

TEMA 5

FRAMEWORKS EN EL LADO DEL SERVIDOR: EXPRESS.JS

SISTEMAS WEB / APLICACIONES WEB



Esta obra está bajo una
Licencia CC BY-NC-SA 4.0 Internacional.

Iván Martínez
DSIA

Facultad de Informática
Universidad Complutense de Madrid

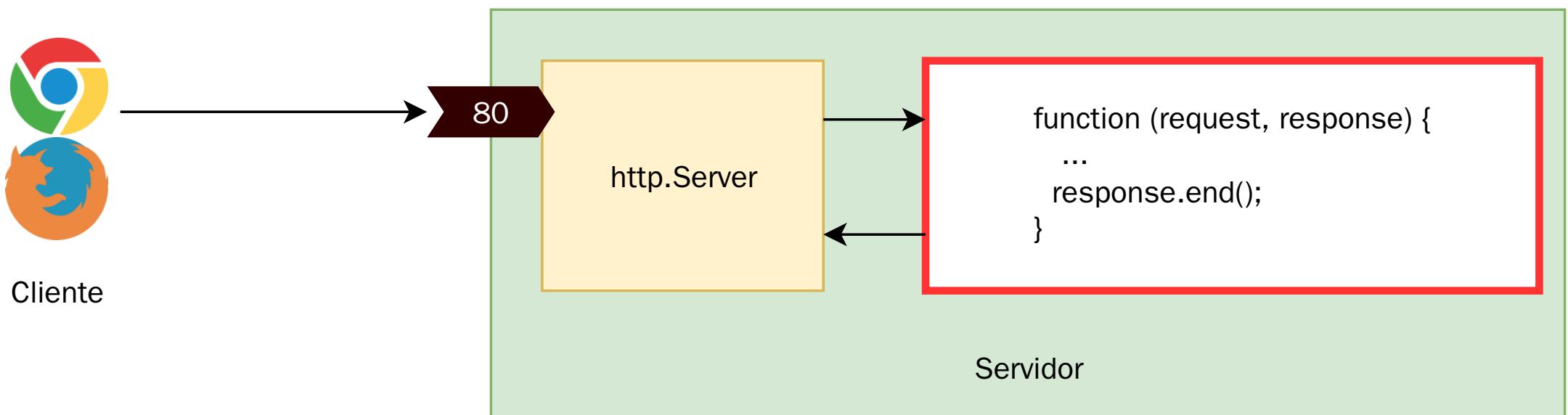
Manuel Montenegro
DSIC

Facultad de Informática
Universidad Complutense de Madrid

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

INTRODUCCIÓN

Conocemos el módulo `http`, que implementa la funcionalidad de un servidor web.



La función callback distingue casos en función de la URL indicada por el cliente en su petición.

```
const servidor = createServer(function(request, response) {
  var method = request.method;
  var url = request.url;
  if (method === "GET" && url === "/index.html") {
    // Servir página principal.
  } else if (method === "GET" && url === "/index.css") {
    // Servir hoja de estilo.
  } else if (...) {
    // ...
    // ...
    // ...
  } else {
    response.statusCode = 404;
  }
});
```

Desventaja: código difícil de mantener.

El módulo `http` resulta insuficiente en aplicaciones web grandes, que normalmente requieren:

- Manejo de cookies
- Validación de formularios
- Gestión de páginas web estáticas
- Registro de peticiones (`logging`)
- Gestión de las vistas de una aplicación (`templates`)

FRAMEWORKS WEB

Un **marco de aplicaciones web** (*web framework*) es un sistema, generalmente en forma de librería, que facilita el desarrollo de aplicaciones web mediante:

- Separación entre la vista y controlador.
 - **Vista**: documento HTML generado.
 - **Controlador**: función *callback* que procesa las peticiones.
- División de un controlador monolítico en distintos mini-controladores.
- Abstracción de aspectos complejos: envío de ficheros, cookies, etc.

EJEMPLOS DE FRAMEWORKS WEB

(en el lado del servidor)

- **Java**: Spring MVC, SpringBoot, Jakarta Faces, etc.
- **Python**: Django.
- **Ruby**: Ruby on Rails.
- **Node**:
 - Express.js - <http://expressjs.com>
 - Sails.js - <http://sailsjs.org>
 - Meteor.js - <http://www.meteor.com>

EXPRESS.JS

Es un framework basado en el módulo `http` que proporciona:

- Posibilidad de dividir la función callback que gestiona las peticiones HTTP en varias fases.
- Mecanismos de alto nivel para acceder a algunas componentes de la petición (cookies, IP del cliente, etc).

Principales características:

- **Flexible**: no impone una determinada estructura.
- **Modular**: basado en concatenar *middleware*.
- **Minimalista**: solo proporciona funcionalidad básica, que puede extenderse mediante librerías externas.

El minimalismo de Express.js responde a la filosofía UNIX:

Write programs that do one thing and do it well.

Es raro el uso de Express.js sin ninguna librería adicional.

Ventajas

- Eficiencia: no hay componentes innecesarios.
- Simplicidad: es fácil comprender el funcionamiento.
- Flexibilidad: componentes intercambiables.

Inconvenientes

- Cantidad abrumadora de componentes externos.
- Requiere tomar decisiones de diseño.

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

PRIMERA APLICACIÓN CON EXPRESS.JS

Creamos un proyecto nuevo y añadimos `express` como dependencia:

```
# npm init  
...  
# npm install express
```

Escribimos un programa `main.js` que comienza del siguiente modo:

```
// main.js
// -------

import express from 'express';
const app = express();
// ...
```

El módulo `express` exporta una única función.

Cada llamada a esta función devuelve una **aplicación**. Una aplicación es un servidor HTTP que escucha en un determinado puerto.

A continuación se definen los **manejadores de ruta**, que especifican las acciones del servidor para cada URL:

`app.get (URL, callback)`

Cuando se reciba una petición de tipo GET sobre la *URL* pasada como parámetro, se llamará a la función *callback*, que será la que gestione la petición.

```
app.get("/", (request, response) => {
  response.statusCode = 200;
  response.setHeader("Content-Type", "text/html");
  response.write("Esta es la página raíz");
  response.end();
});
```

Los objetos `request` y `response` son del mismo tipo que los del módulo `http`, pero con algunos métodos más. [+]

- `response.status (codigo)`
Especifica el código HTTP de respuesta.
- `response.type (codigo)`
Especifica el tipo MIME del cuerpo de la respuesta.
- `response.set (clave, valor)`
Modifica las cabeceras de la respuesta.
- `response.write (cadena)`
Escribe en el cuerpo de la respuesta.
- `response.end ([cadena])`
Envía señal de finalización de respuesta.

La ruta anterior podía haberse escrito del siguiente modo:

```
app.get("/", function(request, response) {  
    response.status(200);  
    response.type("text/plain; charset=utf-8");  
    response.end("Esta es la página raíz");  
});
```

Es conveniente indicar el tipo MIME de la respuesta (`text/plain`), para que el navegador web sepa qué hacer con el fichero recibido.

