

Back-end

CFP

Programador full-stack

Consumir una API

Carrito de compras

A lo largo de este módulo vamos a trabajar con nuestro carrito de compras para lograr que guarde los datos en un servidor.

Vamos a hacerlo paso a paso.

Un paso por clase.

Que tenemos?

Hasta ahora tenemos:

Un sitio web que permite agregar cosas al carrito mediante JS

¿Que pasa al refrescar la página?

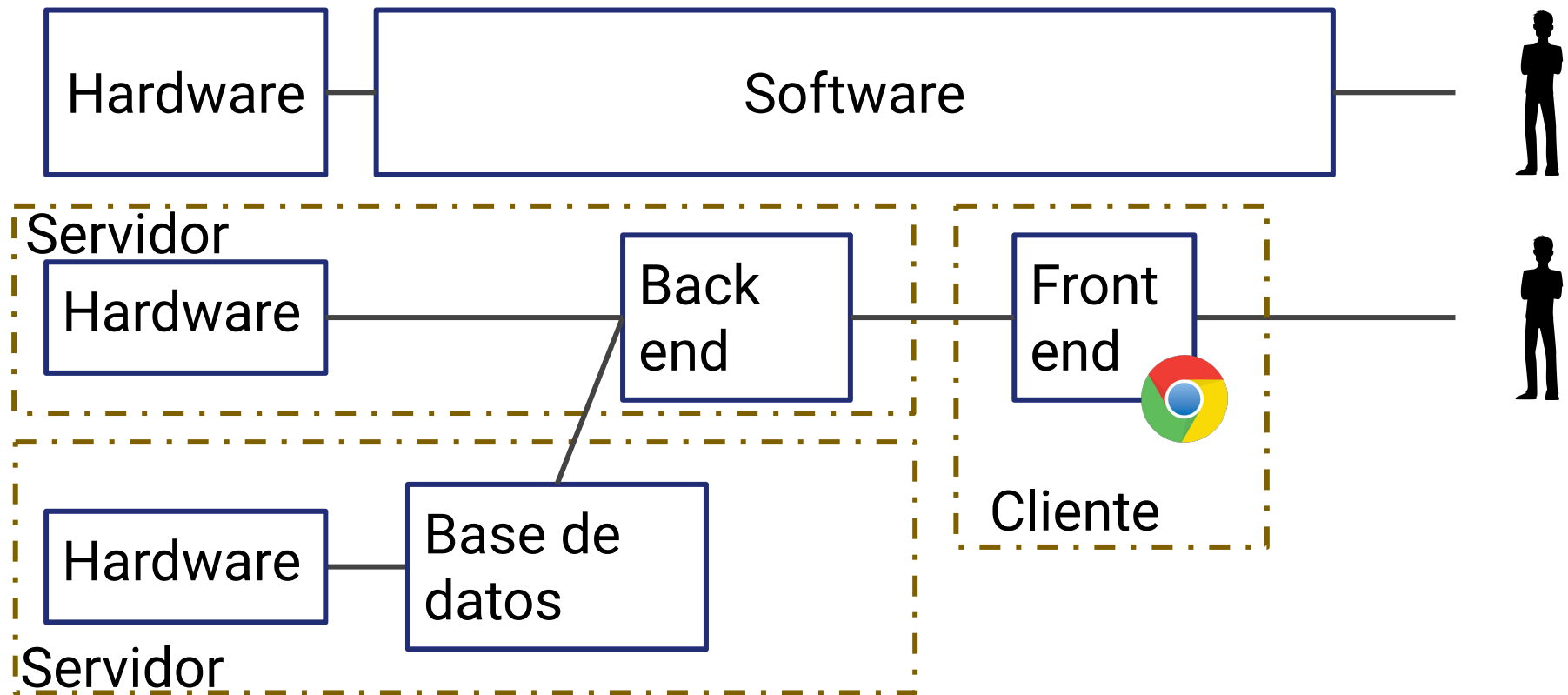
Limitaciones

El código Javascript que se ejecuta en el cliente (navegador/browser) no se persiste

