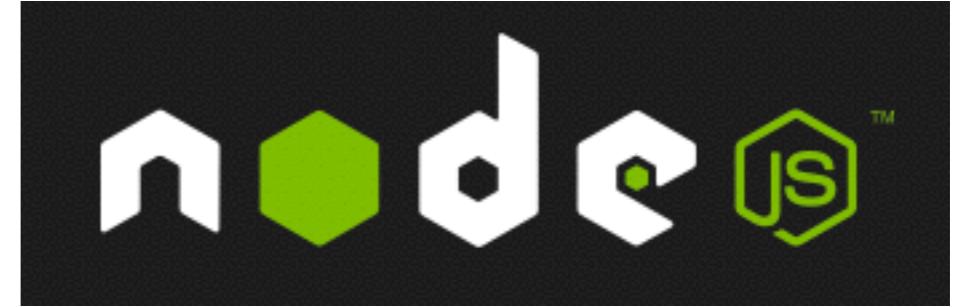


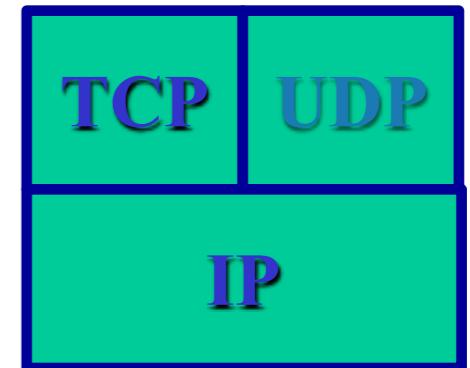


JavaScript



# Introducción a HTTP

Juan Quemada, DIT - UPM



# Creadores, servidores y la nube

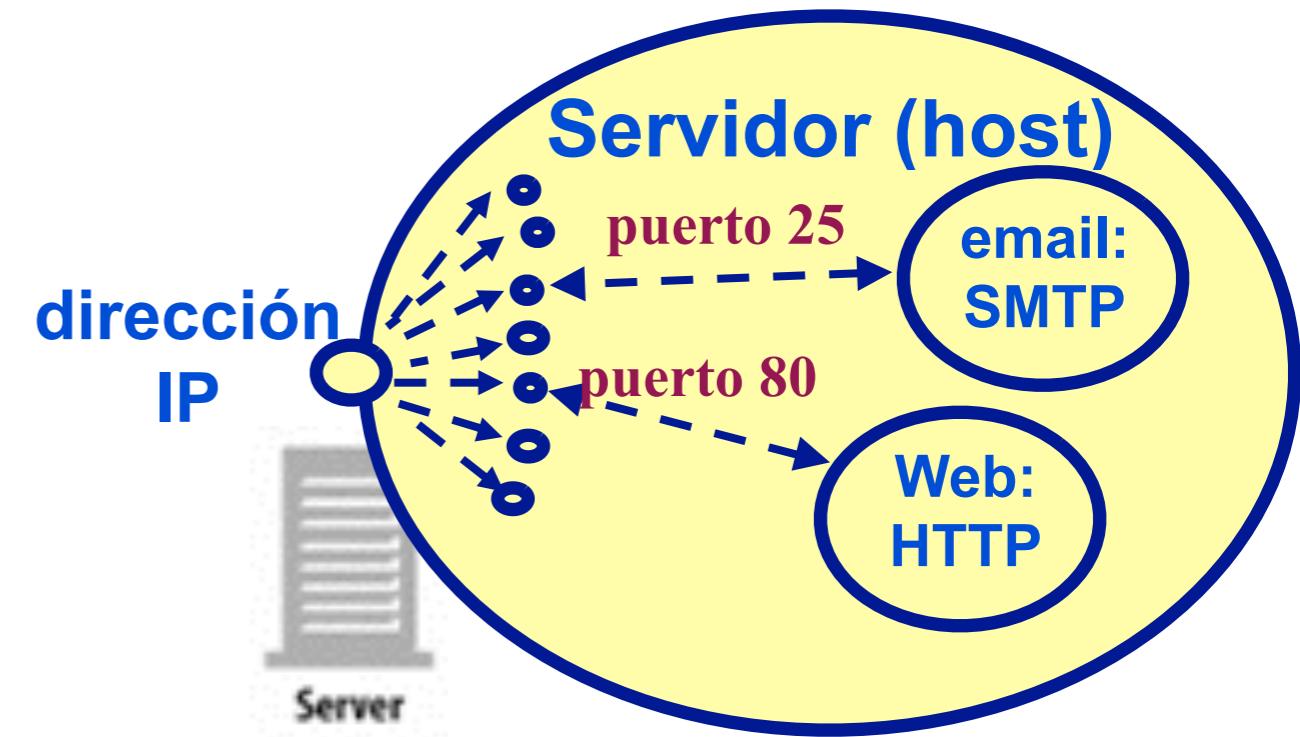
## ◆ Internet

- **Creadores:** dan acceso a los usuarios a la información y los servicios
- **Servidores:** alojan la información y los servicios
- **La nube:** conjunto de terminales y servidores
  - ◆ interconectados con las aplicaciones y protocolos de Internet

## ◆ TCP/IP

- protocolos para crear aplicaciones sobre redes heterogéneas
  - ◆ IP (Internet Protocol): protocolo de interconexión de redes heterogéneas
    - Protocolo del cual hereda su nombre Internet
    - Cada elemento de Internet tiene una dirección IP diferente: **112.9.0.144, ...**
  - ◆ **TCP y UDP:** protocolos de transporte de información
    - Las **mayoría de las aplicaciones** de Internet usan el **servicio TCP**
      - Conectando clientes y servidores con **circuitos virtuales**
    - **Voz y video** sobre IP usan el **servicio UDP**

# Servidores y puertos



## ◆ Puerto (TCP)

- Dirección de aplicación de 16 bits dentro de la máquina servidora
  - ◆ Donde se instala el **programa servidor**
- **El programa servidor** es lo que normalmente denominamos **servidor**
  - ◆ Cliente y servidor se comunican a través de circuitos virtuales TCP (son fiables)
  - Cliente y servidor se comunican con un protocolo de aplicación: HTTP, SMTP, .....

## ◆ Los servicios tienen un protocolo y un puerto por defecto

- **Web:** protocolo HTTP (puerto 80), HTTPS (443)
- **Email:** protocolo SMTP (puerto 25), POP3 (110), IMAP143
- **Shell segura:** protocolo SSH (puerto 22)

## ◆ Si un servidor no está en el puerto por defecto

- Su dirección debe incluir el puerto, p.e. [dit.upm.es:8080](http://dit.upm.es:8080), [upm.es:8000](http://upm.es:8000)

# URIs y URLs

Formato de un URL:

<schema:><//><authority></path><?query><#anchor>

<schema:> = protocolo o esquema de acceso al recurso  
<authority> = <UserInfo@><host><:port>  
</path> = identificador del recurso en el servidor  
<?query> = parámetros enviados al recurso  
<#anchor> = fragmento o parte del recurso

## ◆ URI - Uniform Resource Identifier

- identificador de un recurso (servicio) de Internet
  - ◆ RFC 3986 (2005): <http://tools.ietf.org/html/rfc3986>

## ◆ Existen 2 tipos de URI

- URL - Uniform Resource Locator
  - ◆ Dirección física de un recurso, incluyendo el servidor donde se almacena
    - [http://es.wikipedia.org/wiki/Localizador\\_de\\_recursos\\_uniforme](http://es.wikipedia.org/wiki/Localizador_de_recursos_uniforme)
- URN: Uniform Resource Name
  - ◆ Dirección lógica independiente de lugar físico (poco utilizado)

## ◆ Ejemplos de URL de email, página Web, ftp, .....

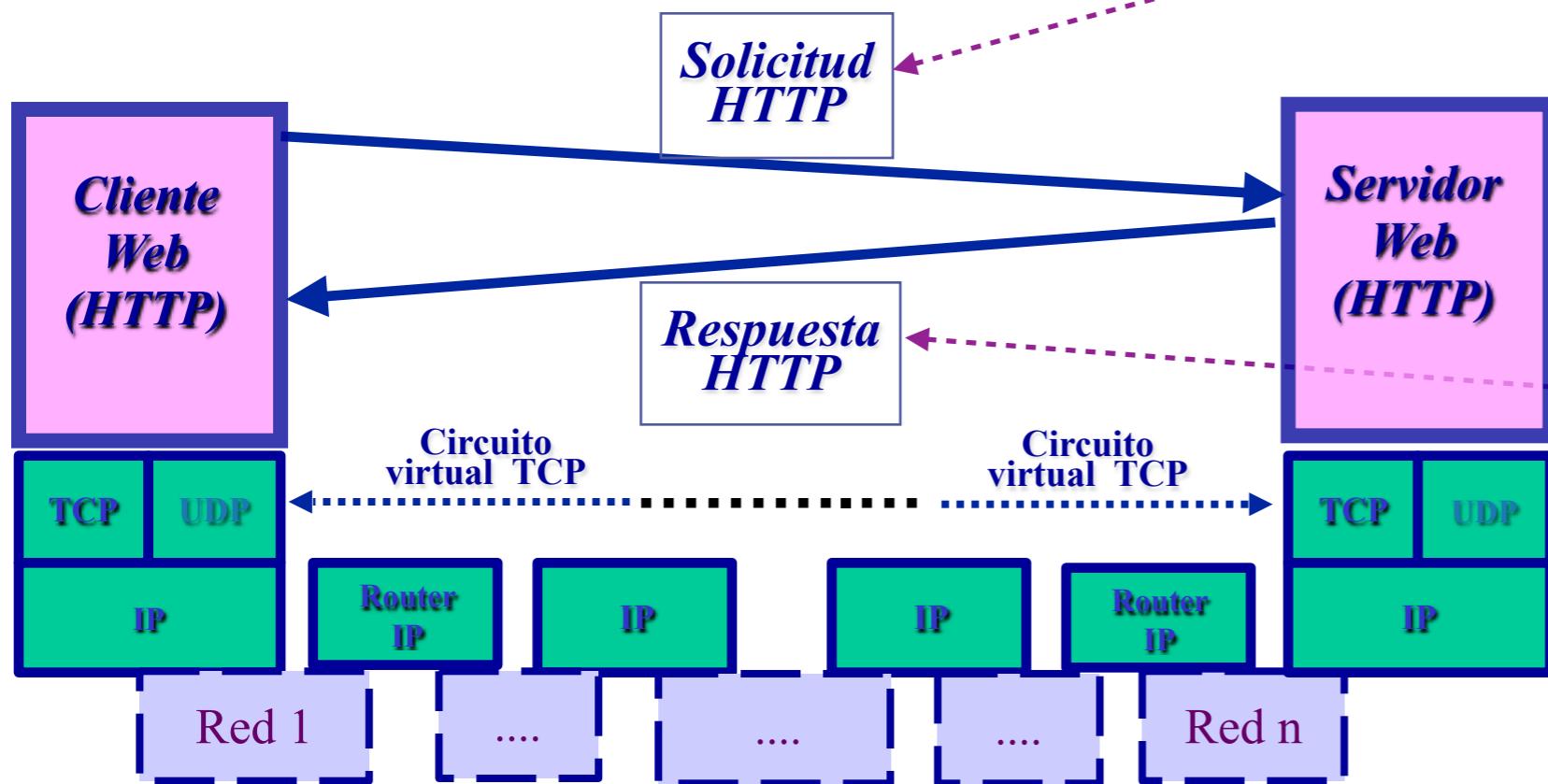
- URL Web: <http://etsit.upm.es/anuncios/index.html>
- URL email: <mailto:jose@dit.upm.es>
- URL ftp: <ftp://ftp.funet.fi/pub/standards/RFC/rfc959.txt>

# HTTP



- ◆ Protocolo transaccional de **acceso a recursos**
  - Transacción HTTP se compone de **solicitud y respuesta**
    - ◆ Método HTTP: tipo de solicitud HTTP (GET, POST, PUT, DELETE, .... )
- ◆ Solicitud y respuesta son **mensajes independientes** (como emails)
- ◆ HTTP no necesita **conectividad TCP extremo a extremo**
  - Permite el uso de proxies, caches, firewalls, ...
- ◆ HTTP es un protocolo **extensible** que ha evolucionado mucho
  - HTTP 0.9, 1.0, 1.1 (1999, 2014,...), extensiones WebDav, ....
    - ◆ Resumen: [http://es.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol)
    - ◆ Norma - RFCs 7230-37: <https://datatracker.ietf.org/doc/rfc7230/>
    - ◆ <http://www8.org/w8-papers/5c-protocols/key/key.html>, <http://www.jmarshall.com/easy/http/>

# Transacción HTTP



*Solicitud HTTP GET*

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

*Respuesta HTTP GET*

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

- ◆ **Navegador** envía solicitud y **servidor** devuelve respuesta, p.e.
  - **Solicitud:** solicitud GET de un fichero o recurso
  - **Respuesta:** respuesta del servidor enviando el fichero o recurso solicitado

- ◆ **Mensajes** (solicitud o respuesta): constan de **Cabecera** y **Cuerpo**.
- ◆ **Cabecera:** string formado por **1a linea** y **parámetros**. Acaba con una línea en blanco (\n\n).
  - **Primera línea de la solicitud:** incluye el **método**, la **ruta o path** que identifica el recurso en el servidor y la **versión** de HTTP utilizada por el cliente.
  - **Primera línea de la respuesta:** incluye **versión** HTTP del servidor, **código** y un **mensaje** de texto explicativo de la respuesta.
  - **Parámetros de la cabecera:** Cada parámetro es un string con el formato: **Nombre: valor**, que ocupa una línea (acaba con \n).
- ◆ **Cuerpo:** incluye el recurso enviado que puede ser de cualquier tipo, p.e. string, imagen, ....

|                  |                               |   |   |
|------------------|-------------------------------|---|---|
| <b>Solicitud</b> | <b>1a linea</b>               | <b>GET /dir/me.htm HTTP/1.1\n</b>   | <b>Método GET, recurso, versión-HTTP 1.1</b>  |
|                  | <b>Parámetros de cabecera</b> | <b>Host: upm.es\n</b><br><b>Accept: text/*, image/*\n</b><br><b>Accept-language: en, sp\n</b><br>.....<br><b>User-Agent: Mozilla/5.0\n</b><br><b>\n</b> | <b>Host:</b> identifica el servidor, se incluye porque el circuito TCP no es extremo a extremo<br><b>Accept:</b> tipos MIME de recursos aceptados<br><b>Accept-language:</b> lenguajes del cliente<br>Acaba con linea en blanco: \n\n |
|                  | <b>Cuerpo</b>                 |   | <b>GET: NO incluye cuerpo en la solicitud</b>   |
| <b>Respuesta</b> | <b>1a linea</b>               | <b>HTTP/1.1 200 OK\n</b>  | <b>Versión HTTP 1.1, 200 (todo ok), mensaje</b>   |
|                  | <b>Parámetros de cabecera</b> | <b>Server: Apache/1.3.6\n</b><br><b>Content-type: text/html\n</b><br>.....<br><b>Content-length: 608\n</b><br><b>\n</b>                                 | <b>Content-type:</b> tipo MIME de recurso, <b>text/html</b> es el tipo de una página Web<br><b>Content-length:</b> número (decimal) de octetos<br>Acaba con linea en blanco: \n\n   |
|                  | <b>Cuerpo</b>                 | <b>&lt;html&gt; .....</b> <b>&lt;/html&gt;</b>  | <b>página HTML (recurso)</b>  |

## 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”

# Códigos de estado (HTTP status codes)

## ◆ Respuestas informativas (1xx)

- 100 Continue // Continuar solicitud parcial

## ◆ Solicitud finalizada (2xx)

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

## ◆ Redirección (3xx)

- 301 Moved Permanently // Recurso se ha movido, actualizar 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
- 405 Method Not Allowed // Método no permitido
- 409 Conflict // Recurso ya no esta
- 410 Gone // Recurso ya no esta

## ◆ Error de Servidor (5xx)

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

*Respuesta HTTP GET*

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

# Métodos (verbos, comandos) de HTTP

## Interfaz Uniforme o CRUD (bases de datos):

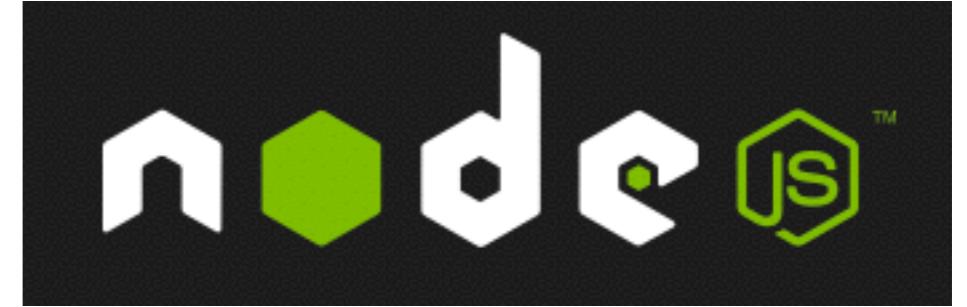
- ◆ **POST:** Crear un recurso en el servidor (Create)
- ◆ **GET:** Pedir un recurso al servidor (Read)
- ◆ **PUT:** Modificar un recurso del servidor (Update)
- ◆ **DELETE:** Borrar un recurso del servidor (Delete)

## más métodos

- ◆ **HEAD:** similar a GET, pero solo pide cabecera al servidor
- ◆ **OPTIONS:** Determinar qué métodos acepta un servidor
- ◆ **TRACE:** Trazar proxies, caches, ... hasta el servidor
- ◆ **CONNECT:** Conectar a un servidor a través de un proxy
- ◆ .....