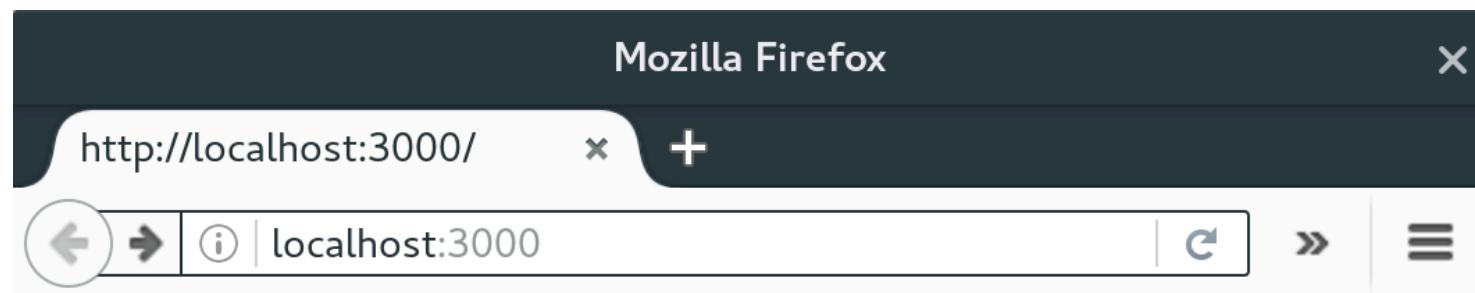
Añadimos otra ruta para gestionar la URL `/users.html`:

```
app.get("/users.html", (request, response) => {
  response.status(200);
  response.type("text/plain; charset=utf-8");
  response.end("Aquí se mostrará la página de usuarios");
});
```

Por último, se llama al método `listen()`, que funciona igual que su homónimo en `http:`

```
app.listen(3000, (err) => {
  if (err) {
    console.error("No se pudo inicializar el servidor: "
      + err.message);
  } else {
    console.log("Servidor arrancado en el puerto 3000");
  }
});
```

Resultado:



Esta es la página raíz

Respuesta HTTP devuelta:

```
HTTP/1.1 200 OK      response.status(200)
```

```
X-Powered-By: Express
```

```
Content-Type: text/plain; charset=utf-8
```

```
response.type(...)
```

```
Date: Tue, 08 Nov 2016 21:07:14 GMT
```

```
Connection: keep-alive
```

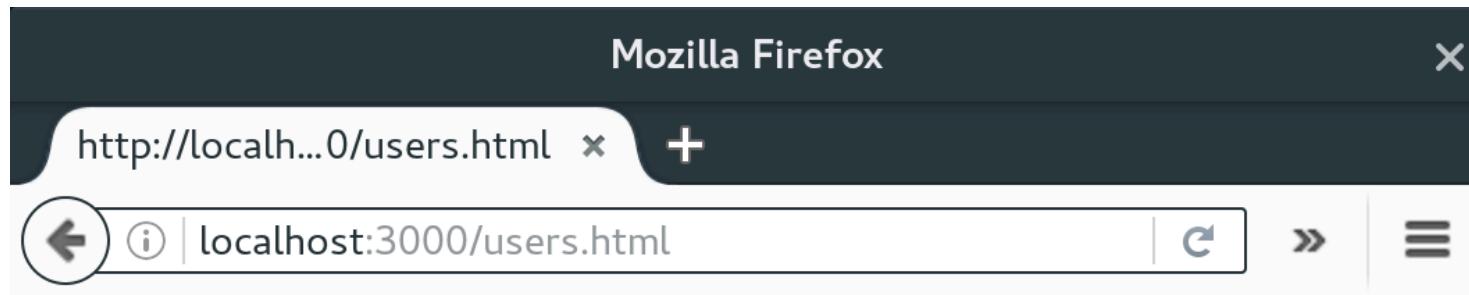
```
Content-Length: 24
```

```
Esta es la página raíz
```

```
response.write(...) / response.end(...)
```

Comprobamos el funcionamiento de la segunda ruta:

`http://localhost:3000/users.html`



Aquí se mostrará la página de usuarios

Mediante la cabecera `Content-Disposition` podemos indicar al navegador que el contenido es para ser descargado, no para ser visualizado en navegador.

```
app.get("/users.html", function(request, response) {  
  // ...  
  response.set("Content-Disposition", "attachment");  
  // ...  
});
```



http://localhost:3000/



Abriendo users.html



Esta es

Ha elegido abrir:

users.html

que es: documento de texto sencillo (41 bytes)

de: http://localhost:3000

¿Qué debería hacer Firefox con este archivo? Abrir con

Editor de textos (predeterminada)

 Guardar archivo Hacer esto automáticamente para estos archivos a partir de ahora.[Cancelar](#)[Aceptar](#)

Existen funciones de más alto nivel para devolver un fichero al cliente:

- `response.sendFile(path)`

Devuelve al cliente el fichero indicado.

- `response.download(path)`

Devuelve al cliente el fichero indicado, utilizando la cabecera `Content-Disposition: attachment`.

```
app.get("/usuario/imagen.png", function(request, response) {  
    response.status(200);  
    response.sendFile("./path/fichero.png");  
});
```

Estas funciones detectan el tipo MIME del fichero enviado.

REDIRECCIONES HTTP

Se envían mediante `response.redirect(url)`:

```
app.get("/usuarios.html", function(request, response) {  
  response.redirect("/users.html");  
});
```

Al acceder a `http://localhost:3000/usuarios.html` se obtendrá la siguiente respuesta HTTP,

```
HTTP/1.1 302 Found  
X-Powered-By: Express  
Location: /users.html  
...
```

Código 302 ⇒ Redirección

y el navegador «saltará» automáticamente a `users.html`.

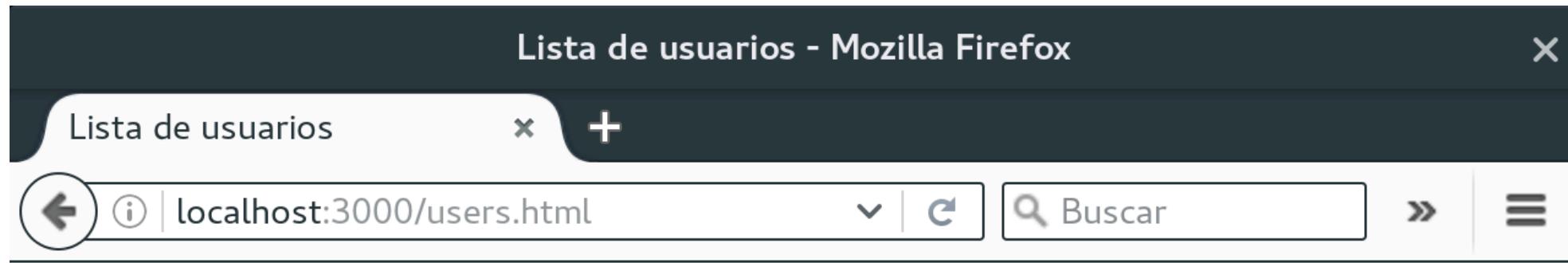
CONTENIDO HTML

Al igual que en el módulo `http`, el objeto `response` puede utilizarse para generar páginas HTML.

```
const usuarios = ["Javier Montoro", "Dolores Vega", "Beatriz Nito"];

app.get("/users.html", (request, response) => {
  response.status(200);
  response.type("text/html");
  response.write("<html>");
  response.write("<head>");
  response.write("<title>Lista de usuarios</title>");
  response.write('<meta charset="utf-8">');
  response.write("</head>");
  response.write("<body><ul>");
  usuarios.forEach((usuario) => {
    response.write(`<li>${usuario}</li>`);
  });
  response.write("</ul></body>");
  response.end("</html>");
});
```

<http://localhost:3000/users.html>



- Javier Montoro
- Dolores Vega
- Beatriz Nito

Hemos generado dinámicamente una página HTML mediante sucesivas llamadas a `response.write()`.

