



JavaScript



express.js, Middleware, Formularios, MVC y AJAX

Juan Quemada, DIT - UPM

Express

1. <u>Aplicación express, MW, static, solicitud (req), respuesta (res) y API REST</u>	3
2. <u>Composición y ejecución secuencial de MWs con next() y next(Error)</u>	10
3. <u>Formularios GET y POST, URL encode y parámetros ocultos</u>	16
4. <u>Patrón MVC de servidor</u>	27
5. <u>MVC, Interfaz Uniforme, method-override, POST, PUT y DELETE</u>	32
6. <u>AJAX - Asynchronous JavaScript & XML</u>	38



JavaScript

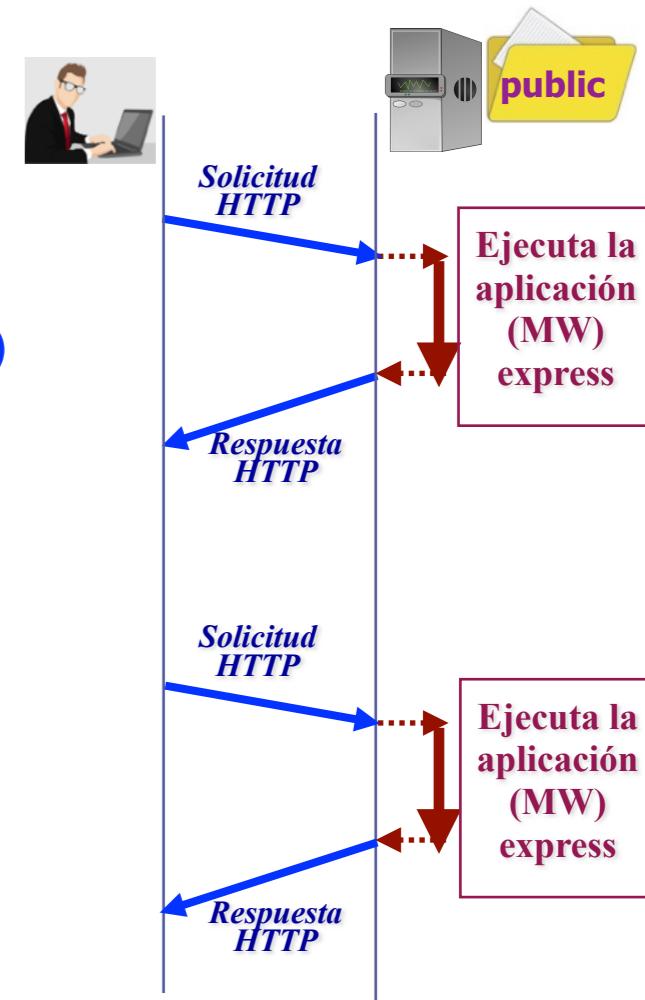


Express

Aplicación, MW, static, solicitud (req), respuesta (res) y APIs REST

Juan Quemada, DIT - UPM

Express: Aplicación y MW



◆ Express

- Librería node para crear **aplicaciones REST** de servidor (<http://expressjs.com>)
 - ◆ "\$ npm i express" instala la última versión de express
 - ◆ "\$ npm i express@4.16.2" instala la versión 4.16.12 (utilizada para estos ejemplos)
- **express-generator**: genera esqueleto de aplicación REST basada en **MVC**
 - ◆ Ver: <http://expressjs.com/en/starter/generator.html>

◆ Aplicación Express

- Permite instalar **middlewares (MWs)** que responden a solicitudes
 - ◆ El método **use(<MW>)** instala MW en la aplicación express en la que se invoca
 - La aplicación express **invoca el MW** instalado cuando llega una **solicitud HTTP**

◆ MiddleWare (MW) Express

- **Función** invocada por la aplicación para **responder a solicitudes HTTP**
 - ◆ La función recibe en su **primer parámetro** un objeto con la **solicitud HTTP**
 - **express.static(<repositorio>)** instala un MW con el servidor estático de express

```
var express = require('express');
var path = require('path');

var app = express();

app.use(express.static(path.join(__dirname, 'public')));

app.listen(8080);
```

El método **use(...)** **instala** el MW en el esqueleto de la aplicación express.

var express = require('express') importa el paquete express (instalado con: **npm i express**).

var path = require('path') importa el módulo path de gestión de rutas.

var app = express() crea el esqueleto de la aplicación express sobre la que se pueden instalar MWs.

path.join(__dirname, 'public'): crea ruta absoluta al directorio del repositorio de recursos (public).

static(<repositorio>) devuelve un MW con un **servidor estático** ejecutado con cada solicitud HTTP. El parámetro <repositorio> debe contener la ruta al **repositorio de recursos** (directorio **public**). El MW responde a cada solicitud enviando una respuesta HTTP con el recurso solicitado (si lo tiene).

Conecta la aplicación express al puerto 8080 para que responda a las solicitudes que llegan a ese puerto.

Instalación y definición de MWs

◆ **use, all, post, get, put, delete, head, ...**

- Métodos para instalar MWs **asociados** a tipos de solicitudes (métodos o verbos)

◆ Un **MW** puede instalarse asociado a una **ruta**

- El MW solo se invoca si la **ruta de la solicitud y la asociada** coinciden, por ej.
 - ◆ `app.use('/user/3', MW)` invocará MW con cualquier solicitud con la ruta '`user/3`'
 - ◆ `app.get('/user/3', MW)` invocará MW con solicitudes GET con la ruta '`user/3`'
 - Rutas <http://expressjs.com/en/guide/routing.html>

◆ Una app evalúa los MWs instalados siguiendo el orden de instalación

- Invoca el primer MW cuya **método y ruta** casan con los de la solicitud recibida

◆ **MW**

- Es una función: `(req, res, next) => {...código...}`
 - ◆ El código de la función define la respuesta a la solicitud que atiende
 - Ver: <http://expressjs.com/en/guide/writing-middleware.html>

◆ **req**

- Objeto JavaScript con los parámetros de la **solicitud** recibida
 - ◆ `req.method`, `req.path`, `req.url`, ...

◆ **res**

- Objeto JS que permite construir y enviar la **respuesta**
 - ◆ `res.type(text/html)` configura el tipo mime de la respuesta HTTP
 - ◆ `res.status(200)` configura el código de la respuesta HTTP
 - ◆ `res.send(..texto-del-body..)` envía la respuesta HTTP con los parámetros configurados

Ejemplo II: MWs



Ejemplo III: Configurar respuestas

`http://localhost:8080/mime/preconfigured`



```
var express = require('express');
var app = express();

app.get('/mime/preconfigured', function (req, res){
    res.send('<html><body><h1>Mi Ruta</h1></body></html>\n');
});

app.get('/mime/text/plain', function (req, res){
    res.type('text/plain');
    res.status(200);
    res.send('<html><body><h1>Mi Ruta</h1></body></html>\n');
});

app.get('/mime/text/html', function (req, res){
    res.type('text/html');
    res.status(200);
    res.send('<html><body><h1>Mi Ruta</h1></body></html>\n');
});

app.listen(8080);
```

`http://localhost:8080/mime/text/plain`



`http://localhost:8080/mime/text/html`



APIs REST con express

◆ Las definición de **rutas** express facilita la creación de APIs **REST**

- Una **ruta** tiene segmentos separados por "/", por ejemplo **/hola/que/tal** (3 segmentos)
 - Documentación: <http://expressjs.com/en/guide/routing.html>

◆ Segmentos de tipo **literal**

▪ /client, /game, /quiz/hola, /que, /23, ...	acepta solo las rutas de los literales
▪ /pep* (* se sustituye por cualquier string)	acepta las rutas /pep, /pepppp, /pep/ito, ...
▪ /car? (? casa con 0 o 1 ocurrencias)	acepta las rutas /ca o /car
▪ /c(ar)?	acepta las rutas /c o /car
▪ /car+ (+ casa con 1 o mas ocurrencias)	acepta las rutas /car, /carr, /carr, ...
▪ /c(ar)+	acepta las rutas /car, /carar, /cararar, ...

◆ Segmentos de tipo **parámetro** (aceptan cualquier valor)

- Los parámetros llevan ":" delante del nombre, por ejemplo: **/:x, /:x1, /:x_1, /:model,**
 - El nombre de un parámetro puede tener **letras ASCII, dígitos decimales o "_"**
- Un segmento puede incluir parámetros separados por "-" o por ".", por ej. **/:from-:to, /:file.:extention**

◆ El valor de un **parámetro** se restringe con una RegExp entre paréntesis

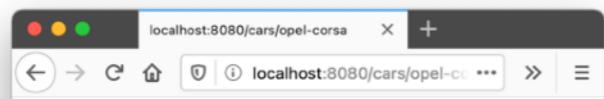
- **/:idioma(es | en | ge)** el parámetro **idioma** solo acepta los strings: "es", "en" o "ge"
- **/:x([0-9]+) o /:x(\d+)** el parámetro **x** solo acepta un número decimal

◆ Los parámetros de la ruta están accesibles en el objeto **req.params** de un MW

- por ejemplo, el valor de **/:x1 o /:from-:to,**
 - Estará accesible en el MW en: **req.params.x1, req.params.from y req.params.to**
 - Documentación: <http://expressjs.com/en/4x/api.html#req.params>

Ejemplo IV: Parámetros de ruta y query

`http://localhost:8080/cars/opel-corsa`



Car model: opel-corsa

`http://localhost:8080/users/13`



User number: 13

`http://localhost:8080/file/example.js`



Route params: name(example), ext(js)