JavaScript



# Servidor Web

Juan Quemada, DIT - UPM

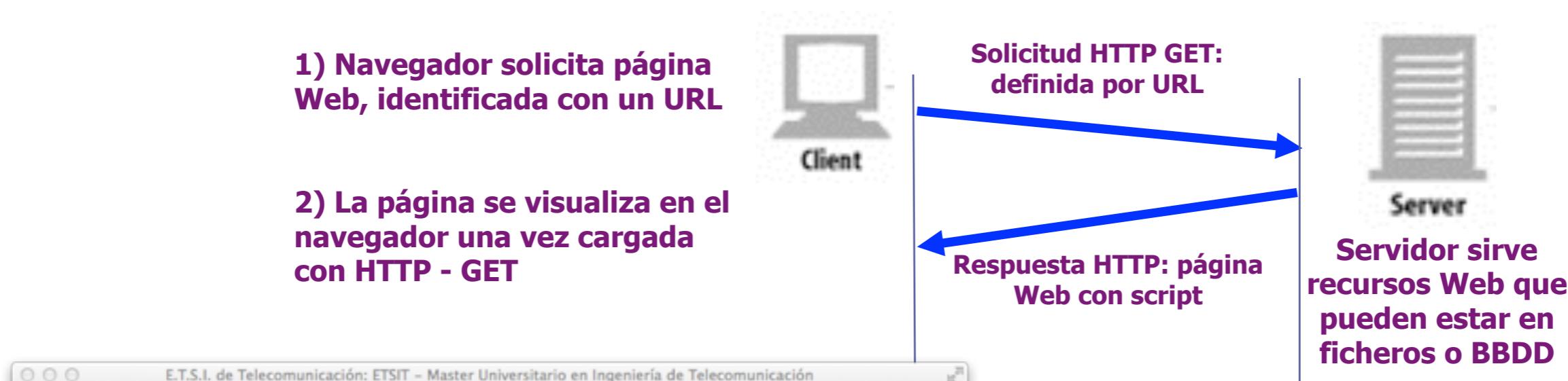
# Servidor Web

## ◆ Programa residente en la máquina servidora

- Sirve recursos Web (páginas) con transacciones HTTP GET
  - ◆ Un recurso se identifica con un **URL**, por ej. <http://etsit.upm.es/anuncios/index.html>
    - El **URL Web** lleva asociado solo el método **GET** del protocolo **HTTP**

## ◆ Los servidores también pueden crear servicios mas complejos

- Tiendas electrónicas, redes sociales, blogs, ...
  - ◆ Se verá en temas posteriores

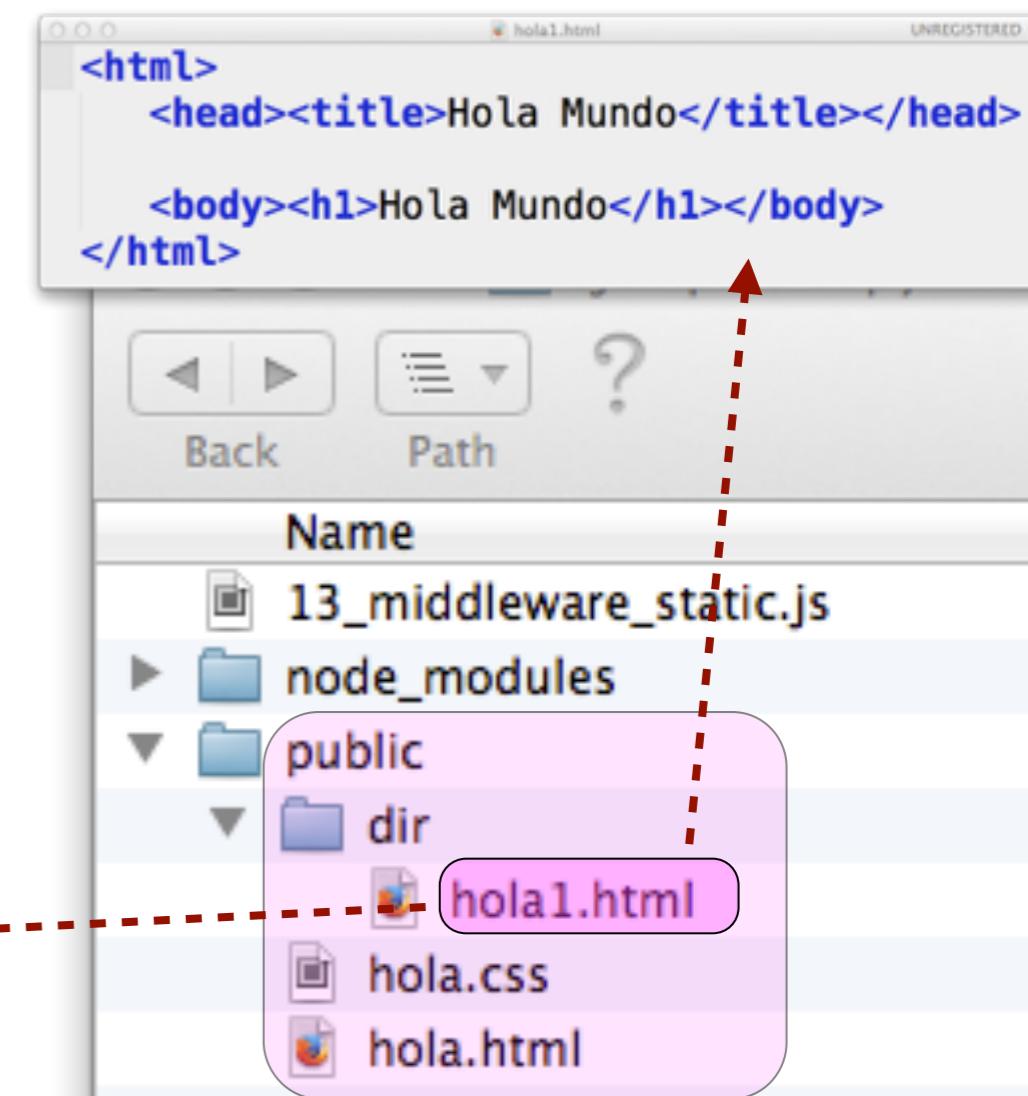
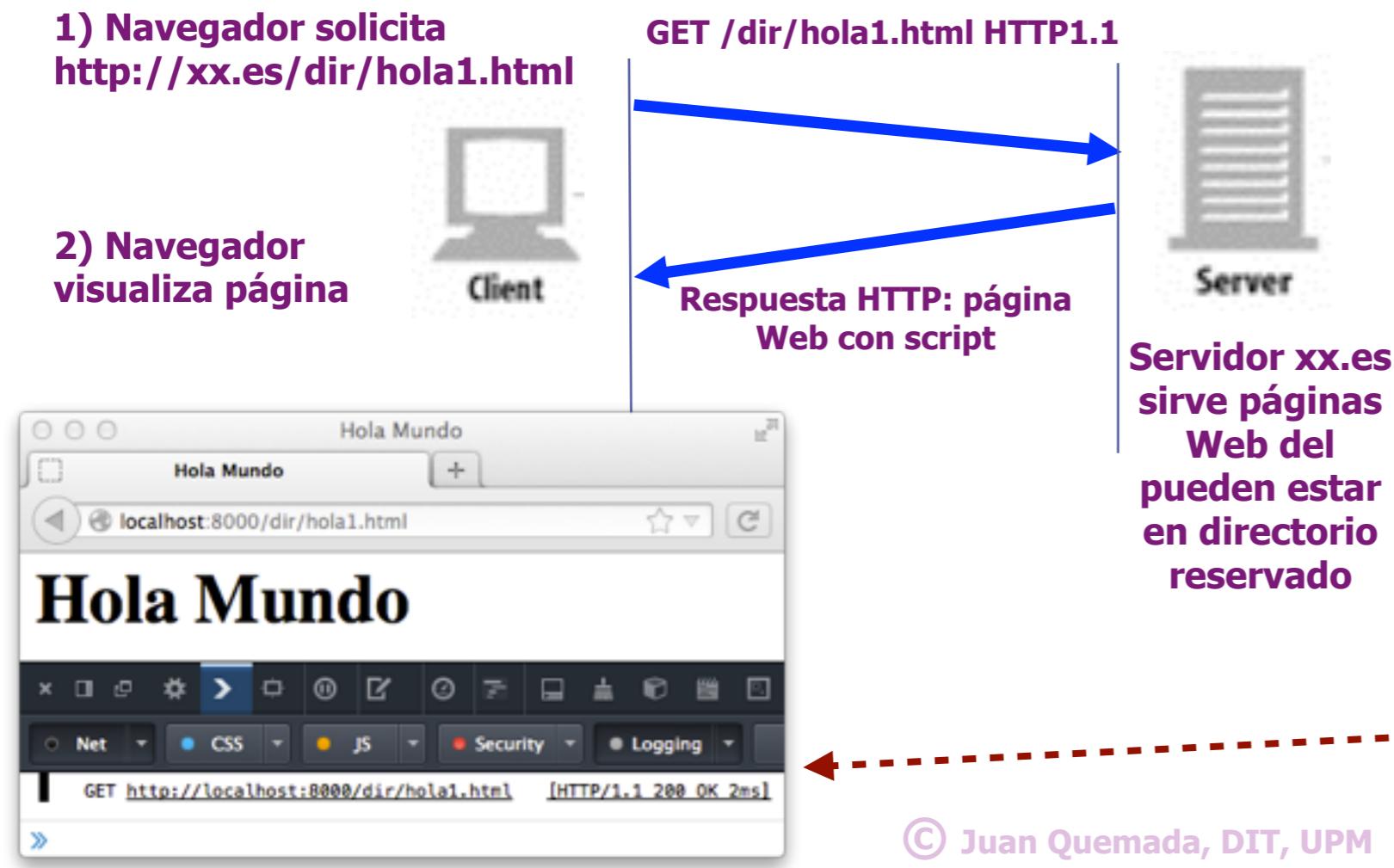


, DIT, UPM

# Repositorio de recursos Web

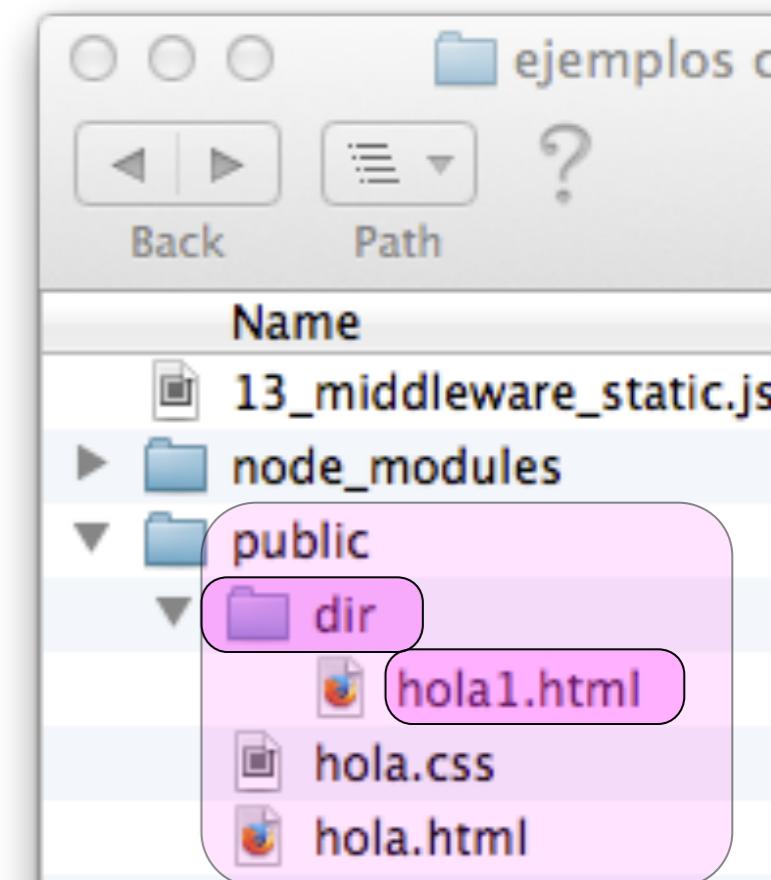
## ◆ Un servidor Web reserva un directorio para recursos Web

- Los ficheros del directorio y sus subdirectorios son accesibles con HTTP GET
- El directorio de recursos (páginas, ..) Web suele denominarse **public**, **www**, ..

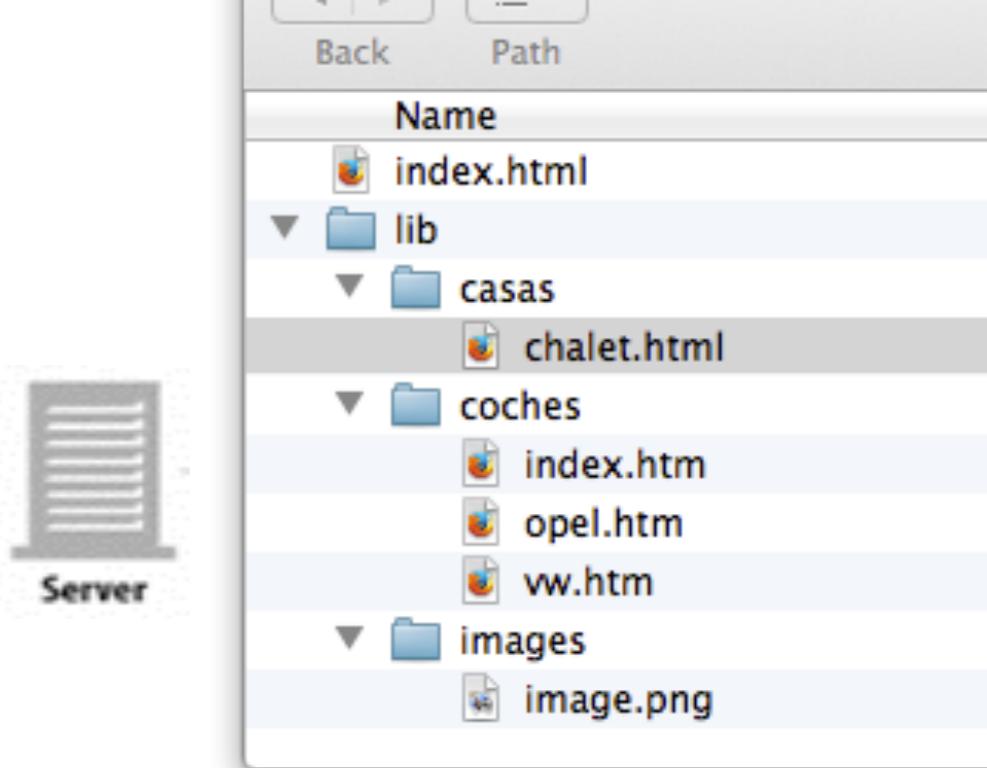


# URL Web

- ◆ **URL Web:** dirección de recurso accesible con HTTP GET
  - EL URL Web solo permite navegar por páginas Web, pero no modificar recursos
- ◆ Componentes básicos del URL: <http://upm.es/dir/hola1.html>
  - **http:** Protocolo o esquema de acceso al recurso (significa HTTP GET)
  - **upm.es:** Dirección del servidor donde reside el recurso
  - **/dir/hola1.html:** Ruta (path o camino) al fichero dentro del servidor
- ◆ Ruta (path)
  - ruta hasta el recurso
    - ◆ desde la raíz del directorio de recursos
- ◆ Los servidores Web suelen configurarse
  - Con el nombre **index.html** o **index.htm** opcional
    - ◆ Es decir, **/dir/index.htm** es equivalente a **/dir/**



# URLs relativos



- ◆ Son relativos al URL (recurso) actual
  - Solo incluyen la ruta (path), el navegador añade host, protocol, ....
    - ◆ Ambos recursos deberán estar en el repositorio del mismo servidor
  
- ◆ Los URLs relativos pueden ser de 2 tipos:
  - **Ruta o path absolutos:** /lib/coches/vw.html
    - ◆ Ruta desde el **directorio raíz del repositorio de recursos** del servidor
  - **Ruta o path relativos:** coches/vw.html, vw.html
    - ◆ Ruta desde el **directorio del recurso actual** en el servidor

# Interacción HTTP con el servidor

## ◆ Cuando el servidor Web recibe una petición HTTP GET

- Envía el recurso en la respuesta con el código “200 OK”, si lo tiene
  - ◆ El parámetro “Content-Type: text/html” contiene el tipo MIME del recurso enviado
- Sino, responde con el mensaje de error correspondiente

## ◆ Si el recurso es un fichero, su extensión determina su tipo MIME

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

## ◆ El navegador interpreta el recurso de acuerdo al tipo MIME recibido

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

# 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)

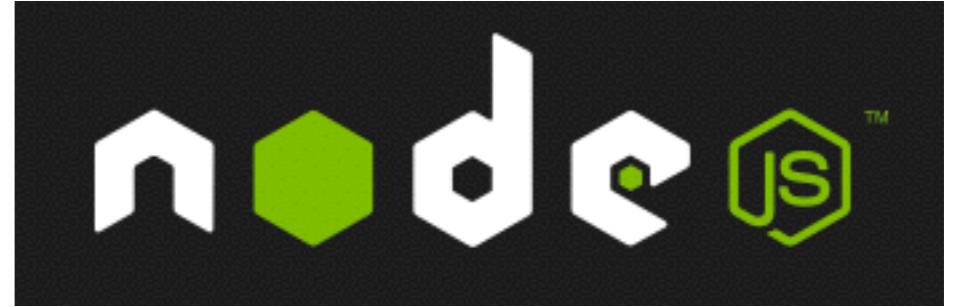
- 400 Bad request // Comando enviado incorrecto
- 404 Not Found // Recurso no encontrado, no hay ningún fichero con ese path
- 405 Method Not Allowed // Método no permitido, p.e. se solicita método POST, PUT, ....
- 409 Conflict // Existe conflicto con el estado del recurso en el servidor
- 410 Gone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

## ◆ Error de Servidor (5xx)

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



JavaScript



# Introducción a express.js y al middleware static

Juan Quemada, DIT - UPM

# node.js y express

# express

## web application framework for node

## ◆ Express: paquete para crear servicios Web

- Accesibles por HTTP desde clientes
    - ◆ Mas información: <http://expressjs.com>

