



JavaScript



express.js, Middleware, Formularios, MVC y AJAX

Juan Quemada, DIT - UPM

Índice: express.js, Middleware, Formularios, MVC y AJAX

1. <u>express.js: aplicación, middleware y static</u>	3
2. <u>Aplicaciones express de servidor: MWs, rutas, req, res, next e interfaces REST</u> ...	8
3. <u>Ejemplos de MWs y de uso de req, res, next</u>	19
4. <u>Composición y ejecución de middlewares: next() y next(Error)</u>	28
5. <u>Formularios GET y POST: Parámetros (en query, en body u ocultos), URL encode y method override</u>	35
6. <u>Patrón MVC Patrón MVC (Model-Vista-Controller)</u>	50
7. <u>AJAX - Asynchronous JavaScript & XML</u>	56



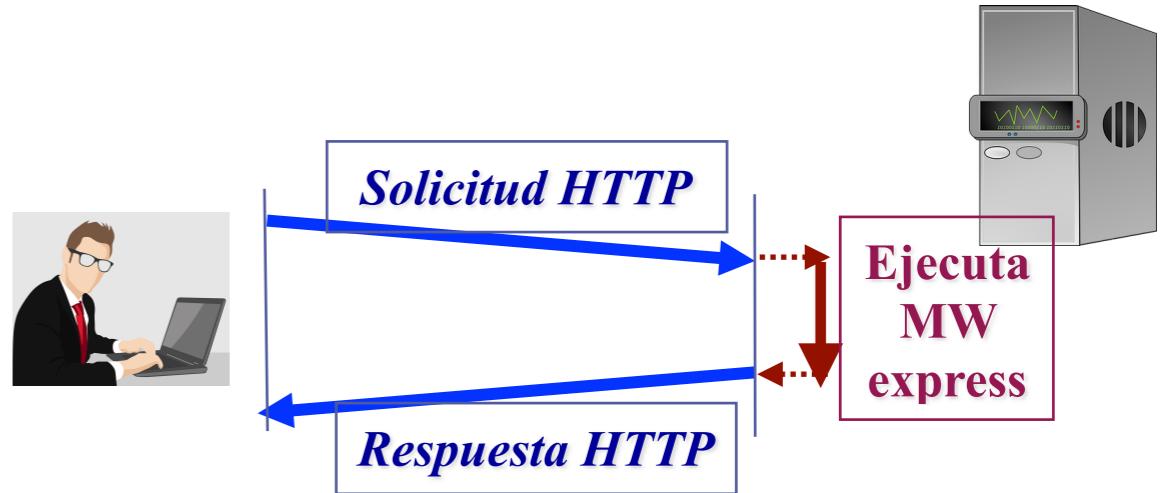
JavaScript



express.js: aplicación, middleware y static

Juan Quemada, DIT - UPM

Paquete, aplicación y middleware express



◆ Paquete express

- Paquete npm muy popular para crear aplicaciones Web con node.js
 - ◆ Mas información: <http://expressjs.com>, <https://www.npmjs.com/package/express>
 - ◆ Documentación: <http://expressjs.com/en/api.html>

◆ npm install express@4.16.2

- Instala la **versión 4.16.2** de express (utilizada para correr estos ejemplos)
 - ◆ Lo trae del registro central (<https://www.npmjs.org>) y lo instala en el directorio **node_modules**

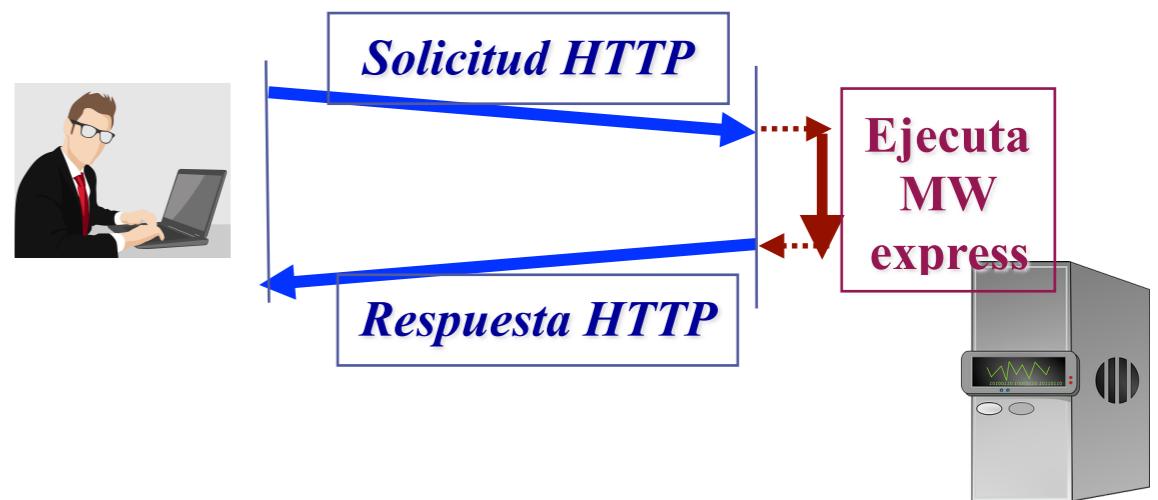
◆ aplicación express

- Las aplicaciones express atienden transacciones HTTP (de clientes)
 - ◆ Las aplicaciones express se construyen con middleware (MW)

◆ Middleware (MW) express

- Función que se instala en una aplicación express para atender una transacción HTTP
 - ◆ <http://expressjs.com/en/guide/using-middleware.html>, <http://expressjs.com/en/guide/writing-middleware.html>

Crear aplicación express con MW static



◆ Crear una aplicación express

- Importar el módulo express:
 - ◆ El modulo solo se puede importar si el paquete npm se ha instalado con: `npm install`
 - Generar la aplicación express:
 - Instalar MW con el método use:
 - Arrancar el servidor en un puerto:
- `const express = require('express')`
- `const app = express();`
- `app.use(MW);`
- `app.listen(<puerto>);`

◆ Middleware static

- MW incluido en el paquete express para crear **servidores Web estáticos**
 - ◆ <http://expressjs.com/en/api.html#express.static>

◆ El MW static se crea invocando

- `express.static(<ruta_al_directorio_de_recursos_Web>);`

Servidor Web estático

Este programa crea un **servidor Web estático** con el middleware express **express.static(...)**.

var express = require('express') importa el módulo express y lo guarda en la **variable** express. **Nota:** los módulos son normalmente objetos (cierres con propiedades y métodos) guardados en variables globales.

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

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

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

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

app.listen(8000) arranca el servidor conectado al puerto 8000 para que responda a solicitudes HTTP en ese puerto.

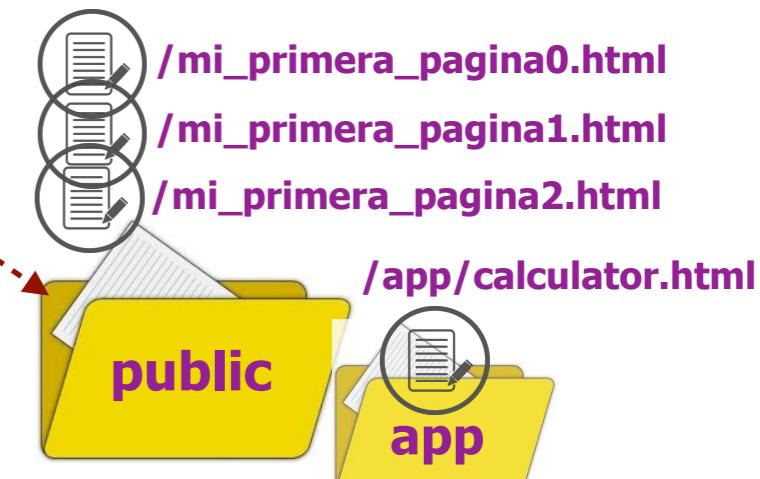
var path = require('path') importa el módulo path de node.js y lo guarda en la variable **path**.

var app = express() crea la aplicación express con el constructor de objetos express() importado y la guarda en la variable **app**.

path.join(__dirname, 'public'): función del módulo path que concatena 2 rutas, devolviendo la ruta absoluta al directorio public donde deben alojarse las páginas Web.

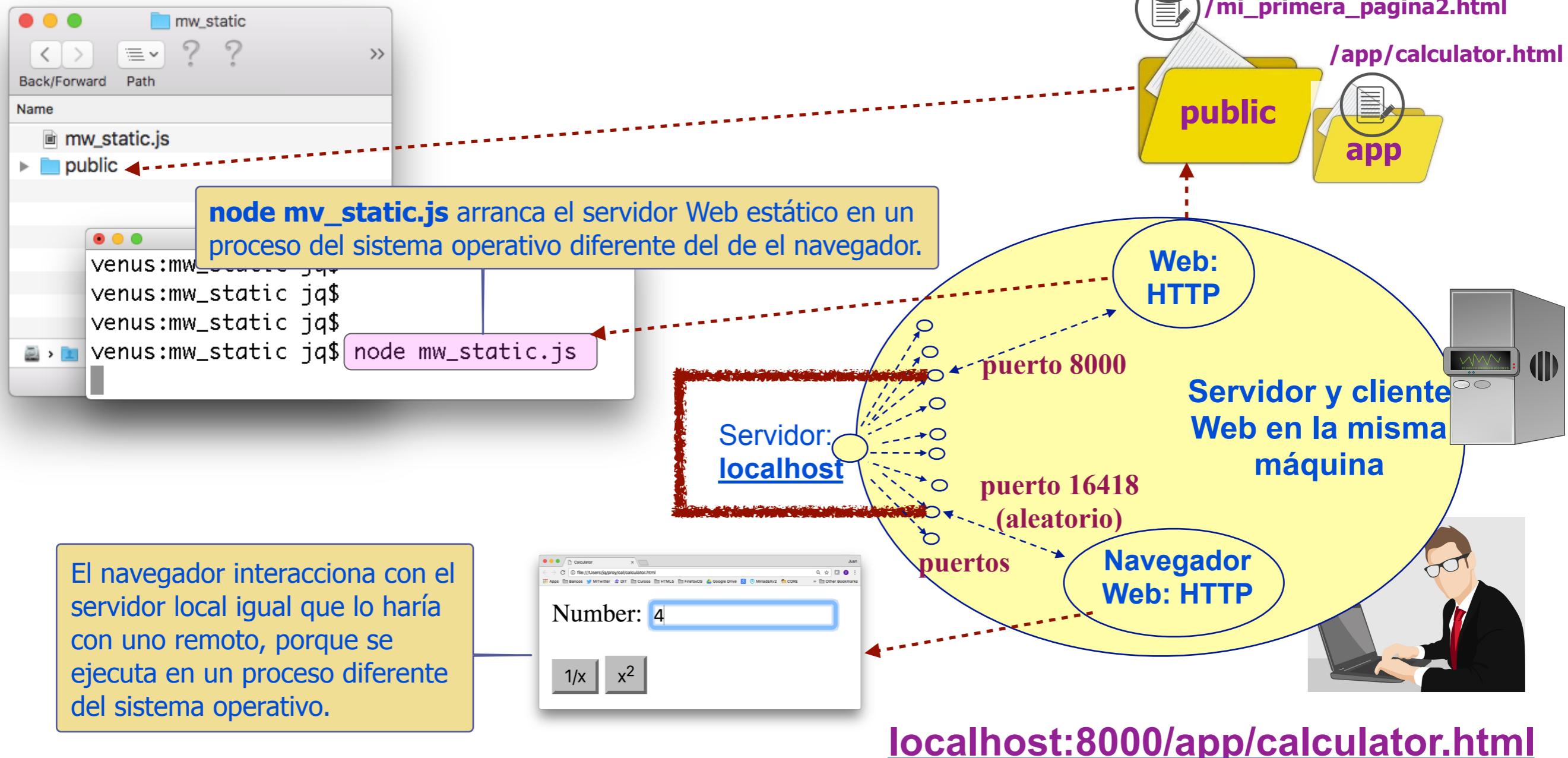
__dirname: variable predefinida de node.js con la ruta absoluta al directorio del programa. Ver: <https://nodejs.org/api/globals.html>

app.use(express.static(path.join(__dirname, 'public'))) instala el middleware static (servidor Web estático) en la aplicación express **app**.



Indica al servidor estatico que el **repositorio de recursos** esta en el directorio public. Este contiene 3 páginas y 1 subdirectorrio app con otra página. Se acceden con las rutas indicadas.

Cliente y servidor en una máquina



◆ El cliente y servidor pueden estar en la misma máquina

- Es algo habitual cuando se hacen pruebas durante el desarrollo
 - ◆ Si el servidor está en el puerto 8000, el recurso `/app/calculadora.html` se accede con
 - localhost:8000/app/calculator.html



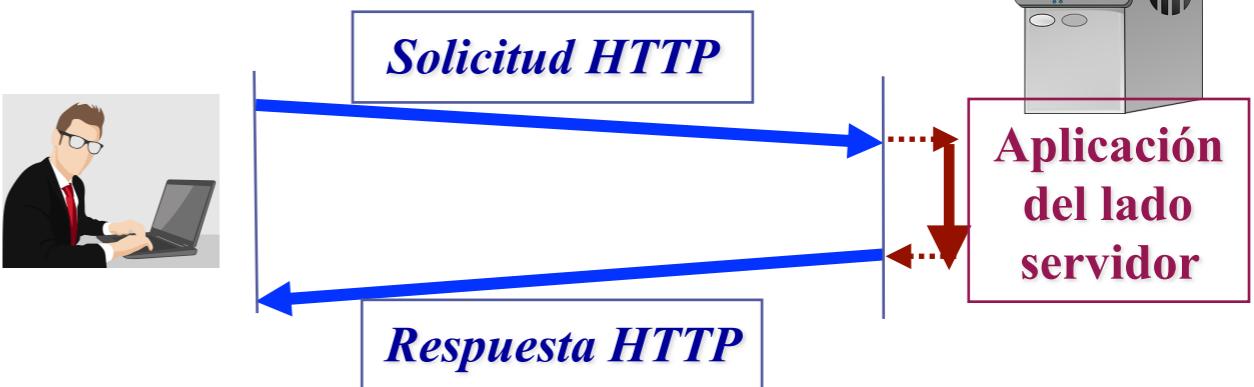
JavaScript



Aplicaciones express de servidor: MWs, rutas, req, res, next e interfaces REST

Juan Quemada, DIT - UPM

Aplicación de servidor



◆ Aplicación de servidor o del lado servidor (server side application)

- Aplicación de servidor que programa respuestas a medida a solicitudes HTTP
 - El protocolo de acceso a servidores más habitual es HTTP, pero hay otros como SOAP (en desuso), ...