`http://localhost:8080/file?name=example&ext=js`



Query params: name(example), ext(js)

```
const express = require('express');
```

```
let app = express();
```

```
app.get('/cars/:model', function (req, res){
```

```
  let model = req.params.model;
```

```
  res.send(`Car model: ${model}`);
```

```
});
```

```
// RegExp '[0-9]+' is equivalent to '\d'
```

```
app.get('/users/:id([0-9]+)', function (req, res){
```

```
  let id = req.params.id;
```

```
  res.send(`User number: ${id}`);
```

```
});
```

```
app.get('/file/:name.:ext', function (req, res){
```

```
  let {name, ext} = req.params;
```

```
  res.send(`Route params: name(${name}), ext(${ext})`);
```

```
});
```

```
app.get('/file', function (req, res){
```

```
  let {name, ext} = req.query;
```

```
  res.send(`Query params: name(${name}), ext(${ext})`);
```

```
});
```

```
app.get('*', function (req, res){
```

```
  res.send('Unknown request');
```

```
});
```

```
app.listen(8080);
```

Unknown request



JavaScript



Express

Composición y ejecución secuencial
de MWs con next() y next(Error)

Juan Quemada, DIT - UPM

Composición secuencial de MWs: next

- ◆ Una aplicación express instala MWs asociados a rutas y métodos
 - Al llegar una solicitud HTTP los evalúa siguiendo el **orden** de instalación
 - ◆ Invoca el primer MW cuyo **método** y **ruta** casan con los de la **solicitud recibida**
 - Ver: <http://expressjs.com/en/guide/writing-middleware.html>
- ◆ Un MW es una función: **(req, res, next) => { ...código... }**
- ◆ **next**
 - Invocar **next()** indica a la app que debe continuar evaluando MWs al finalizar del actual
 - ◆ La app continuara evaluando **MWs** hasta encontrar otro cuyo **método** y **ruta** casen con los recibidos
 - Si un MW **no invoca next()** la atención a la solicitud finaliza con él
 - ◆ Alguno de los **MWs invocados** deberá haber enviado la **respuesta** a la **solicitud recibida** o esta **no se enviará**
- ◆ **req** y **res** son objetos persistentes entre MWs de una transacción
 - Un MW puede **crear** o **procesar** propiedades de **req** y **res** para los MWs siguientes
- ◆ Objetos **app.locals** y **res.locals**
 - Sus propiedades deben utilizarse para **pasar datos entre MWs** de la app express
 - ◆ **app.locals** crea propiedades persistentes entre transacciones HTTP
 - ◆ **res.locals** crea propiedades para una transacción HTTP, que desaparecen al acabar
- ◆ Los middlewares permiten programación secuencial y modular
 - Conservando la eficacia de la programación por eventos

Ejemplo V: evaluación secuencial de MWs



Incrementa el contador e invoca **next** para que app **siga evaluando** los siguientes MWs.

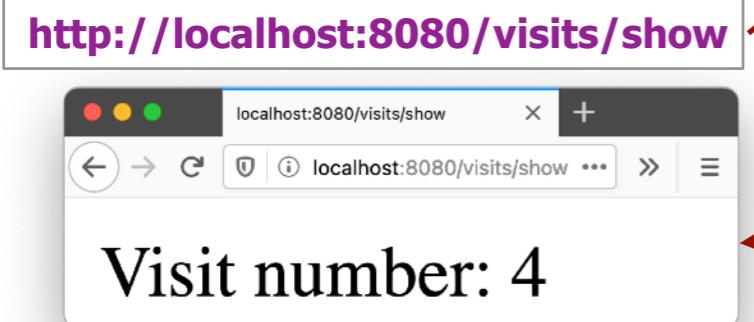
http://localhost:8080/ejemplo.html

```
const express = require('express');
let app = express();
```

app.locals: objeto para crear propiedades locales a medida de la app, como **visits**.

```
app.use(function(req, res, next){
  app.locals.visits = (++app.locals.visits || 1);
  next();
});
```

El MW static **envía la respuesta** con el recurso solicitado y **finaliza**, sí está en el repositorio. Si no lo tiene, invoca **next** para que un **MW posterior atienda** la solicitud.



```
app.use(express.static('public'));
```

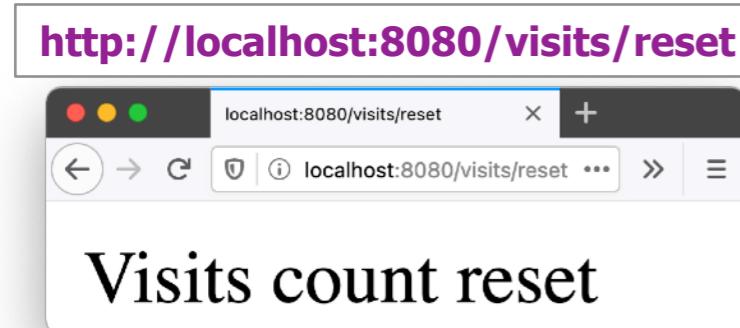
```
app.get('/visits/show', function (req, res){
  res.send(`Visit number: ${app.locals.visits}`);
});
```

```
app.get('/visits/reset', function (req, res){
  app.locals.visits = 0;
  res.send('Visits count reset');
});
```

```
app.get('*', function (req, res){
  res.send('Not supported');
});
```

```
app.listen(8080);
```

Los tres últimos MWs envían la respuesta y finalizan.



MW-de-error: next(err)

◆ Middleware de error

- MW que atiende situaciones de error o excepcionales (como las excepciones)
 - ◆ `next(err)` indica esa **situación excepcional** a la app express, para que invoque el siguiente **MW-de-error**
 - Ver: <http://expressjs.com/en/guide/using-middleware.html>

◆ MW-de-error: `(err, req, res, next) => { ...código... }`

- **err**: parámetro de error pasado por el MW que lanzó el error con `next(err)`
- **req**: objeto con los parámetros de la solicitud HTTP
- **res**: objeto donde se insertan los parámetros de la respuesta HTTP
- **next**: función que indica a la app express que evalúe los siguientes MWs

◆ `next()` vs `next(err)`

- `next()` indica a la app que debe evaluar solo los siguientes **MWs**
- `next(err)` indica a la app que debe evaluar solo los siguientes **MWs-de-error**

◆ Los **MWs de error** se instalan en la app express, igual que los otros MWs

- Se **instalan y asocian a rutas** con: `use`, `all`, `post`, `get`, `put`, `delete`, `head`, ...
 - ◆ La app evalúa los siguientes **MWs** o **MWs-de-error** a partir del que invoca `next()` o `next(err)`, siguiendo el orden de instalación
 - Alguno de los **MWs** o **MWs-de-error** invocados debe enviar la respuesta a la solicitud recibida o no se enviará

◆ Los **MWs-de-error** estructuran la atención a los errores de un programa

- Agrupan la atención a errores en **bloques aparte** (MWs-de-error), normalmente hacia el final

Ejemplo VI: next(err)

Este MW invoca `next()` si la solicitud es de tipo GET y si no, invoca: `next(new Error("405 Method not Allowed"))`.



```
const express = require('express');
let app = express();

app.use('*', function(req, res, next){
  if (req.method.toLowerCase() !== "get") {
    res.locals.status = "405";
    next(new Error("405 Method Not Allowed"));
  }
  next();
});

app.use(express.static('public'));

app.get('*', function(req, res, next){
  res.locals.status = "404";
  next(new Error("404 Not Found"));
});

app.use(function(err, req, res, next){
  res.status(res.locals.status);
  res.send(err.toString());
});

app.listen(8080);
```

Indica que el servidor estático no tiene el recurso solicitado invocando: `next(new Error("404 Not Found"))`.

El **MW-de-error** envía la respuesta de cualquier tipo de error notificado por los otros MWs.

Tipos de MWs

◆ Existen 5 tipos de MWs

- **MW** de nivel de aplicación
- **MW-de-error**
- **Router**: MW de nivel de Router
- MWs incorporados a express
- MWs de terceros
 - ♦ Ver: <http://expressjs.com/en/guide/using-middleware.html>

◆ Router (agrupación de MWs)

- MW para agrupar instalaciones de otros MWs ya asociados a rutas
 - ♦ Un **router** es un MW que puede instalarse, a su vez, en la app express
 - Ver: <http://expressjs.com/en/4x/api.html#router>

◆ Express lleva 3 **MWs** incorporados porque son muy frecuentes

- **express.static**: implementa un servidor Web estático
- **express.json**
 - ♦ Transforma el JSON del body de la solicitud en un objeto equivalente (incluido desde 4.16.0+)
- **express.urlencoded**
 - ♦ Transforma los parámetros del formulario (en body) en propiedades de req.body (incluido desde 4.16.0+)

◆ Existen muchos **MWs de terceros** con funcionalidades habituales

- Deben importarse como paquetes npm en el proyecto
 - ♦ Después se importan en la app express y se instalan como MWs
 - Ver: <http://expressjs.com/en/resources/middleware.html>