◆ express se basa en node.js y se instala con npm

- desde el **servidor central** en <https://www.npmjs.org>

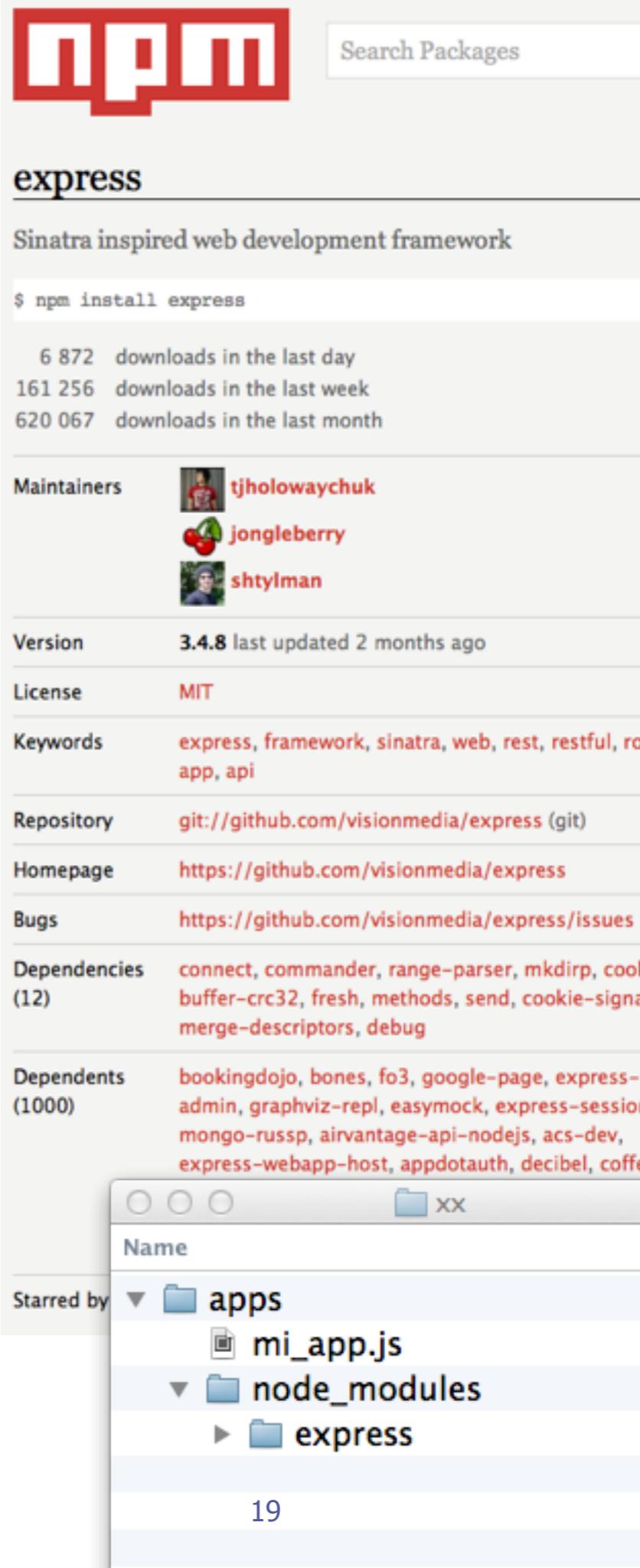
```
..$ mkdir apps // crear un directorio para las apps express
```

```
..$ cd apps // entramos en el directorio de trabajo
```

```
..../apps$ npm install express@4.9.0 // Instala express v4.9.0  
// en el directorio apps/node_modules/express
```

// Editamos una aplicación express, por ejemplo **mi\_app.js** y

```
..../apps$ node mi app.js // la ejecutamos con node
```

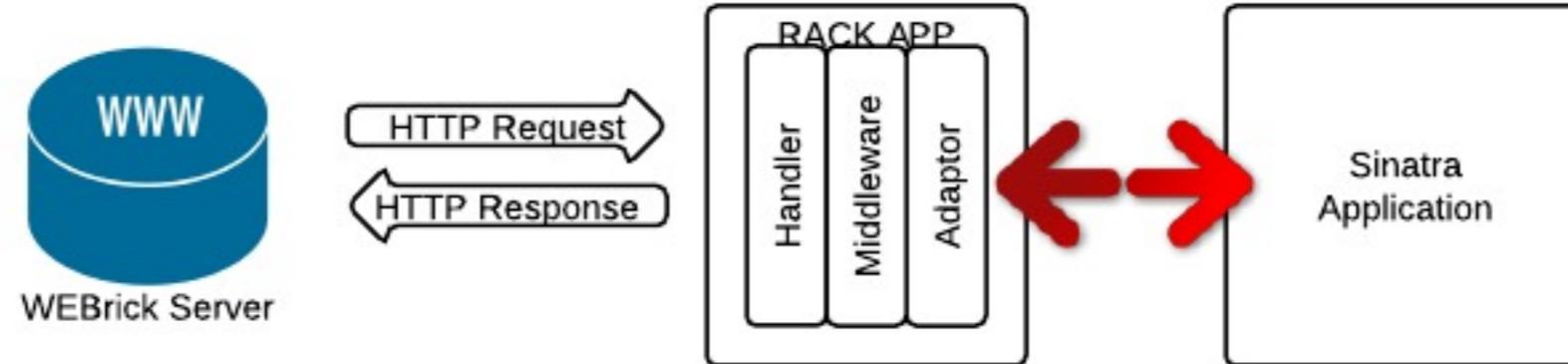


# Middleware (MW) static de express.js

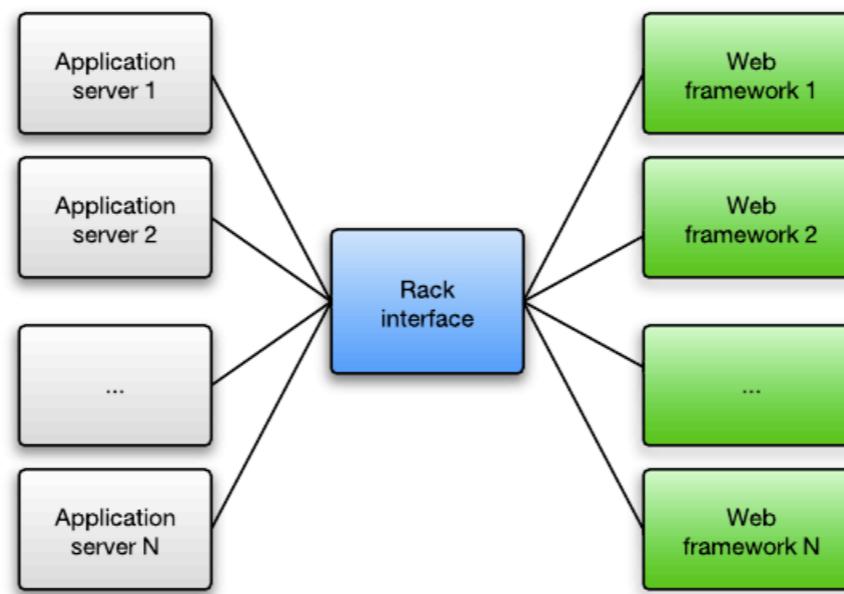
- ◆ Las aplicaciones express se construyen con middlewares (MWs)
  - Un **middleware es una función** que se instala en la aplicación express
- ◆ Un MW se instala en **app** con el método **app.use(MW)**
  - Una vez instalado, un ‘**MW**’ se ejecuta con cada **solicitud HTTP que llega a app**
    - ◆ Doc: <http://expressjs.com/4x/api.html#app.use>
- ◆ **Middleware static:** funcionalidad de un servidor Web estático
  - Sirve páginas Web estáticas, estilos CSS, librerías JavaScript, imágenes, .....
  - ◆ Documentación: <http://expressjs.com/guide/using-middleware.html#middleware.built-in>
- ◆ Express se basa en el modulo connect.js
  - El concepto de middleware lo hereda de connect y puede utilizar sus MWs
    - ◆ <http://expressjs.com/guide/using-middleware.html#middleware.third-party>

Connect es similar a Rack (Ruby)

<http://stackoverflow.com/questions/5284340/what-is-node-js-connect-express-and-middleware>



The fundamental idea behind Rack middleware is – come between the calling client and the server, process the HTTP request before sending it to the server, and processing the HTTP response before returning it to the client.



In the distant past, each Ruby web framework had its own interface, so application servers needed to explicitly add support for each web framework. Nowadays application servers just support Rack.

Express no longer uses Connect. However, Express is still compatible with middleware written for Connect. Express does to Connect what Connect does to the http module:

So all of the functionality of Connect is there, plus view rendering and a handy DSL for describing routes.

Ruby's Sinatra is a good analogy.

Then there are other frameworks that go even further and extend Express!

Zappa, for instance, which integrates support for CoffeeScript, server-side jQuery, and testing.

# Aplicación express.js

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

// Middleware de acceso a páginas Web estáticas
// -> root = directorio 'public'
// -> __dirname: nombre del directorio de ejecución
app.use(express.static(path.join(__dirname, 'public')));

app.listen(8000);
```

The code editor window shows a file named 13\_middleware\_static.js. The code itself is:

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

// Middleware de acceso a páginas Web estáticas
// -> root = directorio 'public'
// -> __dirname: nombre del directorio de ejecución
app.use(express.static(path.join(__dirname, 'public')));

app.listen(8000);
```

Annotations with dashed red arrows point to specific parts of the code:

- An arrow points from the text "Una aplicación express es un programa en JavaScript que responde a solicitudes HTTP." to the line `var express = require('express');`.
- An arrow points from the text "El programa carga en primer lugar el módulo express en una variable." to the line `var express = require('express');`.
- An arrow points from the text "A continuación crea el objeto servidor invocando el modulo cargado como una función que devuelve el objeto (este patrón se denomina una factoría de objetos)" to the line `var app = express();`.
- An arrow points from the text "Por último, se arranca el servidor en el puerto 8000. Así responderá a las solicitudes HTTP que llegan a dicho puerto." to the line `app.listen(8000);`.

Una aplicación express es un programa en JavaScript que responde a solicitudes HTTP.

El programa carga en primer lugar el **módulo express** en una variable.

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

A continuación **crea el objeto servidor** invocando el modulo cargado como una función que devuelve el objeto (este patrón se denomina una factoría de objetos)

**var app = express();**

Por último, **se arranca el servidor en el puerto 8000**. Así responderá a las solicitudes HTTP que llegan a dicho puerto.

**app.listen(8000);**

# Middleware static

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

// Middleware de acceso a páginas Web estáticas
// -> root = directorio 'public'
// -> __dirname: nombre del directorio de ejecución
app.use(express.static(path.join(__dirname, 'public')));

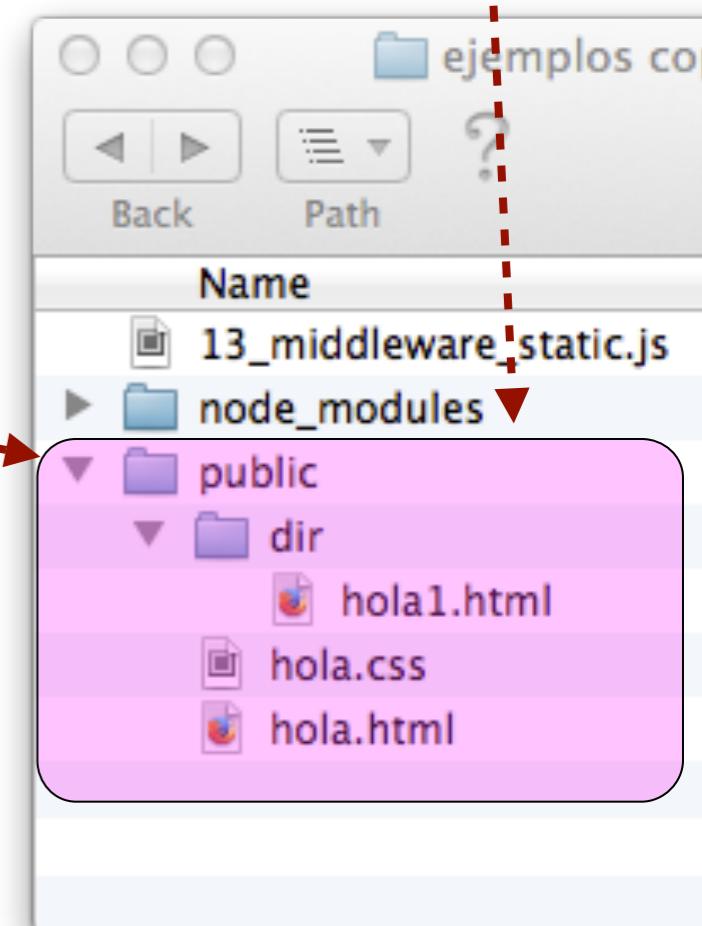
app.listen(8000);
```

Un middleware se instala invocando el método **use()** en **app**. Una vez instalado, se ejecuta cada vez que llega una solicitud HTTP.

**app.use(express.static(path.join(\_\_dirname, 'public')))** instala el middleware con el servidor de páginas Web estáticas, incluido en **express**. Las páginas deben alojarse en el **directorio 'public'**.  
Mas info en: <http://expressjs.com/4x/api.html#middleware.builtin>

**path** es un módulo de **node.js** de gestión de rutas. Y **path.join(..)** concatena dos rutas. Mas info: <http://nodejs.org/api/path.html>

**\_\_dirname**: ruta (path) al directorio de ejecución de la aplicación.  
Mas info en: <http://nodejs.org/api/globals.html>



# Como funciona el MW static

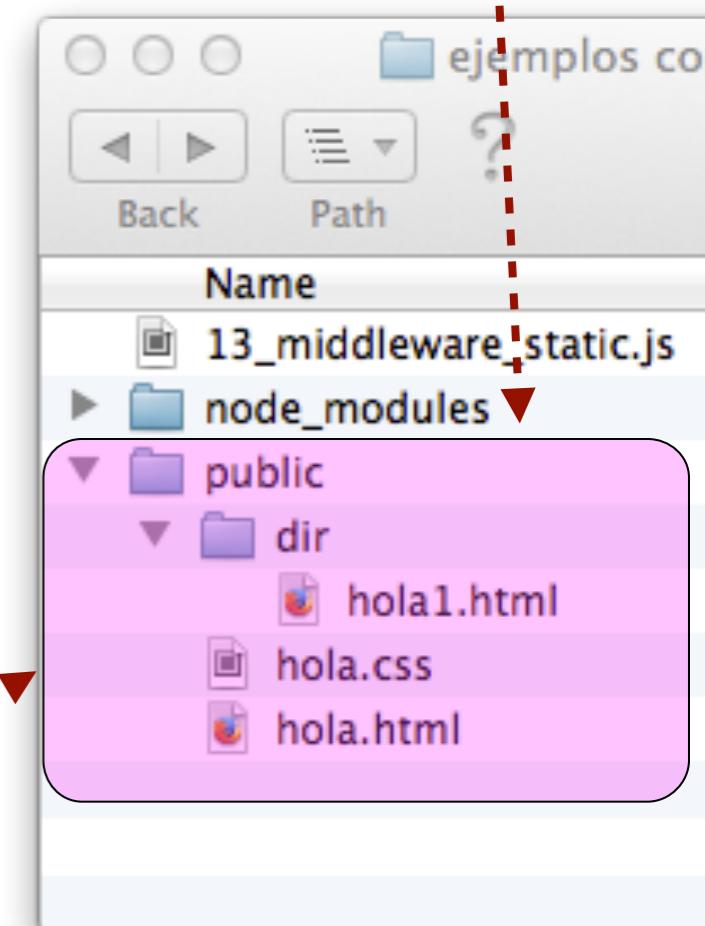
```
var express = require('express');
var path = require('path');
var app = express();

// Middleware de acceso a páginas Web estáticas
// -> root = directorio 'public'
// -> __dirname: nombre del directorio de ejecución
app.use(express.static(path.join(__dirname, 'public')));

app.listen(8000);
```

El **middleware static**, igual que el resto de middlewares de la aplicación express, se ejecuta al llegar una solicitud HTTP al servidor en el puerto 8000. El middleware static analiza el método, la ruta (path) y los parámetros de la solicitud HTTP y responde de la siguiente forma:

- 1) Solo **acepta solicitudes de tipo GET y rechaza el resto** (POST, PUT, DELETE, ....) con **405 METHOD NOT SUPPORTED**.
- 2) Si la **ruta (path)** **referencia un fichero existente** en el repositorio de recursos Web, lo devuelve con **200 OK** y parámetros asociados.
- 3) La respuesta al **resto de solicitudes GET** incluye el código de error correspondiente: **404 NOT FOUND** (recurso no existe), **500 INTERNAL SERVER ERROR** (error de servidor), ....



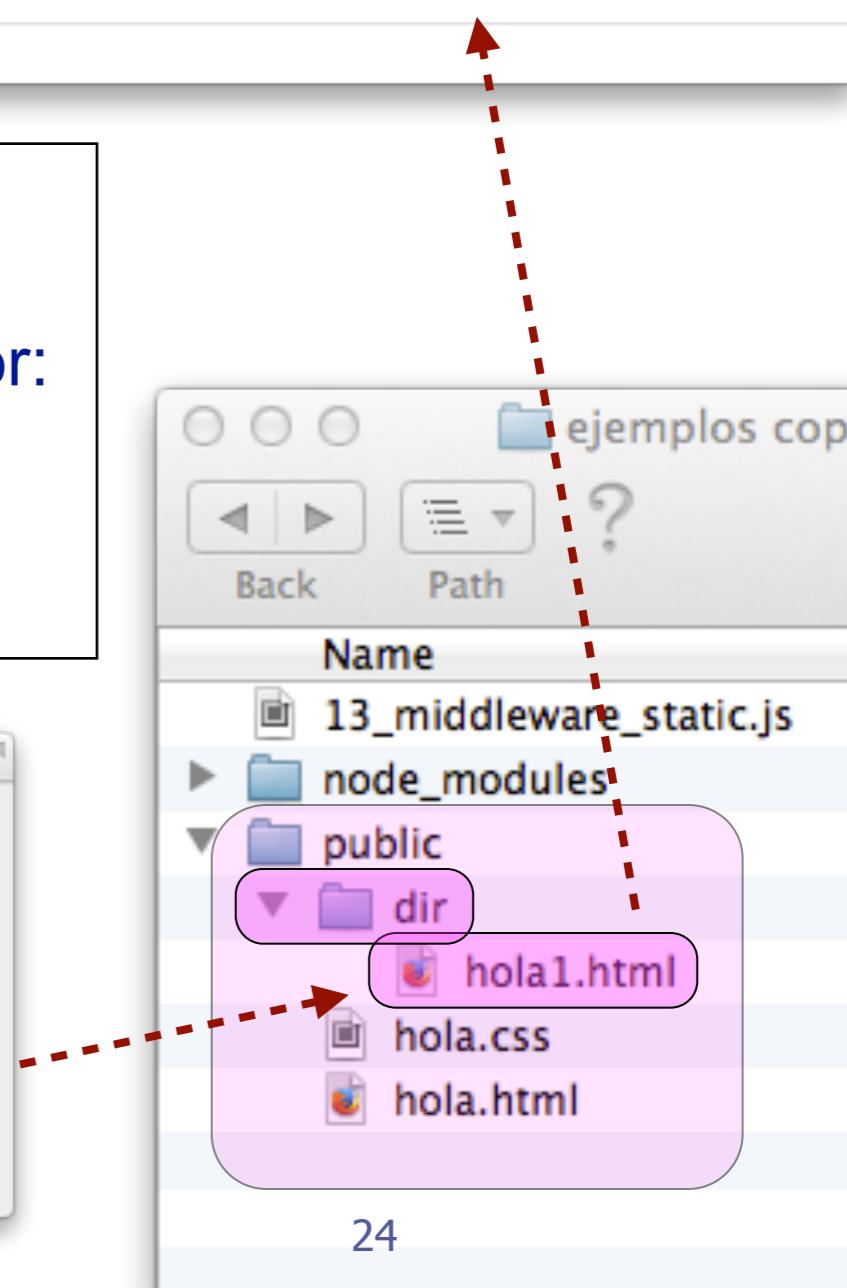
# Página dir/hola1.html

La captura del navegador Firefox con la consola abierta y con red (**Net**) seleccionado, muestra la transacción GET realizado para traer al navegador la página identificada por:

<http://localhost:8000/dir/hola1.html>