◆ REST (Representational State Transfer)

- Estilo arquitectural para aplicaciones hipermedia desarrollado por Roy Fielding
 - El cliente y el servidor intercambian estados, tal y como hacen los navegadores Web con las páginas Web
 - Documentación: https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

◆ Interfaz o API REST

- Interfaz de acceso HTTP a funciones de un servidor basada en rutas
 - Cada **ruta** identifica el **código** (MW en express) que hay que responder a dicha ruta, por ejemplo
 - **/lista/de/clientes** devuelve la lista de clientes del servidor
 - **/cliente/221** devuelve el cliente 221 del servidor
 - **/clientes/VIP** devuelve la lista de clientes VIP del servidor

MWs asociados a ruta y método HTTP

◆ `app.use('ruta', MW)`

- Permite instalar también un **MW** asociado a una **ruta**
 - ◆ El **MW** se invocará para cualquier tipo de solicitud HTTP recibida, pero solo si la **ruta recibida** es la **instalada**
- '**ruta**' es opcional, si se omite el MW se invocará para cualquier ruta recibida

◆ `app.get('ruta', MW)`

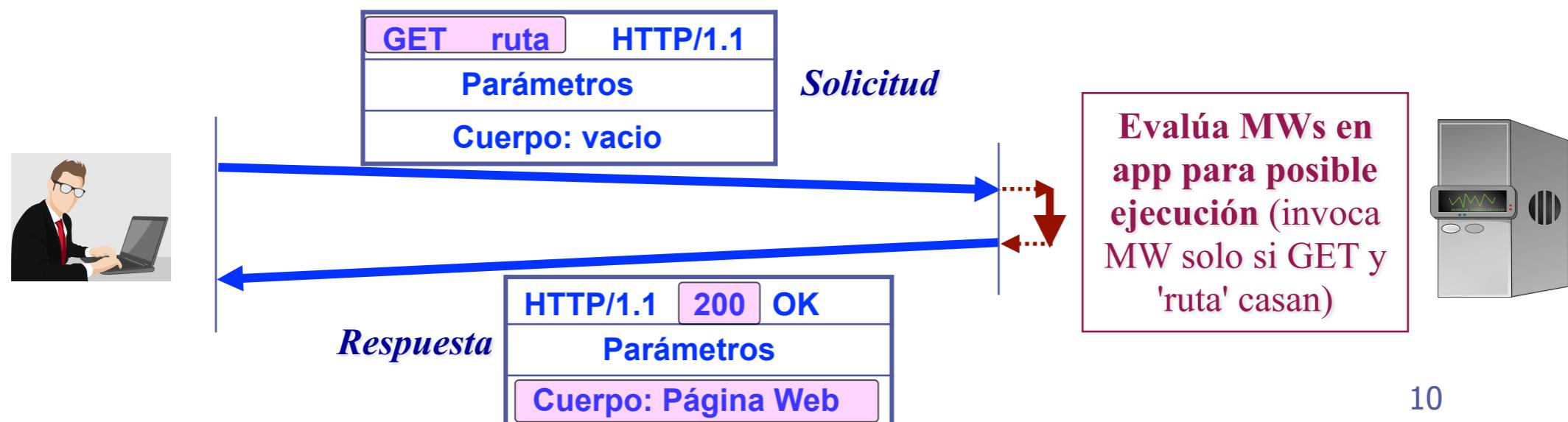
- Instala el middleware **MW** en la aplicación **app** asociado a la **ruta** indicada y a HTTP **GET**
 - ◆ MW se invoca solo si la solicitud recibida es HTTP **GET** y la **ruta recibida** coincide con la **instalada**

◆ Los **MWs** permiten **modularizar** un programa express.js

- Cada MW podemos verlo como un módulo que realiza una función diferente

◆ Los **MW** se evalúan para su ejecución siguiendo el orden de instalación

- Se invocará el primero que case con la ruta y método HTTP recibidos



Parámetros req, res y next de un MW

◆ **MW (req, res, next) { }**

- La aplicación express invoca siempre un MW con los parámetros **req, res** y **next**

◆ **req**

- Objeto JavaScript con los parámetros de la **solicitud HTTP** recibida
 - ♦ Documentación: <http://expressjs.com/en/4x/api.html#req>

◆ **res**

- Objeto JavaScript que permite configurar y enviar la **respuesta HTTP** al cliente
 - ♦ Documentación: <http://expressjs.com/en/4x/api.html#res>

◆ **next**

- función que pasa control a los MWs instalados después al invocarse: **next()**
 - ♦ Los siguientes MWs pueden ejecutarse y seguir procesando la transacción HTTP en curso
 - Documentación: <http://expressjs.com/en/guide/writing-middleware.html>

Enviar la respuesta al cliente con send(..)

◆ **res.send(<body>)**

- método de **res** que envía la respuesta HTTP al cliente
 - ◆ El parámetro **<body>** se incluye como el cuerpo de la respuesta HTTP
 - Documentación: <http://expressjs.com/en/4x/api.html#res.send>

◆ Una página **HTML** es un **string** con **marcas HTML**

- Se puede enviar en la respuesta al cliente con send, por ejemplo
 - ◆ `res.send('<html><head></head> <body><h1> Título </h1></body></html>')`

◆ **send(..)** finaliza la ejecución de la secuencia de MWs

- Node incluye otros métodos para enviar respuesta y finalizar
 - ◆ <http://expressjs.com/en/guide/routing.html>

◆ La atención a cada transacción HTTP debe terminar siempre enviando una respuesta

- Si no se envía, el cliente y la aplicación pueden quedar colgados
 - ◆ <http://expressjs.com/en/guide/writing-middleware.html>

MW GET: rutas

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

var app = express();

app.get('/', function (req, res){
  res.send('Welcome to my first server app\n' + 'Responds to: /');
});

app.get('/first_route', function (req, res){
  res.send('Responds to: /first_route');
});

app.get('/my/route', function (req, res){
  res.send('Responds to: /my/route');
});

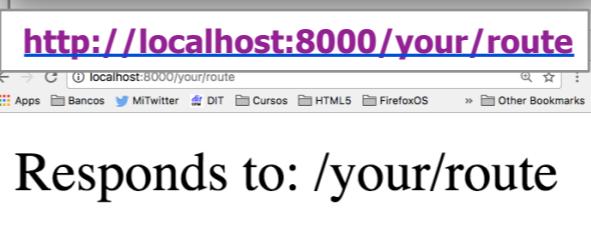
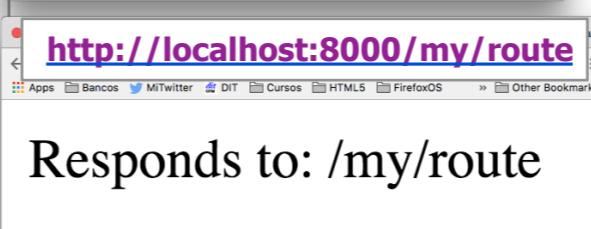
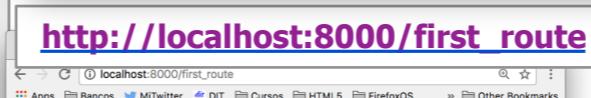
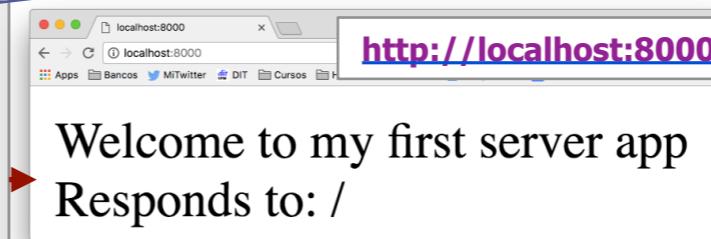
app.get('/your/route', function (req, res){
  res.send('Responds to: /your/route');
});

app.get('*', function (req, res){
  res.send('Responds to: any other route');
});

app.listen(8000);
```

var express = require('express') importa el módulo express (instalado con npm) en la **variable** express.

var app = express() crea la aplicación express donde se van a **instalar los MWs** en la variable **app**.



Un MW se define con un literal de función:

```
function (req, res) {
  res.send('Responds to: /first_route');
}
```

Este MV envía al navegador una **respuesta HTTP** con el siguiente mensaje en el cuerpo:

'Responds to: /first_route'

Cada vez que se recibe una solicitud GET las rutas de los 5 MWs instalados se comparan con la ruta recibida en el orden en que se han instalado:

1. / (ruta base o home)
2. /first_route
3. /my/route
4. /your/route
5. * (* significa cualquier ruta)

Si la ruta recibida coincide con el patrón, su MW se invoca, devuelve la respuesta con **req.send(..)** y la atención a la solicitud HTTP finaliza.

app.listen(8000) arranca esta aplicación express conectada al puerto 8000 para que responda a solicitudes HTTP en ese puerto.

Por ejemplo, para enviar una solicitud **GET** con ruta **/my/route** desde un navegador en el mismo ordenador se puede utilizar el URL:

<http://localhost:8000/my/route>

MW static y rutas

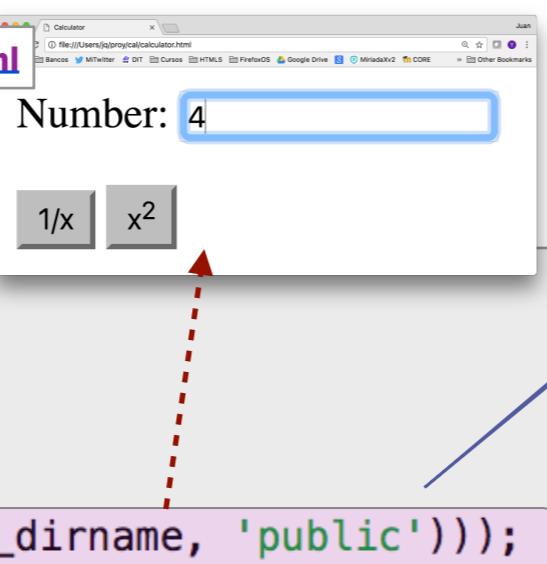
<http://localhost:8000/app/calculator.html>

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

var app = express();

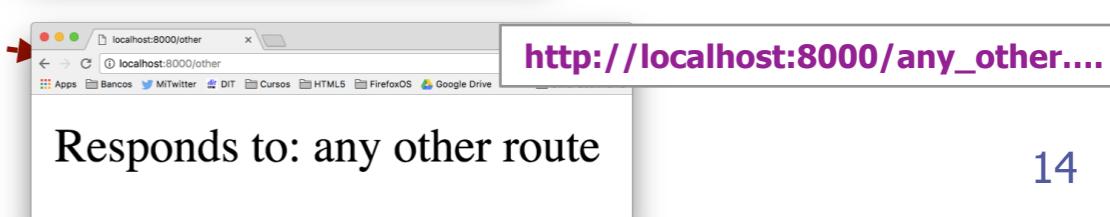
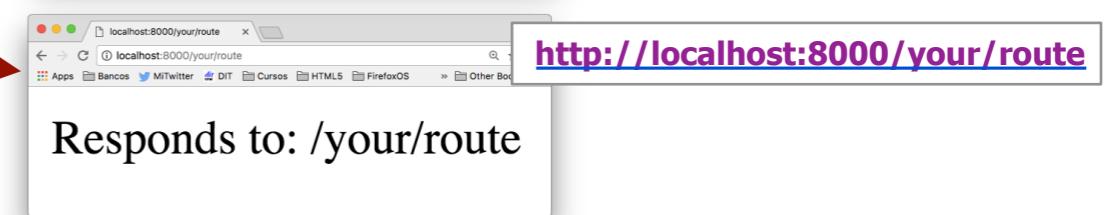
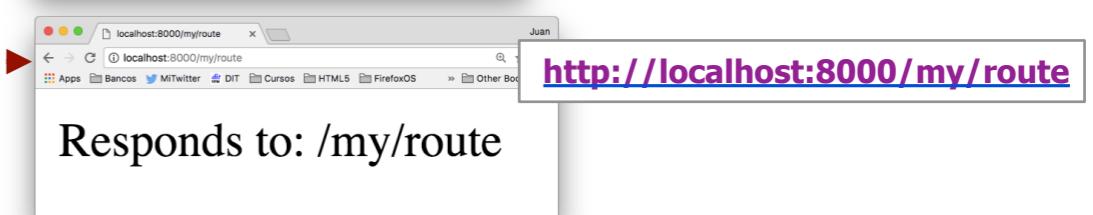
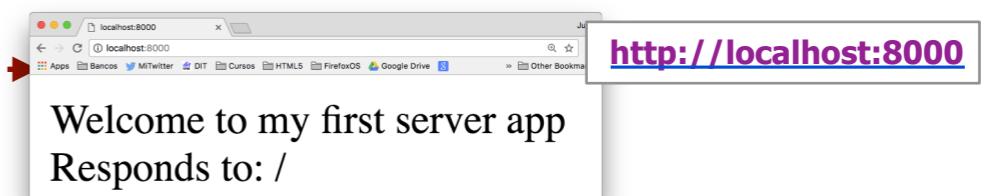
app.use(express.static(path.join(__dirname, 'public')));

app.get('/', function (req, res){-----  
    res.send('Welcome to my first server app\n' + 'Responds to: /');  
});  
  
app.get('/first_route', function (req, res){-----  
    res.send('Responds to: /first_route');  
});  
  
app.get('/my/route', function (req, res){-----  
    res.send('Responds to: /my/route');  
});  
  
app.get('/your/route', function (req, res){-----  
    res.send('Responds to: /your/route');  
});  
  
app.get('*', function (req, res){-----  
    res.send('Responds to: any other route');  
});  
  
app.listen(8000);
```



El **MW static** sirve las páginas Web del repositorio de recursos que pidan los clientes y finaliza. Los demás MWS no se ejecutan.
Si llegase una ruta donde no hay recurso (página Web) en el repositorio, el MW static pasa el control a los siguientes MWS.
Si la ruta a un recurso en el repositorio de recursos estáticos coincide con la de un MW , este último nunca se ejecutará.

Cada **MW** atiende solo transacciones asociados a su(s) ruta(s). Si no puede atender a la transacción pasa control al siguiente MW.



Rutas genéricas: parámetros y regexp

◆ Rutas express: se dividen en segmentos separados por: /

- Por ejemplo, **/hola/que/tal** tiene 3 segmentos: hola, que y tal
 - ◆ Documentación: <http://expressjs.com/en/guide/routing.html>

◆ Los segmentos pueden definirse de diversas formas

- **Literal:** texto con caracteres permitidos en un URL que debe incluirse literalmente
 - ◆ Algunos ejemplos de literales: **/client**, **/game**, **/quiz/hola**, **/que**, **/23**, ...
- **Parámetros:** variables que aceptan cualquier texto
 - ◆ Algunos ejemplos de parámetros de ruta son: **/:x**, **/:x1**, **/:x_1**, **/:model**,
 - ◆ Su valor estará accesible en el MW en: **req.params.x**, **req.params.x1**, **req.params.x_1**,
 - ◆ Un parámetro puede tener letras ASCII, dígitos decimales o _ (sintaxis **:[A-Za-z0-9_]+**)
 - ◆ Un segmento puede incluir varios parámetros separados por - o . , por ej. **/:from-:to**, **/:file.:extention**
- Los parámetros pueden **restringirse** con una **expresión regular**, por ej.
 - ◆ **/:x([0-9]+)** el parámetro x solo puede ser un número decimal
 - ◆ **/:idioma(es | en | ge | it | fr)** el parámetro idioma solo puede contener los strings: es, en, ge, it o fr
- * - casa con cualquier string, por ej. **/pep*** puede casa con: pep, pepppp, pepito, ...
- ? - 0 o 1 vez, por ej. **/cars?** (casa con car o cars), **/c(ars)?** (casa con c o cars), ..
- + - 1 o más veces, por ej. **/cars+** (cars, carss, ...), **/c(ars)+** (cars, carsars, ...), ..

