



JavaScript en el navegador - MÓDULO 7: Excepciones, Promesas, HTTP, AJAX y jQuery

Juan Quemada, DIT - UPM

Índice

MODULO 7 - Excepciones, Promesas, HTTP, AJAX y jQuery

1. Errores, excepciones, throw y try-catch-finally	3
2. Promesas I: new Promise(..), resolve, reject, async, await,	8
3. Promesas II: Orden de ejecución de instrucciones, then, catch y ejemplos	15
4. HTTP: Solicitudes, respuestas, métodos, MIME e interfaces REST	20
5. AJAX, Same Origin, CORS y Fetch API	30
6. MyJSON.com y ejemplo AJAX/REST	36
7. Librería jQuery y CDN Web	42



Excepciones, errores y sentencias throw y try-catch-finally

Juan Quemada, DIT - UPM
Santiago Pavón, DIT - UPM

Excepciones, errores y sentencia throw

◆ Excepción

- es una señal que interrumpe la ejecución de un programa
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/throw>

◆ La excepción se lanza con la sentencia: **throw <expr>**

- Por ejemplo: **throw "Abort execution";**
 - ◆ "Abort execution" es la expresión lanzada, un string, que identifica la excepción

◆ Errores

- Son **excepciones** donde **<expr>** pertenece a la clase predefinida **Error**
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError

◆ Un **error** se lanza también con la sentencia: **throw <expr>**

- Por ejemplo: **throw new Error("Execution error")**

Errores de ejecución del programa

- ◆ El interprete de JavaScript lanza errores durante la ejecución

- Cuando detecta problemas

Se invoca **funcion_no_definida()**
que no existe.

- ◆ Por ejemplo

- Al invocar funciones no definidas
- Al invocar métodos no definidos
- Al utilizar variables no definidas
- Al detectar errores de sintaxis
-

El interprete JS indica que la referencia **funcion_no_definida** no existe.

The screenshot shows a browser's developer tools open to the 'Console' tab. The code area above the console shows a simple HTML page with a script tag containing the call to an undefined function. The console output below shows a red error message: 'Uncaught ReferenceError: funcion_no_definida is not defined at 03-error.html:6'. The number '5' is visible in the bottom right corner of the console panel.

```
<!DOCTYPE html>
<html>
  <body>
    <script type="text/javascript">
      funcion_no_definida()
    </script>
  </body>
</html>
```

Console

✖ 1

✖ 1 | Filter | ⚙

top

✖ Uncaught ReferenceError: 03-error.html:6
funcion_no_definida is not defined
at 03-error.html:6

5

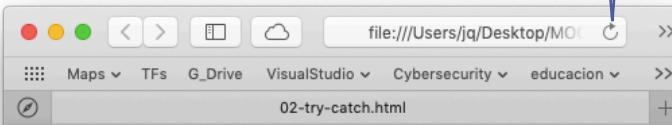
Capturar excepciones: sentencia try-catch-finally

- ◆ La sentencia **try-catch-finally** permite **capturar** excepciones o errores
 - Captura solo las **excepciones** que ocurren dentro del **bloque try**
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch>
- ◆ Cuando ocurre una **excepción** dentro del **bloque try**
 - La ejecución continua en el **bloque catch**
 - ◆ exception contiene el parámetro de throw <param>
- ◆ El bloque **finally** se ejecuta siempre al final
 - Ocurran excepciones o no
- ◆ El interprete JS también lanza errores
 - Son excepciones con un objeto de la clase **Error**
 - ◆ Hay muchas **clases de error** que extienden **Error**
 - ◆ Si ocurren dentro de **try** se pueden capturar con **catch**

```
.....
try {
    .....
    -> throw "Exception"
    or throw new Error("Error")
    .....
} catch (exception) {
    .....
} finally {
    .....
}
.....
```

Ejemplo de try-catch-finally

El programa se recarga haciendo clic aquí.



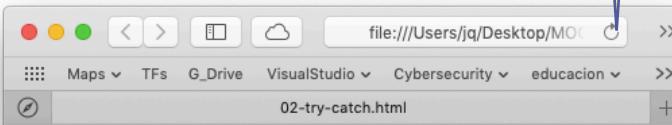
NO ha ocurrido excepción

Fin del programa

Si NO ocurre excepción, se muestra este primer msj.

El mensaje del bloque finally aparece siempre.

El programa se recarga haciendo clic aquí.



HAY EXCEPCIÓN

Fin del programa

El mensaje del bloque finally aparece siempre.

Si ocurre excepción, se muestra este primer msj.

Math.random() se utiliza para que este programa lance una excepción un 50% de las veces que se recarge.

```
<!DOCTYPE html>
<html>
  <head><meta charset="UTF-8"></head>
  <body>
    <div id="i1"></div>
    <div id="i2"></div>

    <script type="text/javascript">
      function show (id, msj) {
        document.getElementById(id).innerHTML = msj;
      }

      try {
        if (Math.random() < 0.5) { throw "HAY EXCEPCIÓN"; }
        show("i1", "NO ha ocurrido excepción");
      } catch (e) {
        show("i1", e);
      } finally {
        show("i2", "Fin del programa");
      }
    </script>
  </body>
</html>
```



Promesas I: new Promise(..), resolve, reject, async, await,

Juan Quemada, DIT - UPM
Santiago Pavón, DIT - UPM

Promesas de ES6

- ◆ Promesa (ES6): objeto de la clase **Promise** (Promesa)
 - Representan tareas que prometen retornar un valor en el futuro
 - Ver: <https://javascript.info/async>
 - Ver: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise
- ◆ Una promesa tiene tres estados para gestionar este proceso
 - **pendiente**: antes de ejecutar la tarea asociada
 - **cumplida**: la tarea tiene **éxito** y retorna el valor prometido
 - **rechazada**: la tarea **falla** y retorna un código de rechazo
- ◆ Ordenan la ejecución de **funciones** (callbacks) en el tiempo
 - Conservando la **eficiencia** de ejecución de los "callbacks" asíncronos
 - Son sencillas de utilizar: separan el **código normal** del código de **atención a errores**
 - Ceden el procesador a otros callbacks hasta que se resuelven

Promesas: constructor

◆ Promesa

- Objeto de la **clase Promise** creado con **new Promise((resolve, reject) => <bloque>)**
 - ◆ El constructor debe recibir como **parámetro** una **función** (con los parámetros **resolve** y **reject**) que se ejecuta al crear el objeto

◆ resolve(<data>)

- Función que finaliza la promesa con éxito al invocarse
 - ◆ Devuelve la expresión **<data>** como valor de **éxito** de la promesa
- Si **<data> es una promesa** se espera a su resolución y se devuelve su valor de retorno

◆ reject(<err>)

- Función que rechaza la promesa al invocarse
 - ◆ Devuelve la expresión **<err>** como valor de **rechazo** (**<err>** suele describir la **causa del rechazo**)

mi_promesa: es un ejemplo de promesa que siempre se resuelve con éxito a los 100 milisegundos (devuelve "undefined").

```
let mi_promesa = Promise ( ( resolve, reject ) => {
```

```
    setTimeout( () => resolve(5), 100)
```

```
}
```

setTimeout(resolve, 100) invoca la función **resolve()** a los 100 milisegundos.

Métodos útiles para crear promesas concisas

◆ **Promise.resolve(<data>)**

- Crea una promesa que finaliza con **éxito** y genera **<data>**
 - ◆ La promesa finalizará con éxito normalmente salvo que <data> genere un rechazo
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/resolve

◆ **Promise.reject(<err>)**

- Crea una promesa que siempre se **rechaza** con **<err>**
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/reject

Ejemplos de promesa sencilla que **siempre se cumple** con **Promise.resolve()** o con constructor.

```
let p2 = Promise.resolve([7, 4, 1, 23]); // equivale a:  
let p1 = new Promise((resolve, reject) => resolve([7, 4, 1, 23]));
```

Ejemplos de promesa sencilla que **siempre se rechaza** con **Promise.reject()** o con constructor.

```
let r2 = Promise.reject("Promesa rechazada"); // equivale a:  
let r1 = new Promise((resolve, reject) => reject("Promesa rechazada"));
```

async/await

◆ **async/await** (ES8) facilita el uso de promesas

- **async/await** es compatible con las promesas de **ES6** y puede mezclarse con ellas
 - ◆ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/async_function
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>
 - ◆ <https://javascript.info/async-await>

◆ **async** define una **función "especial"** que retorna una **promesa**

- **async** se antepone a la definición de la función
 - ◆ Por ejemplo **async function suma (x, y) { }** o **async (x, y) => { }**
- La promesa finaliza con éxito con **return <data>** o se rechaza con **throw <err>**
- **async** no permite definir promesas que se resuelven en callbacks

◆ **await <promesa>**: espera la resolución de la promesa y devuelve su resultado

- Por ejemplo, **let x = await promesa()** asigna a **x** el **valor de éxito** (en caso de éxito)
 - ◆ Pero si se **rechaza**, lanza una **excepción** que puede capturarse con una sentencia **try_catch**
- Otros ejemplos, **2*(await promesa())** o **await promesa()**

◆ **await <promesa>** solo se puede utilizar **dentro** de una **función async**

- No se puede utilizar en otros contextos

Reloj: ejemplo async/await

time(ms): retorna una promesa que finaliza con éxito al cumplirse los milisegundos indicados.

El constructor `new Promise(..)` debe incluir un literal de función que recibe las funciones `resolve` y `reject` como parámetros, para ejecutarlas cuando se quiera **finalizar con éxito o rechazar** la promesa.