```
<html>
  <head><title>Hola Mundo</title></head>

  <body><h1>Hola Mundo</h1></body>
</html>
```



# Cabecera express.js

La consola Firefox permite ver los parámetros de la solicitud y la respuesta al clicar encima.

- **Status 200 OK:** respuesta incluye recurso
- **Content-Type: text/html; charset=UTF-8**
  - > indica página **HTML en UTF-8**
- **Content-Length: 95**
  - > indica que el cuerpo lleva **95 octetos**

Los demás parámetros se ven más adelante.

Hola Mundo

Hola Mundo

localhost:8000/dir/hola1.html

Net CSS JS Security Logging

GET http://localhost:8000/dir/hola1.html [HTTP/1.1 200 OK 2ms]

http://localhost:8000/dir/hola1.html

Inspect Network Request

Request URL: http://localhost:8000/dir/hola1.html  
Request Method: GET  
Status Code: HTTP/1.1 200 OK

19:55:07.000

Request Headers

User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:28.0) Gecko/20100101 Firefox/28.0  
If-None-Match: "94-1397856398000"  
If-Modified-Since: Fri, 18 Apr 2014 21:26:38 GMT  
Host: localhost:8000  
Connection: keep-alive  
Cache-Control: max-age=0  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8

Response Headers

X-Powered-By: Express  
Last-Modified: Sun, 20 Apr 2014 17:55:04 GMT  
Etag: "94-1398016504000"  
Date: Sun, 20 Apr 2014 17:55:07 GMT

Content-Type: text/html; charset=UTF-8  
Content-Length: 94

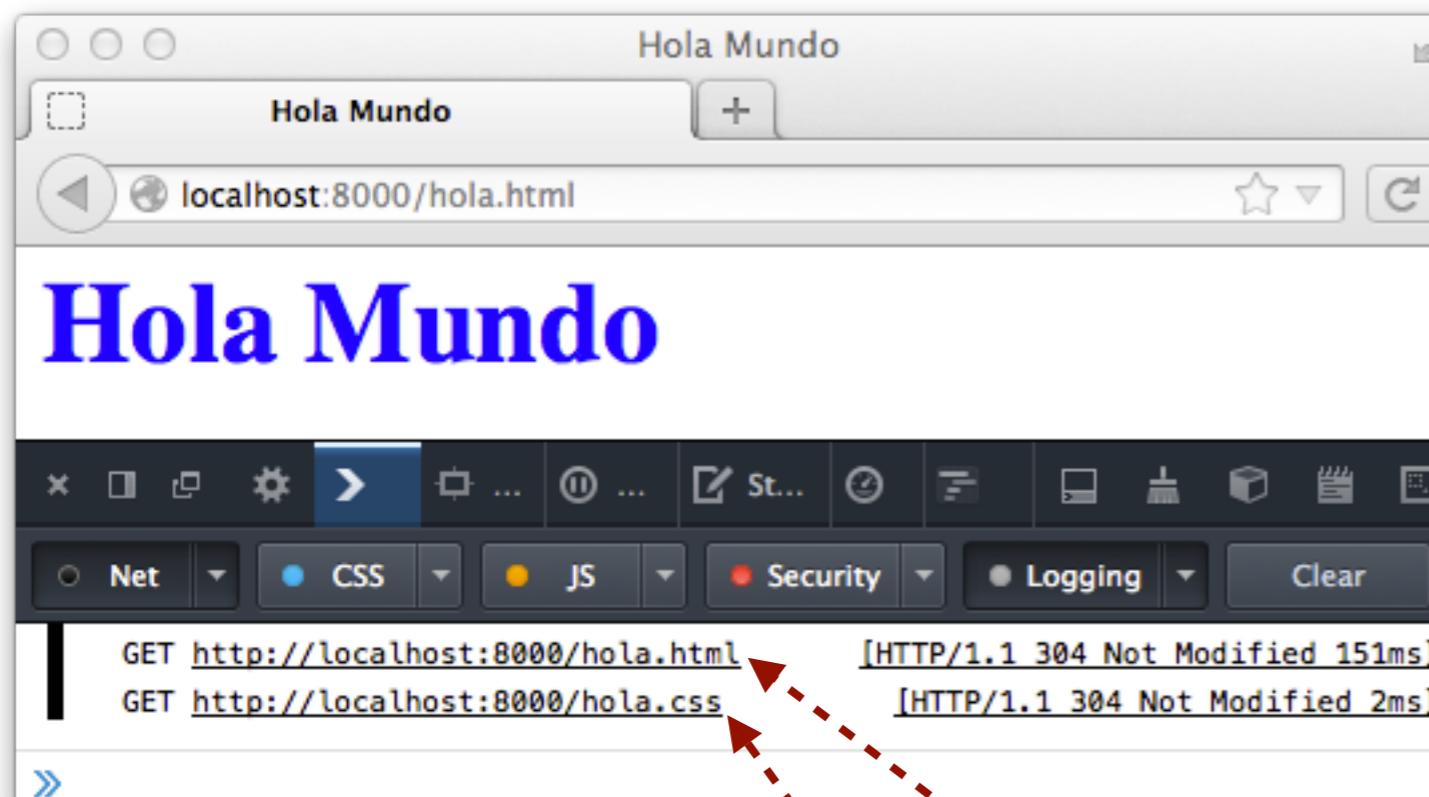
Connection: keep-alive  
Cache-Control: public, max-age=0  
Accept-Ranges: bytes

hola1.html UNREGISTERED

<html><head><title>Hola Mundo</title></head><body><h1>Hola Mundo</h1></body></html>

25

# Página hola.html



A screenshot of a browser window titled "Hola Mundo". The page content is "Hola Mundo". Below the browser is a developer tools interface. The network tab shows two requests:

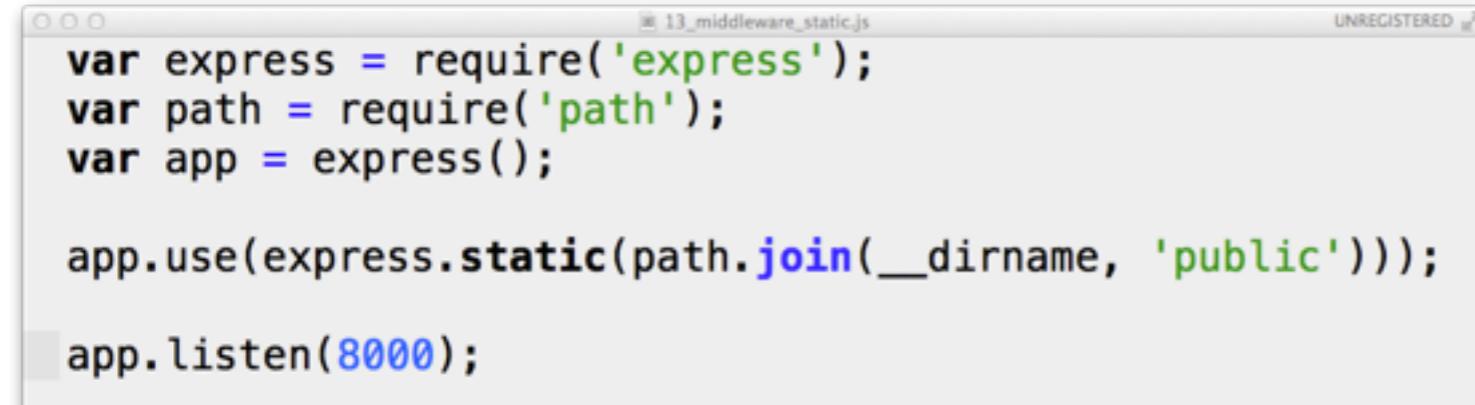
- GET http://localhost:8000/hola.html [HTTP/1.1 304 Not Modified 151ms]
- GET http://localhost:8000/hola.css [HTTP/1.1 304 Not Modified 2ms]



The code editor shows the file "hola.html" containing:

```
<html>
<head>
  <title>Hola Mundo</title>
  <link rel='stylesheet' type='text/css' href='hola.css'>
</head>

<body><h1>Hola Mundo</h1></body>
</html>
```



The code editor shows the file "13\_middleware\_static.js" containing:

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

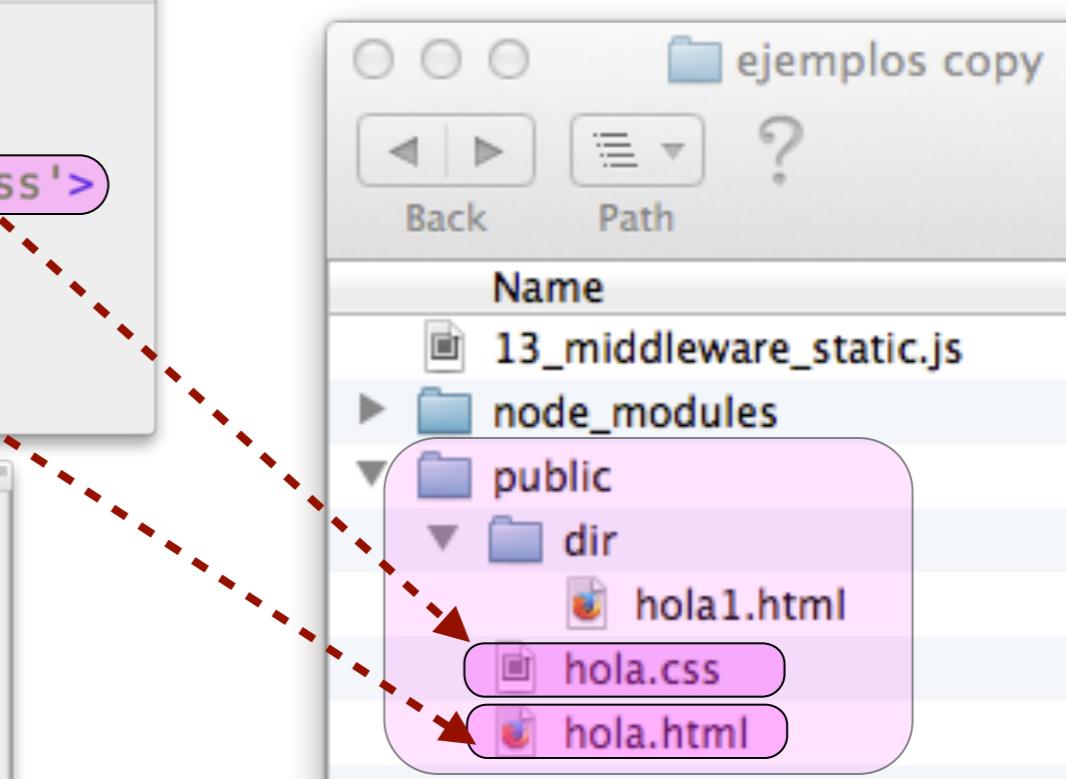
app.use(express.static(path.join(__dirname, 'public')));

app.listen(8000);
```

El servidor ‘static’ sirve páginas alojadas en el directorio ‘public’.

Sirve tanto páginas HTML como otros recursos enlazados, como la hoja de estilo “hola.css” de la página “hola.html”.

Cada recurso se sirve con una operación HTTP GET diferente.



# CURL



**CURL: cliente de acceso a servicios de cliente-servidor programable y muy completo.**

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

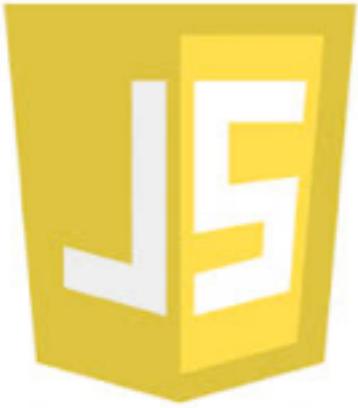
Ver opciones con:

\$ curl --help

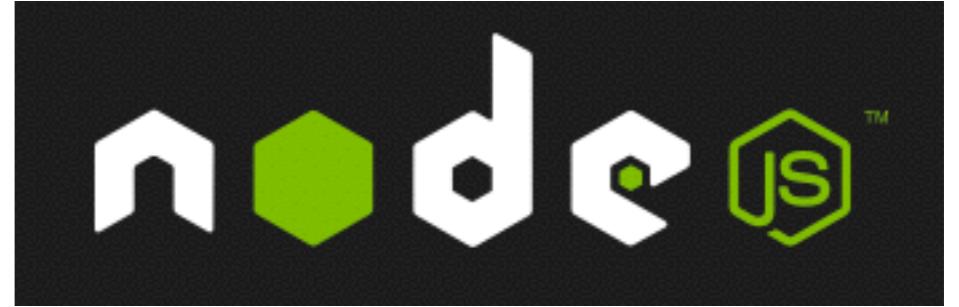
.....

\$ man curl

```
venus:~ jq$  
venus:~ jq$ curl -v http://localhost:8000/hola.html  
* Hostname was NOT found in DNS cache  
*   Trying 127.0.0.1...  
* Connected to localhost (127.0.0.1) port 8000 (#0)  
> GET /hola.html HTTP/1.1  
> User-Agent: curl/7.37.1  
> Host: localhost:8000  
> Accept: */*  
  
>  
< HTTP/1.1 200 OK  
< X-Powered-By: Express  
< Accept-Ranges: bytes  
< Date: Mon, 27 Oct 2014 14:23:28 GMT  
< Cache-Control: public, max-age=0  
< Last-Modified: Sun, 20 Apr 2014 11:24:11 GMT  
< ETag: W/"96-4073527116"  
< Content-Type: text/html; charset=UTF-8  
< Content-Length: 150  
< Connection: keep-alive  
<  
<html>  
<head>  
  <title>Hola Mundo</title>  
  <link rel='stylesheet' type='text/css' href='hola.css'>  
</head>  
  
<body><h1>Hola Mundo</h1></body>  
* Connection #0 to host localhost left intact  
venus:~ jq$ █
```



JavaScript



# Introducción a REST

Juan Quemada, DIT - UPM

# Que es REST

- ◆ **REST: REpresentational State Transfer**
  - El estado se representa en el recurso transferido al cliente
    - ◆ [http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)
- ◆ **REST: Principios arquitecturales para aplicaciones Web escalables**
  - Propuestos por **Roy Fielding** en su Tesis Doctoral (2000)
    - ◆ Roy Fielding fue co-diseñador de HTTP y ha sido uno de los desarrolladores principales del proyecto Apache
  - Conocida como: **Arquitectura Orientada a Recursos (ROA)**
- ◆ Interfaces **REST** o Servicios Web **RESTful**
  - Cliente y servidor interaccionan con Interfaz Uniforme de HTTP
    - ◆ Métodos GET, POST, PUT y DELETE
  - REST está muy extendido: Google, Twitter, Amazon, Facebook, ...

# Interfaces REST

## ◆ Interfaz REST

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

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

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



# Principios REST

- ◆ **Direccionabilidad (Addressability)** de los recursos
- ◆ Uso del **interfaz uniforme de HTTP**: GET, POST, PUT y DELETE
- ◆ **Comunicación sin estado** en el servidor (Statelessness)
- ◆ Servicio **hipermedia** (Connectedness) conectado con URLs
- ◆ **Recursos en formatos abiertos**: HTML, XML, JSON, RSS, texto plano, ...