MWs con rutas y GET

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

var app = express();

app.get('/cars?', function (req, res){
  res.send('Responds to: /car or /cars');
});

app.get('/car/:model', function (req, res){
  var model = req.params.model;
  res.send('Model of your car: ' + model);
});

app.get('/add/:x([0-9]+)/:y([0-9]+)', function (req, res){
  var x = req.params.x;
  var y = req.params.y;
  res.send(x + ' + ' + y + ' = ' + (+x + +y));
});

app.get('*', function (req, res){
  res.send('Unknown request');
});

app.listen(8000);
```

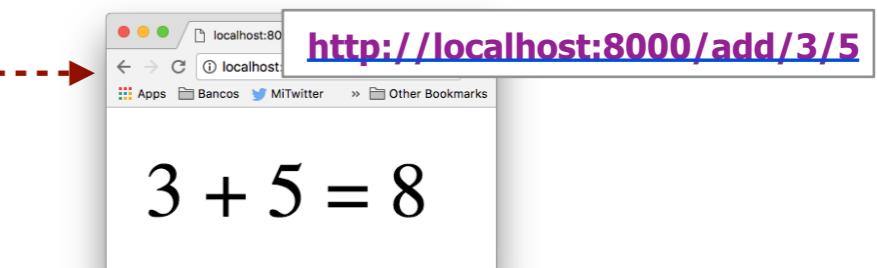
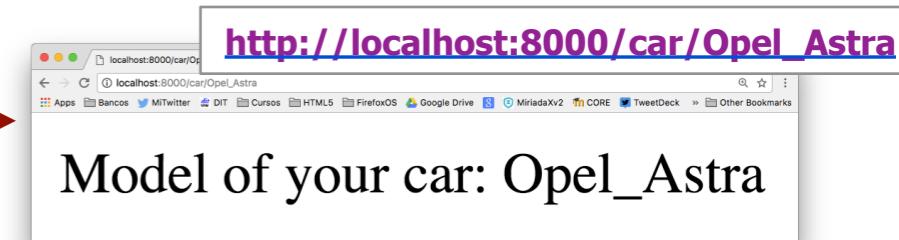
<http://localhost:8000/add/xx/yy>

Unknown request

El operador ? (0 o 1 vez) se puede aplicar a un carácter de un literal, como aquí, o a bloques de caracteres de un literal delimitados por paréntesis.



Esta ruta genérica aceptará una ruta con 2 segmentos. El primero es el literal **car** que debe llegar exactamente igual y el segundo puede contener cualquier string. El string estará accesible en el MW en la propiedad **req.params.model**.



Esta ruta tiene 3 segmentos. El primero es el literal **add** que debe llegar igual y los siguientes son 2 parámetros restringidos a números: strings compuestos únicamente por dígitos decimales, tal y como restringe la regex **[0-9]+**. **[0-9]+** es equivalente a **\d+**.

La ruta **/add/xx/yy** será atendida por este MW, porque la expresión regular del MW anterior limita los parámetros a strings con dígitos decimales.

Ejercicio

Supongamos que en nuestro ordenador local arrancamos la siguiente aplicación express conectada al puerto 80 y que no hay ningún otro servidor en ningún otro puerto:

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

app.get('/coche', function (req, res){res.send( 'Coche' );});
app.get('/casa/*', function (req, res){res.send( 'Casa' );});
app.get('*', function (req, res){res.send( 'Nada' );});
app.listen(80);
```

Como responderá esta aplicación a los siguientes URLs:

<http://localhost>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost:8080>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost:8000/casa>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost/coche>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost:12/coche>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost/casa/casa>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost/casa/coche>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost/coche/casa>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

<http://localhost/coche/coche>

=> 'Coche', 'Casa', 'Nada', Error-NoHayServidor

Ejercicio

Supongamos que en nuestro ordenador local arrancamos la siguiente aplicación express:

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

app.get('/:id1(\d+)/:id2?', function (req,res){
  res.send( req.params.id1 + (req.params.id2 || ""));
});

app.get('*', function (req, res){res.send( 'Nadie' );});
app.listen(80);
```

Con que texto responderá esta aplicación a los siguientes URLs:

<u>http://localhost/3/P2</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/P2</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/3</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/37/signals</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/cuatro/caminos</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/4/caminos</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/4caminos</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/signals</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/3P2</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'
<u>http://localhost/37signals</u>	=> '3P2', '3', 'P2', '37signals', '4caminos', 'cuatrocaminos', 'Nadie'



JavaScript



Ejemplos de MWs y de uso de req, res, next

Juan Quemada, DIT - UPM

Instalación e invocación de MWs

◆ Los MWs se pueden instalar en una aplicación express de forma genérica o asociar a cualquier método HTTP: GET, POST, PUT, DELETE,, por ej.

- **app.use('ruta', MW)** Instala **MW** asociado a **cualquier** método HTTP y a '**ruta**'
- **app.get('ruta', MW)** Instala **MW** asociado al método HTTP **GET** y a '**ruta**'
- **app.post('ruta', MW)** Instala **MW** asociado al método HTTP **POST** y a '**ruta**'
- **app.put('ruta', MW)** Instala **MW** asociado al método HTTP **PUT** y a '**ruta**'
- **app.delete('ruta', MW)** Instala **MW** asociado al método HTTP **DELETE** y a '**ruta**'
- **app.head('ruta', MW)** Instala **MW** asociado al método HTTP **HEAD** y a '**ruta**'
- y así para todos los métodos o verbos de HTTP
 - ◆ Documentación: <http://expressjs.com/en/4x/api.html#app.METHOD>

◆ **MW (req, res, next) { }**

- La aplicación express invoca siempre un MW con los parámetros **req, res** y **next**
 - ◆ **req**: Objeto JavaScript con los parámetros de la **solicitud HTTP** recibida
 - ◆ **res**: Objeto JavaScript que permite configurar y enviar la **respuesta HTTP** al cliente
 - ◆ **next**: función que al ser invocada (**next()**) continua evaluación del resto de MWs instalados

◆ **req y res**

- Son objetos persistentes que pueden **procesarse en serie** por los **MWs**
 - ◆ Documentación: <http://expressjs.com/en/4x/api.html#req>, <http://expressjs.com/en/4x/api.html#res>

Parámetros req y res

La documentación de la API:
<http://expressjs.com/4x/api.html>

function(req, res) {}

req: elementos de la
Solicitud HTTP
representados como
propiedades y métodos
de un objeto JavaScript
accesible en un MW.

Solicitud HTTP

método ruta HTTP/v.v

Host: upm.es

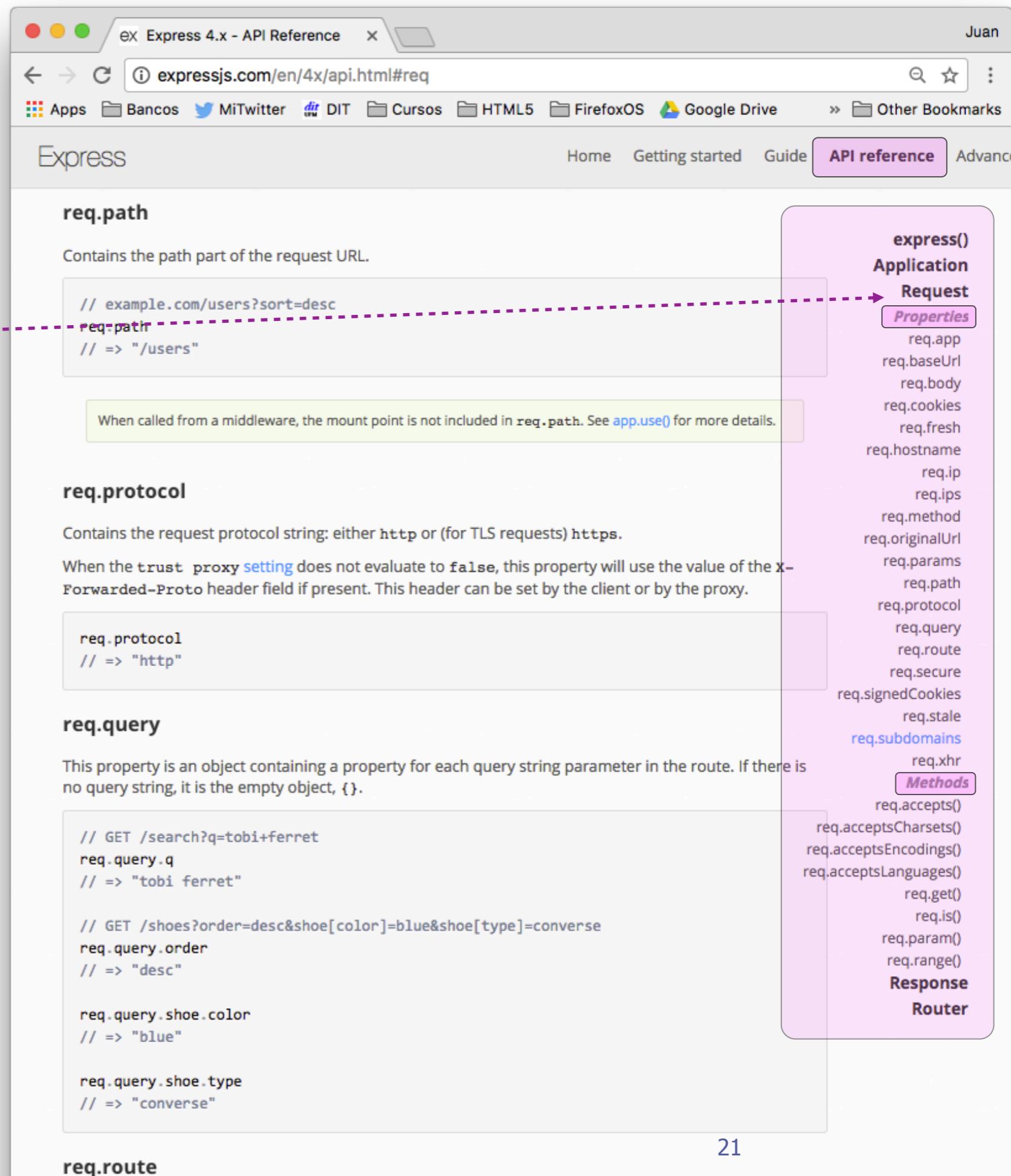
Accept: text/*, image/*

Accept-language: en, sp

.....

User-Agent: Mozilla/5.0

..... Cuerpo



The screenshot shows a browser window displaying the Express 4.x API Reference at expressjs.com/en/4x/api.html#req. The page is titled 'Express' and has tabs for Home, Getting started, Guide, API reference (which is highlighted), and Advanced. The main content is about the 'req.path' property, which contains the path part of the request URL. It includes a code example and a note about middleware. To the right, there is a sidebar with a tree view of the Express API structure, starting with 'express()' and branching into 'Application', 'Request', 'Properties', and 'Methods'. The 'Properties' section lists various req.* properties, and the 'Methods' section lists req.* methods.

req.path

Contains the path part of the request URL.

```
// example.com/users?sort=desc
req.path
// => "/users"
```

When called from a middleware, the mount point is not included in `req.path`. See `app.use()` for more details.

req.protocol

Contains the request protocol string: either `http` or (for TLS requests) `https`.

When the `trust proxy` setting does not evaluate to `false`, this property will use the value of the `X-Forwarded-Proto` header field if present. This header can be set by the client or by the proxy.

```
req.protocol
// => "http"
```

req.query

This property is an object containing a property for each query string parameter in the route. If there is no query string, it is the empty object, `{}`.

```
// GET /search?q=tobi+ferret
req.query.q
// => "tobi ferret"

// GET /shoes?order=desc&shoe[color]=blue&shoe[type]=converse
req.query.order
// => "desc"

req.query.shoe.color
// => "blue"