Si tuviésemos que hacerlo con una página más extensa, el código Javascript sería tedioso de escribir e inmantenible.

Esto no solo ocurre con páginas HTML, sino con otro tipo de recursos: hojas de estilo, imágenes, etc.

Vamos a mejorar el modo de enviar páginas al cliente...

PÁGINAS ESTÁTICAS Y DINÁMICAS

Las páginas **estáticas** son aquellas cuyo contenido es «fijo», en tanto que no dependen de la petición HTTP realizada, o de otros recursos (por ejemplo, BD).

```
app.get("/", (request, response) => {
  response.status(200);
  response.type("text/html");
  response.write("<html>");
  response.write("<head>");
  response.write("<title>Lista de usuarios</title>");
  response.write('<meta charset="utf-8">');
  response.write("</head>");
  response.write("<body>");
  response.write("<h1>¡Bienvenido!</h1>");
  response.write("</body>");
  response.end("</html>");
});
```

Las páginas estáticas **no** suelen devolverse al cliente mediante llamadas a `response.write()`.

Normalmente se almacenan como archivos independientes en el servidor, y se devuelven al cliente tal y como están almacenadas (sin modificaciones).

Esto se aplica a las hojas de estilo, imágenes, etc. de una página, que también se consideran recursos estáticos.

Por ejemplo, si tenemos un fichero `bienvenido.html` con el siguiente contenido:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Lista de usuarios</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>¡Bienvenido!</h1>
  </body>
</html>
```

Podríamos hacer:

```
app.get("/", (request, response) => {
  response.sendFile("bienvenido.html");
});
```

Existe una forma aún más sencilla de realizar esto: el middleware `static`.
(ver apartado 4: Middleware)

Las páginas **dinámicas** sí dependen de la petición realizada o de otros recursos (p.ej. variables de programa o BD).

Por ello han de ser generadas total o parcialmente mediante código Javascript.

```
let usuarios = ...;

app.get("/users.html", (request, response) => {
  // ...
  usuarios.forEach((usuario) => {
    response.write(`<li>${usuario}</li>`);
  });
  // ...
});
```

Aun así, las páginas generadas dinámicamente tienen muchas partes que son estáticas:<head>, encabezados, pies de página, referencias a hojas de estilo, etc.

Por ello, tampoco suele utilizarse `response.write()` directamente para generar páginas dinámicas.

Existe una alternativa mejor: **plantillas**.

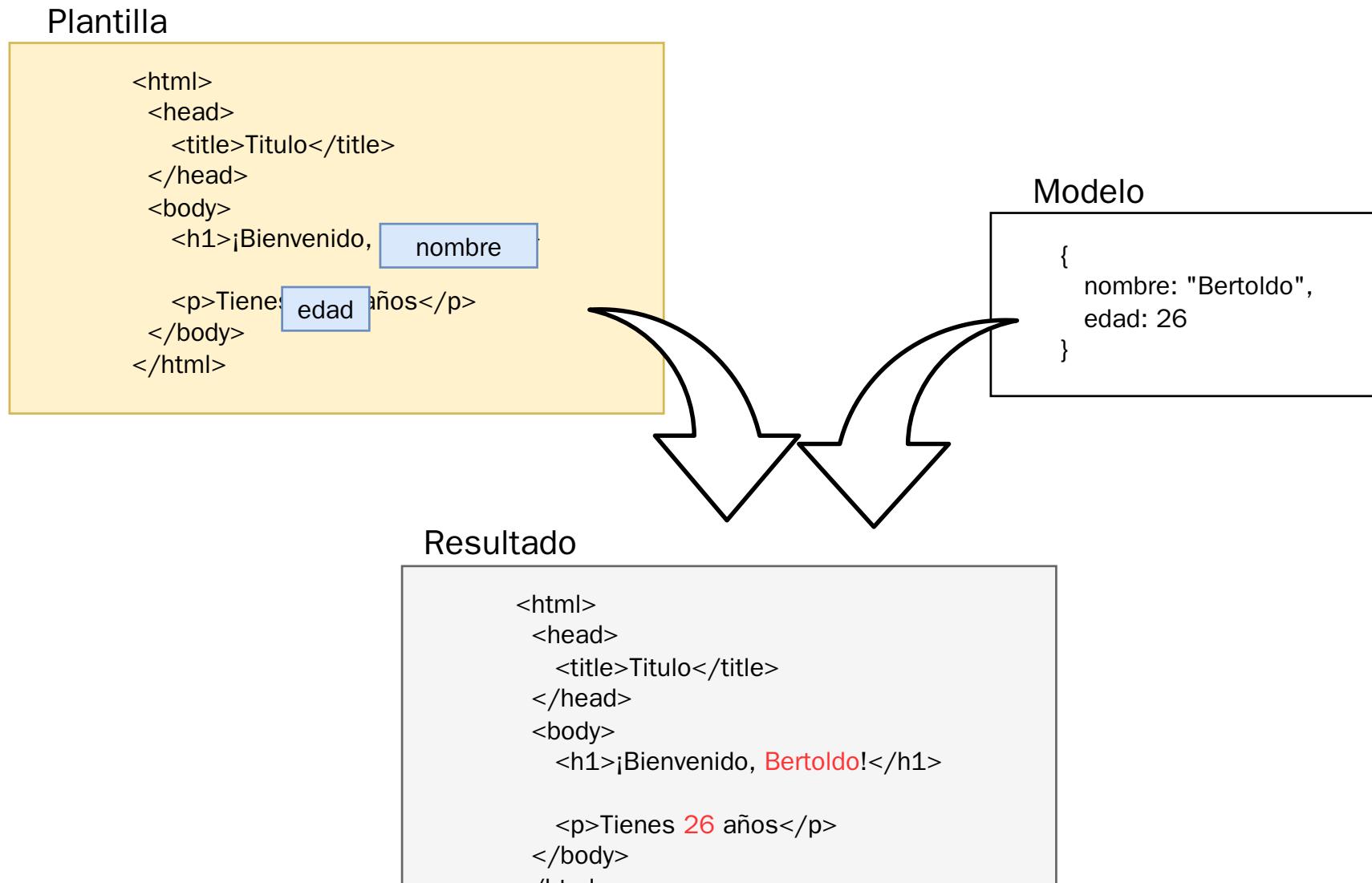
1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

PROCESADORES DE PLANTILLAS

La tarea de generar el documento HTML directamente en el código Javascript, incluso en páginas dinámicas, es tediosa.

Aunque un documento HTML puede tener contenido generado dinámicamente, una buena parte del documento consiste en código invariable: Etiquetas `<html>`, `<head>`, `<body>`, encabezado de la página, etc.

Una **plantilla** es un documento HTML con «huecos». Un **procesador de plantillas** se encarga de colocar el contenido dinámico en estos huecos.



PROCESADORES DE PLANTILLAS EN NODE

Diversos motores de plantillas:

- Mustache.js - <http://mustache.github.io>
- Pug (aka. Jade) - <https://pugjs.org>
- Handlebars - <http://handlebarsjs.com>
- EJS - <http://ejs.co/> Esta asignatura

EJS

Es un paquete externo que se instala mediante `npm`:

```
npm install ejs
```

Las plantillas de EJS son documentos HTML con marcadores especiales de varios tipos:

- Marcadores de **programa**, delimitados por `<%` y `%>`
- Marcadores de **expresión**, delimitados por `<%=` y `%>`

Los marcadores de expresión evalúan la expresión Javascript dada, la convierten en cadena, y se reemplazan por dicha cadena.