Hay formas de guardarlo, pero estaría guardado en la PC del usuario

Necesitamos tener esos datos en el servidor

Arquitectura de Sistemas Web



Cliente

Se encarga de manejar la interfaz gráfica del usuario

Servidor

Se encarga de:

- Almacenar los datos
- Dar seguridad (que cada uno vea solo sus datos, etc)

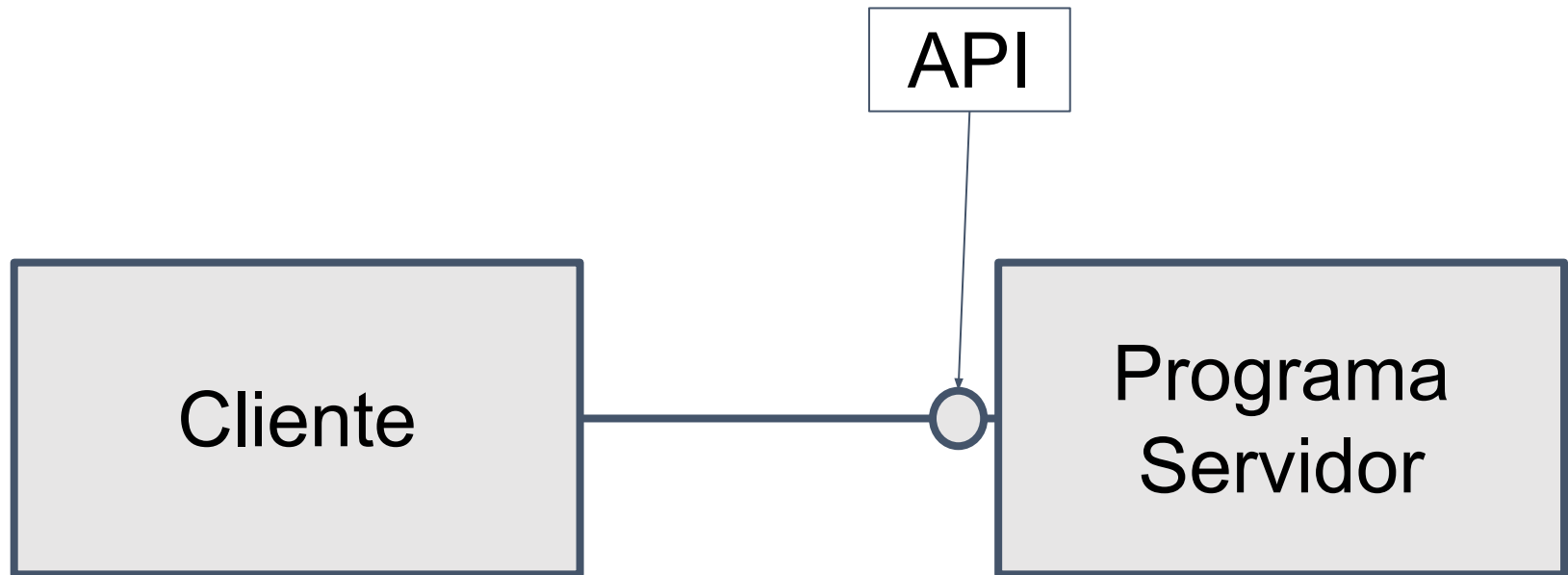
Pasos:

Estos son los pasos que vamos a seguir a través de las clases (el orden puede variar levemente):

1. **Traer datos estáticos que están en el servidor (HOY)**
2. Que esos datos sean dinámicos
3. Que esos datos estén guardados en un archivo
4. Guardar nuevos datos
5. Borrar datos
6. Editar datos

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.