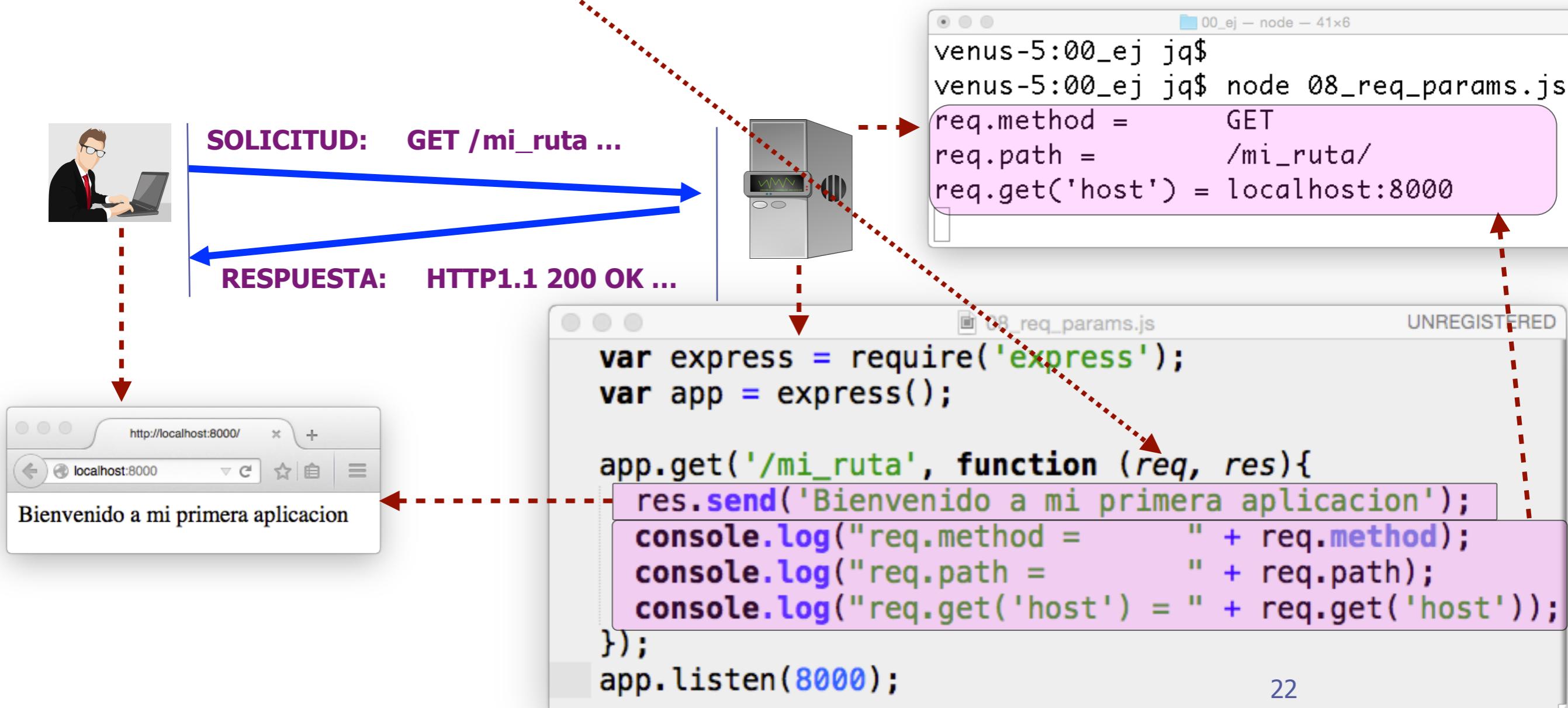
req.query.shoe.type
// => "converse"
```

req.route

Ejemplo req: parámetros de la solicitud

El parámetro **req** es un objeto JS con toda la información de la solicitud HTTP y permite conocer sus parámetros.

El ejemplo lista por **consola del servidor** el método y la ruta de la solicitud obtenidos de las propiedades asociadas de **req** y el parámetro 'host :....' obtenido con el método 'get(...)' de **req**.



Parámetros req y res

La documentación de la API:
<http://expressjs.com/4x/api.html>

function(req, res) {}

res: elementos de la Respuesta HTTP representados como propiedades y métodos de un objeto JavaScript accesible en un MW.

Respuesta HTTP

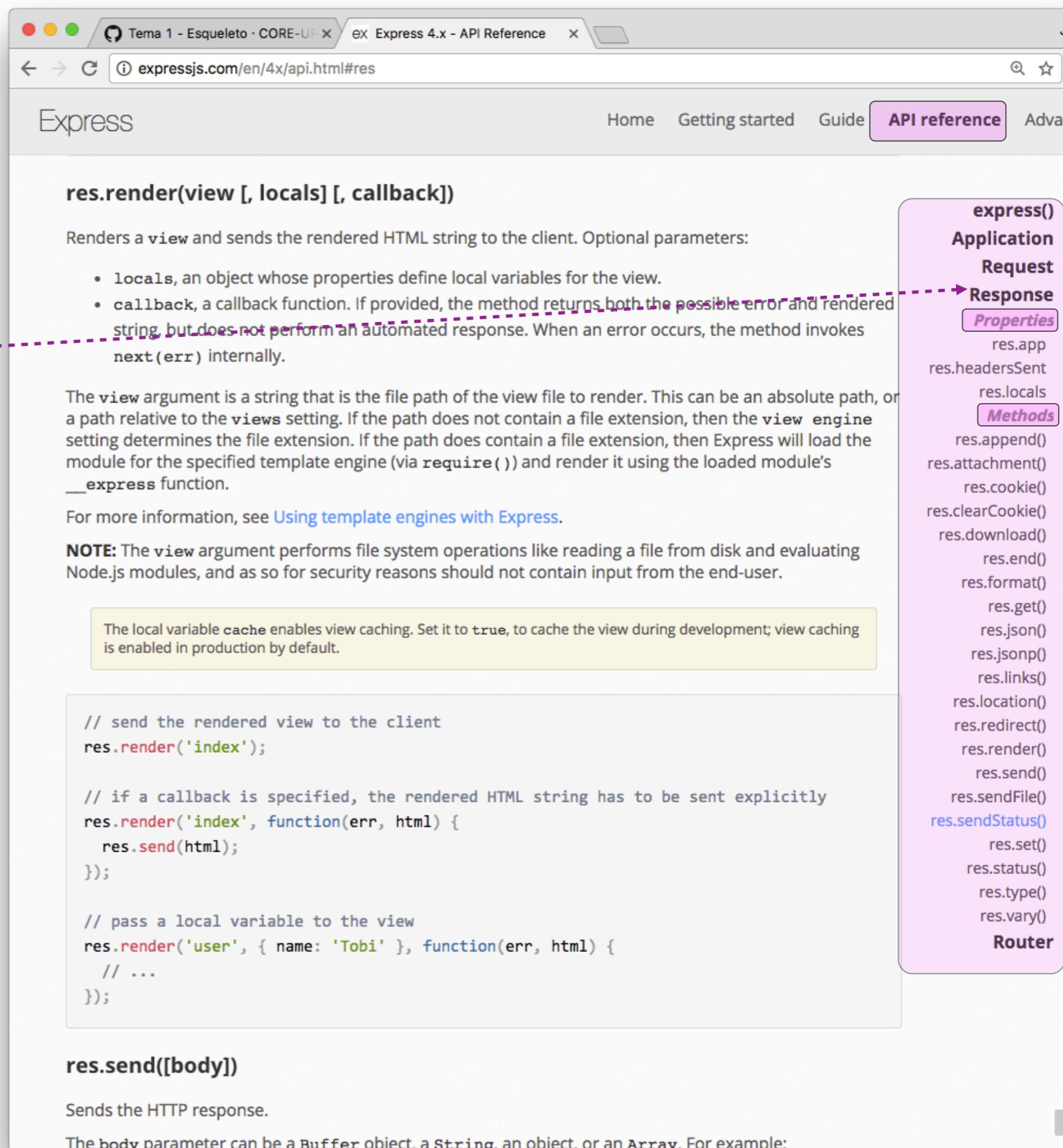
HTTP/v.v código msg

Server: Apache/1.3.6

Content-type: text/html

Content-length: 608

..... Cuerpo



The screenshot shows a browser window displaying the Express.js API documentation at expressjs.com/en/4x/api.html#res. The page title is "Tema 1 - Esqueleto · CORE-UP" and the tab is "Express 4.x - API Reference". The "API reference" tab is highlighted in pink. On the right, there is a sidebar with a tree view of the Express API structure:

- express()
 - Application
 - Request
 - Response
 - Properties
 - Methods
 - res.app
 - res.headersSent
 - res.locals
 - res.append()
 - res.attachment()
 - res.cookie()
 - res.clearCookie()
 - res.download()
 - res.end()
 - res.format()
 - res.get()
 - res.json()
 - res.jsonp()
 - res.links()
 - res.location()
 - res.redirect()
 - res.render()
 - res.send()
 - res.sendFile()
 - res.sendStatus()
 - res.set()
 - res.status()
 - res.type()
 - res.vary()
 - Router

res.render(view [, locals] [, callback])

Renders a `view` and sends the rendered HTML string to the client. Optional parameters:

- `locals`, an object whose properties define local variables for the view.
- `callback`, a callback function. If provided, the method returns both the possible error and rendered string, but does not perform an automated response. When an error occurs, the method invokes `next(err)` internally.

The `view` argument is a string that is the file path of the view file to render. This can be an absolute path, or a path relative to the `views` setting. If the path does not contain a file extension, then the `view` engine setting determines the file extension. If the path does contain a file extension, then Express will load the module for the specified template engine (via `require()`) and render it using the loaded module's `_express` function.

For more information, see [Using template engines with Express](#).

NOTE: The `view` argument performs file system operations like reading a file from disk and evaluating Node.js modules, and as so for security reasons should not contain input from the end-user.

The local variable `cache` enables view caching. Set it to `true`, to cache the view during development; view caching is enabled in production by default.

```
// send the rendered view to the client
res.render('index');

// if a callback is specified, the rendered HTML string has to be sent explicitly
res.render('index', function(err, html) {
  res.send(html);
});

// pass a local variable to the view
res.render('user', { name: 'Tobi' }, function(err, html) {
  // ...
});
```

res.send([body])

Sends the HTTP response.

The `body` parameter can be a `Buffer` object, a `String`, an object, or an `Array`. For example:

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,

Solicitud HTTP GET

1a linea

GET /me.htm HTTP/1.1

Host: upm.es

Accept: text/*, image/*

Accept-language: en, sp

.....

User-Agent: Mozilla/5.0

parámetros
de cabecera

Cuerpo

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 / subtipo**,

■ Tipos: application, audio, example, image, message, model, multipart, text, video

Respuesta HTTP GET

1a linea

HTTP/1.1 200 OK

Server: Apache/1.3.6

Content-type: text/html

.....

Content-length: 608

<html> </html>

parámetros
de cabecera

Cuerpo:

Pág. HTML

◆ 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”

Ejemplo res: parámetros de la respuesta

El parámetro **res** permite configurar la Respuesta HTTP.

El ejemplo configura el **tipo MIME** de la página HTML enviada como texto plano con **res.type('text/plain')**. La respuesta HTTP llevará por lo tanto el parámetro: **Content-Type: text/plain**.

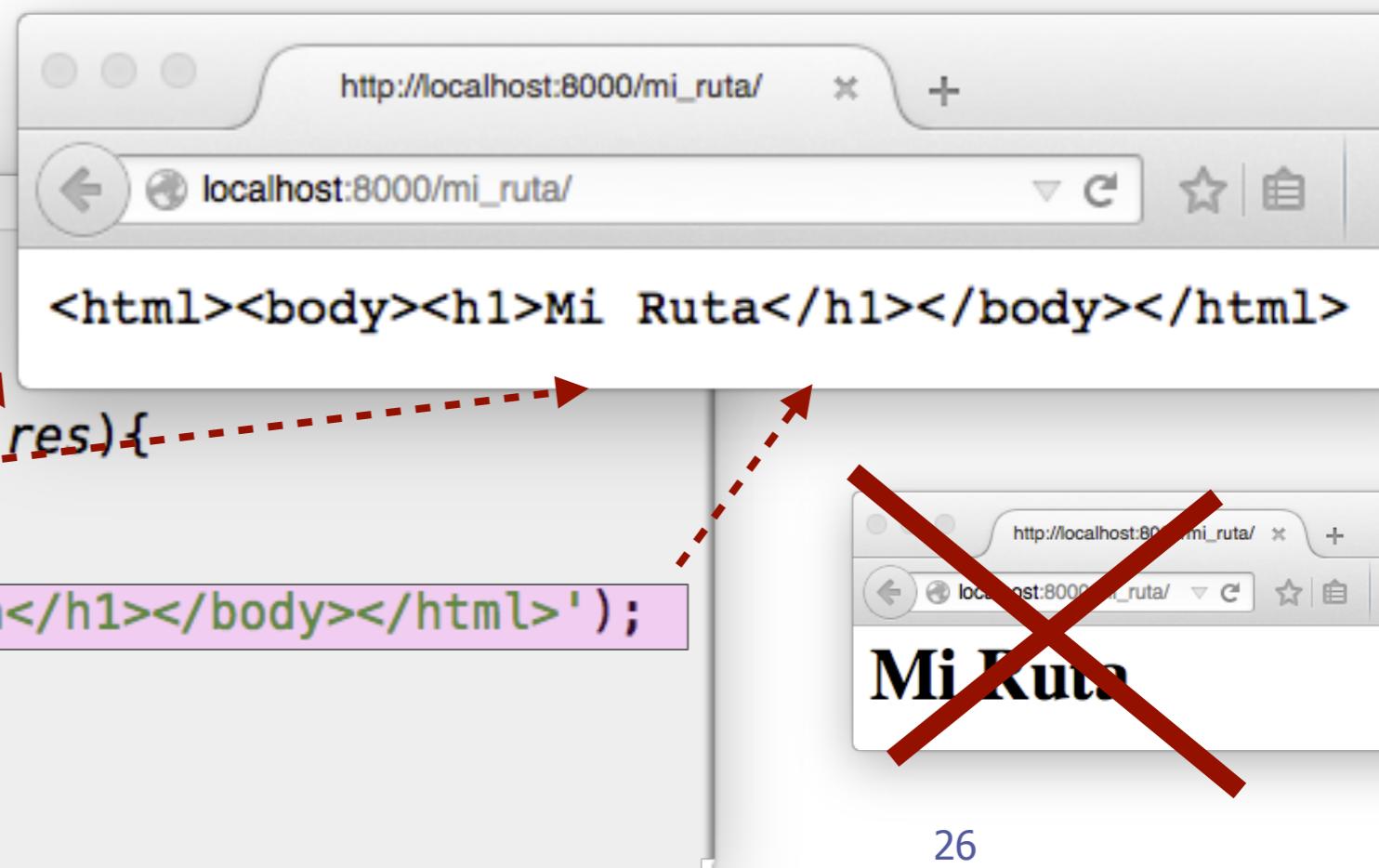
El navegador interpretará la página HTML como texto plano mostrando el código HTML en vez de la página formateada.

El programa configura además explícitamente el el código **200 OK** con el método **status()** de la API.

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

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

app.listen(8000);
```



Ejercicio cabeceras HTTP

Diseñar una aplicación express conectada al puerto 8080 que acepte cualquier transacción GET de HTTP con cualquier ruta (path) y que muestre por consola el path, la dirección IP del cliente y los tipos MIME aceptados por el cliente.



JavaScript



Composición y ejecución de middlewares: next() y next(Error)

Juan Quemada, DIT - UPM

Composición de MWs express

◆ La aplicación express atiende una solicitud HTTP

- Evaluando los MWs instalados siguiendo el orden de instalación
 - ◆ La aplicación express invoca el primer MW cuyo método y ruta casan con los de la solicitud

◆ Formato completo de un MW: **MW(req, res, next)**

- **req**: objeto con los parámetros de la solicitud HTTP
- **res**: objeto donde se insertan los parámetros de la respuesta HTTP
- **next**: función que pasa control a la aplicación express
 - ◆ La aplicación express evaluará los MWs siguientes, a partir del que ha invocado next()

◆ La atención a la solicitud finalizará si el MW invocado no invoca next()

- Un MW que no pasa control al siguiente deberá enviar la respuesta al cliente.

◆ El uso de **next()** permite partir la atención a una solicitud en pasos

- Los pasos se activan con las mismas rutas y ceden control al siguiente con next()

◆ Los middlewares permiten programación secuencial y modular

- Conservando la eficacia de la programación por eventos
 - ◆ <http://expressjs.com/en/guide/writing-middleware.html>

Contador de visitas

Un **MW** puede utilizar **app.locals** y **res.locals** para definir variables locales como propiedades creadas dinámicamente en dichos objetos. **app.locals** es visible en todo app y **res.locals** solo es visible en **res** en el mismo middleware (no se hubiese podido utilizar aquí).

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

app.use(function(req, res, next){
    app.locals.count = (++app.locals.count || 1);
    console.log("Visits: " + app.locals.count);
    next();
});

app.get('*', function(req, res){
    res.send('Visit number: ' + app.locals.count);
});

app.listen(8000);
console.log('Listening on port 8000');
```

Este **MW** crea la variable **count** en **app.locals** para que sea visible en ambos MWs. La variable count se inicializa a 1 en la primera ejecución y en las siguientes se incrementa en 1.

Una aplicación express intenta ejecutar los MWs instalados siguiendo el orden de instalación.

La función **next()** puede pasar el control al siguiente MW. Si **no se invoca next()**, la ejecución de MWs finaliza en el primero MW invocado.



Visit number: 4

Invocación condicional de next()

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

app.get('/user/:id', function(req, res, next){
  var user = req.params.id;
  if (user === 'Ana' || user === 'Pit'){
    res.send('Systems user');
  } else {
    next();
  }
});

app.get('*', function(req, res){
  res.send('Unknown user');
});

app.listen(8000);
console.log('Listening ooon port 8000');
```

El primer MW invoca condicionalmente el siguiente MW. Si la solicitud es GET con una de estas rutas

/user/Ana

/user/Pit

envía directamente la respuesta HTTP al cliente.

Si no ha llegado GET con una de estas rutas pasa en control al siguiente MW.