```
<h1>¡Bienvenido, <%= usuario.nombre %>!</h1>
```

Los marcadores de programa contienen sentencias (o fragmentos de sentencias) Javascript. Suelen utilizarse con bucles, condicionales, etc.

```
<% if (!usuario.nombre) { %>
<p>No estás identificado.</p>
<% } else { %>
<p>¡Bienvenido, <%= usuario.nombre %>!</p>
<% } %>
```

Volviendo a nuestro ejemplo:

```
const usuarios = ["Javier Montoro", "Dolores Vega", "Beatriz Nito"];  
  
app.get("/users.html", (request, response) => {  
    response.status(200);  
    response.type("text/html");  
    response.write("<html>");  
    response.write("<head>");  
    response.write("<title>Lista de usuarios</title>");  
    response.write('<meta charset="utf-8">')  
    response.write("</head>");  
    response.write("<body><ul>");  
    usuarios.forEach((usuario) => {  
        response.write(`<li>${usuario}</li>`);  
    });  
    response.write("</ul></body>");  
    response.end("</html>");  
});
```

Creamos un fichero `views/users.ejs`:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Lista de usuarios</title>
    <meta charset="utf-8">
  </head>
  <body>
    <ul>
      <% users.forEach(user => { %>
        <li><%= user %></li>
        <% } ); %>
      </ul>
    </body>
</html>
```

EXPRESS.JS Y LAS PLANTILLAS

Express.js soporta varios procesadores de plantillas: Jade, Mustache, **EJS**, etc.

Para utilizar EJS es necesario configurar previamente el objeto aplicación mediante el método `set()`.

Opciones a configurar:

- `view engine`: motor de plantillas a utilizar.
- `views`: directorio con las plantillas.

```
import { join } from "node:path";
import express from "express";

const app = express();
```

Utilizamos EJS como motor de plantillas:

```
app.set("view engine", "ejs");
```

El directorio con las plantillas se llamará `views` y estará situado debajo del directorio en el que se encuentra el fichero actual:

```
app.set("views", join(import.meta.dirname, "views"));
// Recuerda: import.meta.dirname = Directorio del script actual
//               join(dir1, dir2, ...) = "dir1/dir2/..."
```

La llamada `response.render()` se encarga de llamar al motor de plantillas y devolver el resultado al cliente.

Recibe dos parámetros:

- Una cadena con el nombre de la vista. Buscará el fichero correspondiente en la carpeta de plantillas.
- El modelo a visualizar. Los «huecos» de la plantilla se llenan con los atributos del modelo.

```
const usuarios = ["Javier Montoro", "Dolores Vega", "Beatriz Nito"];

app.get("/users.html", function(request, response) {
  response.status(200);
  response.render("users", { users: usuarios });
  // Busca la plantilla "views/users.ejs"
  // La variable 'users' que hay dentro de esta plantilla tomará
  // el valor del array usuarios.
});
```

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

MIDDLEWARE

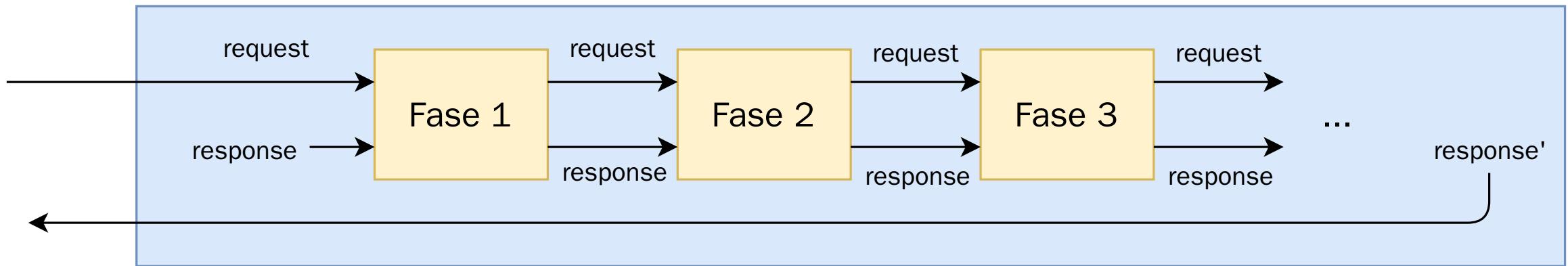
Recordemos el manejo de peticiones utilizando directamente el módulo `http`:

```
const server = createServer(function(request, response) { ... });
```

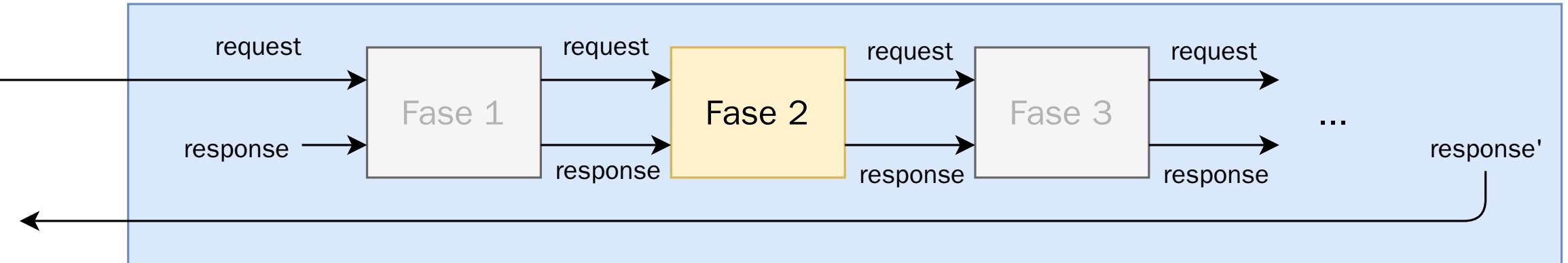
Se gestionan las peticiones mediante una única función monolítica que manipula el argumento `response` para indicar qué respuesta se quiere enviar:



Express.js se basa en dividir esta función monolítica en varias fases:



Cada una de estas fases recibe el nombre de **middleware**.



Un middleware recibe un objeto `request` con los datos de la petición y un objeto `response` con la respuesta acumulada hasta el momento. Durante su ejecución puede:

- **Leer y/o modificar** el objeto `request` (por ejemplo, añadiendo nuevos atributos).
- **Leer y/o modificar** el objeto `response` (por ejemplo, para añadir un fragmento a la respuesta).