La promesa `time(100)` finalizará con éxito 1000 milisegundos después de haber sido arrancada y se utiliza en

`await time(100);`

para introducir un retardo de un segundo en cada vuelta del bucle.

```
<!DOCTYPE html>
<html>
  <head><meta charset="UTF-8"></head>
  <body>
    <div id="i1"></div>

    <script type="text/javascript">

      const time = (ms) => {
        return new Promise (resolve => setTimeout(resolve, ms));
      }

      const start = async () => {
        while (true) {
          document.getElementById("i1").innerHTML = new Date();
          await time(1000); // wait 1 second
        }
      }

      start()
    </script>
  </body>
</html>
```

Este programa muestra como va avanzando la hora y la fecha.

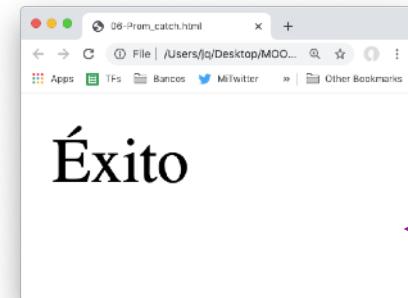
Captura del rechazo con try-catch

`mi_prom()` genera éxito con `resolve("Éxito")` o rechazo con `reject("Promesa rechazada")` aleatoriamente (50%).

◆ `await` espera la resolución de una **promesa**

- Si se rechaza se lanza una excepción con el mensaje de error de la promesa

◆ Los **rechazos** se deben capturar con un **catch** de una sentencia `try-catch`



Si `mi_prom()` finaliza con éxito, el bloque `try` se ejecuta completo mostrando Éxito.



Si `mi_prom(n)` se rechaza, la ejecución del bloque `try` se interrumpe y se pasa a ejecutar el bloque `catch`. `err` recibe el mensaje mostrado: Promesa rechazada.

```
<!DOCTYPE html><html>
<head><meta charset="UTF-8"></head>
<body>
    <div id="d"></div>

    <script type="text/javascript">
        let d = document.getElementById("d");

        const mi_prom = () =>
            new Promise( (resolve, reject) => {
                if (Math.random() < 0.5) {
                    resolve("Éxito");
                } else {
                    reject("Promesa rechazada");
                }
            });

        const start = async () => {
            try {
                let msg = await mi_prom();
                d.innerHTML = msg;
            } catch (err) {
                d.innerHTML = err;
            }
        }

        start();
    </script>
</body></html>
```

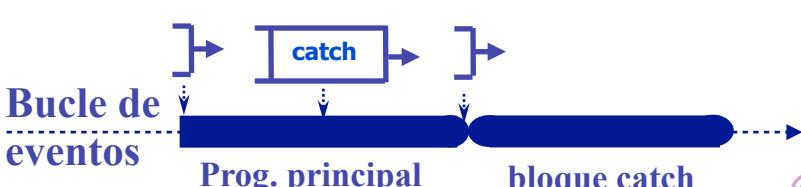


Promesas II: Orden de ejecución de instrucciones, then, catch y ejemplos

Juan Quemada, DIT - UPM
Santiago Pavón, DIT - UPM

Orden de ejecución

- ◆ Debemos recordar, que los promesas, **resolve(..)**, **reject(..)** o las funciones **async** son funciones
 - Se ejecutan como cualquier función y retornan al acabar
- ◆ El ejecutar el código posterior a **await** o el bloque **catch**, depende de la resolución de la promesa
 - Por eso ambos se **encapsulan** como **callbacks**
 - **resolve(..)** envía el **callback await** a la cola de eventos
 - **reject(..)** envía el **callback catch** a la cola de eventos
- ◆ Orden de ejecución de este ejemplo:
 - El **programa principal** se ejecuta primero
 - Define primero las promesas **prom** y **start**
 - A continuación invoca (ejecuta) **start()**, que
 - Muestra primero **E2**
 - después invoca **prom()** que muestra **E1** y se rechaza
 - El **rechazo** envía el **callback catch** a la cola
 - Cuando **start()** retorna, se muestra **E5** y el prog. princ. **finaliza**
 - Entonces se ejecuta el **callback catch**
 - El **callback catch** muestra **E4**



2. **prom()** muestra mensaje **E1** y rechaza la promesa introduciendo el evento que hemos llamado **catch**, que indica que se debe continuar por el bloque **catch**. A continuación retorna finalizando la ejecución de **start()**.

```
<!DOCTYPE html><html><head></head><body><script type="text/javascript">
const prom = () =>
  new Promise((resolve, reject)=>{
    document.write(" E1");
    reject();
  })
const start = async () => {
  try {
    document.write(" E2");
    await prom();
    document.write(" E3");
  } catch (err) {
    document.write(" E4");
  }
}
start();
document.write(" E5");
</script>
</body></html>
```

1. **start()** muestra mensaje **E2** y ejecuta **prom()**.

1. **start()** es el primer código que se ejecuta.

3. En tercer lugar se ejecuta esta sentencia que muestra **E5**.

4. Por último se ejecuta el **callback catch** asociado al evento **catch** del diagrama que el rechazo ha introducido en la cola (al llegarle el turno).

then y catch (ES6)

(Ejemplo equivalente al anterior)

◆ then(..) y catch(..)

- Métodos de instancia de la clase **Promise**, que devuelven **promesas** y pueden encadenarse

◆ p.then(<callback>)

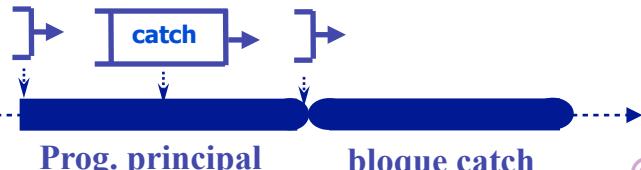
- **<callback>** que atiende el **éxito** de la promesa p
 - Propaga el **rechazo** al siguiente then(..) o catch(..)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then

◆ p.catch(<callback>)

- **<callback>** que atiende el **rechazo** de la promesa p
 - Propaga el **éxito** al siguiente then(..) o catch(..)
 - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch

◆ <callback> de then(..) o catch(..)

- **throw <msg>** rechaza la promesa con el mensaje de error <msg>
- **return <valor>** resuelve la promesa con el valor de éxito <valor>
 - Si <valor> es una **promesa**, se espera a su resolución y se devuelve su **éxito** o **rechazo**



2. **prom()** muestra mensaje **E1** y rechaza la promesa introduciendo el evento que hemos llamado **catch**, que indica que se debe continuar por el bloque **catch**. A continuación retorna finalizando la ejecución de **start()**.

```
<!DOCTYPE html><html><head></head><body><script type="text/javascript">const prom = () => new Promise((resolve, reject)=>{ document.write(" E1"); reject();});const start = () => {document.write(" E2"); prom().then ( () => {document.write(" E3"); return;}).catch( () => {document.write(" E4"); return;});};start();document.write(" E5");</script></body></html>
```

x. **then(..)** encapsula un código que se ejecutara si la promesa tiene éxito (no se ejecuta en este ejemplo).

1. **start()** muestra mensaje **E2** y ejecuta **prom()**.

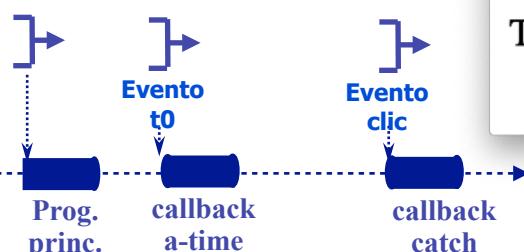
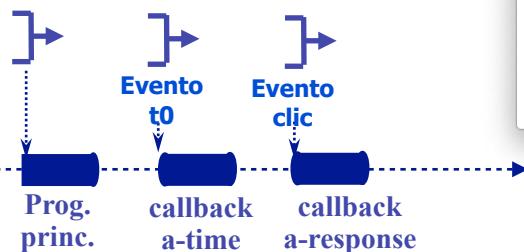
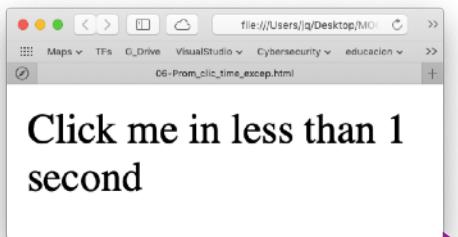
1. **start()** es el primer código que se ejecuta.

3. En tercer lugar se ejecuta esta sentencia que muestra **E5**.

4. Por último se ejecuta el **callback catch** asociado al evento catch del diagrama que el rechazo ha introducido en la cola (al llegarle el turno).

Ejemplo: Reflejos

response(t0) es un ejemplo de promesa asociada a un manejador de eventos. Finaliza con **éxito** si el clic se hace antes de **t0 mseg** y se **rechaza** si el tiempo es mayor.



```
<!DOCTYPE html><html> <head><meta charset="UTF-8"></head>
<body>
  <div id="display"></div>

  <script type="text/javascript">
    let display = document.getElementById("display");

    const time = (ms) => new Promise (resolve => setTimeout(resolve, ms));

    const response = (t0) =>
      new Promise((resolve, reject) =>
        display.addEventListener('click', () => {
          let t = new Date().getTime() - t0;

          if (t <= 1000) resolve(t);
          else reject(`Too late: ${t}ms`);
        })
      )
    }

    const start = async () => {
      await time(5000*Math.random());
      display.innerHTML = "Click me in less than 1 second";

      try {
        let t = await response(new Date().getTime());
        display.innerHTML = `Congratulations: ${t}ms`;
      } catch (msg) {
        display.innerHTML = msg;
      }
    }

    start()
  </script>
</body>
</html>
```

time(ms): es una promesa que finaliza con éxito al cumplirse los milisegundos indicados.