Middleware de error

- ◆ **Middleware de error:** atiende situaciones de error
 - función que podrá invocarse cuando otro MW lance un error con **next(err)**
- ◆ Formato: **MW(err, req, res, next)**
 - **err**: objeto error lanzado en otro middleware
 - **req**: objeto con los parámetros de la solicitud HTTP
 - **res**: objeto donde se insertan los parámetros de la respuesta HTTP
 - **next**: función que pasa control al siguiente middleware al ser invocada (**next()**)
- ◆ Se instalan con **use(..)**, **get(..)**, **put(..)**, .. igual que los demás
 - Un MW, sea de error o no, **invoca el siguiente MW con next()**
 - Un MW, sea de error o no, **invoca el siguiente MW de error con next(err)**
 - ◆ Si **next()** no se invoca, se debe enviar la respuesta, porque la ejecución finaliza en ese MW
- ◆ Los middlewares de error permiten estructurar un programa
 - Agrupando la atención a errores en un bloque aparte, normalmente al final
 - ◆ <http://expressjs.com/guide/using-middleware.html>

MW de error

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

```
app.get('/user/:id', function(req, res, next){
  var user = req.params.id;
  if (user === 'Ana' || user === 'Pit'){
    res.send('Systems user');
  }
  else {
    next(new Error('Unknown user'));
  }
});
```

```
app.get('*', function(req, res){
  res.send('Invalid operation');
});
```

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

```
app.listen(8000);
console.log('Listening on port 8000');
```

El primer MW envía directamente la respuesta HTTP al cliente si la solicitud es GET y la ruta es /user/Ana o /user/Pit y sino pasa control al siguiente MW de error, que notificará el error. El siguiente MW responderá a cualquier solicitud GET cuya ruta diferente de /user/:id.



next(new Error(...)) pasa control al primer MW de error que pueda atender la transacción.



Error: Unknown user



Express

- 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.

Librería
de MWs
express

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>



JavaScript



Formularios GET y POST: Parámetros (en query, en body u ocultos), URL encode y method override

Juan Quemada, DIT - UPM

Formularios: Envío de datos

◆ Formulario

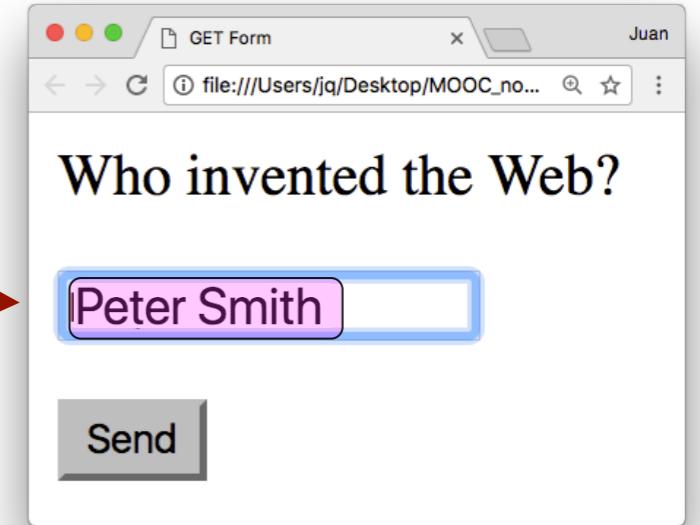
- Elemento HTML capaz de enviar datos desde el cliente al servidor
 - ◆ El formulario se puede configurar para que envíe los datos en solicitudes **GET** o **POST**
 - ◆ El atributo **method** indica GET o POST

◆ Formulario GET

- Envía datos en el query de la ruta
 - ◆ Se utiliza para enviar consultas con parámetros pequeños que no modifican la BBDD

◆ Formulario POST

- Envía datos en el cuerpo (body)
 - ◆ Se utiliza para transacciones que modifican la BBDD.
 - ◆ Puede enviar datos de gran tamaño, tales como, ficheros, o datos masivos, ...



GET /check?answer=Peter+Smith HTTP/1.1

Host: upm.es

Accept: text/*, image/*

Accept-language: en, sp

User-Agent: Mozilla/5.0

POST /hola HTTP/1.1

Host: upm.es

Accept: text/*, image/*

Content-type: application/x-www-form-urlencoded

Content-length: 19

answer=Peter+Smith

Ejemplo: formulario GET

```
<!DOCTYPE html><html>
<head>
  <title>GET Form</title>
  <meta charset="UTF-8">
</head>
```

```
<body>
<form method="get" action="/check">
  Who invented the Web?
  <p>
    <input type="text" name="answer" value="Respond" />
  <p>
    <input type="submit" value="Send" />
</form>
</body>
</html>
```

Un formulario envía solicitudes HTTP **GET** y **POST** al servidor. **<form>** es un elemento HTML complejo que contiene otros elementos:

<form method=.. action=.. >

- > El atributo **method** indica si la solicitud HTTP será **GET** o **POST**
- > El atributo **action** define la ruta asociada a la solicitud HTTP.

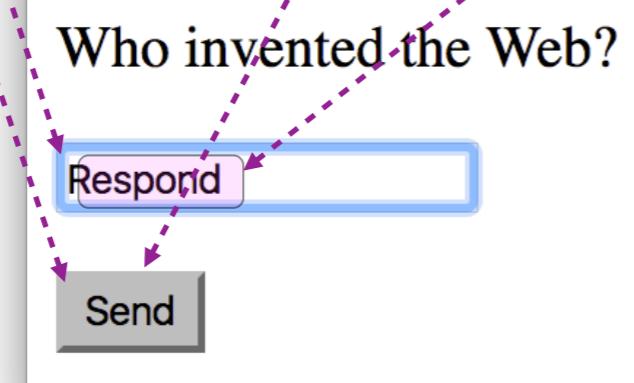
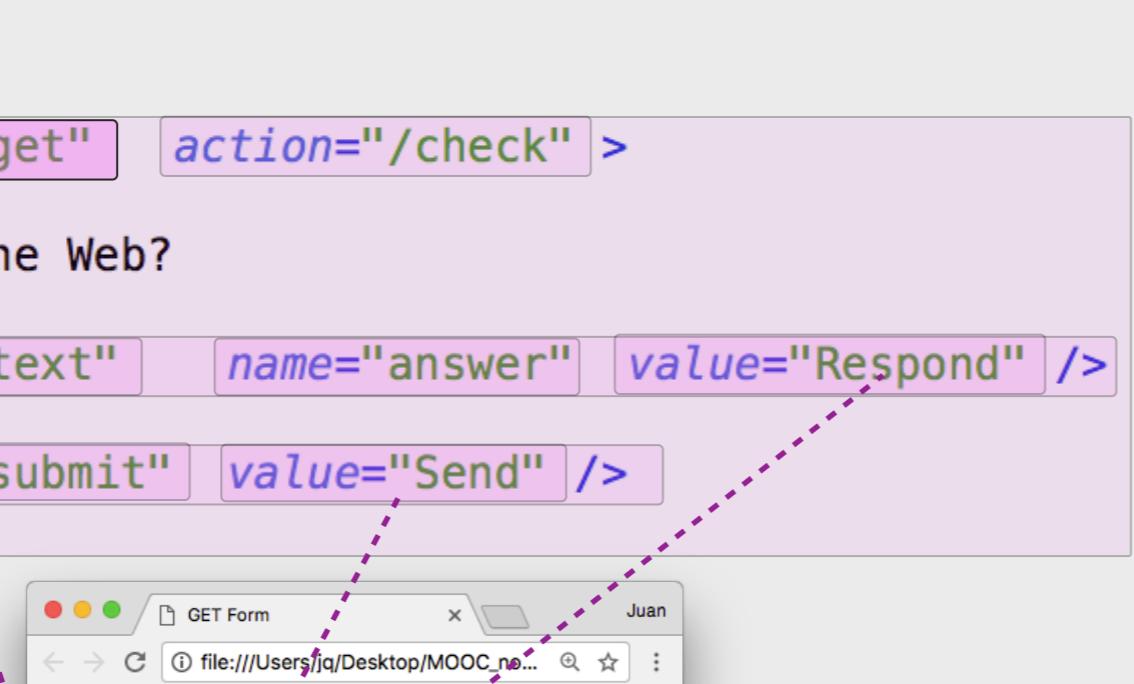
Marca **<input ..>**: permite el cajetín y el botón del formulario.

<input type='text' ..> crea un cajetín para teclear texto.

- > **name='answer'** nombre del parámetro
- > **value='Respond'** texto inicial del cajetín.

<input type='submit' ..> crea botón de envío de HTTP GET.

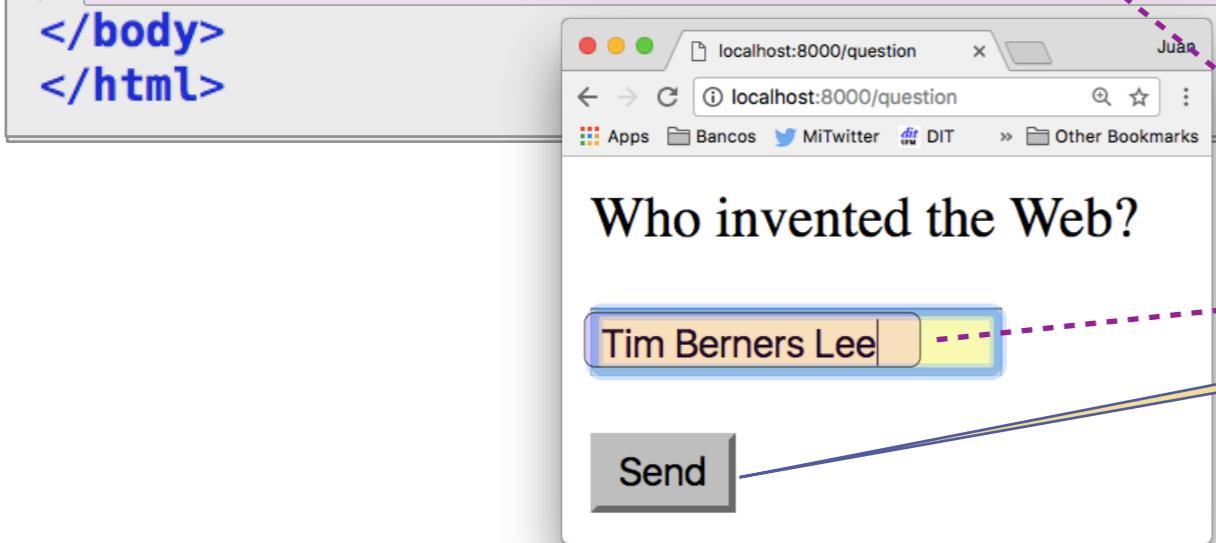
- > **value='Send'** define texto mostrado en el botón



Ejemplo: formulario GET

```
<!DOCTYPE html><html>
<head>
  <title>GET Form</title>
  <meta charset="UTF-8">
</head>

<body>
  <form method="get" action="/check">
    Who invented the Web?
    <p>
      <input type="text" name="answer" value="Respond" />
    <p>
      <input type="submit" value="Send" />
    </form>
</body>
</html>
```

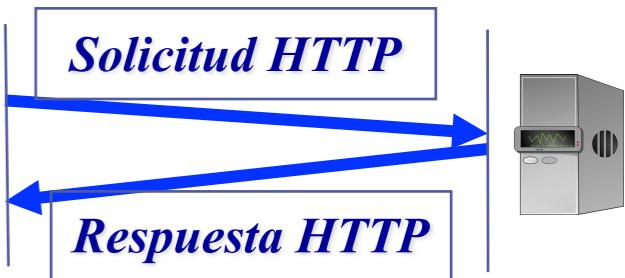


La solicitud HTTP **GET** se envía al pulsar el botón **Send** y su primera línea será:

GET /check?answer='Tim+Berners+Lee' **HTTP/1.1**

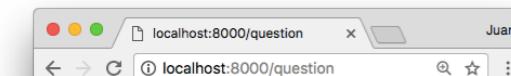
..... parámetros

..... parámetros



Ejemplo GET

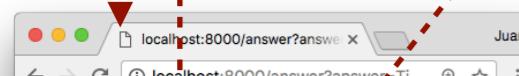
Usar formularios necesita **2 transacciones**: primero se **carga el formulario** y luego se **envían los datos** pedidos en el formulario.



Who invented the Web?

Tim Berners Lee

Send



Correct, Tim Berners Lee
did invent the Web

[try again](#)

Solicitud del formulario

Envío del formulario

Envío de datos

Envío del resultado

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

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

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

```
app.get('/check', function(req, res) {
  var answer = req.query.answer;
  var response = "Correct, " + answer + " did";

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

La transacción **GET** asociada a la ruta **/check** (segunda) envía datos y recibe la respuesta asociada.

```
res.send('<html>
  + <body>
  +   response + " invented the Web <p>
  +   <a href="/question">try again</a>
  + </body>
  + </html>
);
```

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

URL Encoding

URL or percent encoding

◆ Los parámetros del query y el URL se envía por Internet

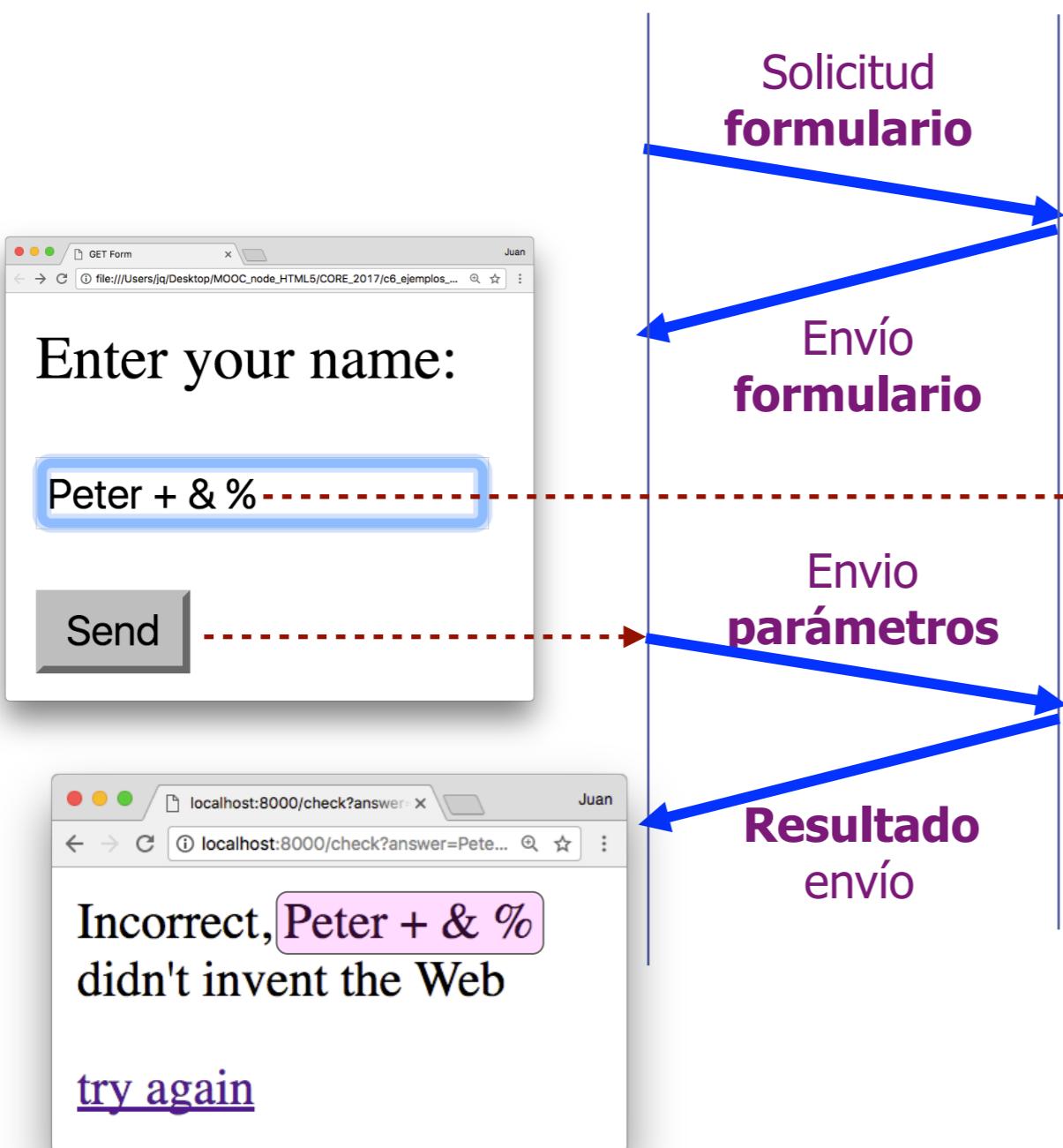
- Codificados con: **percent** o **URL encoding**
 - ◆ Corresponde con tipo MIME: “**application/x-www-form-urlencoded**”
 - **URL encoding** respeta la sintaxis de un URL

◆ 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 asignados a los parámetros del query siempre 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.

Formulario POST y MW urlencode

Ejemplo: formulario POST