JavaScript



Express

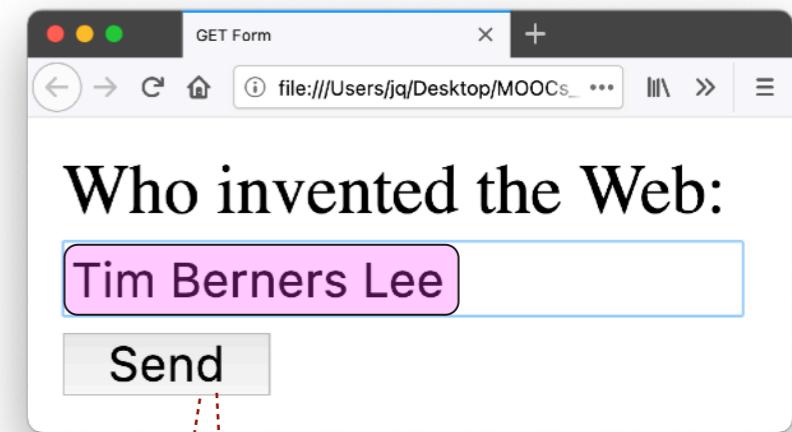
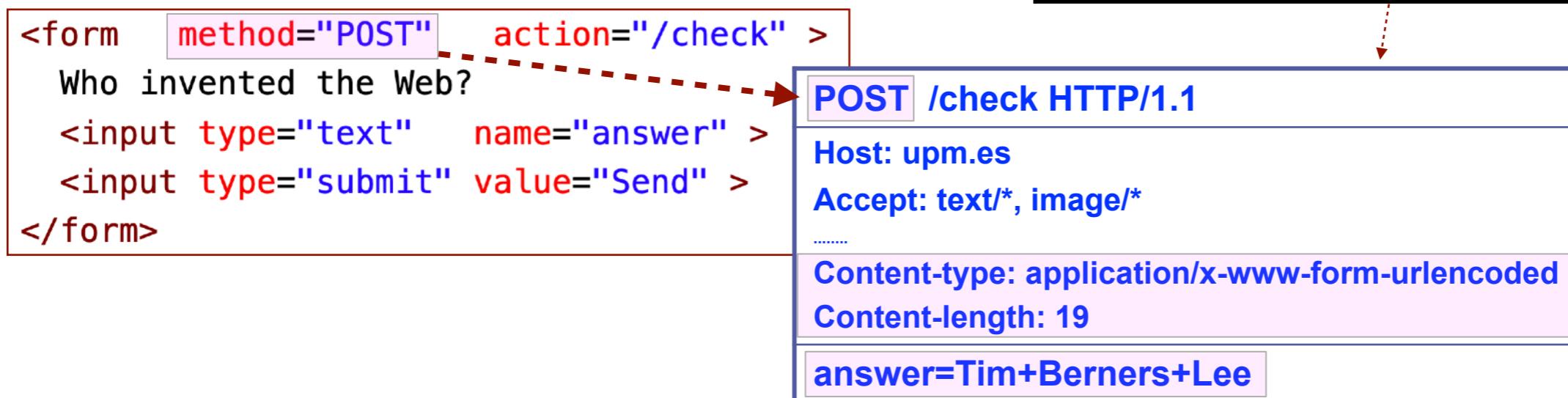
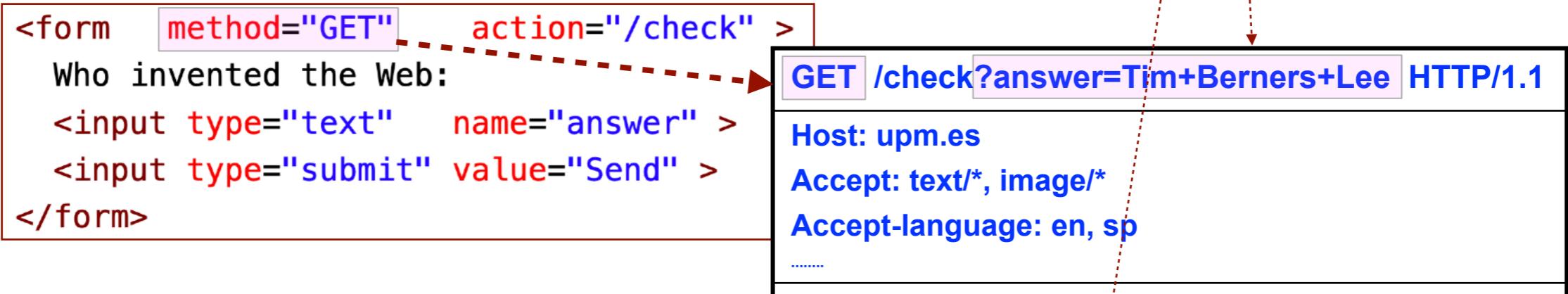
Formularios GET y POST, URL encode
y parámetros ocultos

Juan Quemada, DIT - UPM

Formulario

◆ Formulario

- Elemento HTML de **envío de datos** al servidor
 - ◆ El **formulario** genera un **solicitud HTTP** al clicar el **botón submit**
 - Ver: <https://developer.mozilla.org/es/docs/Web/HTML/Elemento/form>
- Los formularios solo permiten dos tipos de transacciones
 - ◆ **GET**: envía una solicitud HTTP **GET** con los **datos en la query** del URL
 - El tamaño de los datos suele estar limitado a unos pocos KBytes
 - ◆ **POST**: envía una solicitud HTTP **POST** con los **datos en el body**
 - Puede enviar datos de **gran tamaño**, tales como, ficheros, o datos masivos, ...



Elementos de un formulario

◆ <form ..>

- **method:** solo puede ser **GET** o **POST**
 - <https://developer.mozilla.org/es/docs/Web/HTML/Element/form>
- **action:** define la **ruta** de la solicitud HTTP
- **enctype:** codificación (defecto es **urlencoded**)
 - Solo se puede utilizar para cambiarlo con POST
 - Ver: <https://developer.mozilla.org/es/docs/Web/HTTP/Headers/Content-Type>

◆ <label for=".xx..">

- Asocia la etiqueta a <input .. name=".xx.." ..>

◆ <input type=". ." ..>

- Elemento HTML de entrada de datos
 - Tipos: **text**, **submit**, checkbox, date, datetime, email, file, hidden, password,
 - Ver: <https://developer.mozilla.org/es/docs/Web/HTML/Elemento/input>

◆ <input type="text" ..>

- Cajetín de entrada de texto
 - **name:** nombre del parámetro enviado
 - **value:** contenido del cajetín
 - **placeholder:** mensaje si cajetín está vacío

◆ <input type="submit" ..>

- Botón de envío de la solicitud HTTP
 - **value:** texto mostrado en el botón

```
<form method="POST" action="/check" enctype="multipart/form-data">

<label for="answer"> Who invented the Web: </label>
<input type="text" name="answer" value="" placeholder="Type answer" >

<input type="submit" value="Send" >

</form>
```

The screenshot shows a web browser window titled "GET Form". Inside the window, there is a form with the following structure:

```
<form method="POST" action="/check" enctype="multipart/form-data">

<label for="answer"> Who invented the Web: </label>
<input type="text" name="answer" value="" placeholder="Type answer" >

<input type="submit" value="Send" >

</form>
```

Arrows from the text "Who invented the Web:" in the label and the text "Type answer" in the placeholder attribute of the input field point to their respective counterparts in the code. Another arrow points from the "Send" text in the submit button's value attribute to the button itself.

Ejemplo VII: GET

La primera transacción es de tipo **GET** y **carga el formulario**. La segunda transacción es de tipo **GET** y **envía los datos** en el **query** de la ruta en formato **URLencoded**.

<http://localhost:8080/question>



Solicitud del formulario
Envío del formulario



);

Envío de datos
Envío de la respuesta

```
const express = require('express');
let app = express();
```

La transacción **GET** asociada a la ruta **/question** carga el formulario en el navegador.

```
app.get('/question', function(req, res){
```

```
    res.send(`<html>
```

```
    <body>
```

```
        <form method="get" action="/check">
            <label for="answer">Who invented the Web</label>
            <input type="text" name="answer">
            <input type="submit" value="Send">
        </form>
    </body>
</html>`);
```

La transacción **GET**, asociada a **/check**, recibe la respuesta y comprueba si es correcta.

```
app.get('/check', function(req, res) {
```

```
    let answer = req.query.answer;
```

```
    let response = `Correct, ${answer} did`;
```

```
    if (answer !== 'Tim Berners Lee') {
        response = `Incorrect, ${answer} didn't`;
    }
```

```
    res.send(`<html>
```

```
    <body>
```

```
        ${response} invent the Web
```

```
        <br><a href="/question">try again</a>
```

```
    </body>
```

```
</html>`);
```

```
});
```



```
app.listen(8080);
```

Ejemplo VIII: POST

La primera transacción es de tipo **GET** y **carga el formulario**. La segunda transacción es ahora de tipo **POST** y **envía los datos** en el **body** en formato **URLencoded**.

<http://localhost:8080/question>



Who invented the Web:
Tim Berners Lee

localhost:8080/check?answer=Tim-Berners-Lee
Correct, Tim Berners Lee
did invent the Web
[try again](#)

Solicitud del formulario:
GET /question

Envío del formulario

Envío datos en Body:
POST /check

Envío Respuesta



```
const express = require('express');
let app = express();

app.use(express.urlencoded({ extended: true }));

app.get('/question', function(req, res) {
  res.send(`<html>
    <body>
      <form method="post" action="/check">
        <label for="answer">Who invented the Web</label>
        <input type="text" name="answer">
        <input type="submit" value="Send">
      </form>
    </body>
  </html>`);
});

app.post('/check', function(req, res) {
  let answer = req.body.answer;
  let response = `Correct, ${answer} did`;

  if (answer !== 'Tim Berners Lee') {
    response = `Incorrect, ${answer} didn't`;
  }

  res.send(`<html>
    <body>
      ${response} invent the Web
      <br><a href="/question">try again</a>
    </body>
  </html>`);
});

app.listen(8080);
```

Se instala este MW para que los datos del **body** (en formato URLencoded) se añadan a **req.body**.

Hay que indicar a través del atributo **method** que la solicitud debe ser de tipo **POST**.

Este MW debe instalarse con el método **post** para que responda a dicho tipo de solicitudes.

Los parámetros enviados están ahora en **req.body**.