# Direccionabilidad e interfaz uniforme

## ◆ En una arquitectura orientada a recursos

- Todo recurso del servidor tiene una ruta (dirección) diferente
  - ◆ Las rutas (paths) son las direcciones (pueden incluir query o ancla para información adicional)
- Los recursos se procesan solo con los métodos del interfaz uniforme
  - ◆ GET, POST, PUT y DELETE

## ◆ Ejemplo de una colección de usuarios (suele asociarse a una tabla de la DB)

- La ruta suele tomar el nombre de la colección en plural (usuarios)
  - ◆ Las operaciones individuales identifican al usuario con un identificador en la ruta
    - ◆ **POST /usuarios?nombre=Pedro+Ramirez&edad=8** // Crear nuevo usuario
    - ◆ **GET /usuarios** // Traer lista de todos los usuarios
    - ◆ **GET /usuarios/2007** // Traer datos del usuario 2007
    - ◆ **DELETE /usuarios/2007** // Borrar usuario 2007 de la colección
    - ◆ **PUT /usuarios/2007?edad=9** // Actualizar edad del usuario 2007
    - ◆ ademas suele haber primitivas GET para cargar formularios asociados a POST y PUT

# Seguridad e idempotencia



- ◆ Seguridad e idempotencia son 2 propiedades importantes
  - Método **seguro** (safe): no modifica datos en el servidor y puede ser cacheado
  - Método **idempotente**: el resultado es independiente del número de invocaciones

## ◆ Operación idempotente

- El resultado de invocar el método n veces es igual a invocarlo 1 vez
  - ◆ Por ejemplo: **x=2** es idempotente, pero **x=x+1** no es idempotente
- Las operaciones asociadas a un interfaz REST deben ser idempotentes
  - ◆ Por ejemplo: **PUT /usuario/2007?edad=9**

## ◆ Internet no es fiable

- La invocación de una solicitud HTTP puede ejecutarse n veces en el servidor
  - ◆ Si la **solicitud se pierde** y hay reenvío, el método **se ejecuta solo 1 vez**
  - ◆ Si la **respuesta se pierde** y hay reenvío, el método **se ejecuta 2 veces** en el servidor

# Propiedades del interfaz uniforme

## ◆ Interfaz uniforme o CRUD

- Permite crear servicios desacoplados y escalables (<http://restcookbook.com>)

## ◆ Propiedades de los métodos del interfaz uniforme

- POST: **El más peligroso** (puede duplicar recursos)
- GET: **Seguro (cacheable) e idempotente**
- PUT: **idempotente**
- DELETE: **idempotente**

## ◆ Recomendaciones de diseño importantes

- Tratar de minimizar el impacto de no idempotencia de **POST**
- No utilizar **nunca GET** para **modificar recursos** del servidor
  - ◆ Utilizar POST, PUT o DELETE según el tipo de modificación, porque GET puede ser cacheado y no modificará los recursos

# Servicio hipermédia sin estado en el servidor

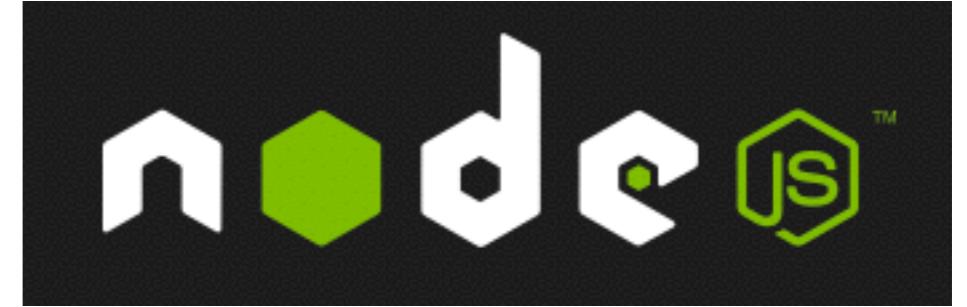
- ◆ **El servicio se usa navegando** por los recursos recibidos del servidor
  - El recurso contiene el estado del cliente (suele ser una página Web)
    - ◆ Las **transiciones** son los enlaces (URLs) y se navega al hacer clic en ellos
- ◆ Los servidores escalan porque **no guardan el estado** de los clientes
  - Solo gestionan recursos a través del interfaz uniforme
    - ◆ Con transacciones HTTP que son independientes entre sí
- ◆ **Los clientes guardan siempre el estado de uso del servicio**
  - en la página Web o recurso cargadas, en cookies, en localStorage, ....

# Representación de los recursos

- ◆ Los recursos transferidos por el servidor al cliente
  - Representan el estado y las transiciones necesarios para navegar por el servicio
    - ◆ Un mismo recurso se puede representar (o serializar) en distintos formatos
- ◆ Formatos más habituales de representación de recursos:
  - **HTML**: para presentar información legible en un browser
    - ◆ XHTML: versión sintácticamente más estricta de HTML
  - **JSON**: Formato de serialización de objetos Javascript
  - **XML**: Formato de datos tipo SGML del W3C
  - **RSS**: formato para representar colecciones (de feeds de blogs)
  - **ATOM**: formato para representar colecciones (de feeds de blogs)
- ◆ HTTP usa el tipo MIME para tipar los recursos que transfiere



JavaScript



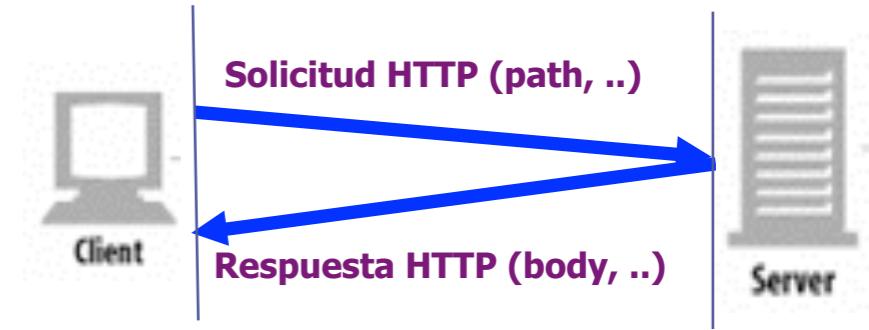
# Aplicaciones REST con express.js

Juan Quemada, DIT - UPM

# Métodos de Ruta

express

Web application  
framework for  
node



- ◆ Express posee métodos que instalan middlewares asociados a
  - verbos HTTP con rutas específicas
    - ◆ **get**(path, MW), **post**(path, MW), **put**(path, MW), **delete**(path, MW), **all**(path, MW)
      - ◆ MW solo se ejecuta si verbo y path coinciden con la solicitud HTTP
- ◆ Los 2 primeros parámetros de un middleware **MV(req, res, ..)** son
  - **req**: objeto JavaScript con parámetros de la Solicitud HTTP
    - ◆ Doc: <http://expressjs.com/4x/api.html#request>
  - **res**: objeto JavaScript para configurar la Respuesta HTTP
    - ◆ Doc: <http://expressjs.com/4x/api.html#response>
- ◆ Documentación
  - Doc: <http://expressjs.com/4x/api.html#app.VERB>

# Ruta GET

El método **get(path, MW)** instala el middleware MW en app para que se ejecute solo cuando llegan Solicitudes HTTP: **GET path**. Si llega otro método (PUT, POST, DELETE) o es GET con otro path, se pasa al siguiente middleware, sin ejecutar este.

La invocación de **res.send(string)** configura la Respuesta HTTP en el objeto **res** del middleware con la configuración estándar:

- Código: 200 OK
- Content-Type: text/html
- Content-length: longitud del string en octetos
- Body: string

y envía la Respuesta HTTP al cliente.

```
05_ruta_get.js UNREGISTERED
var express = require('express');
var app = express();

// el método 'get(path, MW)' responde a peticiones 'GET path'
// como respuesta a 'GET path' ejecuta el middleware MW
// responde solo al path '/mi_ruta'
app.get('/mi_ruta', function (req, res){
  res.send('<html><body><h1>Mi Ruta</h1></body></html>');
});
// devuelve página HTML con tipo MIME: text/html

app.listen(8000);
```

The screenshot shows a terminal window with the file content and a browser window showing the result. The browser address bar shows 'localhost:8000/mi\_ruta/' and the page content is 'Mi Ruta'.

# MW static y ruta

Un middleware se instala con el método **use(MW)** o con una ruta, p.e. **get(path, MW)**. Los MWs se invocan en el orden de instalación.

El middleware static se ejecuta primero. Si la solicitud es GET y si existe el recurso identificado por el path, el recurso se sirve.

Si el recurso no existe, pasa control al siguiente MW, es decir a la ruta **get('/mi\_ruta', ...)** que responde a /mi\_ruta con la respuesta programada en el MW asociado (**res.send(<html> ... </html>)**)

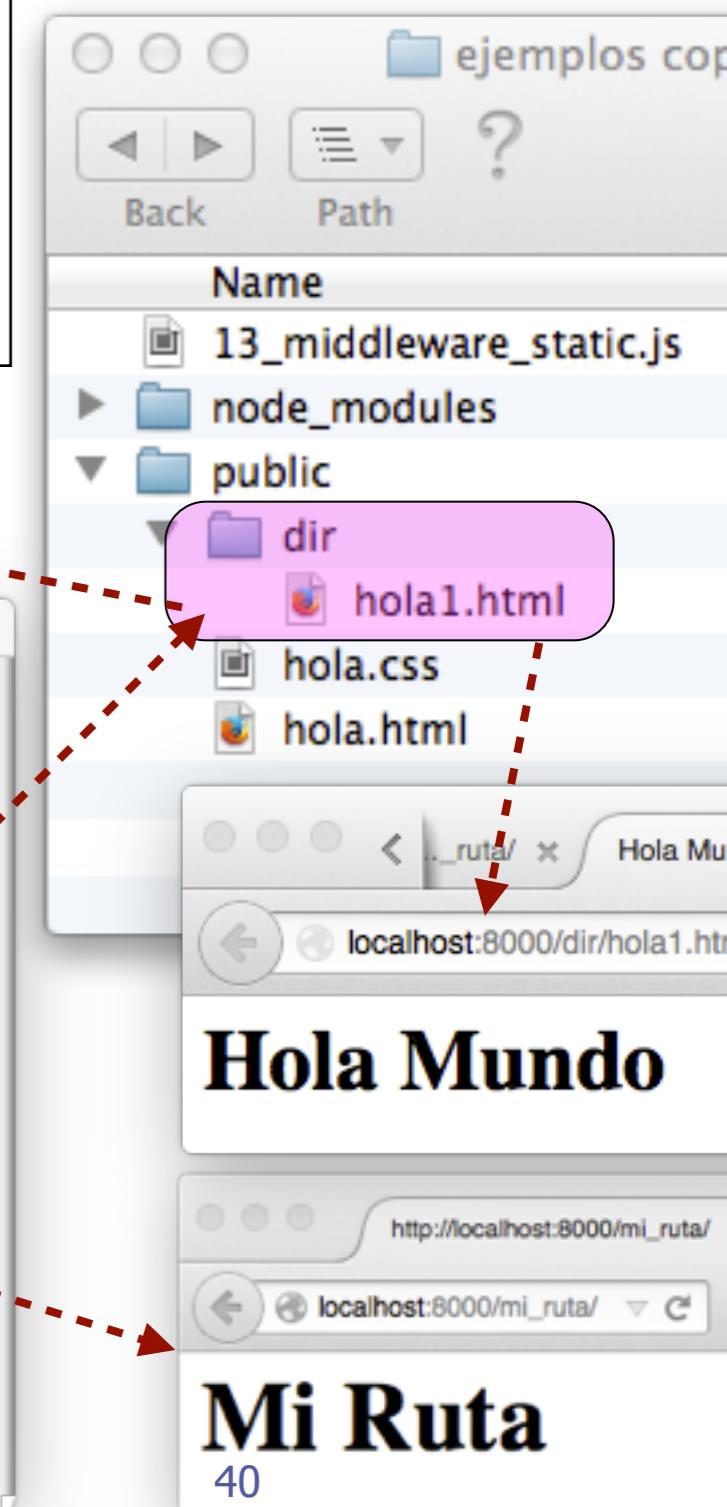
The diagram illustrates the execution flow from the code editor to the browser. A dashed red arrow points from the highlighted middleware code in the code editor to the file structure in the file browser. Another dashed red arrow points from the highlighted route code in the code editor to the browser's address bar and title bar.

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

// sirve páginas del directorio 'public'
app.use(express.static(path.join(__dirname, 'public')));

app.get('/mi_ruta', function (req, res){ // ruta '/mi_ruta'
  res.send('<html><body><h1>Mi Ruta</h1></body></html>');
});

app.listen(8000);
```



# Prioridad

Si en el repositorio de recursos del middleware static existiese un recurso estático '/mi\_ruta/index.html' este recurso se servirá por el middleware static con cualquiera de estos paths:

- /mi\_ruta
- /mi\_ruta/
- /mi\_ruta/index.html

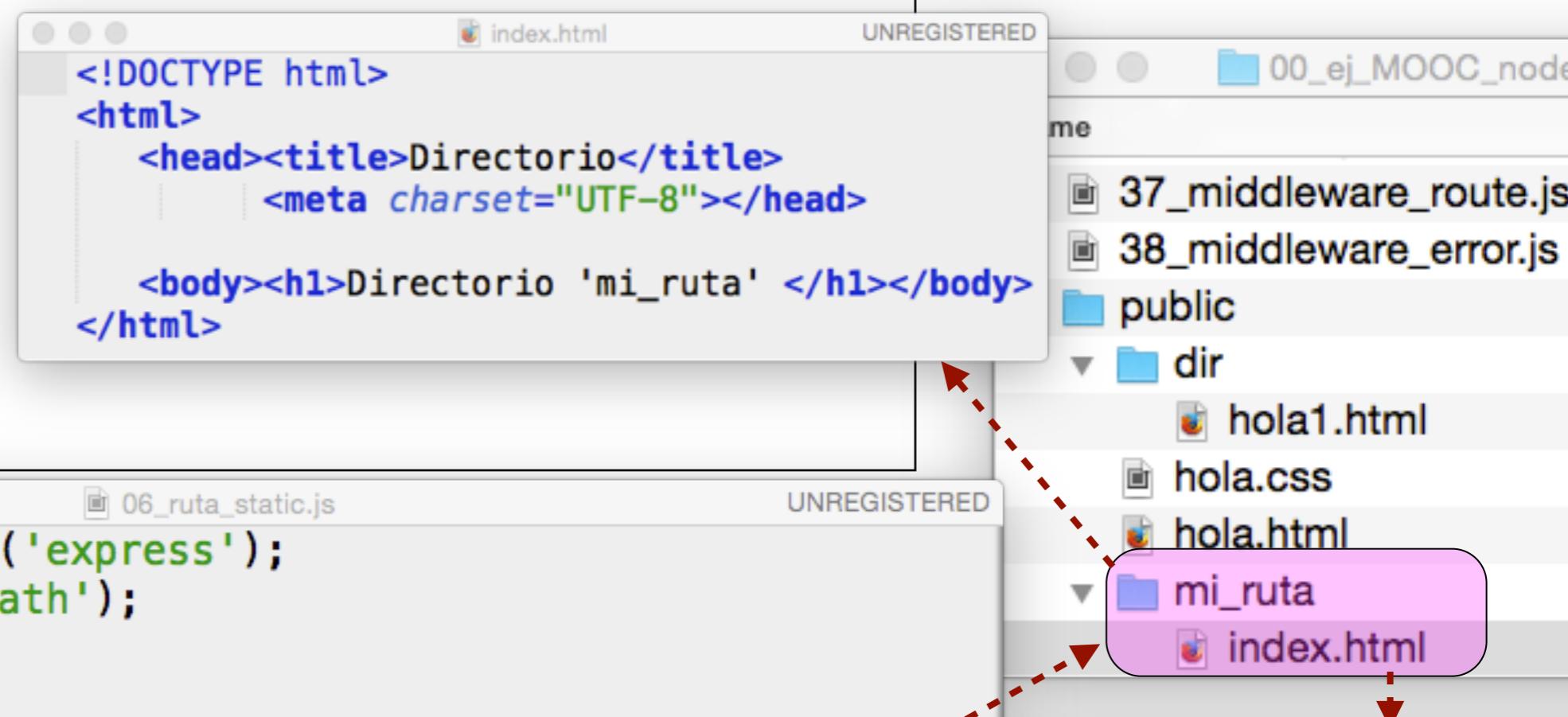
El MW de la ruta **get(...)** nunca se ejecutará, porque el **MW static** atiende antes la petición y responde.

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

// sirve páginas del directorio 'public'
app.use(express.static(path.join(__dirname, 'public')));

app.get('/mi_ruta', function(req, res){ // ruta '/mi_ruta'
  res.send('<html><body><h1>Mi Ruta</h1></body></html>');
});

app.listen(8000);
```



## Directorio 'mi\_ruta'



# Más rutas

Los filtros **get(path, MW)** instalan middlewares que se invocan en el mismo orden que han sido instalados.

Los 3 middlewares de este ejemplo finalizan la atención a la Solicitud HTTP al invocar **res.send(...)**, que envía la respuesta y finaliza la ejecución. Por lo que si uno de ellos atiende la solicitud, los siguientes no se ejecutarán.

Express interpreta la ruta '\*' como cualquier path, por lo que el tercer MW atenderá cualquier path que no haya sido atendido por los 2 primeros.

The diagram illustrates the execution flow of three Express route handlers:

- Path '/' Handler:** Handles the root path. It sends the response "Bienvenido a mi primera aplicacion".
- Path '/mi\_ruta' Handler:** Handles the path "/mi\_ruta". It sends the response "**Mi Ruta**".
- Path '\*' Handler:** Handles any other path. It sends the response "URL incorrecto".

Dashed arrows point from each highlighted code block to its corresponding browser screenshot on the right.

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

app.get('/', function (req, res){ // atiende ruta '/'
  res.send('Bienvenido a mi primera aplicacion');
});

app.get('/mi_ruta', function (req, res){ // ruta '/mi_ruta'
  res.send('<html><body><h1>Mi Ruta</h1></body></html>');
});

app.get('*', function (req, res){ // *: cualquier otra ruta
  res.send('URL incorrecto');
});

app.listen(8000);
```

# Parámetros req y res

`function(req, res) {}`

La documentación de la API de express en

<http://expressjs.com/4x/api.html>

contiene toda la información sobre las aplicaciones express, los middlewares y sobre los objetos que se manejan, como req y res.

The screenshot shows a browser window with the title "Express - api reference". The address bar displays "expressjs.com/3x/api.html#res.status". The page content is the "API Reference" section for the "res" object. The "res.status(code)" method is highlighted with a pink box and a callout arrow pointing to it from the text "function(req, res) {}". The callout also points to a code example: `res.status(404).sendfile('path/to/404.png');`. To the right of the callout, a vertical pink box lists several methods: `res.status()`, `res.set()`, `res.get()`, `res.cookie()`, `res.clearCookie()`, `res.redirect()`, `res.location()`, `res.charset`, `res.send()`, `res.json()`, `res.jsonp()`, `res.type()`, `res.format()`, `res.attachment()`, `res.sendFile()`, `res.download()`, `res.links()`, `res.locals`, `res.render()`, and `Middleware`.

**res.status(code)**

Chainable alias of node's `res.statusCode=`.