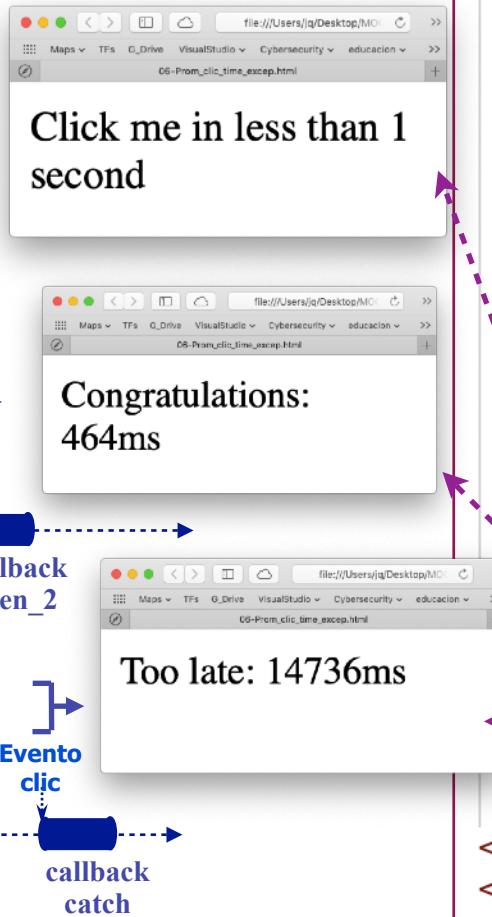
Se muestra el mensaje **"Click me less than 1 second"** después de un retardo aleatorio de entre 0 y 5 segundos.

Se muestra el mensaje **"Congratulations 464ms"** solo si ha habido éxito.

Se muestra el mensaje **"Too late: 14736ms"** si ha habido rechazo.

Ejemplo: Reflejos con then y catch

response(t0) es un ejemplo de promesa asociada a un manejador de eventos. Finaliza con **éxito** si el clic se hace antes de **t0 mseg** y se **rechaza** si el tiempo es mayor.



```

<!DOCTYPE html><html><head><meta charset="U
<body>
  <div id="display"></div>

  <script type="text/javascript">
    let display = document.getElementById("display");

    const time = ms => new Promise ( resolve=>setTimeout(resolve, ms));

    const response = t0 =>
      new Promise((resolve, reject) =>
        display.addEventListener('click', () => {
          let t = new Date().getTime() - t0;

          if (t <= 1000) resolve(t);
          else reject(`Too late: ${t}ms`);
        })
      )
    )

    time(5000*Math.random())
      .then(() => display.innerHTML = "Click me in less than 1 second")
      .then(() =>
        response(new Date().getTime())
          .then( t) =>
            display.innerHTML = `Congratulations: ${t}ms`
          )
      )
      .catch( msg) => display.innerHTML = msg
    )
  </script>
</body>
</html>
  
```

time(ms): es una promesa que finaliza con éxito al cumplirse los milisegundos indicados.

Se muestra el mensaje "Click me less than in 1 second" después de un retardo aleatorio de entre 0 y 5 segundos.

Se muestra el mensaje "Too late: 14736ms" si ha habido rechazo.

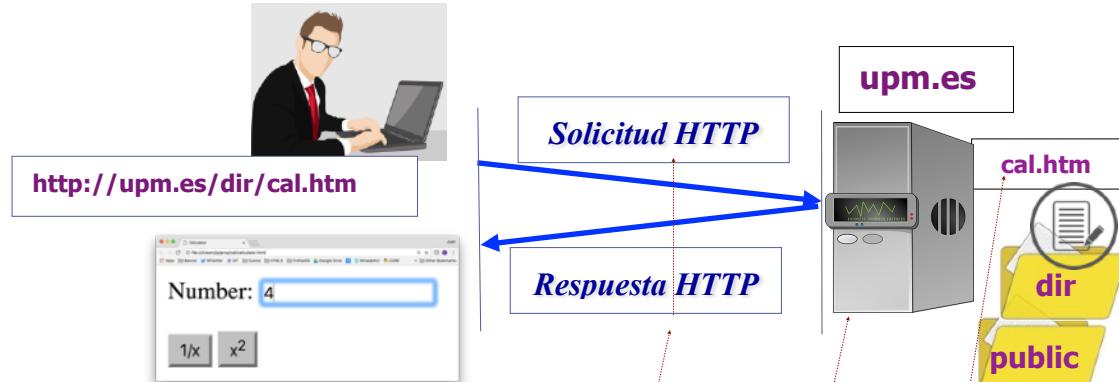
Se muestra el mensaje "Congratulations 464ms" solo si ha habido éxito.



HTTP: Protocolo, solicitudes, respuestas, métodos, MIME e interfaces REST

Juan Quemada, DIT - UPM

La Web inicial



◆ Servidor Web estático

- Programa que **sirve ficheros** solicitados por clientes
 - ◆ Los ficheros están en un directorio de recursos (páginas Web)
 - El directorio de recursos suele ser: www, public, ..

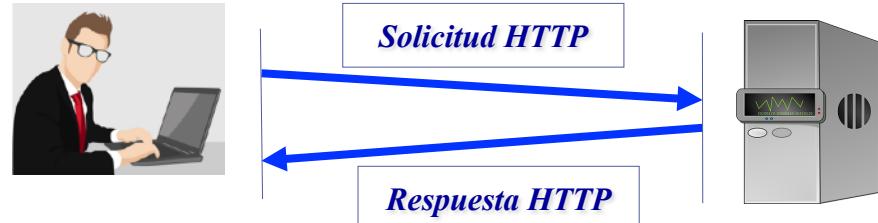
◆ Cliente Web

- Muestra páginas Web traídas de un **servidor en Internet** a un usuario
- El **URL** identifica el recurso Web: <http://upm.es/dir/cal.htm>
 - ◆ **http:** El **protocolo** de acceso al servidor (Inicialmente solo HTTP GET)
 - ◆ **upm.es:** La dirección de dominio del **servidor** que alberga la página
 - ◆ **/dir/cal.html :** La ruta al **fichero** (página Web) en el directorio de recursos del servidor

◆ La transacción HTTP vista desde el cliente:

- Establece una conexión TCP con el servidor (**upm.es**)
- Envía por la conexión una **Solicitud HTTP** con la ruta al recurso Web (**/dir/pagina.htm**)
- Recibe por la conexión la **Respuesta HTTP** con el fichero (**página Web**)
- El servidor cierra la conexión TCP

HTTP hoy



◆ Protocolo HTTP

- Soporta acceso hypermedia a páginas Web y a otros servicios de Internet
 - ◆ Ha evolucionado mucho: HTTP0.9 -> HTTP1.0 -> HTTP1.1 -> HTTP/2
 - HTTP/2 mejora la latencia y las prestaciones de HTTP1.1 (<https://http2.github.io/>)

◆ Transacción HTTP

- Una transacción HTTP se inicia siempre desde el lado cliente
 - ◆ El cliente establece la conexión HTTP con el servidor o reutiliza una existente
 - ◆ El cliente envía **Solicitudes** HTTP al servidor por la conexión HTTP
 - ◆ El servidor contesta al cliente enviando **Respuestas** HTTP por la conexión
- Solicitud y respuesta son **envíos independientes**, aunque relacionados

◆ Solicitud y respuesta tienen el **mismo formato**

- Tienen 3 partes: **Primera línea, parámetros y cuerpo**

<i>Solicitudes y Respuestas HTTP</i>
Primera línea
Parámetros
..... Cuerpo

Tipos de transacciones HTTP: Métodos o verbos

Solicitudes HTTP

POST ruta HTTP/1.1 <parámetros> Cuerpo
GET ruta HTTP/1.1 <parámetros> Cuerpo
PUT ruta HTTP/1.1 <parámetros> Cuerpo
DELETE ruta HTTP/1.1 <parámetros> Cuerpo

Transacciones del Interfaz Uniforme:

- ◆ **POST:** Crear un recurso en el servidor (Create)
- ◆ **GET:** Traer un recurso al servidor (Read)
- ◆ **PUT:** Modificar un recurso del servidor (Update)
- ◆ **DELETE:** Borrar un recurso del servidor (Delete)

La **interfaz uniforme** (CRUD) permite diseñar interfaces **REST** y **arquitecturas orientadas a recursos** (ROA).

El interfaz uniforme gestiona los recursos del servidor remoto como si fuese una **base de datos** con transacciones **POST** (crear), **GET** (leer), **PUT** (actualizar) y **DELETE** (borrar).

HTTP tiene además otros métodos

- ◆ **HEAD:** Pide solo la cabecera al servidor (si el cuerpo)
- ◆ **OPTIONS:** Determinar qué métodos acepta un servidor
- ◆ **TRACE:** Trazar proxies, caches, ... hasta el servidor
- ◆ **CONNECT:** Conectar a un servidor a través de un proxy
- ◆

Respuestas HTTP

HTTP/1.1 código msg Parámetros Cuerpo

Cabecera y cuerpo de mensajes HTTP

- ◆ Los **mensajes** de solicitud o respuesta constan de **cabecera** y **cuerpo**.
- ◆ **Cabecera:** es un **string** formado por **1a línea, parámetros y marca de final** (línea en blanco: `\n\n`).
 - ◆ **1a línea** incluye **método, ruta y versión HTTP**
 - ◆ **Parámetros:** cada uno ocupa 1 línea y su formato es -> **Nombre: valor\n**
- ◆ **Cuerpo:** solo se incluye al enviar recursos. Puede tener cualquier formato: string, binario, imagen,

Solicitud

Primera línea

método ruta vers-HTTP \n

Host: upm.es \n

Accept: text/*, image/* \n

Accept-language: en, sp \n

....