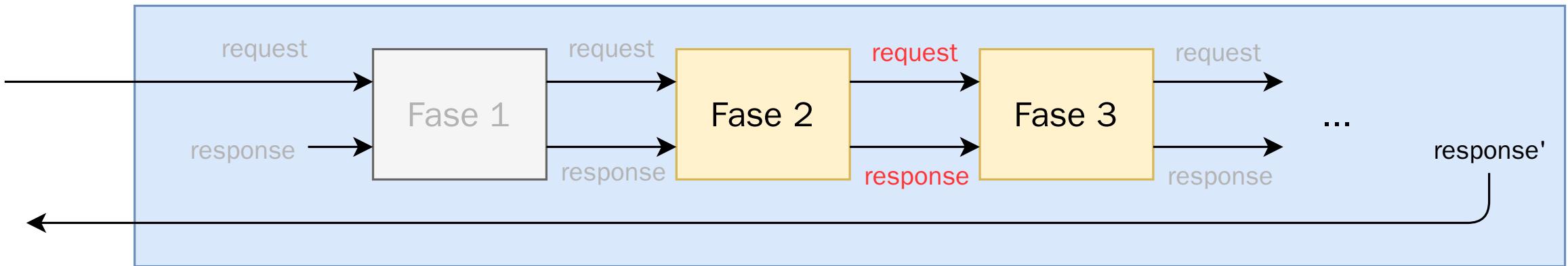
La ejecución de un middleware puede finalizar de tres formas distintas:

1. Pasando el control al **siguiente** middleware en la cadena.
2. **Finalizando** la cadena de middlewares, sin pasar el control al siguiente.
3. Provocando un **error**, que será gestionado por otro middleware.

Pasamos a analizar los dos primeros casos.

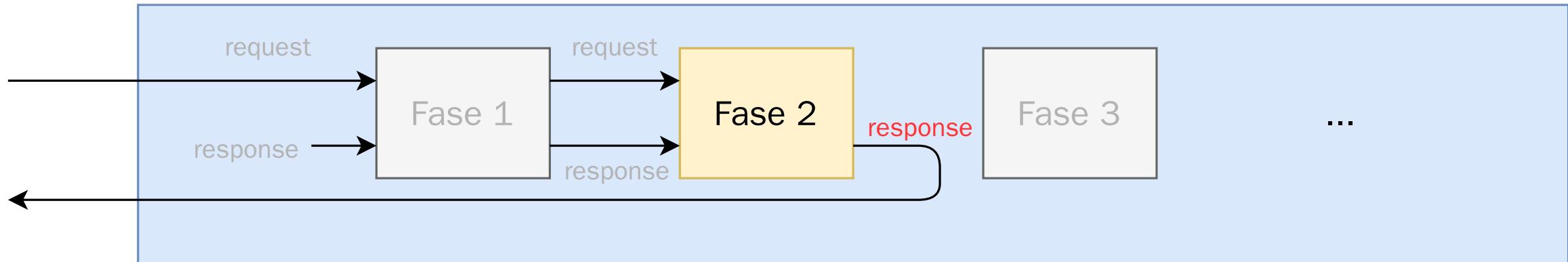
El último se describirá más adelante.

CASO 1: PASAR EL CONTROL AL SIGUIENTE MIDDLEWARE



El middleware «destino» recibe los objetos `request` y `response` tal y como el middleware «origen» los dejó al finalizar su ejecución.

CASO 2: FINALIZAR LA CADENA



Si un middleware decide no invocar al siguiente, la cadena finaliza y la petición se considera atendida.

En este caso, el middleware deberá haber llamado a `response.end()` o `response.redirect()` para devolver una respuesta al cliente.

MÉTODOS TERMINALES DE `response`

Son métodos que dan la respuesta por finalizada.

- `end()`
- `render()`
- `download()`
- `sendFile()`
- `redirect()`
- `json()`

En cada petición solo es posible llamar una vez a alguno de estos métodos.

UTILIZAR UN MIDDLEWARE

La programación de un servidor con Express.js se basa en el encadenamiento de middlewares.

Si `app` representa un objeto aplicación, el método `app.use(...)` añade un middleware a la cadena.

```
import express from "express";
const app = express();

app.use(middleware_1);
app.use(middleware_2);
app.use(middleware_3);

// ...

app.listen(3000);
```



CONSTRUIR UN MIDDLEWARE

Un middleware es, básicamente, una función con tres parámetros: `request`, `response` y `next`.

El parámetro `next` es una función. Cuando el middleware quiera transferir el control al siguiente de la cadena deberá llamar a `next ()` sin parámetros.

```
function miMiddleware(request, response, next) {  
  // ...  
  // Manipular los objetos request y/o response.  
  // ...  
  
  next(); // Saltar al siguiente middleware  
}
```

EJEMPLO 1: REGISTRO DE PETICIONES (LOG)

Creamos un middleware que se limita a mostrar por pantalla las peticiones recibidas, sin alterarlas.

Este tipo de middleware suele ir al principio de la cadena.

```
function logger(request, response, next) {
  console.log(`Recibida petición ${request.method} ` +
    `en ${request.url} de ${request.ip}`);
  // Saltar al siguiente middleware
  next();
}

app.use(logger);
```

Otra forma: utilizar `app.use()` pasando el middleware como una función anónima.

```
app.use((request, response, next) => {
  console.log(`Recibida petición ${request.method} ` +
    `en ${request.url} de ${request.ip}`);
  next();
});
```

EJEMPLO 2: CONTROL DE ACCESO

El siguiente middleware deniega todas las peticiones que provengan de una IP censurada.

```
let ipsCensuradas = [ "147.96.81.244", "145.2.34.23" ];

app.use((request, response, next) => {
    // Comprobamos si la IP de la petición está dentro de la
    // lista de IPs censuradas.
    if (ipsCensuradas.indexOf(request.ip) >= 0) {
        // Si está censurada, devolvemos código 401 (Unauthorized)
        response.status(401);
        response.end("No autorizado");
    } else {
        // En caso contrario, pasamos al siguiente middleware.
        next();
    }
});
```

También puede utilizarse middleware para comprobar si el usuario está identificado y, en caso contrario, redirigir a una página de identificación.

```
app.use((request, response, next) => {
  if /* el usuario actual no está identificado */) {
    response.redirect("/login.html");
  } else {
    next();
  }
});
```

EJEMPLO 3: ADJUNTAR INFORMACIÓN A LA PETICIÓN

El siguiente middleware extiende la petición con un atributo `esUCM` que indica si la IP del usuario es de la forma `147.96.x.x`

```
app.use((request, response, next) => {
  request.esUCM = request.ip.startsWith("147.96.");
  next();
});
```

Colocamos tres middlewares anteriores en secuencia, seguidos de un manejador para la ruta `/index.html`:

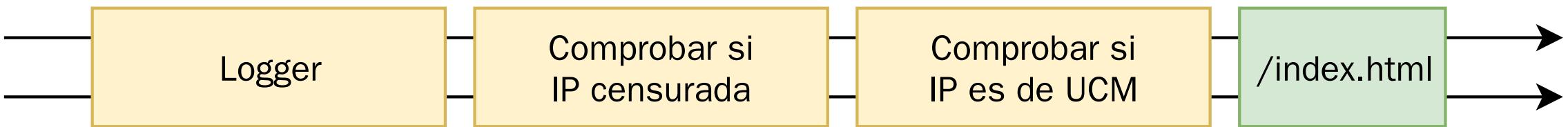
```
const express = require("express");
const app = express();

let ipsBloqueadas = [ ... ];

app.use( ... /* logger */ ...);
app.use( ... /* ip bloqueada? */ ...);
app.use( ... /* ip ucm? */ ...);

app.get("/index.html", (request, response) => {
    response.status(200);
    response.type("text/plain; encoding=utf-8");
    response.write("¡Hola!");
    if (request.esUCM) {
        response.write("Estás conectado desde la UCM");
    }
    response.end();
});
```

Hasta ahora tenemos la siguiente cadena:



El manejador de ruta `/index.html` puede considerarse como un middleware que solo se ejecuta si la URL de la petición es `/index.html`. En caso contrario «salta» al siguiente middleware.