URL Encoding

URL encoding o Percent encoding

◆ URL encoding (percent)

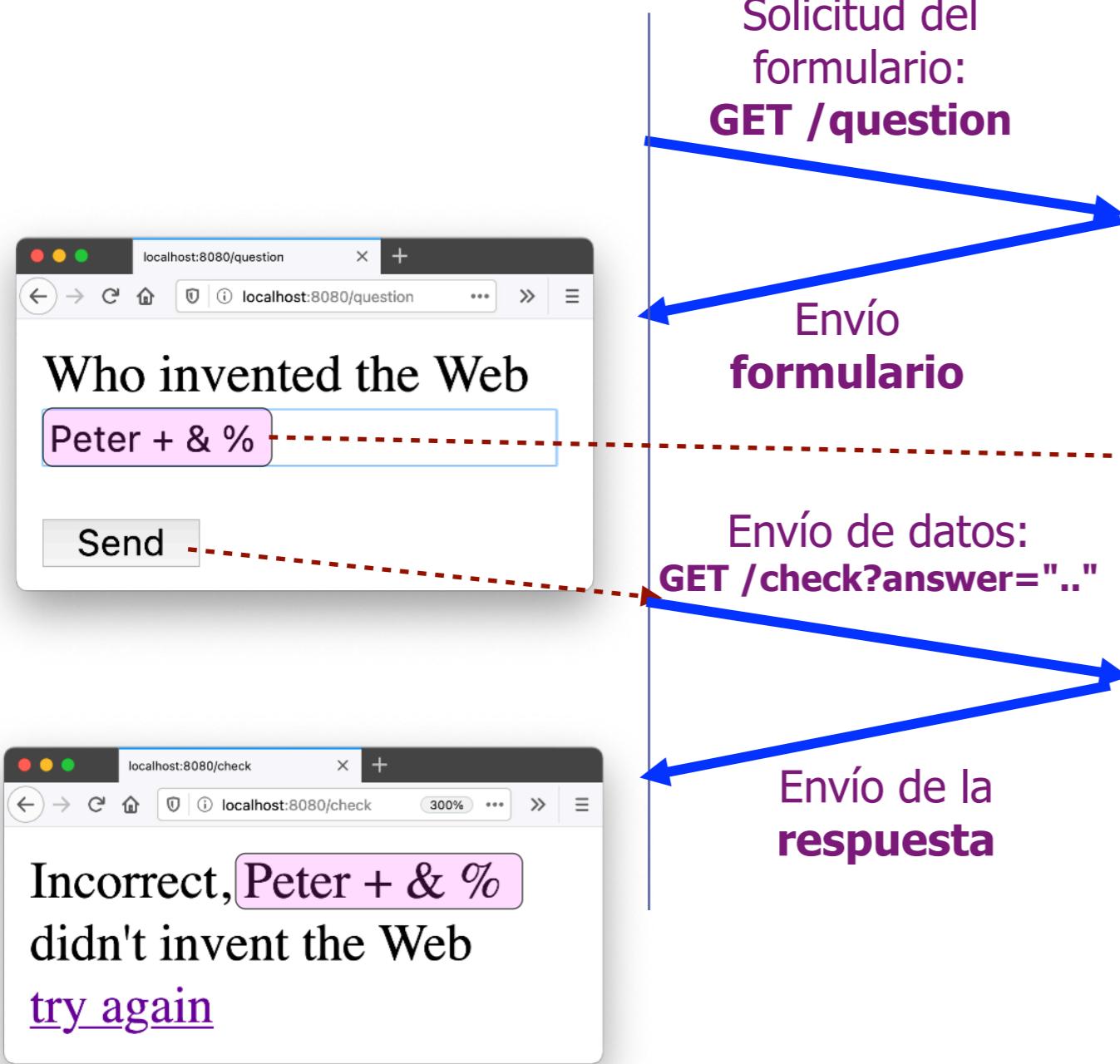
- Codificación para **enviar parámetros** en la query, el body, ..
 - ◆ Corresponde con tipo MIME: “**application/x-www-form-urlencoded**”
 - <https://developer.mozilla.org/en-US/docs/Glossary/percent-encoding>

◆ Reglas de codificación

- Los siguientes caracteres no se codifican: **a-zA-Z0-9-_~!()**
- Resto de caracteres UTF-8
 - ◆ Cada byte se codifica en hexadecimal con tres caracteres ASCII: **%xy**
 - Salvo * ‘ ; : @ & = + \$, / ? % # [] cuando son delimitadores en un URL
 - ◆ El carácter en blanco puede codificarse como **%20** o **+**
- Ejemplo: “<http://wb.es/foto?n=Paco+Garc%C3%ADa>”

!	*	'	()	:	:	@	&	=	+	\$,	/	?	%	#	[]
%21	%2A	%27	%28	%29	%3B	%3A	%40	%26	%3D	%2B	%24	%2C	%2F	%3F	%25	%23	%5B	%5D

Envío de parámetros URL encoded en query



Las Solicitudes HTTP GET llevan los valores de los parámetros del query codificados en URL-encoded.

En el formulario GET anterior el navegador **codifica** los parámetros antes de enviarlos y la aplicación express del servidor **descodifica** los parámetros antes de procesarlos.

`GET /check?answer=Peter%2B%26%25 HTTP/1.1`

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

Parámetro oculto

Parámetro oculto

◆ Parámetro oculto

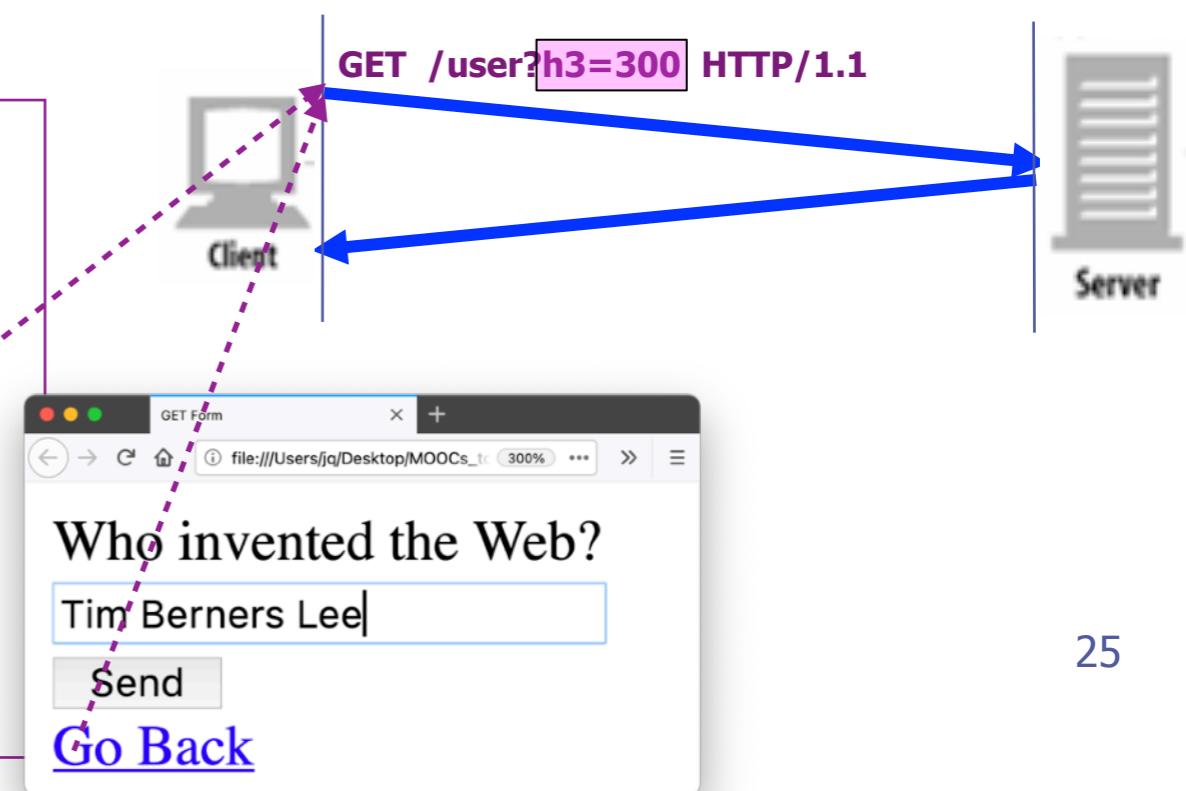
- Parámetro **no visible** de una página HTML
 - ◆ Lleva un dato que volverá del cliente al servidor en la siguiente transacción
- Permite al servidor guardar estado en el cliente
 - ◆ Recordemos que la página HTML codifica el estado del cliente y sus transiciones

◆ Un parámetro oculto se puede guardar en tres lugares

- En el query de un hiperenlace (parámetro **h3=300** del ejemplo)
- En un **<input type="hidden" ...>** (parámetro **h2=200** del ejemplo)
- En la ruta de un formulario (parámetro **h1=100** del ejemplo)

```
<body>
  <form method="POST" action="/check?h1=100" >
    <input type="hidden" name="h2" value="200" >
    Who invented the Web?
    <input type="text" name="answer" >
    <input type="submit" value="Send" >
  </form>

  <a href="http://serv.com/user?h3=300">Go Back</a>
</body>
```



Ejemplo IX: parámetro oculto

Los formularios necesitan **2 transacciones**, primero se **carga el formulario** y luego se **envían los datos** pedidos en el formulario.

http://localhost:8080/question

Who invented the Web?
Type answer
Send

Solicitud del formulario:
GET /question

Envío del formulario

localhost:8080/check?answer=Tim...
Correct, Tim Berners Lee did invent the Web
[try again](#)

Envío de datos

Envío de la respuesta

Solicitud del formulario:
GET /question?my..