User-Agent: Mozilla/5.0 \n

\n

Cuerpo

..... Cuerpo

La primera línea incluye:

- **Método:** GET, POST, PUT, DELETE, HEAD, ...
- **Ruta:** /directorio_1/directorio_2/.../fichero
- **Versión HTTP:** HTTP/1.0, HTTP/1.1, HTTP/2

Algunos parámetros de la solicitud

Host: identifica la dirección del servidor para que este accesible al programa (MW)

Accept: tipos MIME de recursos aceptados

Accept-language: lenguajes del cliente

La **cabecera** tiene formato de texto y se compone de primera línea y parámetros. Cada parámetro ocupa una línea y tiene la siguiente estructura **nombre: valor\n**.

La cabecera y el cuerpo van separados del cuerpo por una **línea en blanco:** `\n\n`.

HTTP/2 pasa la cabecera a formato binario y la comprime para mayor eficiencia, pero manteniendo la misma estructura (primera línea y parámetros) y significado.

Respuesta

Primera línea

vers-HTTP código msg \n

Server: Apache/1.3.6 \n

Content-type: text/html \n

....

Content-length: 608 \n

\n

Cuerpo

..... Cuerpo

La primera línea incluye:

- **Versión HTTP:** HTTP/1.0, HTTP/1.1, HTTP/2
- **Código:** 100, 101, .., 200, 201, .., 300, ...
- **Mensaje:** Ok, Continue, Created, ...

Algunos parámetros de la respuesta

Content-type: tipo MIME de recurso, **text/html** es el tipo de una página Web

Content-length: número (decimal) de octetos del cuerpo

Códigos de estado de un servidor Web

◆ Respuestas informativas (1xx)

- 100 Continue // Continuar solicitud parcial

◆ Solicitud finalizada (2xx)

- 200 OK // Operación GET realizada satisfactoriamente, recurso servido
- 201 Created // Recurso creado satisfactoriamente con POST, PUT
- 206 Partial Content // para uso con GET parcial

◆ Redirección (3xx)

- 301 Moved Permanently // Recurso se ha movido, cliente debe actualizar el URL
- 303 See Other // Envía la URI de un documento de respuesta
- 304 Not Modified // Cuando el cliente ya tiene los datos

◆ Error de cliente (4xx)

- 400 Bad request // Comando enviado incorrecto
- 404 Not Found // Recurso no encontrado, no hay ningún fichero con ese path
- 405 Method Not Allowed // Método no permitido, p.e. se solicita método POST, PUT,
- 409 Conflict // Existe conflicto con el estado del recurso en el servidor
- 410 Gone // Recurso ya no esta

◆ Error de Servidor (5xx)

- 500 Internal Server Error // El servidor tiene errores, p.e. error lectura disco,

Tipos MIME

◆ Tipos MIME: definen el tipo de un recurso

- Aparecieron en email para tipar ficheros adjuntos
 - Su uso se ha extendido a otros protocolos y en particular a HTTP
 - Tipos: <http://www.iana.org/assignments/media-types/media-types.xhtml>

◆ Un tipo MIME tiene 2 partes **tipo / subtípo**,

- Los tipos registrados actualmente son:
 - application, audio, font, example, image, message, model, multipart, text, video

◆ Ejemplos:

- **image/gif, image/jpeg, image/png, image/svg,**
- **text/plain, text/html, text/css,**
- **application/javascript, application/msword,**
-

◆ HTTP utiliza el tipo mime para tipar el contenido del cuerpo (body)

- Cabecera Request: “**Accept: text/html, image/png, ...”**
- Cabecera Response: “**Content-type: text/html**”

Solicitud HTTP GET

1a linea	GET /me.htm HTTP/1.1
parámetros de cabecera	Host: upm.es Accept: text/*, image/* Accept-language: en, sp User-Agent: Mozilla/5.0
Cuerpo	

Respuesta HTTP GET

1a linea	HTTP/1.1 200 OK
parámetros de cabecera	Server: Apache/1.3.6 Content-type: text/html
Cuerpo:	Content-length: 608
Pág. HTML	<html> </html>

Ejemplo de transacción HTTP GET

◆ Cuando el servidor recibe la solicitud GET

- Responde con los siguientes parámetros
 - ◆ Devuelve al cliente el **fichero solicitado** en el **cuerpo** de la respuesta junto con el código "**200 OK**"
 - ◆ El parámetro "**Content-Type: text/html**" indica el **tipo MIME** del recurso enviado
 - ◆ El parámetro "**Content-length: 608**" indica el **número de octetos** del cuerpo
- Si el fichero solicitado no existe, envía solo el mensajes de error: **304 Not found**

◆ La extensión del fichero genera su tipo MIME

- xx.htm y xx.html -> text/html
 - xx.gif -> image/gif
 - xx.css -> text/css
 -
- ◆ ver: <http://webdesign.about.com/od/multimedia/a/mime-types-by-file-extension.htm>

◆ El tipo MIME indica al navegador el formato del recurso recibido

- el navegador muestra el código HTML si una página **HTML** lleva el tipo "**text/plain**"
 - ◆ En vez de el tipo mime "**text/html**", que es el tipo que debería llevar

Solicitud HTTP GET

1a linea

parámetros de cabecera

GET /me.htm HTTP/1.1

Host: upm.es
Accept: text/*, image/*
Accept-language: en, sp
User-Agent: Mozilla/5.0

Cuerpo

Respuesta HTTP GET

1a linea

parámetros de cabecera

HTTP/1.1 200 OK

Server: Apache/1.3.6

Content-type: text/html

.....

Content-length: 608

<html> </html>

Cuerpo:

CURL



CURL: comando para transferir datos identificados por un URL, que soporta muchos protocolos: HTTP, HTTPS, IMAP, SMTP, Kerberos,

La opción -v (verbosa) muestra todos los detalles del proceso.

Ver opciones con:

\$ curl --help

.....

\$ man curl

```
venus:~ jq$ curl -v http://localhost:8000/mi_ruta
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET /mi_ruta HTTP/1.1
> User-Agent: curl/7.37.1
> Host: localhost:8000
> Accept: */*
>
< HTTP/1.1 200 OK
< X-Powered-By: Express
< Content-Type: text/plain; charset=utf-8
< Content-Length: 43
< ETag: W/"2b-1b6a7631"
< Date: Mon, 27 Oct 2014 14:31:05 GMT
< Connection: keep-alive
<
<html><body><h1>Mi Ruta</h1></body></html>
* Connection #0 to host localhost left intact
venus:~ jq$ 
```

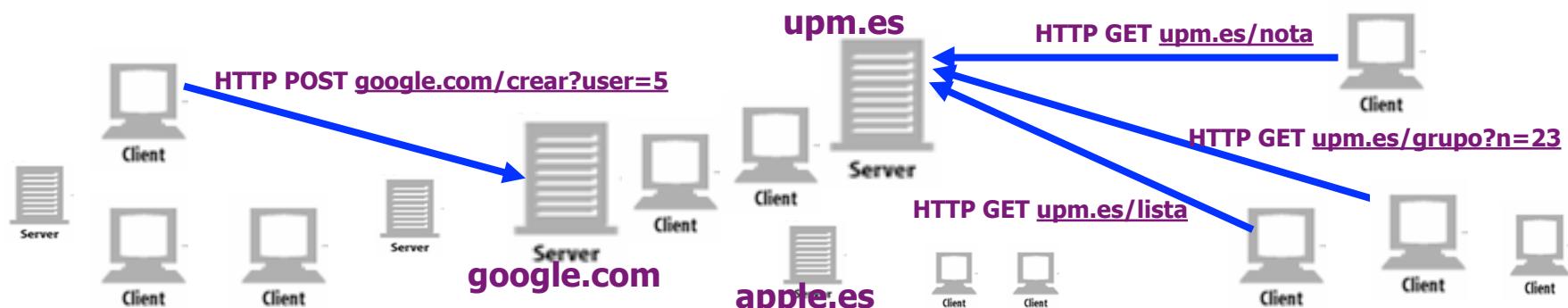
Interfaces REST

◆ Interfaz REST

- Cliente y servidor **interaccionan con HTTP**
 - Cada operación identifica el recurso con una **ruta (path)** diferente
 - Por ejemplo, `/nota/5`, `/user/10`, `/notas/user/5`, `/grupo?n=23`,

◆ Solo utilizan métodos o comandos del **interfaz uniforme**

- **GET:** trae al cliente (lee) un recurso identificado por una ruta
- **POST:** crea un recurso identificado por una ruta
- **PUT:** actualiza un recurso identificado por una ruta
- **DELETE:** borra un recurso identificado por una ruta
- (HTTP tiene mas métodos, pero no pertenecen al interfaz uniforme)





AJAX, Same Origin, CORS y Fetch API

Juan Quemada, DIT - UPM

AJAX - Asynchronous JavaScript & XML|JSON|text|...