EJEMPLO 4: ERROR 404 (NOT FOUND)

Al final de la cadena suele colocarse un middleware que gestione los casos en los que la URL no haya sido capturada por ningún manejador anterior.

Esta middleware devuelve el código de error 404.

```
app.use((request, response, next) => {
  response.status(404);
  response.render("error", { url: request.url });
});
```

views/error.ejs

```
<html>
  <head><title>ERROR 404</title><meta charset="UTF-8"></head>
  <body>
    <h1>Error 404</h1>
    <p>La dirección <code><%= url %></code> no existe</p>
  </body>
</html>
```



MIDDLEWARE BÁSICO

- `static` (viene incluido con Express.js)
- `morgan`
- `compress`

static: SERVIR FICHEROS ESTÁTICOS

Recordatorio: [páginas web estáticas y dinámicas](#)

Los recursos estáticos (imágenes, páginas web estáticas, hojas de estilo) suelen almacenarse en una carpeta dentro del servidor.

Cuando se recibe una petición GET para acceder a alguno de estos recursos estáticos, se lee el fichero correspondiente y se envía su contenido en la respuesta.

El middleware `static` se encarga de todo esto.

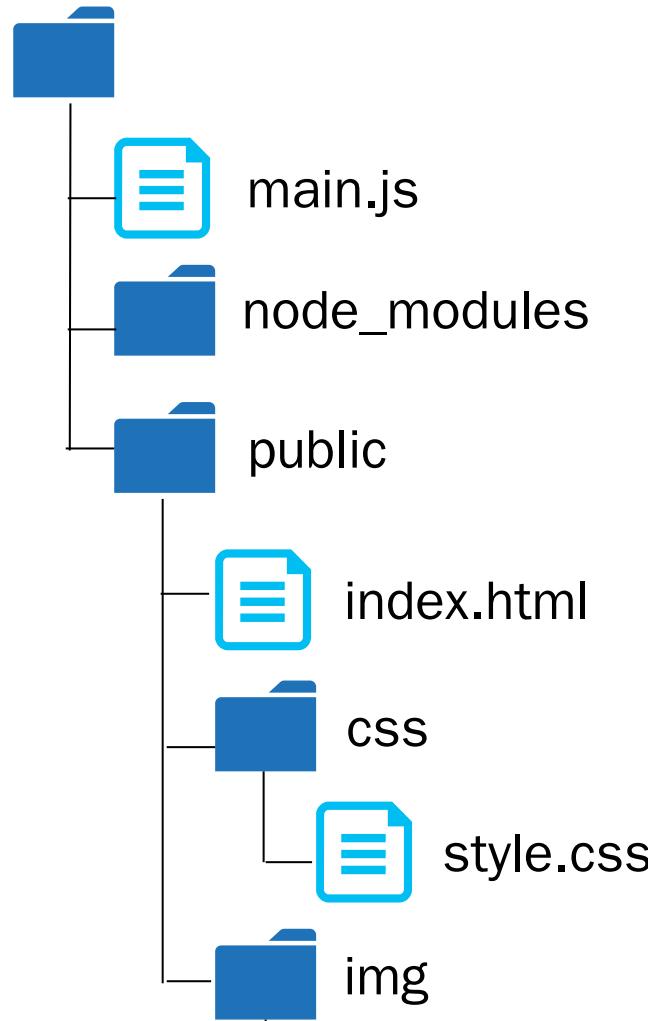
```
express.static(root)
```

Recibe un nombre de directorio `root` y devuelve el middleware que realiza (a grandes rasgos) lo siguiente:

- Analiza `request.url` y comprueba si coincide con algún fichero del directorio `root`.
- En caso de existir, devuelve su contenido mediante `sendFile()`.
- En caso de no existir, pasa al siguiente middleware.

EJEMPLO DE USO

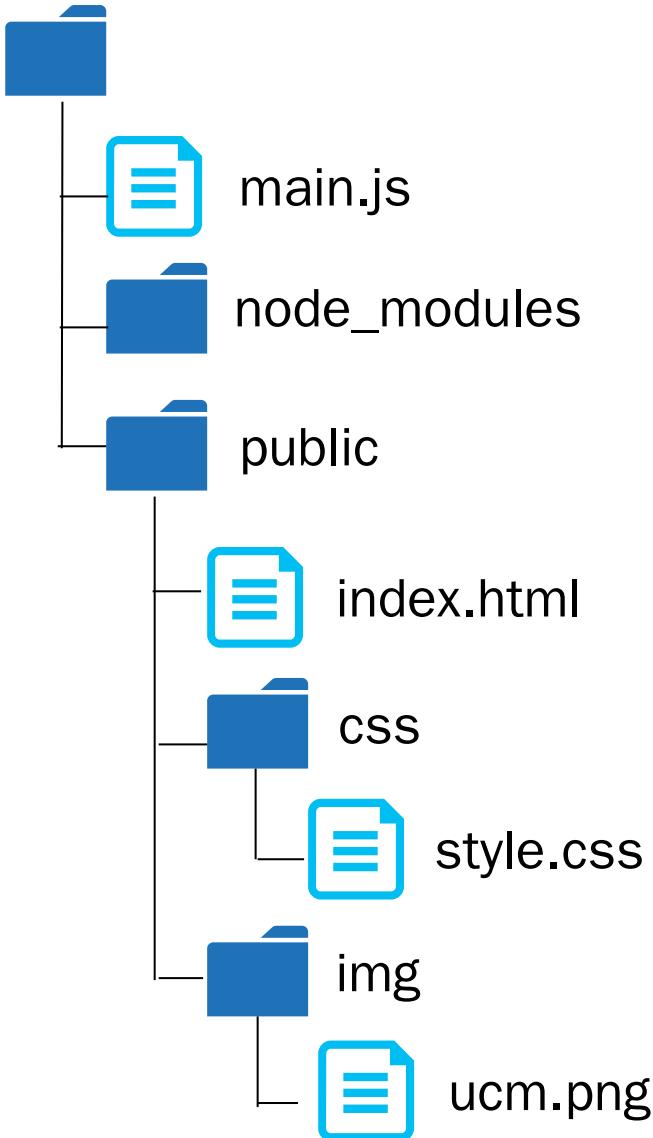
Los recursos estáticos suelen almacenarse en un directorio llamado `public`, que se sitúa dentro del proyecto.



