webUnicen/pen/MGoYEz>



# Referencias

---

- <http://api.jquery.com/jquery.ajax>
- <https://eamodeorubio.wordpress.com/category/webservices/rest/>
- <https://developer.mozilla.org/es/docs/AJAX>
- <http://www.restapitutorial.com/lessons/whatisrest.html>
- [https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar\\_promesas](https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Usar_promesas)
- <https://developers.google.com/web/fundamentals/primers/promises?hl=es>
- “BulletProof AJAX” Jeremy Keith

**AHORA LES TOCA PRACTICAR :D**



mockAPI