- ◆ **AJAX** permite realizar **transacciones HTTP** desde JavaScript en el cliente
 - El soporte inicial de AJAX era el objeto **XMLHttpRequest** (cuando no existía fetch)
 - ◆ Documentación: <https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX>
- ◆ **ES6** añade el método global **fetch** para realizar transacciones HTTP
 - **fetch** permite el uso de **promesas** y es muy simple y eficaz
 - ◆ Documentación: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
 - ◆ <https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch>
- ◆ La librería **jQuery** simplificó el uso de AJAX
 - Facilitó la realización de aplicaciones AJAX, cuando fetch no se soportaba
 - ◆ Independizaba de los detalles de los navegadores
 - Métodos soportados: `jQuery.ajax(...)` o `$.ajax(...)`, `jQuery.get(...)`, `jQuery.post(...)`,
 - ◆ Documentación: <http://api.jquery.com/category/ajax/>
- ◆ Una aplicación AJAX debe economizar proceso y ancho de banda
 - Debe intercambiar con el servidor solo los datos necesarios para actualizar sus vistas
 - ◆ Una aplicación AJAX es más rápida y ágil que la carga de páginas Web completas
 - También se denominan RIA (Rich Internet Applications) o Single Page Applications

Same-origin Security Policy

◆ Origen

- **Protocolo, dominio y puerto** del servidor de donde vino la página o script
 - ♦ Por ejemplo, en la página <http://upm.es/dir/p1.htm> los siguientes recursos tienen
 - Mismo origen: <http://upm.es/lib.js>, <http://upm.es/dir1/style.css>,
 - Origen diferente: <https://upm.es/lib.js>, <http://upm.es:81/dir1/style.css>, <http://google.com>,

◆ La política de **same-origin** es el mecanismo básico de seguridad Web

- Restringe transacciones HTTP-AJAX a otros servidores diferentes de **origin**
 - ♦ Doc:https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

◆ Las transacciones HTTP tradicionales de HTML no están afectadas por la "Same-origin Security Policy"

- **Transacciones GET** generadas con
 - ♦ ``, `<link rel="stylesheet" href="..." ..>`, `<script ..>`, ``, `<audio ..>`,
`<video ..>`, `<iframe ..>`, `<object ..>`, `<form method="GET" >`, ..
- **Transacción POST** generada con: `<form method="POST" >`

CORS: Cross-Origin Resource Sharing

- ◆ La política de seguridad basada en "same-origin" es restrictiva
 - CORS permite flexibilizar esta política manteniendo controles de seguridad
 - ◆ CORS utiliza el navegador como un intermediario confiable
- ◆ CORS necesita extensiones de HTTP para garantizar la seguridad
 - Son extensiones complejas definidas como parte del objeto fetch de **ES6**
 - ◆ Doc: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>,
 - ◆ <https://fetch.spec.whatwg.org> <https://javascript.info/fetch-crossorigin>
- ◆ Las solicitudes HTTP CORS consultan a un servidor antes de acceder
 - El navegador solo realiza la transacción HTTP si servidor acepta el **origin** de la app
- ◆ CORS incluye transacciones simples y complejas
 - Las transacciones simples son similares a las ya existentes en HTML
 - Las transacciones complejas permiten cualquier transacción
 - ◆ Pero el servidor accedido tiene que dar su consentimiento con antelación

Fetch API

◆ Fetch API

- Realiza transacciones AJAX: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

◆ **fetch(url)**

- Realiza una transacción **GET** (trae el **recurso** identificado por **url**)

◆ **fetch(resource, init)**

- Realiza cualquier transacción AJAX (HTTP)
 - Devuelve una **promesa** con un objeto **Response**
- **resource**: puede ser el **url** (dirección) de un recurso o un objeto **Request**
- **init**: objeto opcional que permite configurar parámetros de la solicitud HTTP
- <https://developer.mozilla.org/en-US/docs/Web/API/WindowOrWorkerGlobalScope/fetch>

◆ Headers objeto genérico para gestionar cabeceras (**Request** y **Response**)

- <https://developer.mozilla.org/en-US/docs/Web/API/Headers>

◆ Objetos **Request** y **Response** de la transacción HTTP

- <https://developer.mozilla.org/en-US/docs/Web/API/Request>
- <https://developer.mozilla.org/en-US/docs/Web/API/Response>

Ejemplo AJAX GET: pelis.htm

Esta app está publicado en neocities junto con el fichero JSON (neocities no tiene CORS activado):
<https://juan-quemada.neocities.org/ex/pelis.htm>

AJAX accede al fichero JSON sin problemas, porque la página Web es del mismo origen (neocities) que el fichero JSON. (hacer clic en URL).



Ejemplo AJAX

Respuesta:

["Superlópez", "Jurassic Park", "Interstellar"]

Si cargamos este fichero HTML de otro origen, p.e. un fichero local (<file:///Users/jq/ej/20-pelis.htm>) dará error, salvo que cambiemos el URL del fichero JSON por <https://api.myjson.com/bins//a7k7a>, que tiene CORS activado.



Ejemplo AJAX

Error: TypeError: Origin null is not allowed by Access-Control-Allow-Origin.

```
<!DOCTYPE html><html>
<head><meta charset="UTF-8"></head>
<body>
    <h3>Ejemplo AJAX</h3>

    <div id='res'></div>

    <script type="text/javascript">

        const get = async (url) => {
            try {
                let response = await fetch(url);
                let myJson = await response.json();
                show('res', `Respuesta: <br> ${JSON.stringify(myJson)} `);
            } catch (e) {
                show('res', `Error: ${e}`);
            }
        }

        const show = (id, msj) => document.getElementById(id).innerHTML = msj;

        get('https://juan-quemada.neocities.org/ex/pelis.json');

        // get('https://api.myjson.com/bins//a7k7a');

    </script>
</body>
</html>
```

fetch está configurada por defecto para hacer transacciones GET. Cuando se pasa un URL como parámetro trae del servidor, el recurso asociado al URL, con una transacción GET.

fetch devuelve un objeto de tipo response (respuesta HTTP) con métodos como json(), text() o blob() para obtener el contenido de body en esos formatos.

Servidor que no acepta CORS

Servidor que acepta CORS

Ambos URLs enlazan ficheros JSON con este contenido:
["Superlópez", "Jurassic Park", "Interstellar"]



MyJSON.com y ejemplo AJAX/REST

Juan Quemada, DIT - UPM

MyJSON.com

◆ MyJSON.com

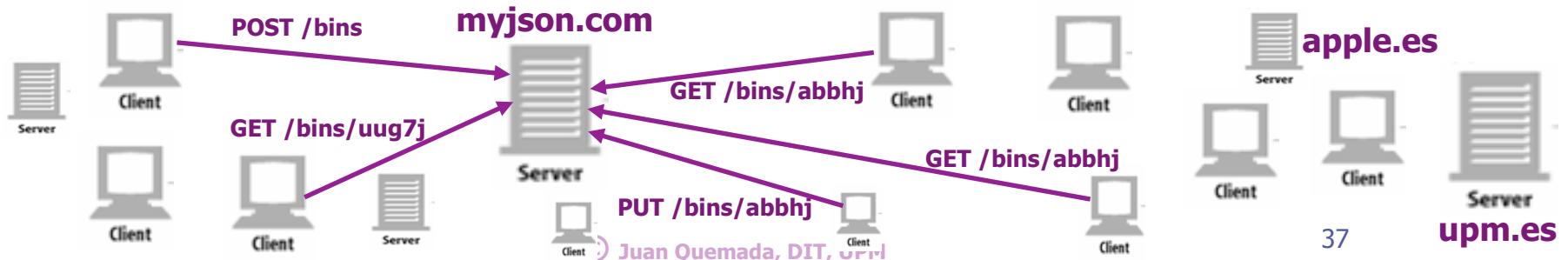
- Repositorio de recursos JSON muy sencillo de utilizar
 - ◆ Acceso **Web** para usuarios (<http://myjson.com>)
 - ◆ Acceso **REST** utilizando AJAX para aplicaciones