`public/index.html`

```
<!DOCTYPE html>
<html>
  <head>
    <title>Middleware estático</title>
    <link rel="stylesheet" href="css/style.css">
    <meta charset="utf-8">
  </head>
  <body>
    <h1>¡Bienvenido!</h1>
    <p>Esto es una página web estática</p>
    
  </body>
</html>
```

main.js



```
import express from "express";
import { join } from "node:path";

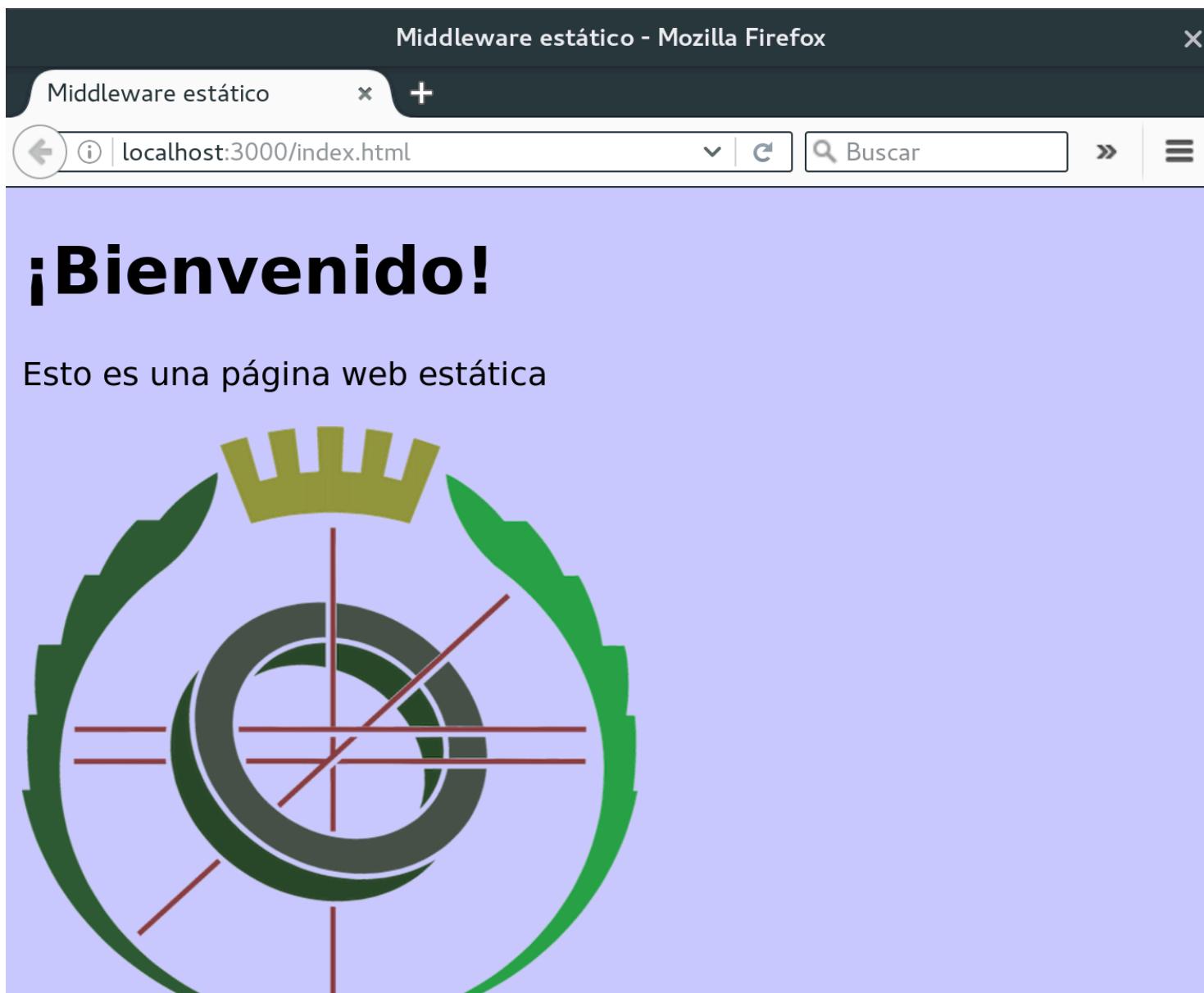
const app = express();

// La variable ficherosEstaticos guarda el
// nombre del directorio donde se encuentran
// los ficheros estáticos:
// <directorioProyecto>/public
const ficherosEstaticos =
    path.join(import.meta.dirname, "public");

app.use(express.static(ficherosEstaticos));

app.listen(3000, () => {
    console.log("Escuchando en el puerto 3000");
})
```

Resultado:



morgan: REGISTRO DE PETICIONES

Anteriormente hemos implementado un middleware sencillo que escribía por pantalla las peticiones recibidas.

El middleware **morgan** permite hacer lo mismo, pero proporciona muchas más opciones.

Instalación en el proyecto:

```
npm install morgan
```

<https://github.com/expressjs/morgan>

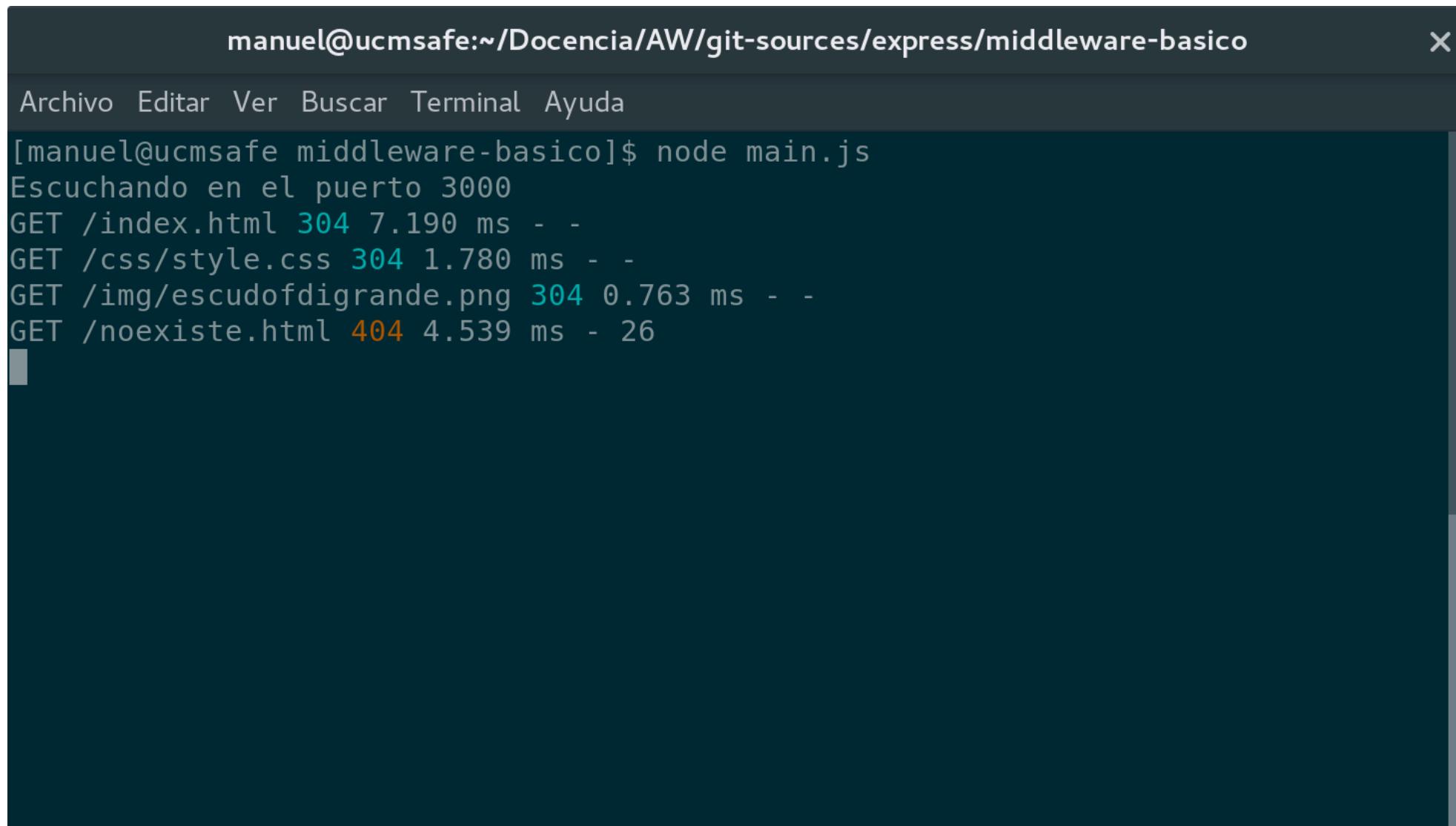
El módulo `morgan` exporta una única función que recibe el tipo de información a imprimir y devuelve un middleware.

```
import morgan from "morgan";  
...  
app.use(morgan("dev"));
```

Tipos disponibles: `dev`, `combined`, `common`, `short`, `tiny`.