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

CFP

Programador full-stack

Mockear nuestro back-end

Mockear una API

Primero vamos a mockear nuestra API

Mockear significa “reemplazar una implementación real por una falsa y simple”

Como las API devuelven JSON vamos a hacer nuestro archivo JSON estático

Para qué mockeamos?

Nos permite trabajar con el front end aunque nuestro back end no esté implementado

Creemos un archivo JSON

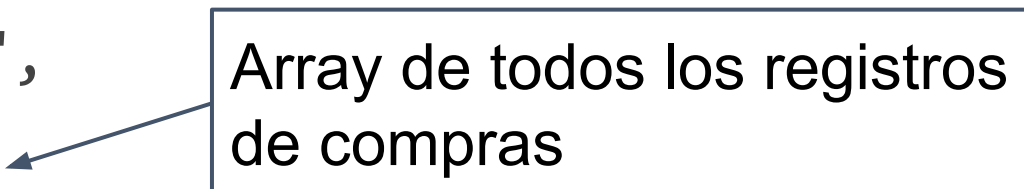
En los archivos de nuestro servidor hagamos un “mock.json” con el contenido que queremos que nos devuelva

Ejemplo

Vamos a inventar una respuesta (estilo de respuesta típica de una API):

```
{  
  "status": "OK",  
  "compras": [  
    {  
      "producto": "Manzana",  
      "precio": "20"  
    }  
  ]  
}
```

Array de todos los registros de compras



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)

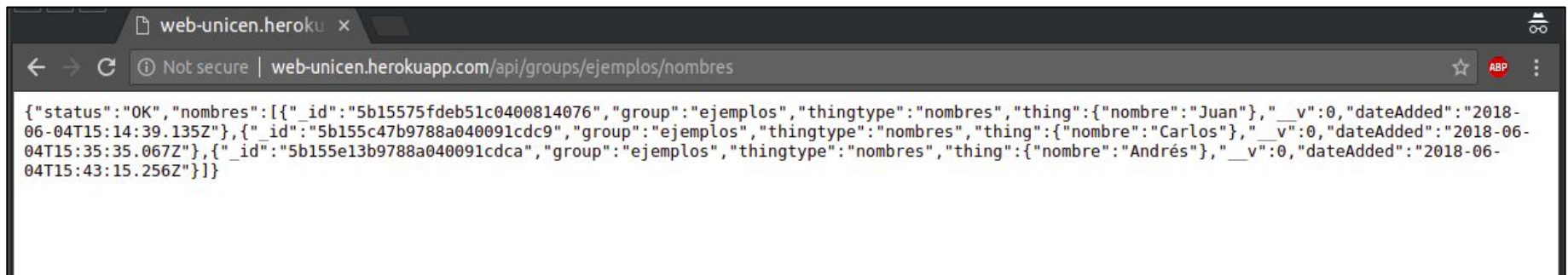
JSON en el navegador

```
▼ {  
  "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"
```

Extensión para
Chrome:
[JSON Formatter](#)

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 `web-unicen.herokuapp.com/api/groups/ejemplos/nombres`. The page content displays a JSON response from a REST API. The JSON is formatted with syntax highlighting, showing an array of three objects under the `nombres` key. Each object contains an `_id`, a `group` of `ejemplos`, a `thingtype` of `nombres`, and a `thing` object with a `nombre` field. The `__v` field is 0 and the `dateAdded` field shows timestamps.

```
{"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

```
{  
  "status": "OK",  
  "compras": [  
    {  
      "producto": "Manzana",  
      "precio": "20"  
    }  
  ]  
}
```

← Array de todos los registros

JSON en el navegador

```
▼ {  
  "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"
```

Extensión para
Chrome:
[JSON Formatter](#)

CFP

Programador

full-stack

Consumir la API

Consumir la API mockeada

Ahora nos falta consumir esa API desde el front-end.

Lo que tenemos que ver es que el producto que pusimos en el JSON aparezca en la página

Nuestra situación:

Hasta ahora teníamos:

- DOM (HTML)
- Estilos (CSS)
- Programar y editar el DOM (JS)

Todo dentro del cliente

Y ahora vamos a agregar:

- Conexión al servidor desde JS

Ajax en ES7

ES7 incorpora la interfaz **fetch()**