Envío del formulario

Who invented the Web?
Tim Berners Lee
Send

```
const express = require('express');
let app = express();
```

```
app.get('/question', function(req, res){
```

```
    let myanswer = (req.query.myanswer || "");
```

```
    res.send(`<html>
```

```
        <body>
```

```
            <form method="get" action="/check">
```

```
                <label for="answer">Who invented the Web</label>
```

```
                <input type="text" name="answer" value="${myanswer}">
```

```
                <input type="submit" value="Send">
```

```
            </form>
```

```
        </body>
```

```
</html>`
```

```
});
```

```
app.get('/check', function(req, res) {
```

```
    let answer = req.query.answer;
```

```
    let response = `Correct, ${answer} did`;
```

```
    if (answer !== 'Tim Berners Lee') {
```

```
        response = `Incorrect, ${answer} didn't`;
```

```
}
```

```
res.send(`<html>
```

```
        <body>
```

```
            ${response} invent the Web
```

```
            <br><a href="/question?myanswer=${answer}">try again</a>
```

```
        </body>
```

```
</html>`
```

```
);
```

```
app.listen(8080);
```

La transacción **GET** asociada a la ruta **/question** carga el formulario en el navegador. Inicializa el cajetín con "" o con el parámetro oculto **myanswer**, si este se ha enviado.

La transacción **GET**, asociada a **/check**, recibe la respuesta y comprueba si es correcta. El parámetro oculto **myanswer** llevará la respuesta anterior al servidor al clicar **try again**.



JavaScript



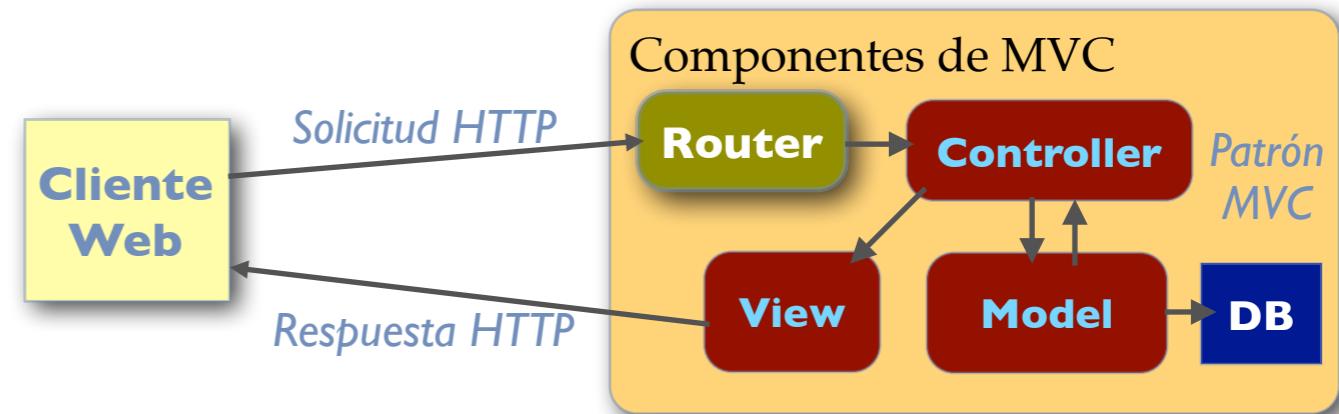
Express

Patrón MVC de servidor

Juan Quemada, DIT - UPM

MVC (Modelo-Vista-Controlador) de Servidor

MVC: patrón propuesto por Trygve Reenskaug en 1979, muy utilizado hoy en Internet.



◆ Router

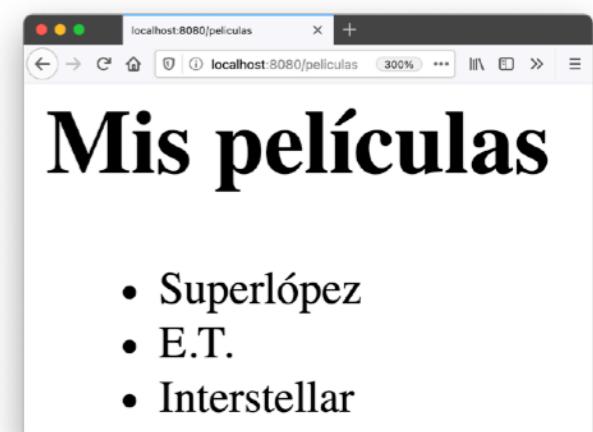
- Decide que **controlador** invocar en función la solicitud HTTP recibida

◆ Modelo

- Conjunto de **datos** (o de recursos) de la aplicación
 - ◆ Pueden estar en variables, ficheros, BBDDs

◆ Vista

- Generador de la **representación visual**
 - ◆ Generador (función) de un documento HTML con los datos solicitados



◆ Controlador

- Orquesta la respuesta a las peticiones del usuario
 - ◆ Normalmente, busca los datos en el modelo, los formatea con la Vista y los envía al cliente

MVC y la estructura de un equipo

- ◆ MVC divide el trabajo de diseño entre distintos especialistas
 - **Modelo:** especialista en bases de datos (técnico)
 - **Vistas:** diseñador (creativo, artista)
 - **Controlador:** programador de la lógica de la aplicación (técnico)
 - **Router:** arquitecto diseñador del interfaz de la aplicación (técnico)
- ◆ Además existen otras especialidades
 - **Diseñador de UX:** usabilidad, interacción, marca, (mixto)
 - **Administrador:** administrador de sistemas en la nube (técnico)
 - **SEO (Search Engine Optimizatióñ):** posiciona portal en buscadores (Google, ...)
 - **Community Manager:** gestiona la relación con los usuarios
 -

Ejemplo X: MVC Index

```
const express = require('express');
const app = express();
```

Modelo no persistente en variable con títulos de películas.

```
// MODEL
var peliculas = ["Superlópez", "E.T.", "Interstellar"];
```

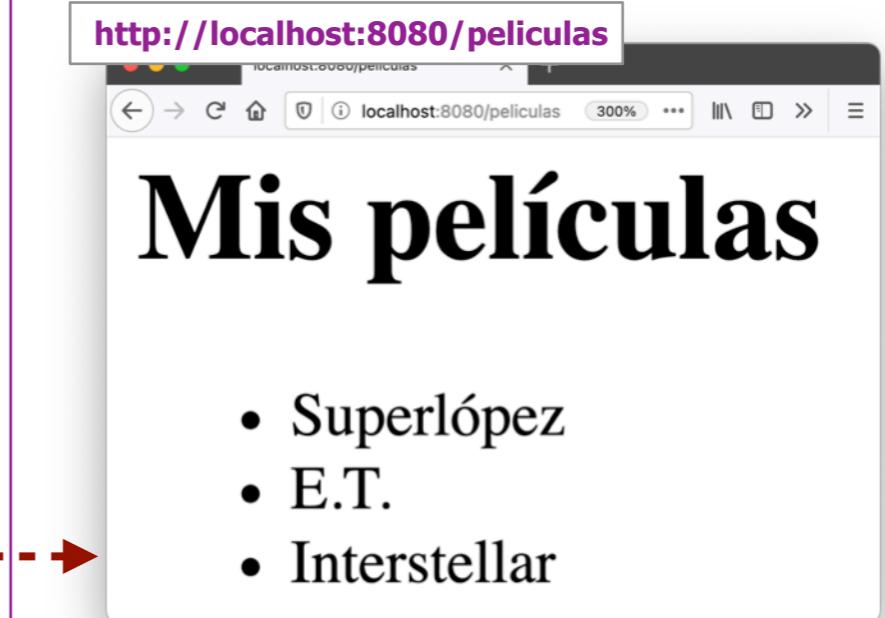
// VIEWS

```
let indexView = (pelis) =>
<!doctype html><html>
<head><meta charset="utf-8"></head>
<body>
  <h1> Mis películas </h1>
  <ul>
    ${pelis
      .map((elem) => `<li> ${elem} </li>`)
      .join('')}
  </ul>
</body>
</html>
```

Vista **Index** que muestra la lista de recursos (películas).

// CONTROLLERS

```
let indexContr = (req, res) => res.send( indexView(peliculas) );
```



MW controlador que genera la página HTML con la lista de recursos y la envía al cliente.

// ROUTER

```
app.get('/peliculas', indexContr);
```

Instalación del MW controlador asociado al método **GET** y a la ruta '**/peliculas**' asociadas a su primitiva en la interfaz REST.

```
app.use('*', (req, res) => req.status(405).send("Not Supported."));
```

MW de respuesta HTTP '**405 Not Supported**' a todos los métodos y rutas nos soportados por esta aplicación.

```
app.listen(8080);
```

Ejemplo X: MVC Index, Show

```
const express = require('express');
const app = express();
```

// MODEL

```
var peliculas = [
  {titulo: "Superlópez", director: "Javier Ruiz Caldera", estreno: "2018"},
  {titulo: "E.T.", director: "Steven Spielberg", estreno: "1982"},
  {titulo: "Interstellar", director: "Christopher Nolan", estreno: "2014"}
];
```

Modelo no persistente en variable con los detalles de las películas.