◆ Primitivas del **interfaz REST**

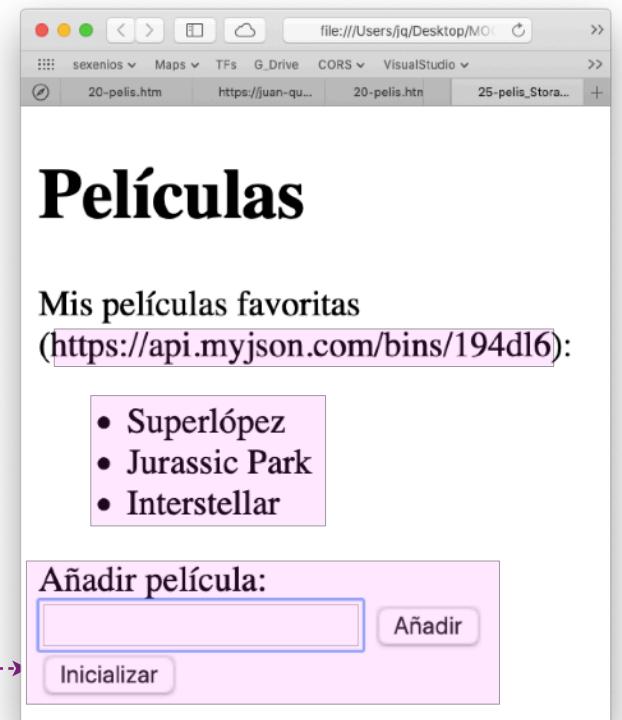
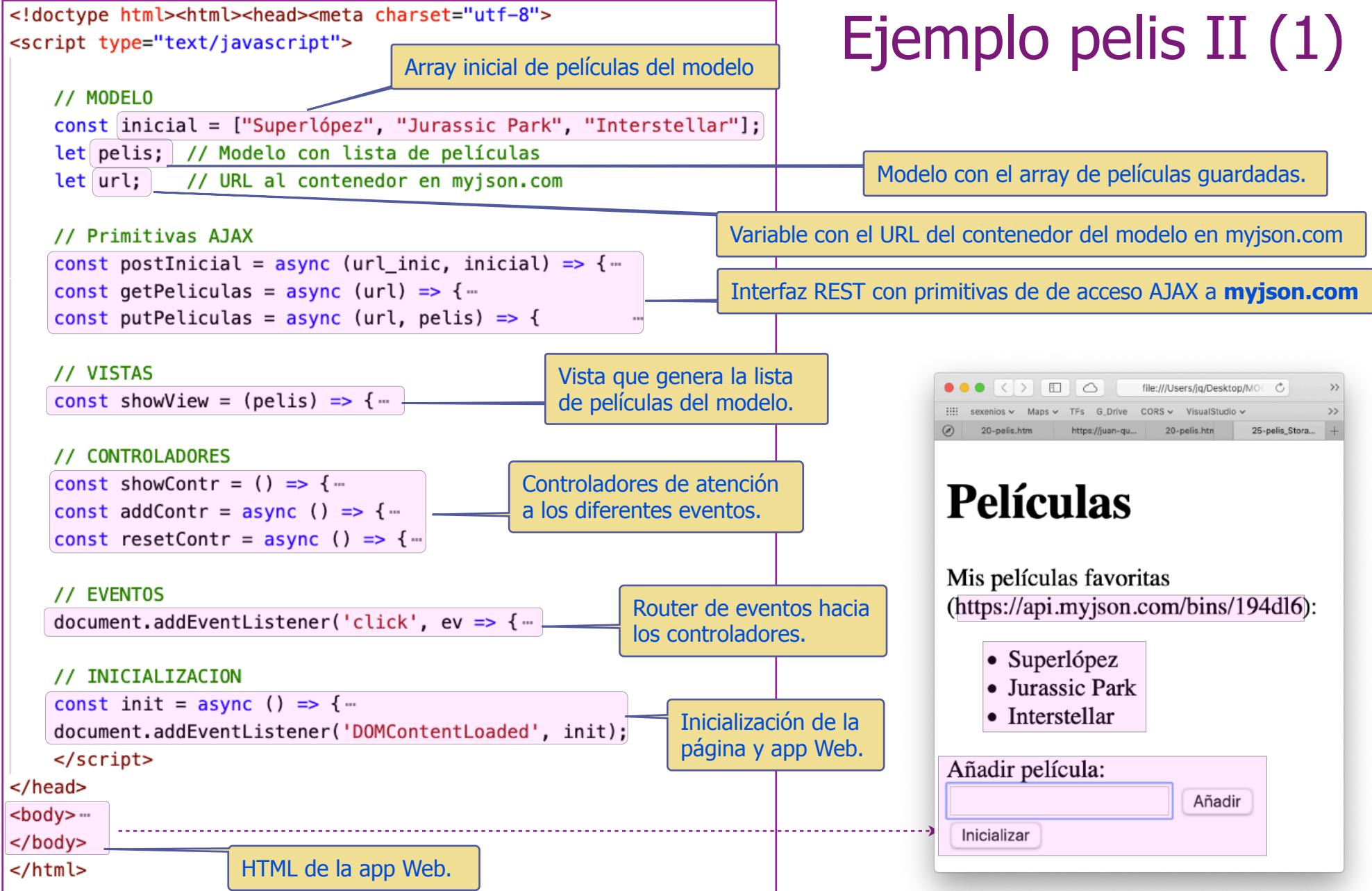
- **GET /bins/:id** trae al cliente el recurso JSON identificado por **:id**
- **POST /bins** crea un recurso JSON y devuelve **url** con **:id** del recurso
- **PUT /bins/:id** actualiza el recurso JSON identificado **:id**

◆ Para usar el repositorio no se necesita crear una cuenta

- solo se necesita el **:id** para traer y actualizar un recurso JSON



Ejemplo pelis II (1)



Ejemplo pelis II (2)

```
// EVENTOS
document.addEventListener('click', ev => {
  if (ev.target.matches('#add')) addContr();
  else if (ev.target.matches('#reset')) resetContr();
})
```

Router de eventos (clic en botones) identificados por id.

Si el contenedor 'url' existe ya en localStorage, se **reutiliza** su contenido.

// INICIALIZACION

```
const init = async () => {
  try {
    url = localStorage.getItem('url')
    || await postInicial('https://api.myjson.com/bins', inicial);
    localStorage.setItem('url', url);
    pelis = await getPeliculas(url);
    showContr();
  } catch (e) { document.getElementById('lista').innerHTML = `Error: ${e}`; }
}
document.addEventListener('DOMContentLoaded', init);
```

URL para crear un contenedor en myjson.com con POST.

Actualizar con nuevo URL o contenido previo.

Traer modelo del contenedor de myjson.com.

Mostrar modelo

Crear contenedor en myjson.com con el contenido de la variable **initial**, si no existe 'url' en localStorage.

postInicial() solo se ejecuta la primera vez que esta app se descarga en el navegador.

```
</script>
```

Bloque vacío para películas del **modelo**.

Bloque vacío para **url** del contenedor en myjson.com.

```
</head>
<body>
```

<h1>Películas</h1>

Mis películas favoritas
():

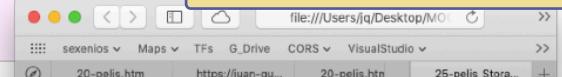
```
<ul id="lista"></ul>
```

Añadir película: <input type="text" id="pelicula">
<button id="add">Añadir</button>

<button id="reset">Inicializar</button>

```
</body>
</html>
```

HTML de los botones y del cajetín.



Películas

Mis películas favoritas

(<https://api.myjson.com/bins/194dl6>):

- Superlópez
- Jurassic Park
- Interstellar

Añadir película:

Ejemplo pelis II (3)

```
const showContr = () => {
  document.getElementById('lista').innerHTML = showView(pelis);
  document.getElementById('url').innerHTML = url;
};
```

```
const showView = (pelis) => {
  let i=0, view = "";
  while (i < pelis.length) view += `<li> ${pelis[i++]}</li>`;
  return view;
};
```

showContr() muestra el URL y las películas de la variable **pelis**, utilizando la vista **showView()**.

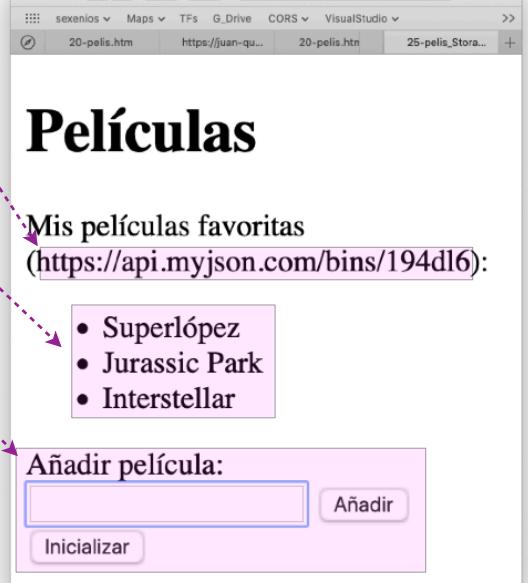
showView() genera un elemento de la lista por cada película del array con este bucle.

```
const addContr = async () => {
  let p = document.getElementById('pelicula').value;
  pelis.push(p);
  await putPeliculas(url, pelis);
  showContr();
};
```

addContr() extrae la película tecleada del cajetín, la añade como último elemento del array guardado en **pelis**, sincroniza el estado de la variable **pelis** con el contenedor de **myjson.com** y muestra el modelo actualizado.

```
const resetContr = async () => {
  pelis = [...inicial];
  await putPeliculas(url, pelis);
  showContr();
};
```

resetContr() inicializa la variable **pelis** con el array **inicial** (el array se clona con el operador spread, porque sino los cambios en **pelis** modificarían **inicial** también), sincroniza el estado de la variable **pelis** con el contenedor de **myjson.com** y muestra el modelo actualizado.



```

const postInicial = async (url_inic, inicial) => {
  let response = await fetch(
    url_inic,
    { method: 'POST',
      headers: { "Content-Type": "application/json", },
      body: JSON.stringify(inicial)
    }
  );
  return (await response.json()).uri;
}

```

postInicial() realiza un **POST** con el array inicial de películas (variable **inicial**) en JSON al servidor <https://api.myjson.com/bins/>.
let pelis = ["Superlópez", "Jurassic Park", "Interstellar"]

Ejemplo pelis II (4)

json() espera a que llegue todo el JSON del body y lo devuelve como objeto JavaScript.

```

const getPeliculas = async (url) => {
  let response = await fetch(url);
  return await response.json();
}

```

json() espera a que llegue todo el JSON del body y lo devuelve como objeto JavaScript.

Para que **fetch** realice una transacción **POST** hay que pasarle el segundo parámetro **init** con datos de la cabecera.

getPeliculas(url) trae con **GET** el array de películas guardado en el contenedor del servidor myjson.com identificado por <https://api.myjson.com/bins/194dl6>.

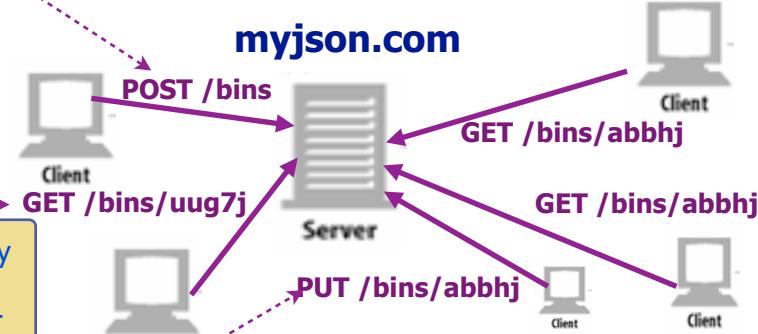
fetch está configurado por defecto para **GET** y solo necesita el url para ordenar esta transacción.

```

const putPeliculas = async (url, pelis) => {
  let response = await fetch(
    url,
    { method: 'PUT',
      headers: { "Content-Type": "application/json", },
      body: JSON.stringify(pelis)
    }
  );
  return await response.json();
}

```

json() espera a que llegue todo el JSON del body y lo devuelve como objeto JavaScript.