```
<!DOCTYPE html><html>
<head>
  <title>GET Form</title>
  <meta charset="UTF-8">
</head>
```

```
<body>
<form method="post" action="/check">
  Who invented the Web?
  <p>
    <input type="text" name="answer" value="Respond" />
  <p>
    <input type="submit" value="Send" />
</form>
</body>
</html>
```

Un formulario envía solicitudes HTTP **GET** y **POST** al servidor. **<form>** es un elemento HTML complejo que contiene otros elementos:

<form method=.. action=.. >

- > El atributo **method** indica si la solicitud HTTP será **GET** o **POST**
- > El atributo **action** define la ruta asociada a la solicitud HTTP.

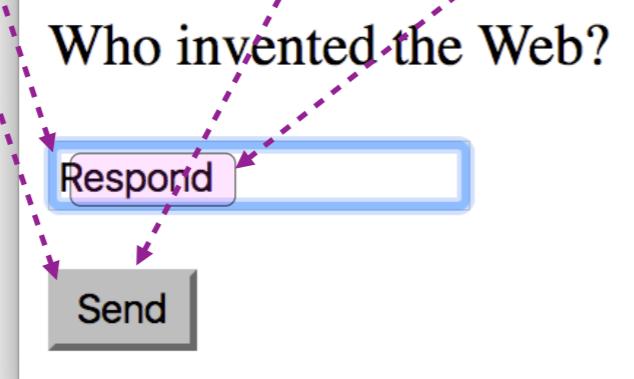
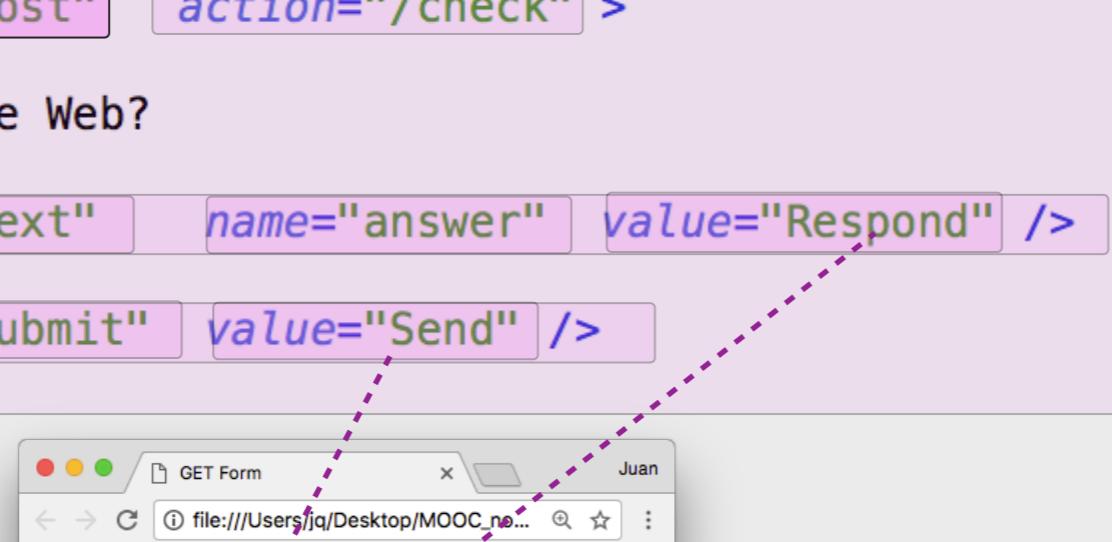
Marca **<input ..>**: permite el cajetín y el botón del formulario.

<input type='text' ..> crea un cajetín para teclear texto.

- > **name='answer'** nombre del parámetro
- > **value='Respond'** texto inicial del cajetín.

<input type='submit' ..> crea botón de envío de HTTP GET.

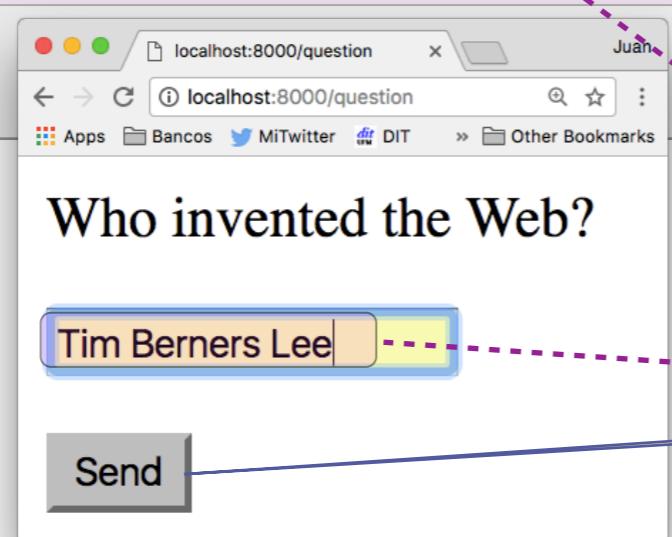
- > **value='Send'** define texto mostrado en el botón



Ejemplo: formulario POST

```
<!DOCTYPE html><html>
<head>
  <title>GET Form</title>
  <meta charset="UTF-8">
</head>

<body>
  <form method="post" action="/check">
    Who invented the Web?
    <p>
      <input type="text" name="answer" value="Respond" />
    <p>
      <input type="submit" value="Send" />
    </form>
</body>
</html>
```



localhost:8000/question

Who invented the Web?

Tim Berners Lee

Send



Solicitud HTTP



Respuesta HTTP

La solicitud HTTP **POST** se envía al pulsar el botón **Send** y su estructura será:

POST /check **HTTP/1.1**

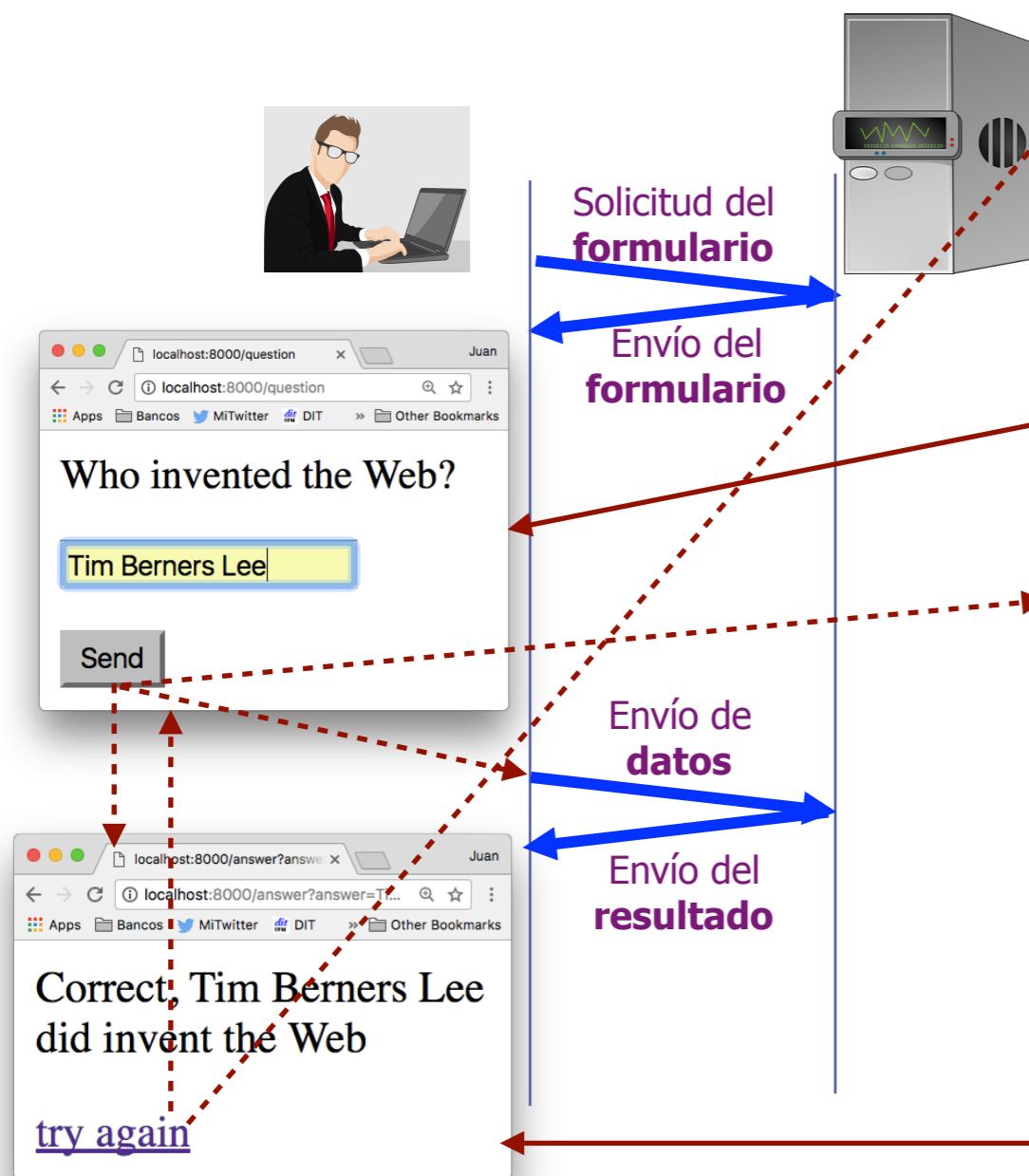
..... parámetros

answer='Tim+Berners+Lee'

<- body

Ejemplo POST

Usar formularios necesita **2 transacciones**: primero se **carga el formulario** y luego se **envían los datos** pedidos en el formulario.



Se debe instalar con npm el módulo **body-parser**
\$ **npm install body-parser**
Una vez instalado en el proyecto se importa el módulo **body-parser** en el programa y se instala el MW **bodyParser.urlencoded**. Más información en: <https://github.com/senchalabs/connect#middleware>

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

var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({ extended: true }));

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

app.post('/check', function(req, res) {
  var answer = req.body.answer;
  var response = "Correct, " + answer + " did";

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

  res.send('<html>
    + <body>
    +   response + " invent the Web <p>
    +   <a href="/question">try again</a>
    + </body>
    + </html>');
});

app.listen(8000);
```

El formulario debe indicar el uso del método POST.

Este MW debe instalarse con **post(...)**.

MW urlencoded: decodifica los parámetros en **req.body**.

Esta nueva aplicación express con formulario POST es similar a la de GET salvo los cambios marcados en **marrón**.

Parámetro oculto y method override

Parámetro oculto

◆ Parámetro oculto

- Parámetro **no visible** en una página HTML **enviado por el servidor al cliente**
 - ◆ Se pueden enviar en el **query de una ruta** o como una **entrada input oculta** de un formulario
- El **cliente devuelve el parámetro** al servidor al **enviar la solicitud HTTP** asociada

◆ Ejemplo de envío en el **query de una ruta** de en un hiperenlace

- ` Envío de parámetro oculto`

◆ Ejemplo de envío como **input de tipo "hidden"** en un formulario

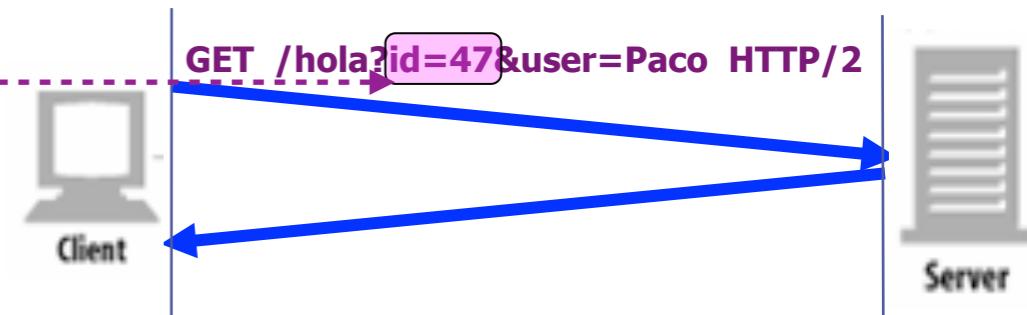
- `<input type="hidden" name="id" value="47">`

```
25_formulario_hidden.html UNREGISTERED 20
<!DOCTYPE html><html>
<head><title>form</title><meta charset="UTF-8"></head>

<body>
<form method="get" action="/hola" >

<input type="hidden" name="id" value="47" /><br>

Su nombre: <br>
<input type="text" name="user" value="nombre" /><br>
<input type="submit" value="Enviar" />
</form>
</body>
</html>
```



Method Override

- ◆ Un formulario solo puede enviar GET y POST al servidor
 - Aplicaciones REST deben enviar también PUT y DELETE
 - ◆ Las técnicas de **method override** (anulación) envían PUT o DELETE
 - Encapsulados en un **parámetro oculto** dentro de una **transacción POST**
- ◆ PUT y DELETE se envían en el parámetro oculto: **_method**
 - **_method=PUT** o **_method=DELETE**
 - ◆ Es una **convención** que indica que **PUT** o **DELETE** se **encapsulan** en **POST**
 - El paquete npm **method-override** de express gestiona este convenio
 - ◆ Doc: <https://www.npmjs.com/package/method-override>
- ◆ PUT y DELETE podrían encapsularse en GET
 - Siempre que no necesiten llevar información en el body
 - ◆ Pero es bastante antinatural y pueden no funcionar con caches mal diseñadas