// VIEWS

```
let layout = (view) => `<!doctype html><html>
  <head><meta charset="utf-8"></head>
  <body><h1> Mis películas </h1>${view}</body></html>`;
```

Función **layout** que añade el marco HTML fijo a las vistas.

```
let indexView = (p) =>
`<ul>${p
  .map((e,i) => `<a href="/peliculas/${i}"><li> ${e.titulo}</li></a>`)
  .join('')
}
</ul>`;
```

Vista **Index** que genera una lista HTML con los títulos de las películas.

```
let showView = (p) => `<b>${p.titulo}</b>: película estrenada en el año <b>
  ${p.estreno}</b> y dirigida por <b>${p.director}</b>.
<p> <a href="/peliculas"><button> Volver </button></a> </p>`;
```

Vista **Show** que muestra los detalles de la película.

// CONTROLLERS

```
let indexContr = (req, res) => res.send(layout(indexView(peliculas)));
let showContr = (req, res) => res.send(layout(showView(peliculas[req.params.id])));
```



Mis películas

- [Superlópez](#)
- [E.T.](#)
- [Interstellar](#)



Mis películas

Interstellar: película estrenada en el año **2014**, dirigida por **Christopher Nolan**.

// ROUTER

```
app.get('/peliculas', indexContr);
app.get('/peliculas/:id', showContr);
```

Instalación de los dos MWs controladores asociados a los métodos y rutas de cada primitiva de la interfaz REST.

```
app.use('*', (req, res) => req.status(405).send("Not Supported."));
app.listen(8080);
```

MW de respuesta HTTP '**405 Not Supported**' a todos los métodos y rutas no soportados por esta aplicación.



JavaScript



Express

MVC, Interfaz Uniforme, method-override, POST, PUT y DELETE

Juan Quemada, DIT - UPM

Interfaz Uniforme y Method Override

- ◆ HTML solo puede enviar solicitudes GET y POST
 - Aplicaciones REST deben enviar también PUT y DELETE
 - ◆ **Method override** define un convenio para enviar PUT o DELETE
- ◆ Method-Override
 - Simula enviar **PUT** y **DELETE** con un **parámetro oculto** en GET o POST
 - ◆ `_method=PUT` o `_method=DELETE`
 - El paquete npm **method-override** de express gestiona este convenio
 - ◆ Instalación: `app.use(method-override('_method', {methods: ["POST", "GET"]}))`
 - Doc: <https://www.npmjs.com/package/method-override>
- ◆ PUT y DELETE pueden encapsularse en GET
 - No permiten datos voluminosos
 - ◆ Deben ser operaciones idempotentes, para no interferir con las **caches**

Express middleware

Juan

expressjs.com/en/resources/middleware.html

Express

Home Getting started Guide API reference Advanced topics Resources

body-parser
compression
connect-rid
cookie-parser
cookie-session
cors
csurf
errorhandler
method-override
morgan
multer
response-time
serve-favicon
serve-index
serve-static
session
timeout
vhost

Express middleware

The Express middleware modules listed here are maintained by the [Expressjs team](#).

Middleware module	Description	Replaces built-in function (Express 3)
body-parser	Parse HTTP request body. See also: body , co-body , and raw-body .	express.bodyParser
compression	Compress HTTP responses.	express.compress
connect-rid	Generate unique request ID.	NA
cookie-parser	Parse cookie header and populate <code>req.cookies</code> . See also cookies and keygrip .	express.cookieParser
cookie-session	Establish cookie-based sessions.	express.cookieSession
cors	Enable cross-origin resource sharing (CORS) with various options.	NA
csurf	Protect from CSRF exploits.	express.csrf
errorhandler	Development error-handling/debugging.	express.errorHandler
method-override	Override HTTP methods using header.	express.methodOverride
morgan	HTTP request logger.	express.logger
multer	Handle multi-part form data.	express.bodyParser
response-time	Record HTTP response time.	express.responseTime
serve-favicon	Serve a favicon.	express.favicon
serve-index	Serve directory listing for a given path.	express.directory
serve-static	Serve static files.	express.static
session	Establish server-based sessions (development only).	express.session
timeout	Set a timeout period for HTTP request processing.	express.timeout
vhost	Create virtual domains.	express.vhost

Additional middleware modules

These are some additional popular middleware modules.

MWs express de terceros

Los middlewares de la librería de express se pueden mezclar con los que crea un usuario respetando los convenios y requisitos de cada uno.

<http://expressjs.com/en/resources/middleware.html>

34

Ejemplo XI: Index, New, Create y Delete

Primitiva	Solicitud HTTP	MV controlador	Respuesta: Vista o redirección
Index	GET /movies	indexContr(..)	indexView()
Search	GET /movies?search=..	indexContr(..)	indexView()
New	GET /movies/new	newContr(..)	newView()
Create	POST /movies	createContr(..)	redirección a /movies
Delete	DELETE /movies/:id	deleteContr(..)	redirección a /movies

Delete se implementa con un hiperenlace con **method-override**.

◆ Index

- Muestra la lista de recursos (películas)

◆ Search

- Muestra las películas que casan con el patrón de búsqueda

◆ New (complementaria de Create)

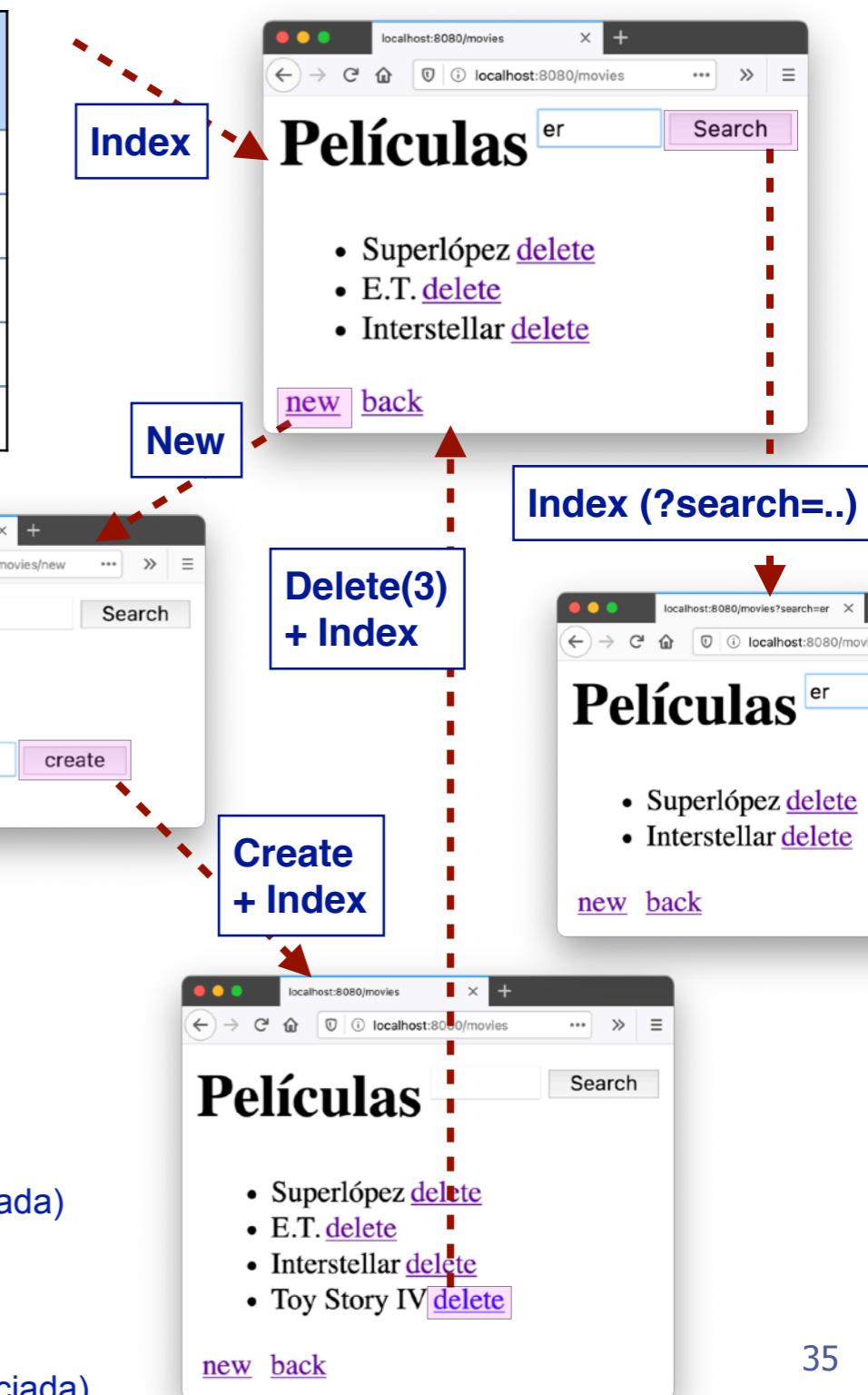
- Muestra el formulario para teclear el nuevo recurso (película)

◆ Create

- Añade la nueva película al modelo (al final del array)
 - Una vez añadido, redirige a la vista Index con **res.render(<url>)** (no tiene vista asociada)

◆ Delete

- Elimina un recurso (película) del modelo
 - Una vez eliminado, redirige a la vista Index con **res.render(<url>)** (no tiene vista asociada)



```
const express = require('express');
const methodOverride = require('method-override');
const app = express();
```

Instalar MWs para **URLencoded**, **method-override** y **static** Web server.

```
app.use(express.urlencoded({extended: true})); // parse POST params in BODY
app.use(methodOverride('_method', {methods: ["POST", "GET"]}));
app.use(express.static('public'));
```