Películas

Mis películas favoritas
<https://api.myjson.com/bins/194dl6>:

- Superlópez
- Jurassic Park
- Interstellar

Añadir película:



Librería jQuery y CDN Web

Juan Quemada, DIT - UPM

Librería jQuery



The screenshot shows the official jQuery website. At the top, there's a navigation bar with links for 'Download', 'API Documentation', 'Blog', 'Plugins', 'Resource Directory', and 'Search'. Below the navigation, there are three main sections: 'Lightweight', 'Fastest', 'CSS3 Compliant', and 'Cross-Browser'. A central text block explains what jQuery is, mentioning it's a fast, small, and feature-rich JavaScript library. To the right, there's a sidebar titled 'Resources' with links to 'jQuery Core API Documentation', 'jQuery Plugins Doctor', 'jQuery UI', 'Get Started', 'Contributing to jQuery', 'About the jQuery Foundation', and 'Direct or Submit a GitHub Issue'. At the bottom, there are links for 'jQuery Unit', 'Sizzle', and 'jQuery Mobile'.

◆ Librería jQuery

- Librería **multi-navegador** con el lema: **write less, do more**
 - Ejecuta en Chrome, Firefox, Safari, Edge, IE, Opera, ...
 - Documentación y descarga: <http://jquery.com/>

◆ jQuery incluye muchas funcionalidades

- Manipulación de DOM, eventos, estilos CSS, AJAXs, ...
 - En este tema se ven ejemplos de usos significativos de jQuery

◆ ES6 hace jQuery innecesario, se utiliza "Vanilla JavaScript"

- <https://tobiasahlin.com/blog/move-from-jquery-to-vanilla-javascript/>
 - Pero todavía existen muchos sites que utilizan jQuery



Objetos y función jQuery: `$(..)`

- ◆ **jQuery** representa los **objetos DOM** como **objetos jQuery**
 - Los **objetos jQuery** permiten **procesar** el árbol DOM de forma **más eficaz**
 - ◆ **Arrays de objetos jQuery** se procesan con **un solo método**, sin necesidad de bucles
- ◆ Función **jQuery**: `jQuery("<selector:CSS>")` o `$(<selector_CSS>")`
 - devuelve la colección de **objetos jQuery** que casan con el **<selector CSS>**
 - ◆ Si no casa ninguno, devuelve un **objeto jQuery vacío**
 - **<selector CSS>** utiliza la **sintaxis CSS** para seleccionar objetos DOM

```
document.getElementById("fecha")  o  document.querySelectorAll("#fecha")
// es equivalente a:
$("#fecha")
```

Selectores tipo CSS de jQuery

SELECTORES DE ELEMENTOS HTML CON ATRIBUTO ID

```
$("h1#d83")      /* devuelve objeto con marca h1 e id="d83" */  
$("#d83")         /* devuelve objeto con con id="d83" */
```

SELECTORES DE ELEMENTOS HTML CON ATRIBUTO CLASS

```
$("h1.princ")    /* devuelve array de objetos con marcas h1 y class="princ" */  
$(".princ")       /* devuelve array de objetos con class="princ" */
```

SELECTORES DE ELEMENTOS HTML CON ATRIBUTOS

```
$("h1[border]")   /* devuelve array de objetos con marcas h1 y atributo border */  
$("h1[border=yes]") /* devuelve array de objetos con marcas h1 y atributo border=yes */
```

SELECTORES DE ELEMENTOS HTML UTILIZANDO LA MARCA

```
$("h1, h2, p")    /* devuelve array de objetos con marcas h1, h2 y p */  
$("h1 h2")         /* devuelve array de objetos con marca h2 después de h1 en el árbol */  
$("h1 > h2")       /* devuelve array de objetos con marca h2 justo después de h1 en arbol */  
$("h1 + p")        /* devuelve array de objetos con marca p adyacente a h1 del mismo nivel */
```

.....

Métodos de jQuery: ejemplos



◆ Método **html(< código html >)**

- `$("#h1").html("Hello World!")` sustituye por **Hello World!** el **innerHTML** de todos los elementos **h1**

◆ Método **html()**

- `$("#h1").html()` devuelve el **innerHTML** del primer **h1** encontrado

◆ Método **append("Hello World!")**

- `$("#id3").append("Hello World!")` añade "Hello World!" al **innerHTML** del elemento con `id="id3"`

◆ Método **val(< valor >)**

- `$("#id3").val("3")` asigna el **valor "3"** al atributo **value** del elemento con `id="id3"`

◆ Método **attr(< atributo >, < valor >)**

- `$(".lic").attr("rel", "license")` asigna "license" al atributo **rel** a todos los elementos con `class="lic"`

◆ Método **addClass(< valor >)**

- `$("ul").addClass("visible")` asigna el **valor "visible"** al atributo **class** de todos los elementos ``

◆ Métodos **hide()** y **show()**

- **Oculta o muestra** objetos DOM de la página HTML cargada en el navegador

◆ Ejemplo de composición serie: `$("#h1").html("Title").addClass("view").show()`

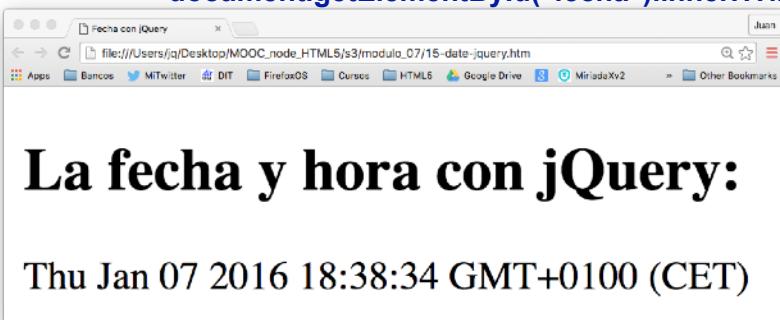
- Asigna el HTML "**Title**" a todos los `<h1>`, les añade el atributo `class="view"` y los hace visibles

◆ La API de jQuery es grande y esta bien estructurada (es útil conocer sus detalles)

- Más información en: <http://api.jquery.com/category/manipulation/>, <https://www.w3schools.com/jquery/>

Fecha y hora con jQuery

- ◆ Una librería JavaScript externa se identifica por su **URL**:
 - <script type="text/javascript" src="jquery-2.1.4.min.js.js" > </script>
- ◆ **\$("#fecha")** obtiene el objeto jQuery
 - del **elemento HTML** con **id="fecha"**
- ◆ **\$("#fecha").html(new Date())**
 - inserta **new Date()** como **HTML interno**
 - del **objeto jQuery** devuelto por **\$("#fecha")**
 - es equivalente a
 - **document.getElementById("fecha").innerHTML = new Date();**



Selecciona el elemento DOM con atributo id="fecha": <div id="fecha"></div>.

```
<!DOCTYPE html>
<html>
<head>
<title>Fecha con jQuery</title>
<script type="text/javascript"
       src="jquery-3.4.1.min.js">
</script>
</head>

<body>
<h2>La fecha y hora con jQuery:</h2>

<div id="fecha"></div>

<script type="text/javascript">
  $('#fecha').html(new Date());
</script>
</body>
</html>
```

Asigna la fecha y hora a innerHTML del objeto DOM seleccionado.

Función ready: árbol DOM construido

◆ `$(document).ready(function() { ..código.. })`

- Ejecuta el código (bloque) de la función cuando el **árbol DOM está construido**
 - Es decir, dicho bloque se ejecuta cuando ocurre el evento `onload` de `<body>`
- Se recomienda utilizar la invocación abreviada: `$(function() { ..código.. })`

```
<html>
<head>
<script type="text/javascript" src="jquery-3.4.1.min.js" >
</script>

<script type="text/javascript">
  $(function() { $('#fecha').html(new Date()); });
</script>
</head>
<body>
<h2>Fecha y hora (ready):</h2>

<div id="fecha"></div>
</body>
</html>
```



Cache y CDN (Content Distribution Network)

- ◆ Cache: contiene recursos cargados anteriormente durante la navegación
 - La cache identifica los recursos por igualdad de URLs
 - ◆ Un nuevo recurso se carga de alguna cache (navegador, ..) si tiene el mismo URL que otro ya guardado
 - Cargarlo de la cache es más rápido que bajarlo del servidor, especialmente de la del navegador
- ◆ CDNs Web: utilizan el mismo URL (a Google, jQuery, ...) en muchas páginas
 - Así se maximiza la probabilidad de que los recursos estén ya en la cache

```
<html>
<head>
<script type="text/javascript"
        src="https://code.jquery.com/jquery-3.4.1.min.js">
</script>

<script type="text/javascript">
    $(function() { $('#fecha').html(new Date( )); });
</script>
</head>
<body>
<h2>Fecha y hora, con CDN de jQuery</h2>

<div id="fecha"></div>
</body>
</html>
```





Los 4 usos de la función jQuery: \$(..)