```
res.status(404).sendfile('path/to/404.png');
```

**res.set(field, [value])**

Set header `field` to `value`, or pass an object to set multiple fields at once.

```
res.set('Content-Type', 'text/plain');

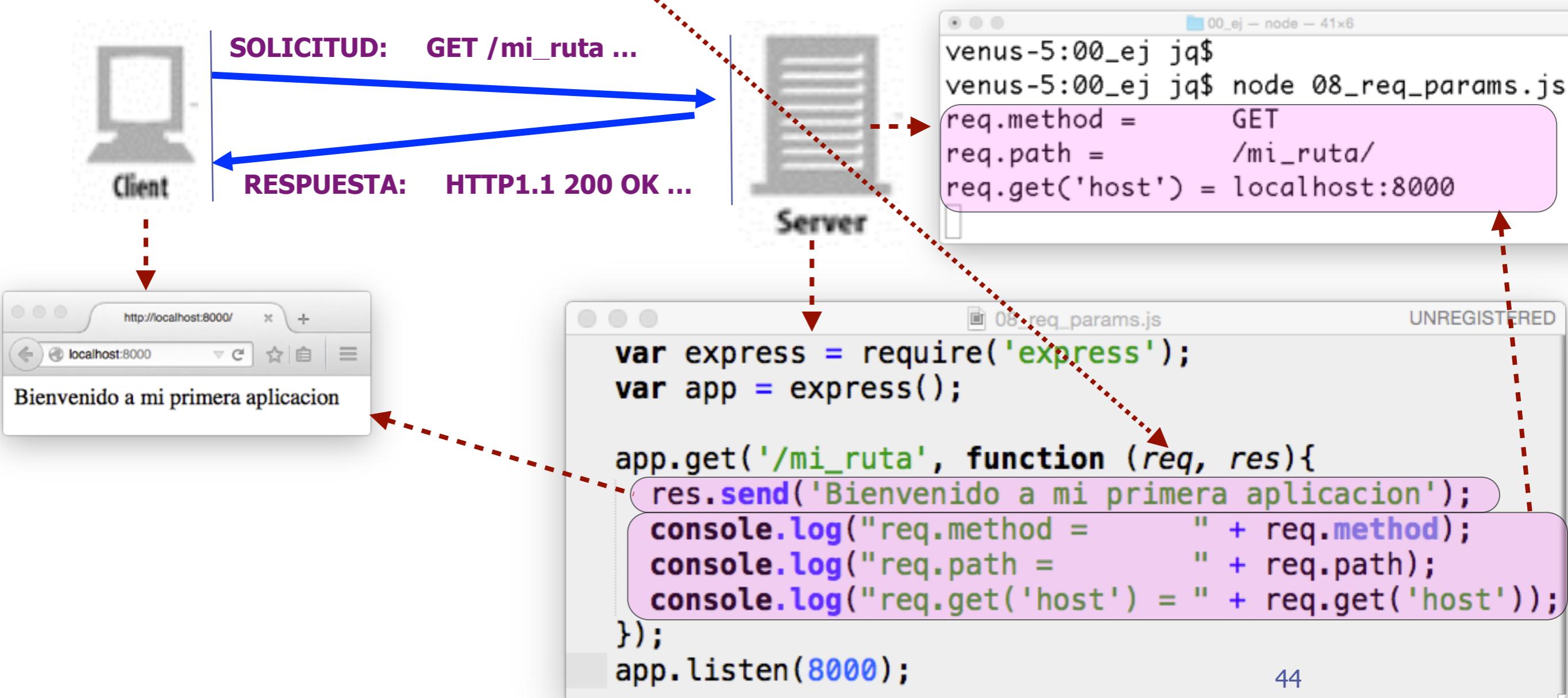
res.set({
  'Content-Type': 'text/plain',
  'Content-Length': '123',
  'ETag': '12345'
})
```

Aliased as `res.header(field, [value])`.

# Objeto res: paráms de la respuesta

El parámetro **req** de un MW express es un objeto con todo lo relativo a la solicitud HTTP y permite conocer sus parámetros.

El ejemplo lista por **consola del servidor** el método y el path obtenidos de las propiedades **method** y **path** del objeto **req** y el parámetro 'host ....' obtenido con el método 'get(...)' de la librería express.



# Objeto res: paráms de la respuesta

El parámetro **res** de 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 llevara por lo tanto el parámetro

**Content-Type: text/plain**

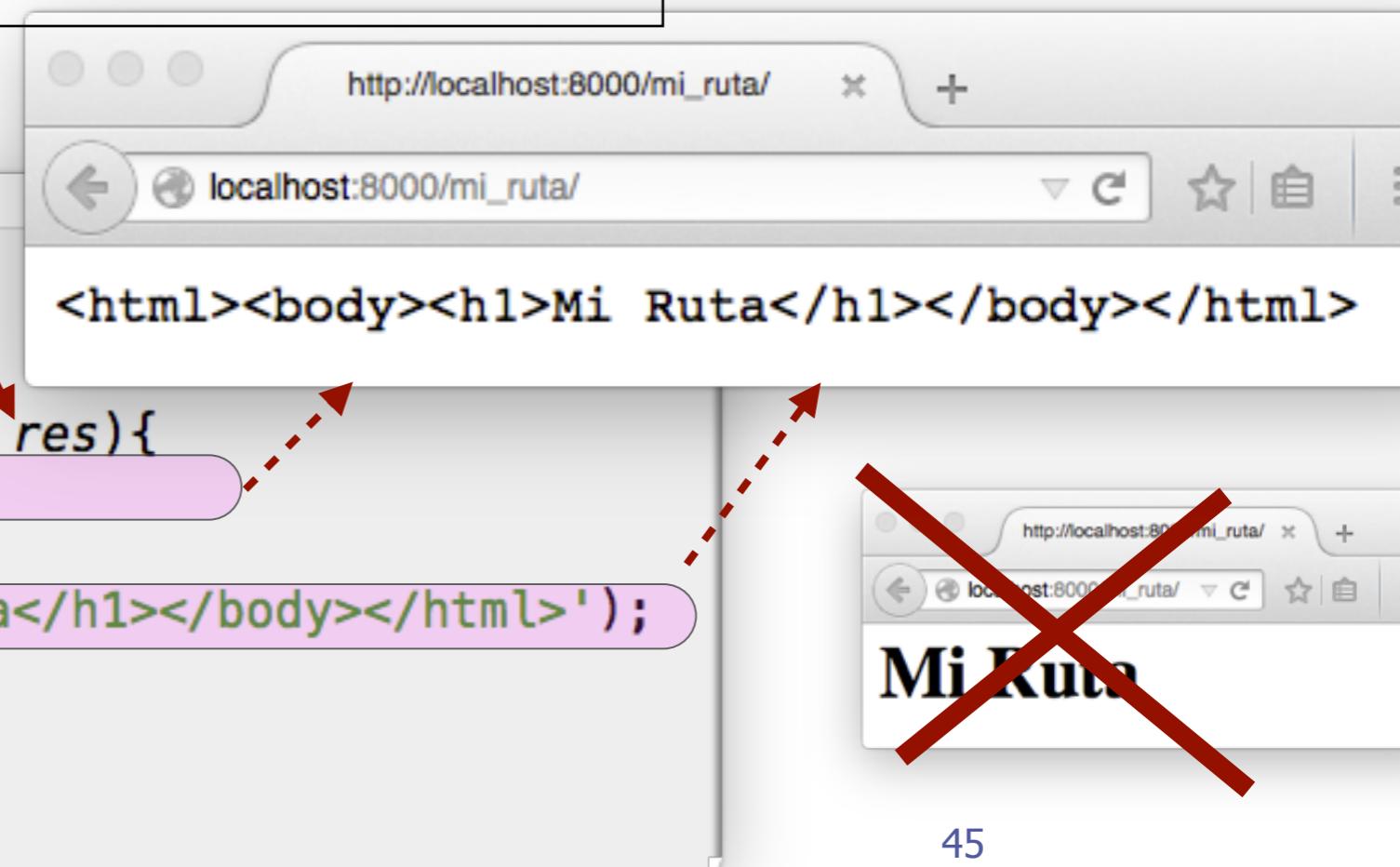
y el navegador interpretara la página HTML como texto plano mostrando el código HTML en vez de la página formateada.

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);
```



# CURL



**CURL:** cliente de acceso a servicios de cliente-servidor programable y muy completo.

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

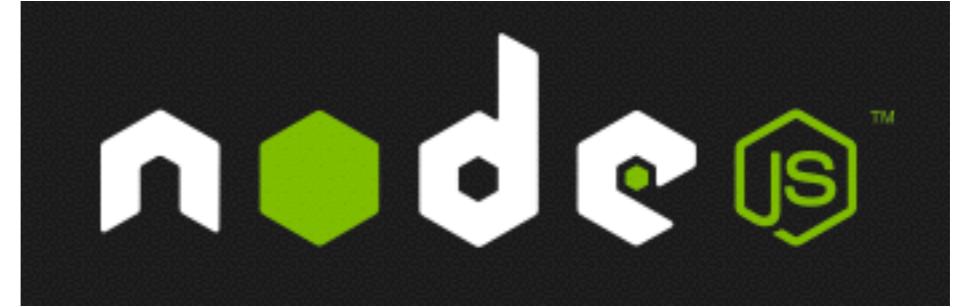
Ver opciones con:

```
$ curl --help  
.....  
$ man curl
```

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



JavaScript



# Parámetros de ruta (path)

Juan Quemada, DIT - UPM

# Acceso parámetros en la ruta (path)

- ◆ Un cliente puede enviar parámetros
  - como parte de la ruta (path)
    - ◆ Estos URLs se denominan URLs pesados o fat URLs
- ◆ express.js permite identificar parámetros
  - en la definición de rutas que acepta utilizando el carácter ":"
    - ◆ Ejemplos
      - '/hola/:n' // donde :n es un parámetro
      - '/service/:op/user/:id' // donde :op e :id son parámetros
- ◆ Los parámetros pueden tomar cualquier valor
  - La ruta fija solo la parte que no son parámetros

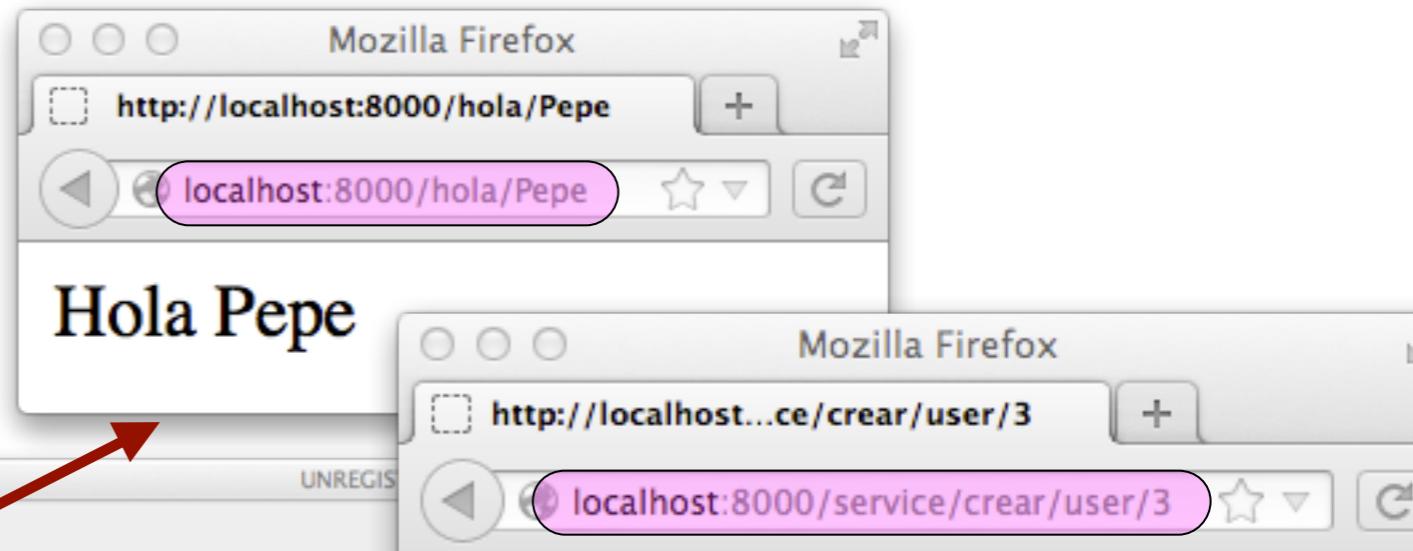
# Parámetros de ruta

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

// URL con parámetro: /hola/:n
// -> Reconoce /hola/Pepe, /hola/14, ...
app.get('/hola/:n', function(req, res){
    // :n accesible como req.params.n
    res.send('Hola ' + req.params.n);
});

// 2 parámetros :op e :id
app.get('/service/:op/user/:id', function(req, res){
    // :p accesible como req.params.p
    res.send('Usuario ' + req.params.id
        + ' solicita ' + req.params.op);
});

app.listen(8000);
```



Los parámetros **req**, **res** dan acceso desde el manejador **function (req, res) {}** a las cabeceras HTTP de solicitud y respuesta.

**req** guarda en **params** los parámetros del **URL** con el mismo nombre:

- **req.params.n**
- **req.params.id**
- **req.params.op**

# Condiciones en parámetros

Este ejemplo acepta URLs con paths /user, /user/Pepe, /user/1, .. o /user1/27, /user1/1, ..

pero no responderá a URLs con paths /usuario, /user2, /user1/Pepe, ..

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

// en :id? la interrogación indica opcional
// -> Reconoce /user /user/Pepe, /user/14, ...
app.get('/user/:id?', function(req, res){
  res.send('User ' + (req.params.id || 'anónimo'));
});

// Parámetros se restringen con RegExp entre paréntesis
// '/user/:id(\d+)' reconoce '/user1/23' o '/user1/1'
// pero no '/user1/a' o '/user1
app.get('/user1/:id(\d+)', function(req, res){
  res.send('User1 ' + req.params.id);
});

app.listen(8000);
```

The code editor shows an Express.js application with two routes. The first route, `/user/:id?`, uses a question mark after the parameter name to indicate it is optional. The second route, `/user1/:id(\d+)`, uses a regular expression constraint to require a digit sequence. Arrows from the highlighted code segments point to three Mozilla Firefox browser windows. The top window shows the result of the first route with a non-existent user ID, displaying 'User anónimo'. The middle window shows the result of the first route with a valid user ID, displaying 'User Pepe'. The bottom window shows the result of the second route with a valid user ID, displaying 'User1 23'.



JavaScript



# Composición y ejecución de middlewares

Juan Quemada, DIT - UPM

# Middleware express

## ◆ Middleware express

- Función JavaScript que se ejecuta al llegar una transacción HTTP
  - ◆ Recibe la solicitud HTTP en el **parámetro req** y prepara la respuesta en el **parámetro res**
  - ◆ Pasa control con **next()** al siguiente MW o con **next(err)** al siguiente MW de error

## ◆ Se instalan con **use(..)** o con rutas como **get(..)**, **put(..)**, ..

- El orden de ejecución (invocación) es el mismo en que han sido instalados

## ◆ Los MWs permiten una programación secuencial, modular y legible

- Conservan la eficacia de la programación por eventos
  - ◆ <http://expressjs.com/guide/using-middleware.html>
  - ◆ <http://expressjs.com/resources/middleware.html>

## ◆ express.js se basa en el concepto de middleware de connect.js

- Hereda muchos MWs: <http://expressjs.com/resources/middleware.html>
- Proyecto connect: <http://www.senchalabs.org/connect/>

# Composición y terminación de middlewares

- ◆ express estructura la respuesta a una solicitud HTTP
  - Como una secuencia de **middlewares** donde
    - ◆ cada middleware procesa algún elemento u opción de la transacción HTTP
  - La función **next()** permite pasar control al siguiente middleware
- ◆ Un middleware puede terminar de 3 formas
  - **Finalizando el proceso** (si MW finaliza sin invocar **next()**)
    - ◆ En este caso se suele enviar la respuesta HTTP al cliente invocando **send()**
      - ◆ La atención a la solicitud HTTP acaba y los siguientes MWs ya no se ejecutarán
  - **Pasando control al siguiente middleware**
    - ◆ Un middleware cede el control al siguiente MW invocando la función **next()**
  - **Pasando control al siguiente middleware de error**
    - ◆ Un MW cede el control al primer **MW de error** invocando la función **next(err)**

# Parámetros de un middleware

- ◆ Un **middleware** tiene estos 3 primeros parámetros
  - **middleware(req, res, next)**
    - ◆ el MW se puede invocar como **MW()**, **MW(req, res)**, **MW(req, res, next, id)**, ...
- ◆ Un **middleware de error** incluye un **error** como primer parámetro
  - **middleware\_de\_error(err, req, res, next)**
- ◆ Parámetros de un middleware
  - **err**: parámetro con error (solo en middleware de error)
  - **req**: parámetro con objeto con la información de la solicitud HTTP
  - **res**: parámetro con objeto donde se construye la respuesta HTTP
  - **next**: función de paso de control al siguiente middleware

# Contador de visitas

Los middleware de express se ejecutan en el orden en que se ha instalado con `app.use(<mideleware>)`. El middleware es una función con hasta 3 parámetros: `function(res, req, next)`

`req` y `res` dan acceso a la solicitud y a la respuesta HTTP. La función `next()` pasa el control al siguiente middleware.

Como `next()` no se invoca, la ejecución finaliza en este middleware, que envía la respuesta al cliente con `send(...)`.