// MODEL

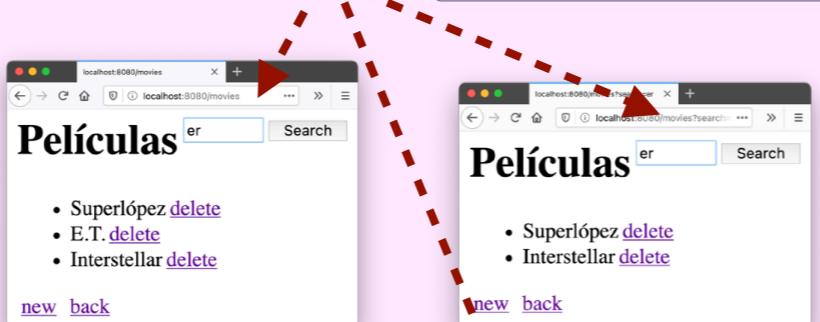
```
let movies = ["Superlópez", "E.T.", "Interstellar"];
```

// VIEWS

```
const style = `<style>.search{float:right;} a{padding:0 3px 0 3px }</style>

function addLayout (view) {
  return `<!doctype html><html><head><meta charset="utf-8">${style}</head>
<body>
  <form method="GET" action="/movies" class="search">
    <input type="text" name="search" size="5">
    <input type="submit" class="button" value="Search">
  </form>
  <h1>Películas</h1>
  ${view}
</body></html>`;
}

function indexView (movies) {
  let view = `<ul>
    ${movies
      .map((e,i)=>`<li>${e}<a href="/movies/${i}?_method=DELETE">delete</a></li>`)
      .join("\n")}`
    </ul>
  <a href="/movies/new">new</a> <a href="/movies">back</a>`;
  return addLayout(view);
}
```



Formulario con el cajetín de búsqueda, que genera la solicitud:
GET /movies?search=..

```
function newView () {
  let view = `<form method="POST" action="/movies">
    <label for="title">Nueva película</label>
    <input type="text" name="title">
    <input type="submit" class="button" value="create">
  </form>
  <a href="/movies">back</a>`;
  return addLayout(view);
}
```



Ejemplo XI: MVC

// CONTROLLERS

```
const indexContr = async (req, res, next) => {
  let s = req.query.search;
  let p = (s) ? movies.filter((e)=>e.includes(s)) : movies;
  res.send(indexView(p));
};
```

```
const newContr = async (req, res, next) => {
  res.send(newView());
};
```

```
const createContr = async (req, res, next) => {
  let title = req.body.title;
  movies.push(title);
  res.redirect('/movies');
};
```

```
const destroyContr = async (req, res, next) => {
  let id = req.param.id;
  movies.splice(id, 1);
  res.redirect('/movies');
};
```

res.redirect('/movies')
redirecciona a la vista
'/movies' con la
respuesta 302 de HTTP.

// ROUTER

```
app.get(['/', '/movies'], indexContr);
app.get(['/movies/new'], newContr);
app.post('/movies', createContr);
app.delete('/movies/:id', destroyContr);
```

```
app.all('*', (req, res) =>
  res.status(404).send("Error: not found or supported."));
};
```

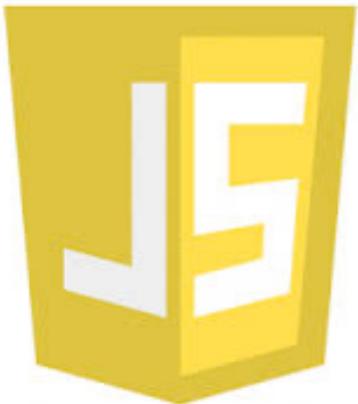
```
app.use((error, req, res, next) => {
  console.log("Error:", error.message || error);
  res.redirect("/");
});
```

```
app.listen(8080);
```

Interfaz CRUD Web para gestionar una lista de recursos

- ◆ **Index** y **Show** muestran la lista de recursos (Index) o los detalles de un recurso (Show)
 - Primitivas **Read**: muestran al usuario datos del modelo (El Ejemplo III solo tiene Index)
- ◆ **New** y **Create** añaden un nuevo recurso al modelo
 - Primitivas **Create**: primero se muestra el formulario (New) y luego se actualiza el modelo (Create)
- ◆ **Edit** y **Update** modifican el contenido de un recurso en el modelo
 - Primitivas **Update**: mostrar el formulario (Edit) y luego actualizar el modelo (Update)
- ◆ **Delete** elimina un recurso del modelo
 - Primitiva **Delete**: suele pedir confirmación antes de eliminar el recurso

Primitiva	Solicitud HTTP	MV controlador	Respuesta: Vista o redirección
Index	GET /quizzes	indexContr(..)	indexView()
Show	GET /quizzes/:id	showContr(..)	showView()
New	GET /quizzes/new	newContr(..)	newView()
Create	POST /quizzes	createContr(..)	redirección a /quizzes o /quizzes/:id
Edit	GET /quizzes/:id/edit	editContr(..)	editView()
Update	PUT /quizzes/:id	updateContr(..)	redirección a /quizzes o /quizzes/:id
Delete	DELETE /quizzes/:id	deleteContr(..)	redirección a /quizzes



JavaScript



AJAX - Asynchronous JavaScript & XML

Juan Quemada, DIT - UPM

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

◆ Aplicación de cliente que solo consulta el servidor si es necesario

- Una aplicación AJAX es más rápida y ágil que la carga de páginas Web
 - ◆ Solo pide al servidor las vistas necesarias, y en las actualizaciones solo pide los valores que cambian
- También se denominan RIA (Rich Internet Applications) o Single Page Application

◆ Los navegadores soportan AJAX con el objeto **XMLHttpRequest**

- XMLHttpRequest permite realizar transacciones HTTP con el servidor
 - ◆ Documentación: <https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX>

◆ ES6 añade el método global Fetch para realizar transacciones HTTP

- Fetch esta adaptado al uso de promesas
 - ◆ Documentación: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

◆ La librería jQuery da un interfaz uniforme para AJAX

- Simplifica la realización de aplicaciones AJAX
 - ◆ Independiza de los detalles de los navegadores
- Métodos soportados: `jQuery.ajax(...)` o `$.ajax(...)`, `jQuery.get(...)`, `jQuery.post(...)`,
- Documentación: <http://api.jquery.com/category/ajax/>

Ajax jQuery: Ej. Person I

Acciones del controlador que responden a las transacciones HTTP con las rutas asociadas.

Creación de la aplicación express.

```
const express = require('express');
const app = express();
```

// CONTROLADORES

```
const indexController = (req, res, next) => { ..... };
const resultController = (req, res, next) => { ..... };
```

// RUTAS

```
app.get(['/', '/index'], indexController);
app.get('/result', resultController);
app.get('*', (req, res) => res.send("Error: resource not found"));
```

// MODELO

```
const model = [ {name:'Peter', age:22},
               {name:'Anna', age:23},
               {name:'John', age:30}
             ];
```

// ESTILOS

```
const style = `<style> ..... </style>`
```

// VISTAS

```
const vista_index = `<html> .....</html>`
```

// Arranque del servidor en puerto 8000

```
app.listen(8000);
```

Arrancar el servidor en el puerto 8000.

Transacciones con estas 2 rutas ('/', '/index') devuelven la vista única que contiene la aplicación AJAX.

Transacciones con la ruta '/result' no carga ninguna vista, solo consulta si el nombre introducido en el formulario es el correcto utilizando una transacción GET de tipo AJAX. La aplicación AJAX inserta el resultado en la vista ya cargada.

Router de solicitudes HTTP: asocia las rutas de la interfaz REST con las acciones del controlador.

Modelo con los datos de la aplicación.

Estilos utilizados por las vistas.

Vistas única de la aplicación.

<http://localhost:8000>
<http://localhost:8000/index>

MVC Example

Access the query form

CORE 2018

<http://localhost:8000/form>

Request form

What is the age of

Anna Send

CORE 2018

<http://localhost:8000/result?name=Anna>

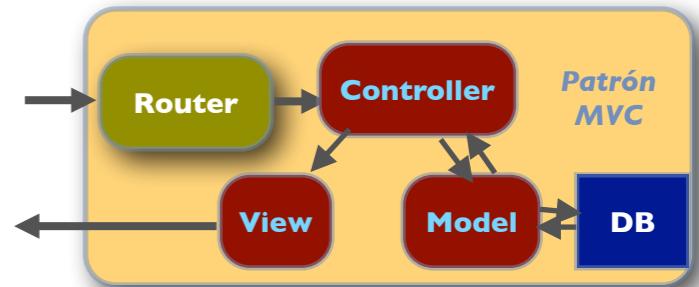
Response: result

Anna is 23 years old

[Go back to the age query form](#)

CORE 2018

Ajax jQuery: Ej. Person I



Esta acción del controlador envía al cliente la vista única que contiene la aplicación AJAX.

```
const express = require('express');
const app = express();
```

// CONTROLADORES

```
const indexController = (req, res, next) => {
  res.send(vista_index);
};
```

```
const resultController = (req, res, next) => {
  let name = req.query.name, response;

  let person = model.find(p => p.name === name);
  if (person) {
    response = name + " is " + person.age + " years old";
  } else {
    response = name + " is not in our DB";
  }

  res.send(response);
};
```

```
// RUTAS
app.get(['', '/index'], indexController); // router
app.get('/result', resultController);
app.get('*', (req, res) => res.send("Error: resource not found") );
```

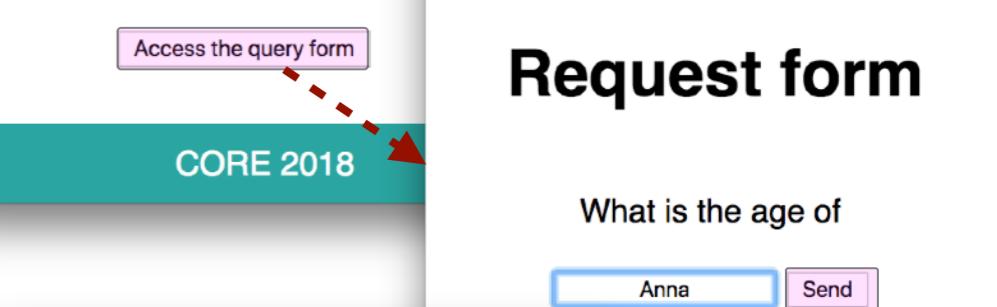
// MODELO

```
const model = [
  {name:'Peter', age:22},
  {name:'Anna', age:23},
  {name:'John', age:30}
];
```