```
let promise = fetch(url);
```

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](#))

AWAIT y ASYNC

- En **ES8** se incorpora dos palabras reservadas para facilitar la escritura de código con promesas
 - ASYNC
 - Hace que una función devuelva una promesa
 - El return se encapsulará en la promesa automáticamente
 - AWAIT
 - Desencapsula el contenido de una promesa.
 - Se reescribe como el THEN de la promesa
 - Solo puede usarse dentro de funciones ASYNC

Await

```
let promise = fetch(url);  
let response = await promise;
```

El `await` lee el objeto dentro de la promesa cuando esté listo.

Mientras tanto libera al navegador para que pueda hacer otra cosa.

SINCRÓNICO

Meto comida
en el horno

Miro el horno hasta que esté lista

Ceno

Se hace la comida

ASINCRÓNICO

Meto comida
en el horno y
prendo timer

Preparo la
mesa

Cargo la
serie

...

Ceno con
una serie

Se hace la comida

RING!

Responde
la promesa

Recibir JSON

```
response.json()
```

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

Devuelve una promesa, dado que convertir a JSON puede demorar mucho

¿Y si el archivo no existe?

El objeto “response” tiene información de la respuesta obtenida del servidor.

Con `response.ok` nos dice si la descarga pudo hacerse correctamente (Código HTTP 200)




```
async function load() {  
  let container =  
document.querySelector("#use-ajax");  
  let response = await fetch(url);  
  if (response.ok) {  
    let t = await response.json();  
    container.innerHTML = ...;  
  }  
}
```

Ejercicio: usar el json

El ... de la slide anterior es lo que procesa el JSON para mostrarlo en la tabla

Completar la adaptación del código para que funcione correctamente

¿Cómo agrego el cartelito de
“Loading...”?

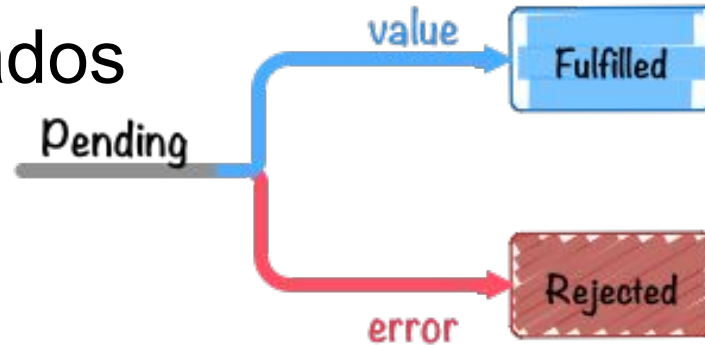
```
async function load() {  
  let container =  
document.querySelector("#use-ajax");  
  container.innerHTML = "<h1>Loading...</h1>";  
  let response = await fetch(url);  
  if (response.ok) {  
    let t = await response.json();  
    container.innerHTML = t;  
  }  
}
```

Y qué pasa si la descarga no se puede hacer?

Promesas (Terminología)

Una promesa tiene 4 estados

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



Término **then**

En vez de pasar funciones callback a una función, a la promesa le “encadenamos” las funciones callback.

Promise



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

```
async function load() {  
  let container = document.querySelector("#use-ajax");  
  container.innerHTML = "<h1>Loading...</h1>";  
  try {  
    let response = await fetch(url);  
    if (response.ok) {  
      let t = await response.json()  
      container.innerHTML = ...;  
    }  
    else  
      container.innerHTML = "<h1>Error - Failed URL!</h1>";  
  }  
  catch (response) {  
    container.innerHTML = "<h1>Connection error</h1>";  
  };  
}
```

Funcionamiento