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

// Middleware - contador de visitas
app.use(function(req, res, next){
    app.locals.cont = (app.locals.cont || 0);
    app.locals.cont += 1;
    console.log("Visitas: " + app.locals.cont);
    next(); // next() pasa a siguiente middleware
            // sinó los siguientes no se ejecutarían
});

app.get('*', function(req, res){
    res.send('Visita número: ' + app.locals.cont);
});

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

**express.js** utiliza `app.locals` y `res.locals` para definir variables locales en MWs.

`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)



# middleware get()

Los métodos **get()**, **post()**, **put()** y **delete()** instalan también middlewares.

En este MW la condición de if/else decide si

- se envía la respuesta al cliente con la función `send()` y finaliza la atención a la solicitud HTTP.
- se invoca la función `next()` y se pasa control al siguiente middleware.

The diagram illustrates the execution flow of the provided Express.js code. A red arrow points from the explanatory text in the top right to the conditional logic in the code. Another red arrow points from the 'User del sistema' response in the first browser window to the 'User desconocido' response in the second browser window, indicating the flow of control between the two middleware functions.

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

app.get('/user/:id', function(req, res, next){
  if (req.params.id === "Ana" || req.params.id === "Eva"){
    res.send('Usuario del sistema');
  }
  else {
    next();
  }
});

app.get('*', function(req, res){
  res.send('Usuario desconocido');
});

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

UNREGISTERED Mozilla Firefox http://localhost:8000/user/Ana localhost:8000/user/Ana Usuario del sistema

Mozilla Firefox http://localhost:8000/user/Luis localhost:8000/user/Luis Usuario desconocido

La **function(err, res, req, next)** define un middleware de error, donde el primer parámetro **err** representa una condición de error encontrada en otro middleware.

Si el **user** del primer `get()` no es **Ana**, **Pepe** o **Eva**, se invoca `next( new Error(...))` pasando directamente al middleware de error (ver ejemplo).

The diagram illustrates the execution flow of an Express.js application. It starts with a code editor window containing the file `38_middleware_error_eq.js`. The code defines an application with two routes: one for user IDs and another for wildcard routes. It also includes a global error middleware. Arrows point from specific code snippets to their corresponding browser outputs:

- An arrow points from the line `res.send('Usuario del sistema');` in the `/user/:id` route to a browser window showing "Usuario del sistema".
- An arrow points from the line `res.send('Operación inválida');` in the wildcard route to a browser window showing "Operación inválida".
- A dashed arrow points from the `next(new Error('Usuario desconocido'));` line in the `/user/:id` route to the global error middleware block at the bottom.
- An arrow points from the global error middleware block to a browser window showing "Error: Usuario desconocido".

**middleware de error**

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

app.get('/user/:id', function(req, res, next){
  if (req.params.id === "Ana" || req.params.id === "Eva"){
    res.send('Usuario del sistema');
  } else {
    next(new Error('Usuario desconocido'));
  }
});

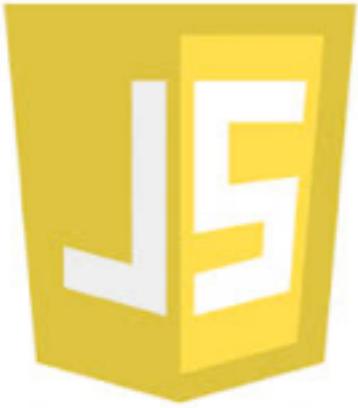
app.get('*', function(req, res){
  res.send('Operación inválida');
});

// Middleware de error
app.use(function(err, req, res, next){
  res.send(err.toString()); // Envía error
});

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

# Middlewares existentes (connect y otros)

- **logger** request logger with custom format support
- **csrf** Cross-site request forgery protection
- **compress** Gzip compression middleware
- **basicAuth** basic http authentication
- **bodyParser** extensible request body parser
- **json** application/json parser
- **urlencoded** application/x-www-form-urlencoded parser
- **multipart** multipart/form-data parser
- **cookieParser** cookie parser
- **session** session management support with bundled MemoryStore
- **cookieSession** cookie-based session support
- **methodOverride** faux HTTP method support
- **responseTime** calculates response-time and exposes via X-Response-Time
- **staticCache** memory cache layer for the static() middleware
- **static** streaming static file server supporting Range and more
  - **directory** directory listing middleware
  - **vhost** virtual host sub-domain mapping middleware
  - **favicon** efficient favicon server (with default icon)
  - **limit** limit the bytesize of request bodies
  - **query** automatic querystring parser, populating req.query
  - **errorHandler** flexible error handler
- y más módulos de terceros (<http://expressjs.com/resources/middleware.html>)



JavaScript



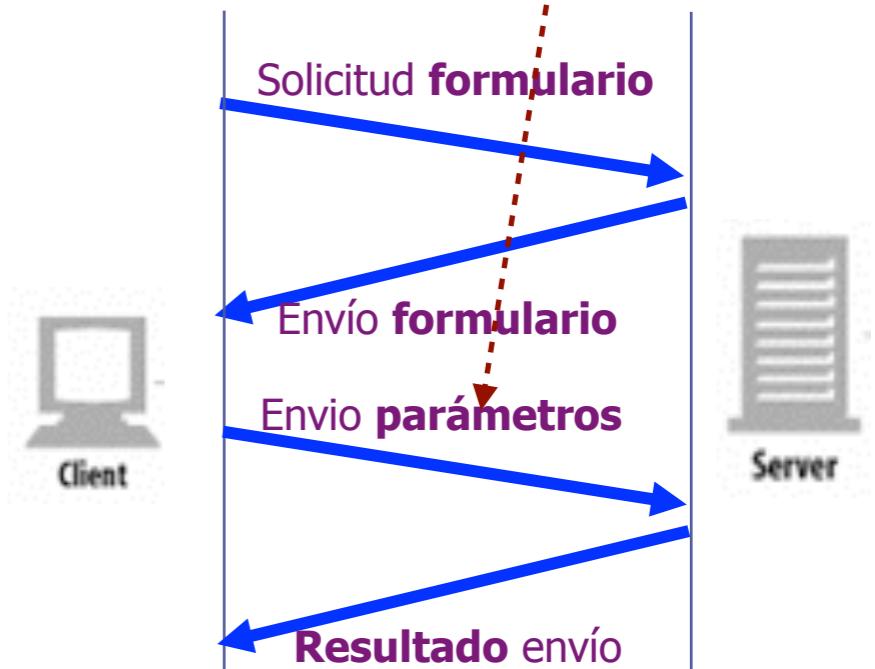
# Formulario GET

Juan Quemada, DIT - UPM

# Parámetros con GET

```
GET /hola?user=Paco&id=7 HTTP/1  
Host: upm.es  
Accept: text/*, image/*  
Accept-language: en, sp  
.....  
User-Agent: Mozilla/5.0
```

- ◆ Un formulario permite enviar parámetros al servidor
  - solo mediante **Solicitudes HTTP** de tipo **GET** y **POST**
    - ◆ Los parametros se teclean en cajetines del formulario
      - Los cajetines se definen con la marca <input ....> y otras
- ◆ GET envía parámetros en **pregunta (query)** del path
  - Solicitud HTTP: **GET /hola?user=Paco&id=7**
    - ◆ Valores de parámetros: van en el query del path
      - Son strings de tamaño limitado
- ◆ Un envío de parámetros al servidor
  - Suele incluir dos transacciones
    - ◆ Transacción 1: Carga del formulario
    - ◆ Transacción 2: Envío y proceso de datos



# Formulario: ejemplo

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

<body>
    <form method="get" action="/hola" >
        Su nombre: <br>
        <input type="text"
            name="user"
            value="teclée su nombre"
        /><br>
        <input type="submit"
            value="Enviar"
        />
    </form>
</body>
</html>
```

Un formulario se crea con la marca **<form..>**

**<form method=.. action=.. >**

-> **method** indica si es **GET** o **POST**

-> **action** define la ruta (path) de la transac.

Las marcas **<input ..>** definen los cajetines y botones del formulario:

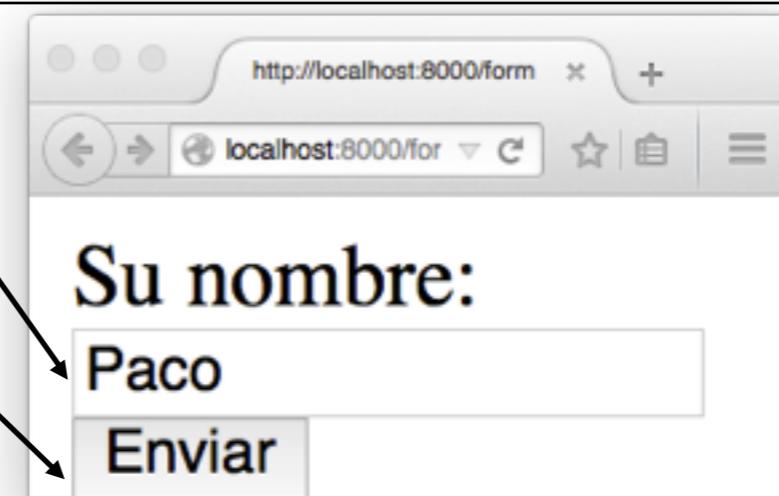
**<input type='text' ..>** crea un cajetín donde teclear un parámetro.

-> **name='user'** nombre del parámetro

-> **value='teclée ..'** texto del cajetín.

**<input type='submit' ..>** crea el botón de realizar transacción.

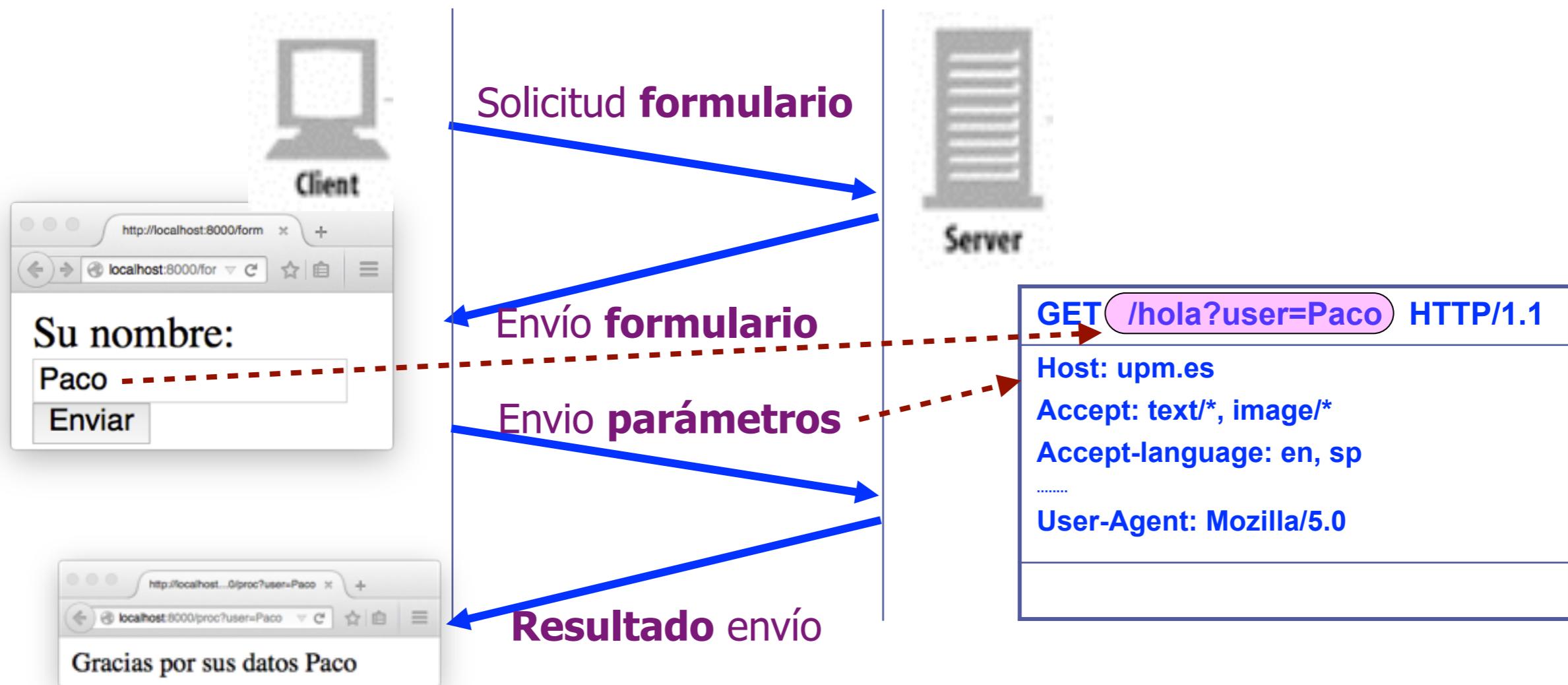
-> **value='Enviar'** define texto del botón



# Envío de parámetros en query

El envío de datos de un cliente a un servidor necesita un formulario, por lo que una aplicación de envío de datos necesita soportar 2 transacciones.

- Transacción 1: cargar de la página Web con el formulario que permite enviar los datos.
- Transacción 2: envío de datos en **query** con **GET** desde el formulario, seguida de respuesta.



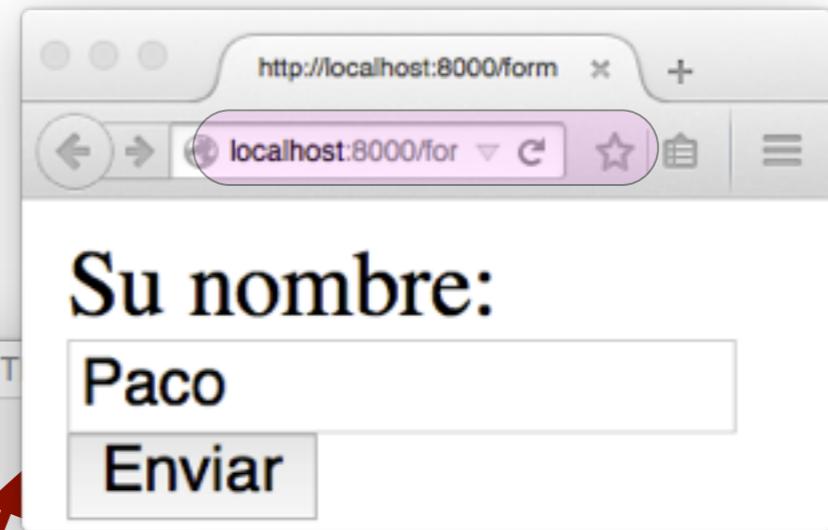
# Formulario con GET

```
21_pregunta_GET_query.js  
UNREGISTERED  
  
var express = require('express');  
var app = express();  
  
// Transacción 1: carga del formulario  
// → GET /form ...
```

```
app.get('/form', function(req, res){  
  res.send('<html><body>'  
    + '<form method="get" action="/proc">'  
    + 'Su nombre: <br>'  
    + '<input type="text" name="user" /><br>'  
    + '<input type="submit" value="Enviar"/>'  
    + '</form>'  
    + '</body></html>');  
});
```

```
// Transacción 2: envío y proceso de datos  
// → GET /hola?user=Paco ...
```