JavaScript



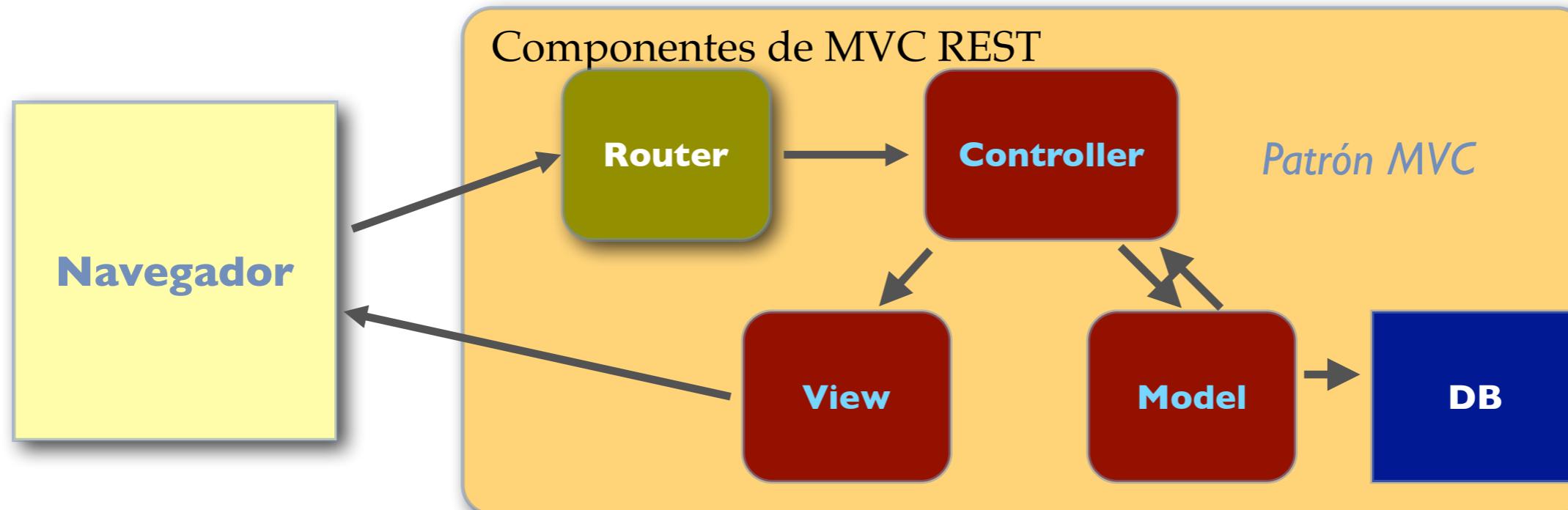
Patrón MVC (Model-Vista-Controller)

Juan Quemada, DIT - UPM

Patrón MVC (Adaptado a Web)

MVC: propuesto por **Trygve Reenskaug** en 1979 para estructurar aplicaciones basadas en ventanas

- ◆ **Modelo:** Datos de la aplicación
 - En una BD, en un fichero, ...
- ◆ **Vista:** Página Web con parámetros
 - Permite generar la página Web enviada al cliente
- ◆ **Controlador:** lógica de la aplicación
 - Acciones asociadas a las rutas de la interfaz REST
- ◆ **Router:** asocia rutas con acciones del controlador



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
 - **Administrador:** administrador de sistemas en la nube (técnico)
 - **Diseñador de experiencia:** usabilidad, interacción, marca, (mixto)
 - **SEO (Search Engine Optimizatióñ):** posiciona portal en buscadores (Google, ...)
 - **Community Manager:** gestiona la relación con los usuarios
 -

Creación de la aplicación express.

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

// CONTROLADORES

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

// RUTAS

```
app.get(['/', '/index'], indexController); // router
app.get('/form', formController);
app.get('/result', resultController);
app.get('*', () => 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>`
```

Estilos utilizados por las vistas.

// VISTAS

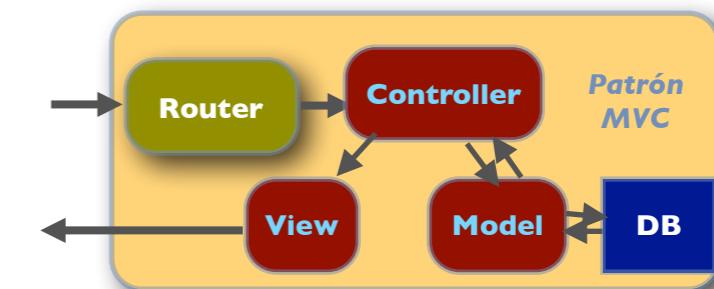
```
const vista_index = `<html> ..... </html>`;
const vista_form = `<html> ..... </html>`;
const vista_result = `<html> ..... </html>`
```

// Arranque del servidor en puerto 8000

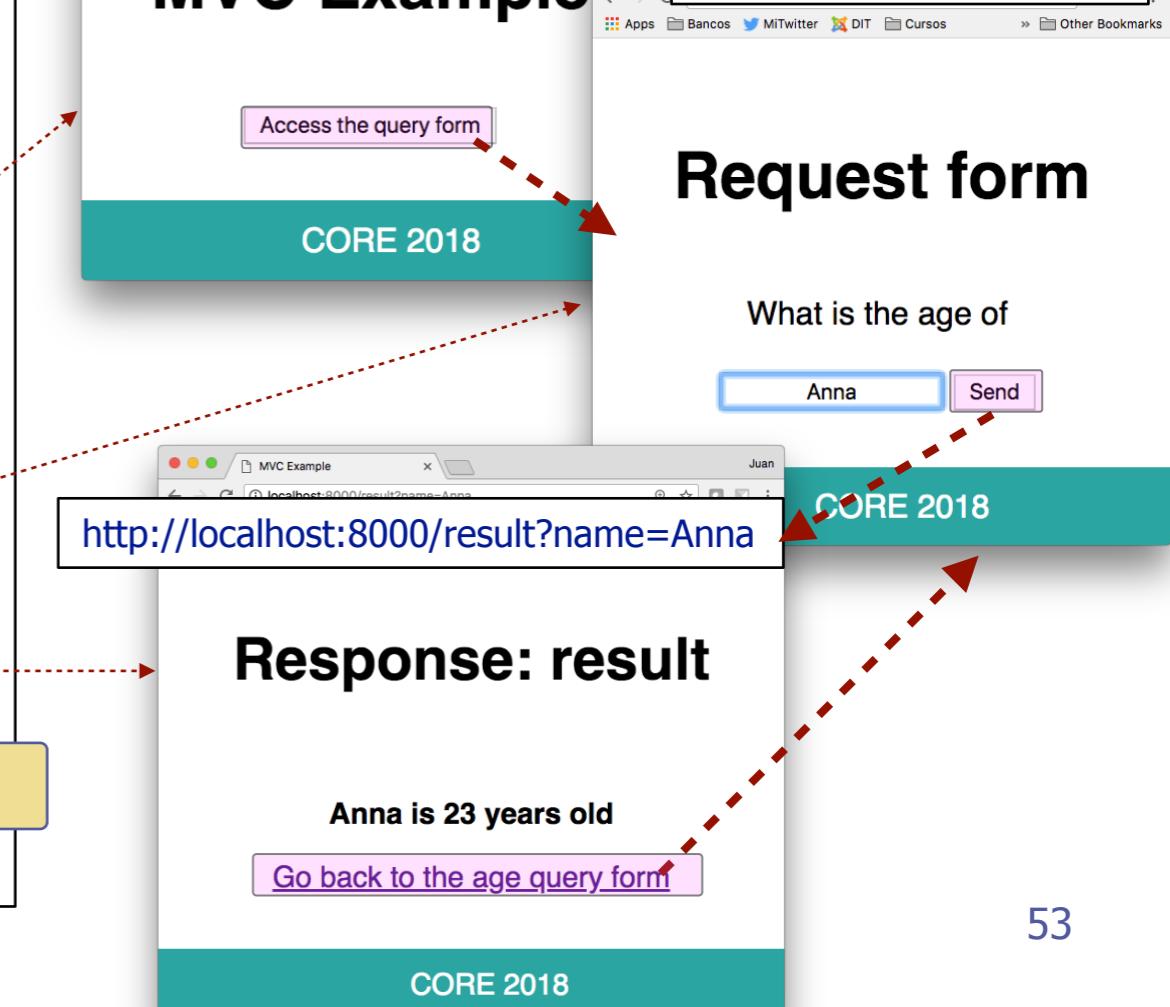
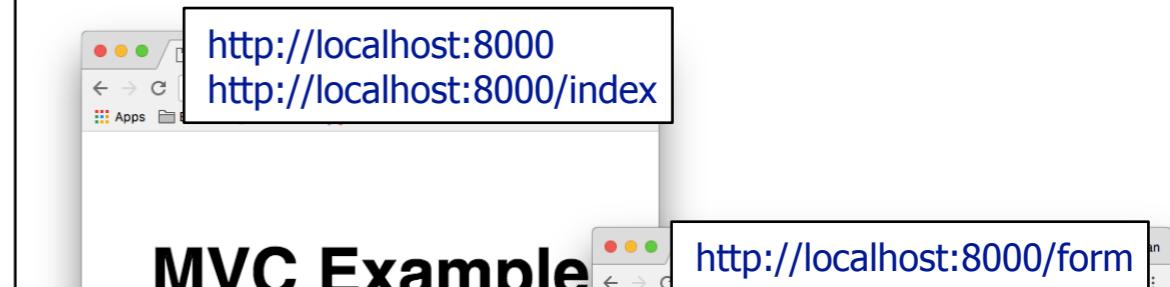
```
app.listen(8000);
```

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

MVC: Ej. Person I



Router de solicitudes HTTP que asocian las rutas de la interfaz REST con las acciones del controlador.



```

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

// CONTROLADORES

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

const formController = (req, res, next) => {
  res.send(vista_form);
}

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(vista_result.replace("<% response %>", response));
};

// RUTAS

app.get('/', '/index', indexController); // router
app.get('/form', formController);
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}
];

```

Estos controladores envían vistas sin parámetros.

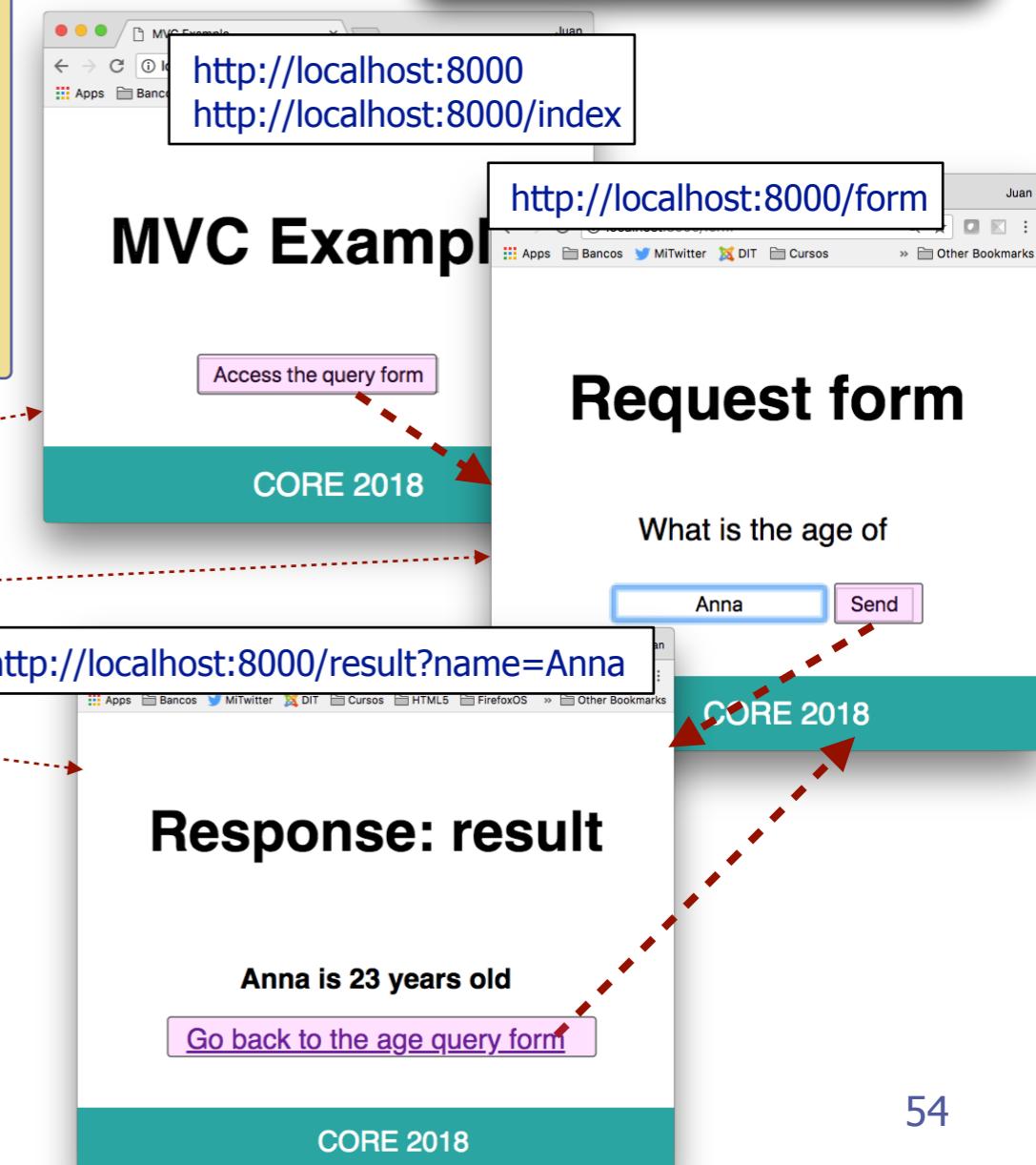
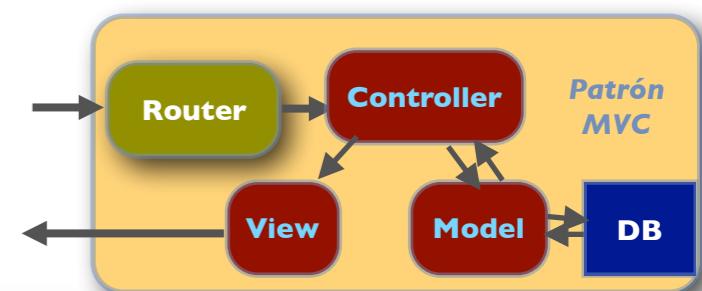
resultController consulta en el modelo la edad del nombre enviado, genera la respuesta (result) y la inserta en la vista que envía al cliente.

El router asocia las **rutas** atendidas por la aplicación (interfaz REST) con las acciones del controlador.

El modelo contiene los datos.

MVC: Ej. Person II

Cada acción del controlador es un MW express que es invocado por el router cuando llega una transacción HTTP con la ruta asociada. El MW consulta el modelo (si necesita datos) y envía la respuesta al cliente.