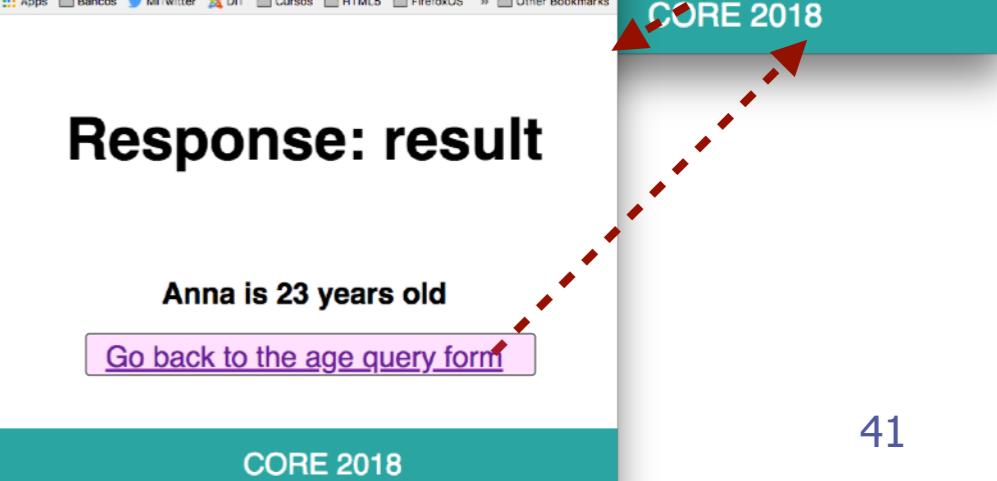
Este acción del controlador comprueba si el nombre introducido en el formulario es el correcto y devuelve el resultado. Para ello consulta el modelo.



Request form



Response: result



Ajax jQuery: Ej. Person III

```
// VISTAS
const vista_index = ` <!-- HTML index view -->
<html>
<head>
<title>MVC Example</title><meta charset="utf-8">${style}
<script type="text/javascript" src="https://code.jquery.com/jquery-3.3.1.min.js" > </script>
<script type="text/javascript">
$(function(){
    $('#msg').on('click', function(){
        $('#msg').hide();
        $('#form').show();
    });
    $('#submit').on('click', function(){
        $.ajax( { type:'GET',
                  url: '/result?name=' + $("#name").val(),
                  success: function(response){
                      $('#msg').html('<strong>' + response + '</strong><br><br><button>Back to form</button>');
                  }
        })
        $('#msg').show();
        $('#form').hide();
    });
    $('#msg').show();
    $('#form').hide();
});
</script>
</head>
<body>
<h1>MVC Example</h1>
<div id='msg'><button>Get query form</button></div>
<div id='form'>
    What is the age of <br><br>
    <input type="text" id="name" placeholder="Name" />
    <button id="submit">submit</button>
</div>
<footer>CORE 2018</footer>
</body>
</html>`
```

Evento hacer **click en el mensaje** (#msg): muestra formulario y oculta mensaje.

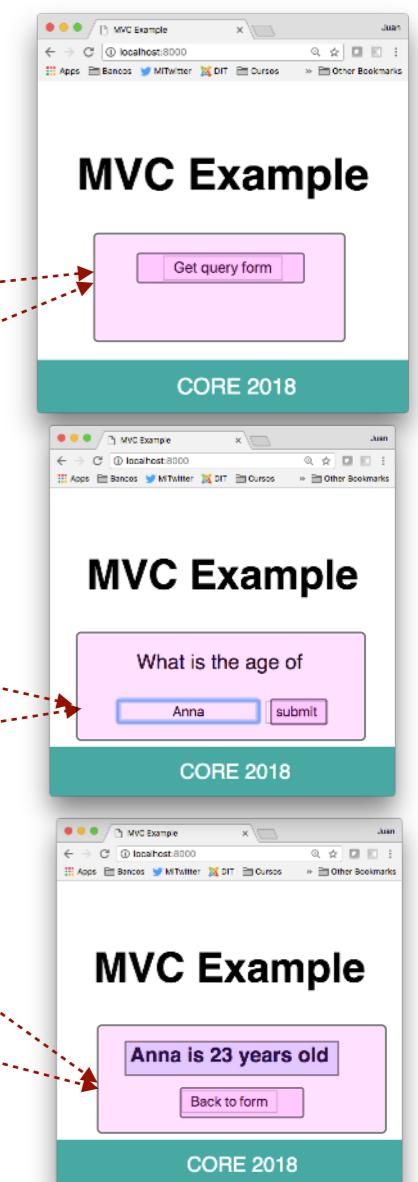
Evento hacer **click en el formulario** (#form): muestra mensaje y oculta formulario.

Transacción AJAX HTTP GET con método ajax(..) de jQuery para consultar la edad al servidor.

configuración inicial: muestra mensaje y oculta formulario.

Bloque <div> donde se muestra el mensaje con la edad de la persona consultada.

Bloque <div> con el cajetín de consulta de la edad de una persona.



Ajax: Ej. Person IV

```

const vista_index = ` <!-- HTML index view -->
<html>
<head>
  <title>MVC Example</title> <meta charset="utf-8">
  <script type="text/javascript" src="https://code.jquery.com/jquery-3.3.1.min.js" > </script>
<script type="text/javascript">
$(function(){
  $('#msg').on('click', function(){
    $('#msg').hide();
    $('#form').show();
  });

  $('#submit').on('click', function(){
    var url = '/result?name=' + $("#name").val()
    if(self.fetch) {
      console.log('Fetch');
      fetch(url)
      .then(function(response) {
        return response.text().then(function(text){
          $('#msg').html(text + '<button>Back to form</button>');
        });
      });
    } else {
      console.log('XHR')
      var xhr = new XMLHttpRequest();
      var request = new XMLHttpRequest();
      xhr.open('GET', url, true);
      xhr.onload = function() {
        $('#msg').html(xhr.response + '<button>Back to form</button>');
      }.bind(this);
      xhr.onerror = function(e) {
        console.log(e);
      }
      xhr.send();
    }
    $('#msg').show();
    $('#form').hide();
  });
  $('#msg').show();
  $('#form').hide();
});
</script>
</head>
```

```

Evento hacer **click en el mensaje** (#msg): muestra formulario y oculta mensaje.

Evento hacer **click en Submit** (#form): muestra mensaje y oculta formulario.

Transacción AJAX HTTP GET con método **fetch(..)** y profesas para consultar la edad al servidor.

Transacción AJAX HTTP GET con **XMLHttpRequest** que consulta la edad al servidor.

config. inicial:  
muestra mensaje y oculta formulario.

```

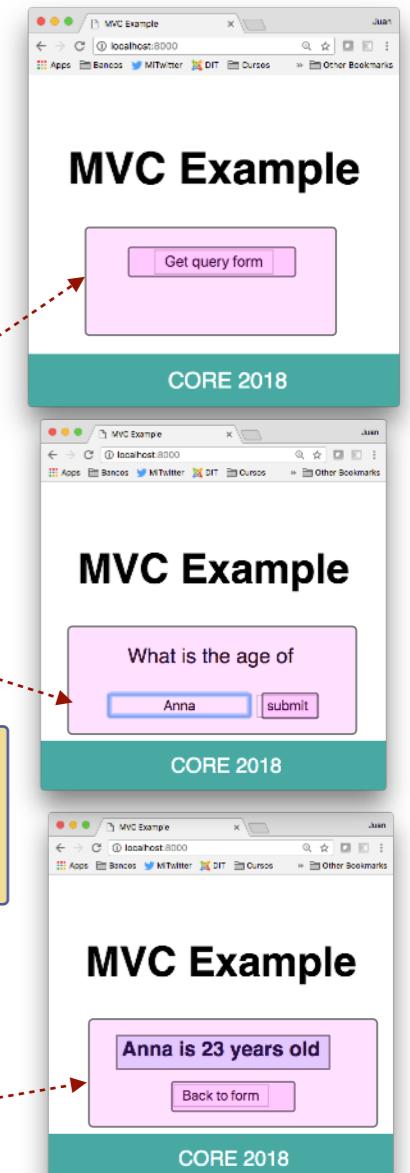
<body>
 <h1>MVC Example:</h1>
 <div id='msg'><button>Get query form</button></div>
 <div id='form'>
 What is the age of:

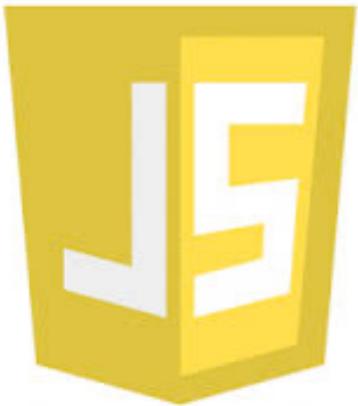
 <input type="text" id="name" placeholder="Name" />
 <button id="submit">submit</button>
 </div>
</body>
</html>
```

```

Bloque <div> donde se muestra el mensaje con la edad de la persona consultada.

Bloque <div> con el cajetín de consulta de la edad de una persona.





JavaScript



Final del tema