```
app.get('/proc', function(req, res) {  
  // ..... (los datos se procesan)  
  res.send('Gracias por sus datos ' + req.query.user);  
});  
app.listen(8000);
```



Transacción 1: carga el formulario en el cliente

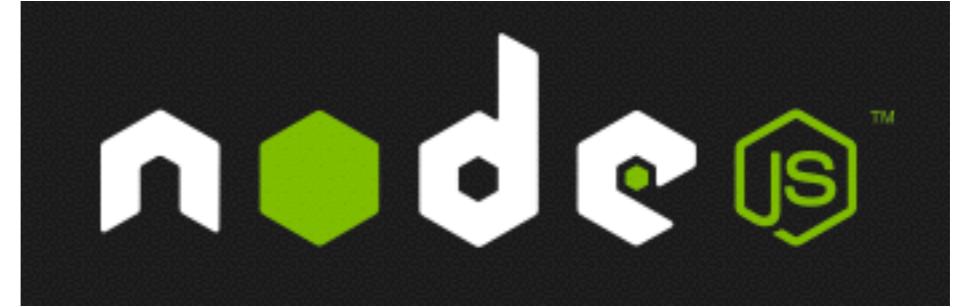
Transacción 2: envía datos al servidor y recibe respuesta.

Los datos de query se guardan en express en propiedades de **req.query** del mismo nombre que cada **<input ... name="user">** del formulario: **req.query.user**





JavaScript



# URL Encode

Juan Quemada, DIT - UPM

# URL or percent encoding

## ◆ Un URL se codifica en UTF-8 para su envío por Internet

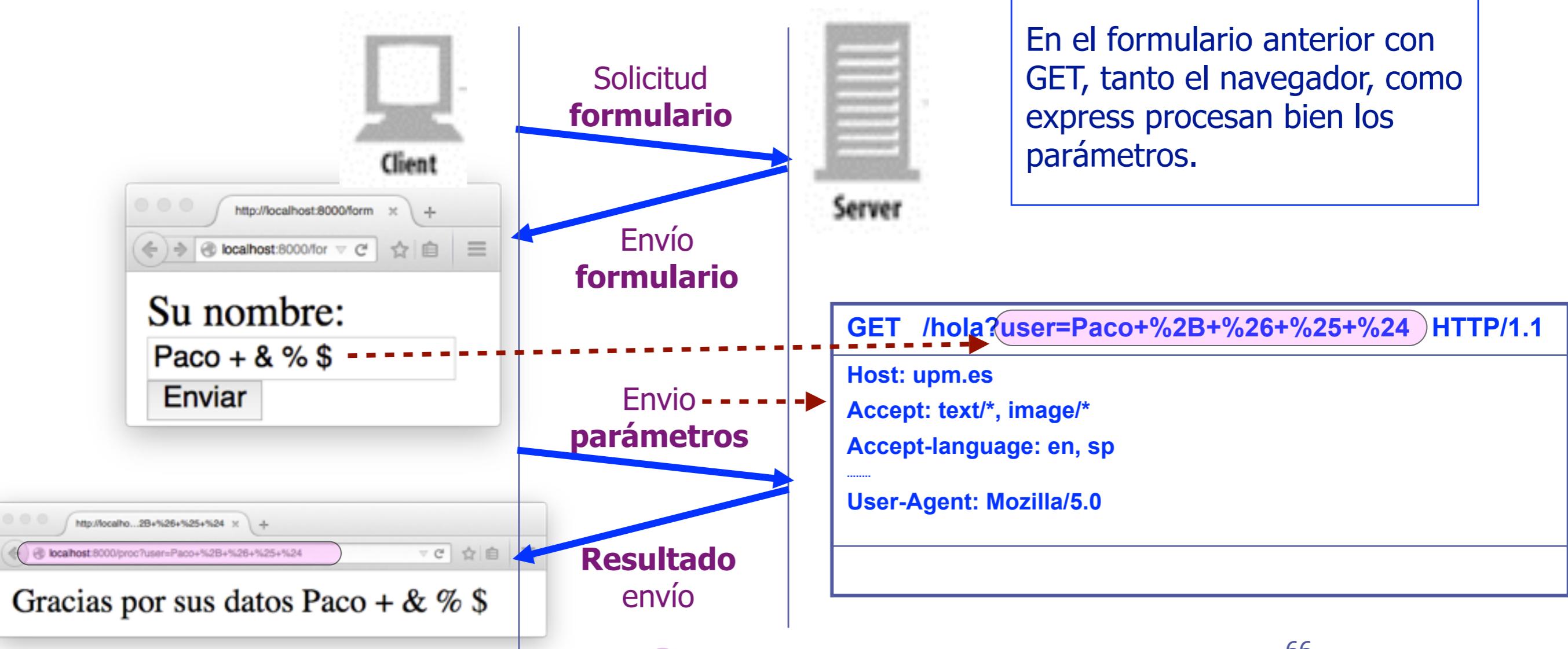
- Utilizando percent o **URL encoding**
  - ◆ Corresponde con tipo MIME: “**application/x-www-form-urlencoded**”
  - **URL encoding** respeta la sintaxis de un URL

## ◆ 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 |

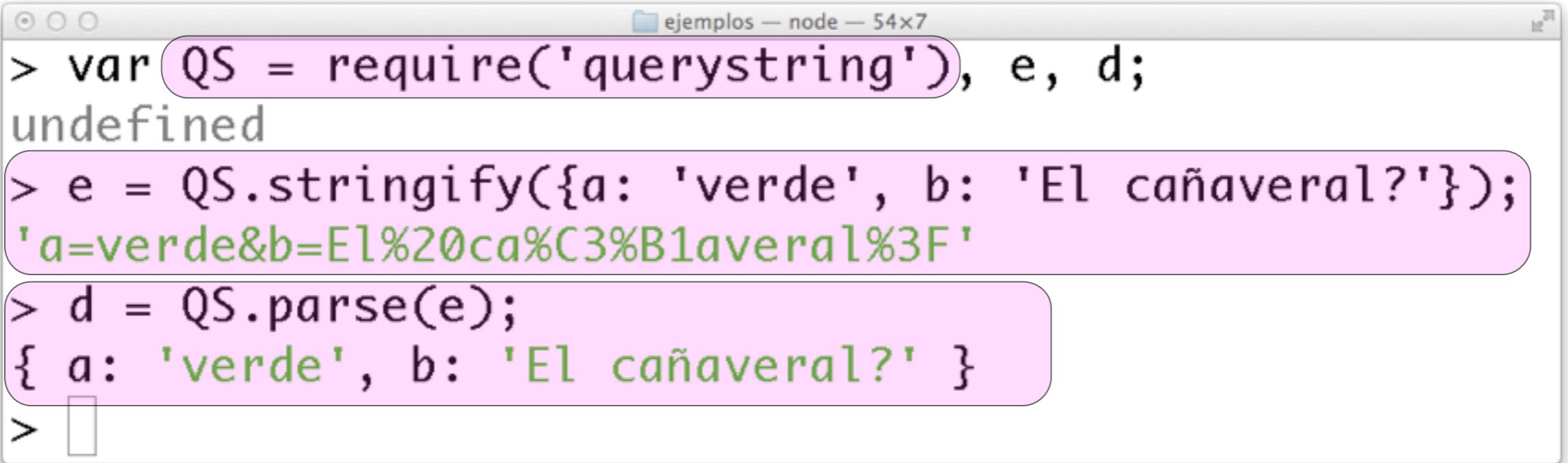
# Envío de parámetros URL encoded en query



# Modulo “querystring” de node.js

## ◆ Codifica y decodifica en formato URL encoded

- Los métodos **stringify** y **parse**
  - ◆ transforman un objeto en un query (string URL encoded) y viceversa
- Doc: <http://nodejs.org/api/querystring.html>



```
> var QS = require('querystring'), e, d;
undefined
> e = QS.stringify({a: 'verde', b: 'El cañaveral?'});
'a=verde&b=El%20ca%C3%B1averal%3F'
> d = QS.parse(e);
{ a: 'verde', b: 'El cañaveral?' }
>
```



JavaScript



# Formulario POST

Juan Quemada, DIT - UPM

# Parámetros en POST

```
POST /hola HTTP/1.1
Host: upm.es
Accept: text/*, image/*
Content-type: application/x-www-form-urlencoded
Content-length: 17
user=Paco+Perez
```

## ◆ Un formulario permite enviar parámetros al servidor

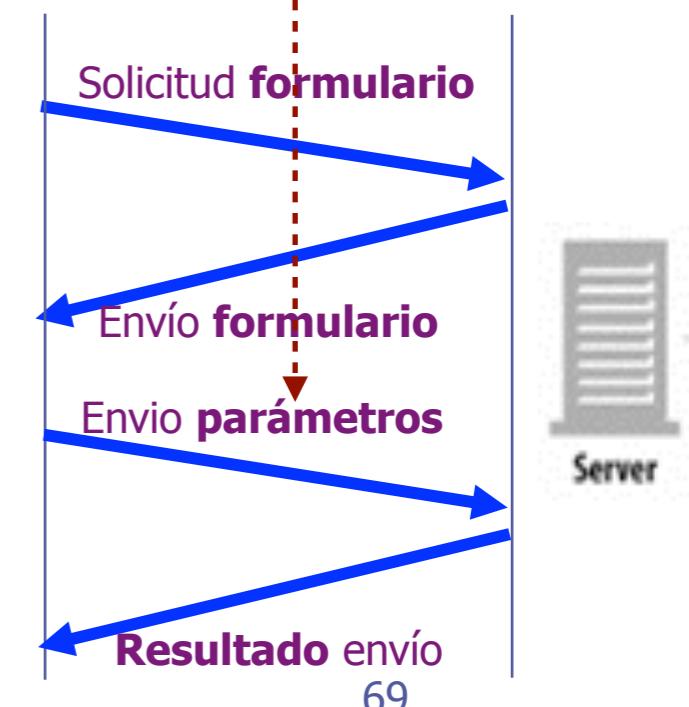
- solo mediante **Solicitudes HTTP** de tipo **GET** y **POST**
  - ◆ Los parametros se teclean en cajetines del formulario
    - Los cajetines se definen con la marca <input ....> y otras

## ◆ POST envía parámetros en el **cuerpo (body)** de la solicitud

- Los parámetros strings en formato URL-encoded similar a la query
  - ◆ Ejemplo de contenido del body: **id=47&user=Paco+Perez**
    - Los parámetros pueden tener cualquier tamaño

## ◆ Un envío de parámetros al servidor

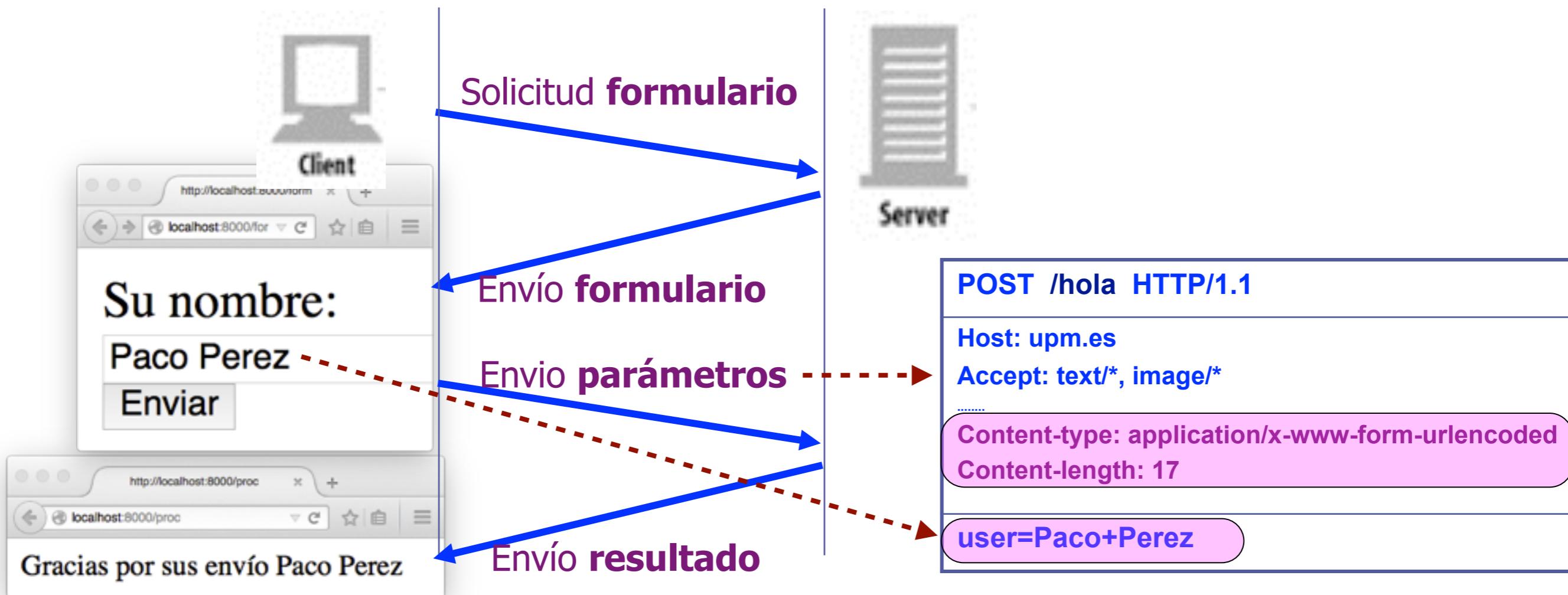
- Suele incluir dos transacciones
  - ◆ Transacción 1: Carga del formulario
  - ◆ Transacción 2: Envío y proceso de datos



# Envío de parámetros en body

El envío de datos de un cliente a un servidor necesita un formulario, por lo que una aplicación de envío de datos suele cargar el formulario antes de realizar el envío.

- Transacción 1: carga de la página Web con el formulario que permite enviar los datos.
- Transacción 2: envío de datos en body con POST desde el formulario, seguida de respuesta.



# Formulario con POST

```
var express = require('express');
var path = require('path');
var bodyParser = require('body-parser');

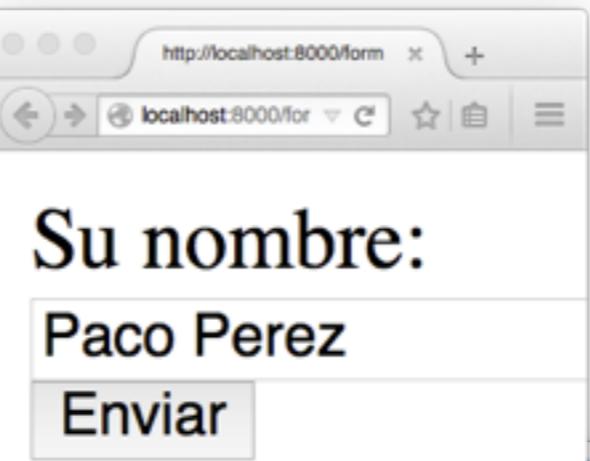
var app = express();

app.use(bodyParser.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')));

app.get('/form', function(req, res){
  res.send('<html><body>
    <form method="post" action="/proc">
      Su nombre: <br>
      <input type="text" name="user" /><br>
      <input type="submit" value="Enviar"/>
    </form>
  </body></html>');
});

app.post('/proc', function(req, res) {
  // ..... (los datos se procesan aquí)
  res.send('Gracias por sus envío ' + req.body.user);
});

app.listen(8000);
```



Los parámetros del formulario se envían en formato **URL encoded** en el **cuerpo (body)** de la respuesta.

Para decodificarlos se debe instalar el paquete **body-parser** de connect y usar su MW **urlencoded()**, que los guarda decodificados en propiedades de **req.body** con el mismo nombre de los parámetros del formulario **<input .. name="user">**: **req.body.user**.

Además se ha instalado el MW **static** visto anteriormente .



# Cabeceras de transacción POST

Gracias por sus envío Paco

Method: POST    proc    localhost:8000

Request URL: http://localhost:8000/proc  
Request method: POST  
Status code: 200 OK

Headers

Response headers (0.159 KB)

- Connection: "keep-alive"
- Content-Length: "27"
- Content-Type: "text/html; charset=utf-8"
- Date: "Tue, 28 Oct 2014 06:01:49 GMT"
- X-Powered-By: "Express"

Request headers (0.330 KB)

- Host: "localhost:8000"
- User-Agent: "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:32.0) Gecko/20100101 Firefox/32.0"
- Accept: "text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8"
- Accept-Language: "en-US,en;q=0.5"
- Accept-Encoding: "gzip, deflate"
- Referer: "http://localhost:8000/form"
- Connection: "keep-alive"

Request headers from upload stream (0.063 KB)

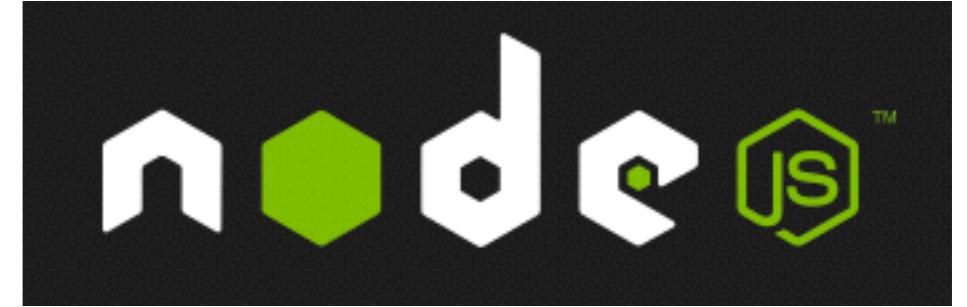
- Content-Type: "application/x-www-form-urlencoded"
- Content-Length: "9"

All    HTML    CSS    JS    XHR    Fonts    Images    Media    Flash    Other    One request, 0.02 KB, 0.00 s    Clear

Esta captura ilustra los parámetros de las cabeceras de la solicitud y la respuesta HTTP POST de la aplicación express anterior.



JavaScript



# Parámetro oculto y method override

Juan Quemada, DIT - UPM

# Parámetro oculto

## ◆ Parámetro oculto

- Parámetro enviado al servidor, que no está visible en la página HTML
  - ◆ Se pueden enviar en el **query** (GET y POST) o en el **body** (POST)

## ◆ Envío en el query de un hiperenlace

- `<a href="/dir/hola.htm?id=47"> Envío de parámetro oculto</a>`

## ◆ Envío en el query de la acción de un POST

- `<form method="POST" action="/dir/hola?id=47"> ..... </form>`

## ◆ Envío en el body de un formulario con marca input, de tipo "hidden"

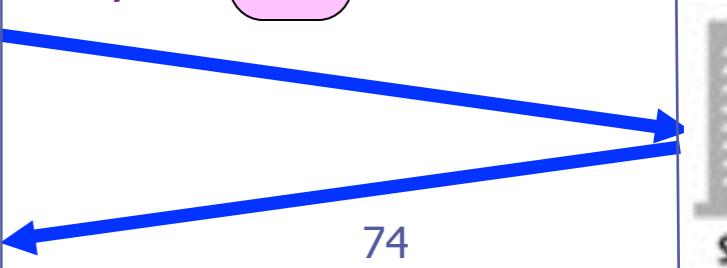
- `<input type="hidden" name="id" value="47">`

The diagram illustrates the flow of data from a code editor to a browser and then to a server. A dashed arrow points from the code editor window to the browser window. A solid blue arrow points from the browser window to the server. The code editor window shows the HTML code for a form with a hidden input field:

```
<!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="Enviár" />
  </form>
</body>
</html>
```



GET /hola?id=47&user=Paco HTTP/1.1



# 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 encapsulan en el parámetro: **\_method**
  - **\_method=PUT** o **\_method=DELETE**
    - ◆ Es una **convención** que indica que **PUT o DELETE van encapsulados POST**
      - ver **method-override** de express: <https://github.com/expressjs/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

# methodOverride()

```
41_mdware_put_encapsulate.js UNREGISTERED
var express = require('express');
var path = require('path');
var bodyParser = require('body-parser');
var methodOverride = require('method-override');

var app = express();

app.use(bodyParser.urlencoded({ extended: true }));
app.use(methodOverride('_method'));
app.use(express.static(path.join(__dirname, 'public')));

app.get('/form', function(req, res){
  res.send(
    '<html><body>
      <form method="post" action="/proc?_method=put">
        Su nombre: <br>
        <input type="text" name="user" /><br>
        <input type="submit" value="Enviar"/>
      </form>
    </body></html>');
});

// method="post" y action="/proc?_method=put" es PUT
app.put('/proc', function(req, res) {
  // ....
  res.send('Hola ' + req.body.user);
});

app.listen(8000);
```



**methodOverride()** se instala con **use()** como otros middlewares y desencapsula **PUT** y **DELETE** encapsuladas dentro de **POST**.

En el ejemplo se envían en body los parámetros (urlencoded):

**\_method=put&user=Paco**

Así las acciones **PUT** o **DELETE** se definen con **put()** y **delete()**, igual que **get()** o **post()**.



# Final del tema