MVC: Ej. Person III

```
const style = `<style>
*{text-align: center;}
body{padding: 20px; font-family: sans-serif; height: 80%; display: flex; flex-direction: column}
footer{position: fixed;bottom: 0;left: 0; width: 100%; padding: 10px; background: #2aa8a2; color: white;}
h1{margin-bottom: 40px;}`
```

El código CSS es común a todas las vistas y se define como una variable que se sustituye en cada vista.

```
// VISTAS
const vista_index = `<!-- HTML view --&gt;
&lt;html&gt;
  &lt;head&gt;&lt;title&gt;MVC Example&lt;/title&gt;&lt;meta charset="utf-8"&gt;${style}&lt;/head&gt;
  &lt;body&gt;
    &lt;h1&gt;MVC Example&lt;/h1&gt;
    &lt;div id='msg'&gt;&lt;button onclick="window.location.href='/form'"&gt;Access the query form&lt;/button&gt;&lt;/div&gt;
    &lt;footer&gt;CORE 2018&lt;/footer&gt;
  &lt;/body&gt;
&lt;/html&gt;`</pre>

```

```
const vista_form = `<!-- HTML view --&gt;
&lt;html&gt;
  &lt;head&gt;&lt;title&gt;MVC Example&lt;/title&gt;&lt;meta charset="utf-8"&gt;${style}&lt;/head&gt;
  &lt;body&gt;
    &lt;h1&gt;Request form&lt;/h1&gt;
    &lt;form method="get" action="/result"&gt;
      What is the age of &lt;br&gt;&lt;br&gt;
      &lt;input type="text" name="name" placeholder="Name" /&gt;
      &lt;input type="submit" value="Send"/&gt; &lt;br&gt;
    &lt;/form&gt;
    &lt;footer&gt;CORE 2018&lt;/footer&gt;
  &lt;/body&gt;
&lt;/html&gt;`</pre>

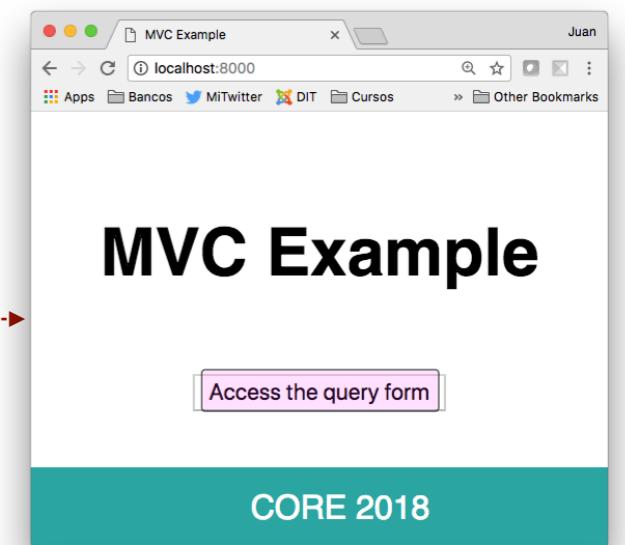
```

Las vistas contienen el código HTML que se envía como respuesta a una solicitud HTTP.
Las vistas son strings que contienen HTML.

```
const vista_result = `<!-- HTML view --&gt;
&lt;html&gt;
  &lt;head&gt;&lt;title&gt;MVC Example&lt;/title&gt;&lt;meta charset="utf-8"&gt;${style}&lt;/head&gt;
  &lt;body&gt;
    &lt;h1&gt;Response: result&lt;/h1&gt;
    &lt;p id='answer'&gt;&lt;strong&gt;${response}&lt;/strong&gt;&lt;/p&gt;
    &lt;a href="/form"&gt;Go back to the age query form&lt;/a&gt;
    &lt;footer&gt;CORE 2018&lt;/footer&gt;
  &lt;/body&gt;
&lt;/html&gt;`</pre>

```

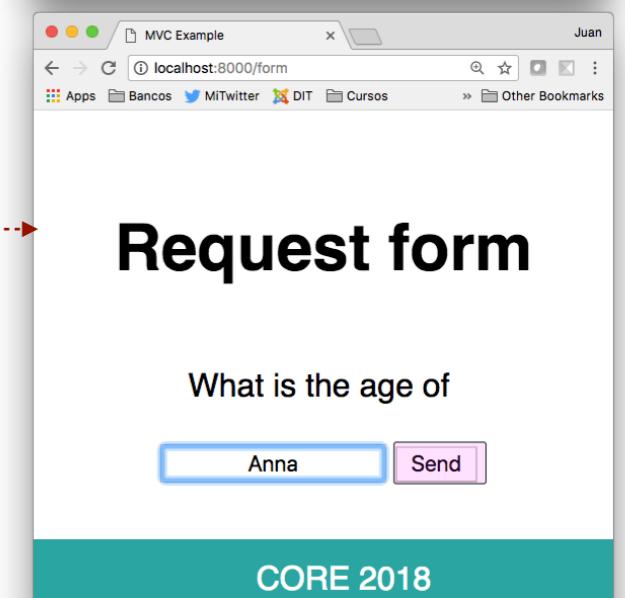
La vista **vista_result** tiene el parámetro **<% response %>** que se sustituirá por la respuesta que genere el controlador.



MVC Example

Access the query form

CORE 2018

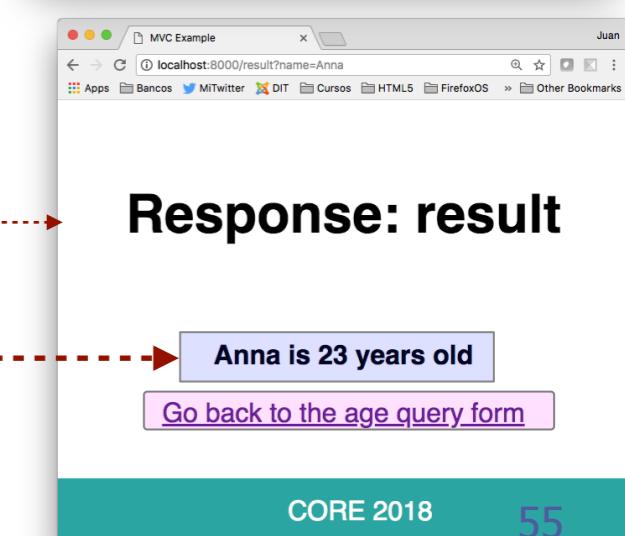


Request form

What is the age of

Anna Send

CORE 2018



Response: result

Anna is 23 years old

Go back to the age query form

CORE 2018



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);
```

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.

Arrancar el servidor en el puerto 8000.

<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>

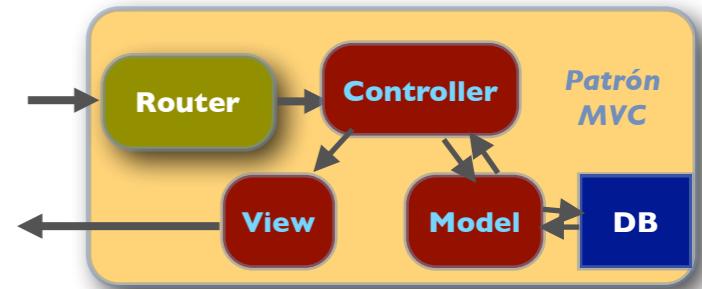
Response: result

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;

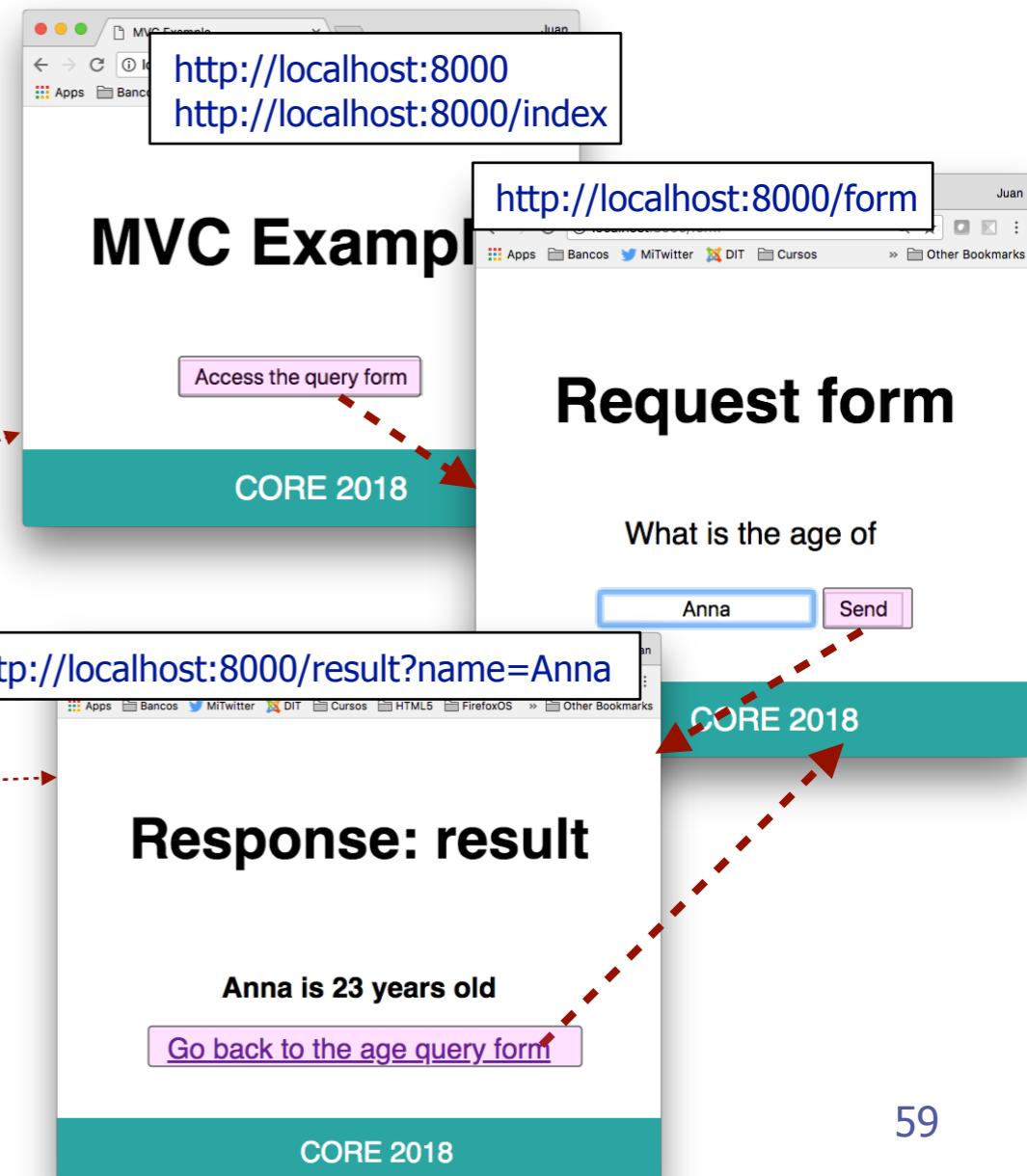
  let person = model.find(p => p.name === name);
  if (person) {
    response = name + " is " + person.age + " years old";
  } else {
    response = name + " is not in our DB";
  }

  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.



El router define las rutas atendidas por la aplicación.

El modelo contiene los datos.

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>`
```

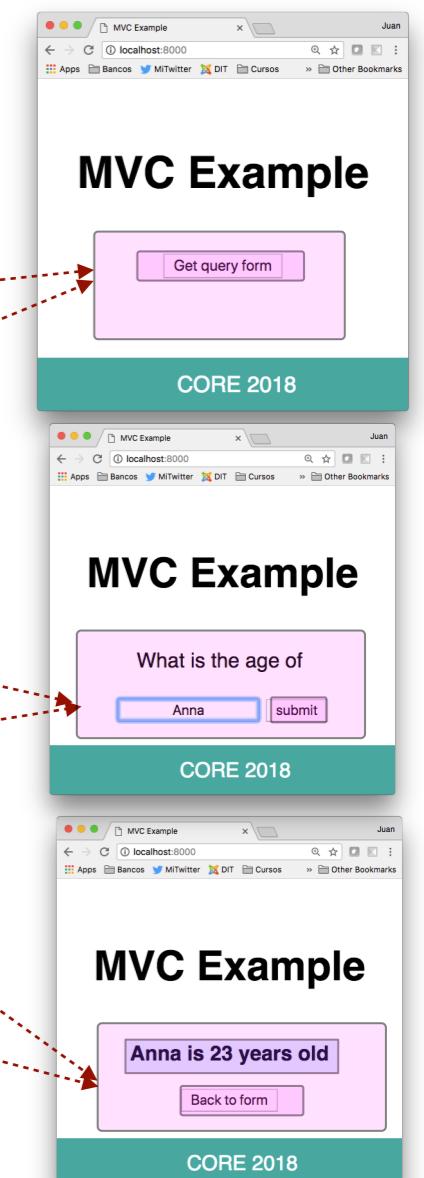
Evento hacer **click en el mensaje** (#msg): muestra formulario y oculta mensaje.

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.

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>
  <body>
    <h1>MVC Example:</h1>
    <div id='msg'><button>Get query form</button></div>
    <div id='form'>
      What is the age of: <br>
      <input type="text" id="name" placeholder="Name" />
      <button id="submit">submit</button>
    </div>
  </body>
</html>`
```

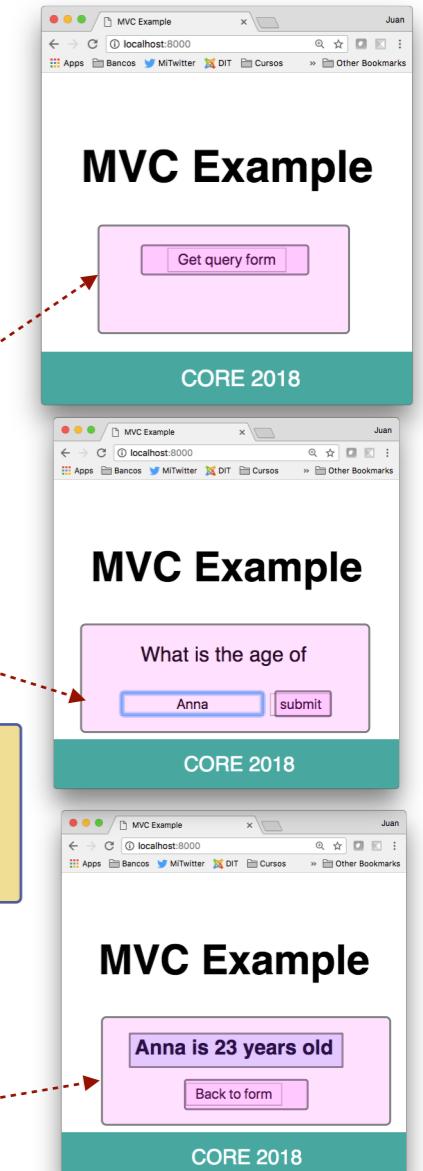
Evento hacer **click en el mensaje** (#msg): muestra formulario y oculta mensaje.

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.



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