◆ Acceso a DOM: `$(“selector CSS”)`

- Devuelve un **array** con los **objetos jQuery** que casan con **<selector CSS>**
 - ◆ Programas mas **cortos, eficaces y multi-navegador** que con JavaScript directamente

◆ Esperar a DOM-construido: `$(function(){..código..})`

- **Ejecuta el código** de la función cuando el **árbol DOM está construido**
 - ◆ Equivalente a **ejecutar el código** asociado al evento **onload**

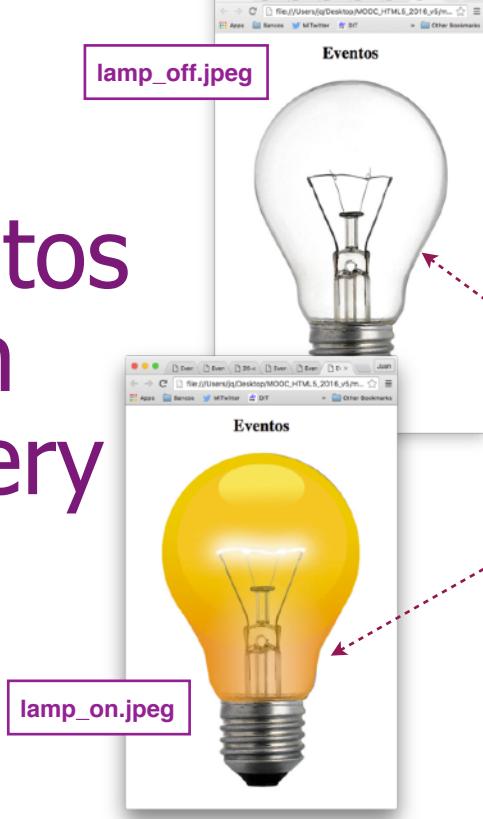
◆ Convierte HTML a objeto jQuery: `$(“UnoDos”)`

- Devuelve **objeto jQuery** equivalente al **HTML**
 - ◆ Mecanismo simple para convertir **HTML** en **jQuery**

◆ Convierte objeto DOM a jQuery: `$(document.getElementById("fecha"))`

- Transforma **objeto DOM** en **objeto jQuery** equivalente
 - ◆ Tiene compatibilidad total con **DOM** y con otras librerías basadas en **DOM**

Eventos en jQuery



```
<!DOCTYPE html>
<html><head><title>Evento jQuery</title><meta charset="UTF-8">
<style> body{text-align:center;} </style>

<script type="text/javascript" src="jquery-3.4.1.min.js" > </script>
<script type="text/javascript">
$(function(){
    let img = $('#i1');

    img.on('dblclick', function(){img.attr('src', 'lamp_on.jpg')});
    img.on('click',     function(){img.attr('src', 'lamp_off.jpg')});
});

</script>
</head>
<body>
    <h1>Eventos</h1>
    
</body>
</html>
```

- ◆ jQuery permite también la definición de eventos en objetos con el método **on()**
 - **objetojQuery.on(tipo_ev, manejador)**
 - Mas información en: <https://api.jquery.com/on/> o <http://api.jquery.com/category/events/>
- ◆ El **manejador** es un literal de función: **(event) => { .. código.. }** o **function (event) { .. código.. }**
 - **event:** objeto con información del evento ocurrido: <http://api.jquery.com/category/events/event-object/>
 - El parámetro **event** no necesita utilizarse en este ejemplo, pero se ha incluido para hacerlo explícito
- ◆ Los **tipos de eventos (tipo_ev)** utilizados por **on(..)** y **off()** son los de **addEventListener(..)**
 - Más información en <https://developer.mozilla.org/en-US/docs/Web/Events>

Calculadora jQuery

Obtener objeto jQuery (DOM) deL cajetín: `$("#n1")`

Obtener objeto jQuery (DOM) del botón: `$("#b1")`

◆ jQuery simplifica la calculadora

◆ Modificaciones

- Debemos **importar la librería jQuery**
- Definir eventos en **función ready**
 - con **método on(..)**
 - y con árbol DOM ya construido
- **Obtener objetos jQuery con `$("#...")`**
- **Obtener texto de cajetín con `val()`**
- **Asignar texto en cajetín con `val(texto)`**



```
<!DOCTYPE html>
<html><head><title>Calculadora</title>
    <meta charset="utf-8">
    <script type="text/javascript"
        src="jquery-3.4.1.min.js">
    </script>
    <script type="text/javascript">
$(function() {
    $("#n1").on("click",
        function(){ $("#n1").val(""); });
    );
    $("#b1").on("click",
        function() {
            let num = $("#n1");
            num.val(num.val() * num.val());
        });
    );
    </script>
</head>
<body>
    Número:
    <input type="text" id="n1">
    <button id="b1"> x<sup>2</sup> </button>
</body>
</html>
```

Importar librería jQuery del mismo directorio de la app

Evento click en cajetín

Vaciar el cajetín

Evento click en botón x^2

Calcular resultado obteniendo el string tecleado en cajetín con `num.val()` y guardando el resultado con `num.val(..resultado..)`.

"Bubbling" de eventos (o eventos delegados)

- ◆ Los eventos burbujean ("bubbling") hacia arriba en el árbol DOM
 - Primero se ejecutan los manejadores (si existen) del elemento DOM sobre el que se ha hecho clic
 - Luego se ejecutan los manejadores (si existen) del elemento DOM que **contiene al anterior**
 - Y así hasta llegar a la raíz
- ◆ **elemento.on(event, selector, manejador)**
 - **selector:** restringe **los descendientes** de **elemento** que disparan el evento, por ejemplo
 - ".class_a" restringe a los elementos de esta clase
 - "#n1" restringe al elemento con este identificador
 - Doc: <https://api.jquery.com/on/>
- ◆ **this** referencia el elemento DOM asociado al manejador en ejecución
- ◆ **event.target** es una referencia al elemento que provocó el evento
 - **event** es el parámetro del manejador en
 - `on(tipo_ev, selector, (event) => {.. código ..})`



```
<!DOCTYPE html>
<html><head><title>Calculadora</title>
| | | | | <meta charset="utf-8">
| | | | | <script type="text/javascript"
| | | | | | src="jquery-3.4.1.min.j
| | | | | </script>
| | | | | <script type="text/javascript">
| | | | | $("body").on("click", "#n1",
| | | | | | function(){ $("#n1").val("");}
| | | | | );
| | | | | $("body").on("click", "#b1",
| | | | | | function() {
| | | | | | let num = $("#n1");
| | | | | | num.val(num.val() * num.val());
| | | | | }
| | | | | );
| | | | | </script>
| | | | </head>
| | | <body>
| | | | Número:
| | | | <input type="text" id="n1">
| | | |
| | | | <button id="b1"> x<sup>2</sup> </button>
| | | |
| | | </body>
| | </html>
```

Evento click en cajetín:
click en elemento
contenido en **<body>**
con **id="n1"**

Evento click en botón x²:
click en elemento
contenido en **<body>**
con **id="b1"**.

Los dos **manejadores** de eventos se
asocian a **<body>**, pero la activación del
evento se filtra en función del
identificador del evento clicado ("target").

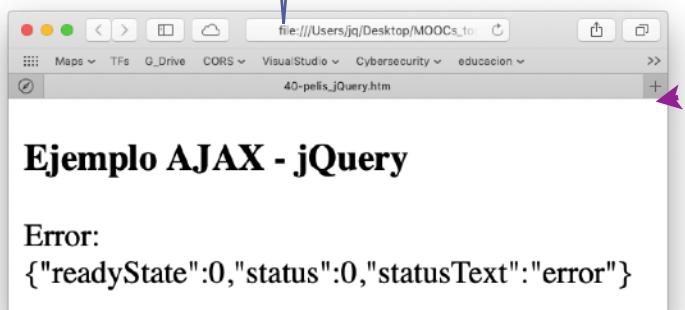
Fichero en neocities junto con fichero JSON:
https://juan-quemada.neocities.org/ex/pelis_jQuery.htm

AJAX accede al fichero JSON sin problemas, porque la página Web se ha descargado desde un URL del mismo origen que el fichero JSON. Acceder a este ejemplo a través del URL anterior para comprobarlo.



En cambio, cuando esta página se carga de otro origen, por ejemplo de un fichero local, la política de "same-origin" rechaza el intento de acceso AJAX al fichero JSON desde otro origen.

El fichero local funciona bien si sustituimos el URL por <https://api.myjson.com/bins/a7k7a>, que enlaza con el mismo fichero JSON pero en otro servidor, que permite CORS con cualquier origen.



Ejemplo AJAX: pelis-jQuery.htm

- ◆ El método ajax(..) es sencillo de utilizar
 - Se puede configurar de diversas formas
- ◆ Aquí se configura con un objeto de configuración
 - <https://api.jquery.com/jquery.ajax/>
 - <https://openclassrooms.com/en/courses/4309491-simplifica-tus-proyectos-con-jquery/4981961-el-metodo-ajax>

```
<!DOCTYPE html><html>
<head><meta charset="UTF-8">
<script type="text/javascript" src="jquery-3.4.1.min.js"></script></head>
<body>
    <h3>Ejemplo AJAX - jQuery</h3>

    <div id='res'></div>

    <script type="text/javascript">
        $ajax( { type: 'GET',
                  url: 'https://juan-quemada.neocities.org/ex/pelis.json',
                  success: function(response){
                      $('#res').html(`Respuesta: <br> ${JSON.stringify(response)}`);
                  },
                  error: function(err){
                      $('#res').html(`Error: <br> ${JSON.stringify(err)}`);
                  }
        })
    </script>
</body>
</html>
```

En este URL hay un fichero JSON que contiene:
["Superlópez", "Jurassic Park", "Interstellar"]

Callback de error.

Callback de éxito.



JavaScript



Final del tema