<https://github.com/expressjs/morgan#predefined-formats>

Al realizar distintas peticiones con el navegador se imprime esta información por pantalla:



A screenshot of a terminal window titled "manuel@ucmsafe:~/Docencia/AW/git-sources/express/middleware-basico". The window has a dark background and light-colored text. At the top, there's a menu bar with options: Archivo, Editar, Ver, Buscar, Terminal, and Ayuda. Below the menu, the terminal prompt is "[manuel@ucmsafe middleware-basico]\$" followed by the command "node main.js". The output shows the server listening on port 3000 and processing four requests: GET /index.html (status 304), GET /css/style.css (status 304), GET /img/escudofdigrande.png (status 304), and GET /noexiste.html (status 404). The status codes are highlighted in blue.

```
manuel@ucmsafe:~/Docencia/AW/git-sources/express/middleware-basico
Archivo Editar Ver Buscar Terminal Ayuda
[manuel@ucmsafe middleware-basico]$ node main.js
Escuchando en el puerto 3000
GET /index.html 304 7.190 ms - -
GET /css/style.css 304 1.780 ms - -
GET /img/escudofdigrande.png 304 0.763 ms - -
GET /noexiste.html 404 4.539 ms - 26
```

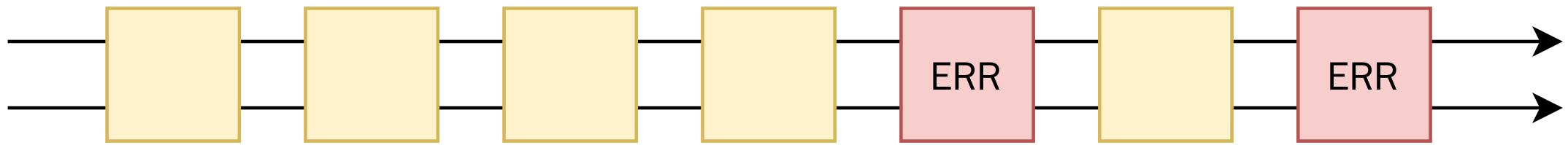
MANEJO DE ERRORES

Recordemos cómo puede finalizar un middleware:

1. Pasando el control al siguiente middleware en la cadena.
2. Finalizando la cadena de middlewares, sin pasar el control al siguiente.
3. Provocando un **error**, que será gestionado por otro middleware.

Pasamos a describir este último caso.

Dentro de la cadena de middlewares podemos incorporar algunos middleware especiales que se encargan de gestionar los errores producidos.

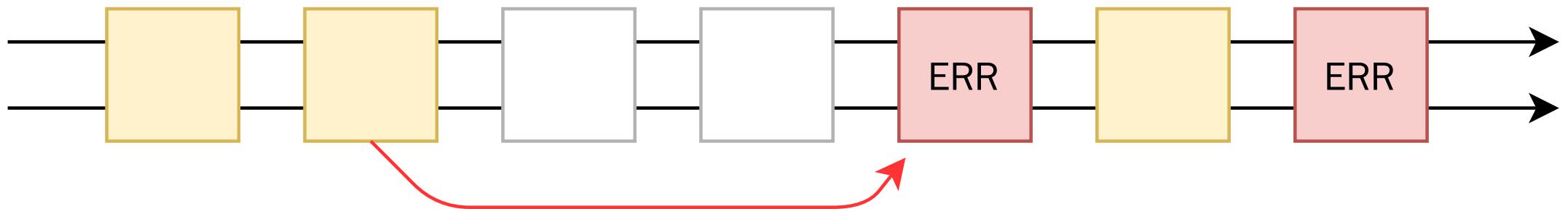


Este tipo de middleware se caracteriza por recibir **cuatro** parámetros, en lugar de tres:

```
function(error, request, response, next) { ... }
```

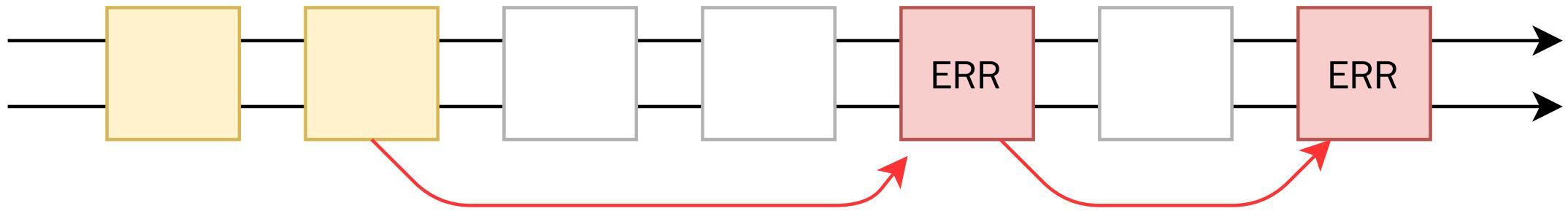
El primer parámetro contiene información sobre el error producido.

Cuando un middleware llama a su función `next` **pasándole un argumento**, el control pasa directamente al primer manejador de errores disponible en la cadena, «saltándose» los middlewares intermedios.



El argumento pasado a `next` será el objeto `Error` recibido por el manejador de errores como primer parámetro.

A su vez, el manejador de error puede pasar el control al siguiente manejador de error de la cadena. Para ello ha de llamar a su función `next ()` con el mismo objeto error recibido, o con otro distinto.



EJEMPLO

```
import express from "express";
import { join } from "path";
import { readFile } from "fs:node";

const app = express();

app.set("view engine", "ejs");
app.set("views", join(import.meta.dirname, "views"));

app.get("/usuarios", function(request, response, next) {
  readFile("noexiste.txt", function(err, contenido) {
    if (err) {
      next(err); // Saltar al manejador de error
    } else {
      request.contenido = contenido;
    }
  });
}); // ... continúa ...
```

```
// Manejador del error

app.use(function(error, request, response, next) {
  // Código 500: Internal server error
  response.status(500);
  response.render("error", {
    mensaje: error.message,
    pila: error.stack
  });
});

app.listen(3000, function() {
..  console.log("Escuchando en el puerto 3000");
}) ;
```

views/error.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Error del servidor</title>
    <meta charset="UTF-8">
  </head>
  <body>
    <h1>500 - Error interno del servidor</h1>
    <p>Se ha producido el siguiente error: <%= mensaje %></p>
    <p>Pila de ejecución:</p>
    <pre><%= pila %></pre>
  </body>
</html>
```

Error del servidor



| localhost:3000/usuarios



Buscar



500 - Error interno del servidor

Se ha producido el siguiente error: ENOENT: no such file or directory, open 'noexiste.txt'

Pila de ejecución:

```
Error: ENOENT: no such file or directory, open 'noexiste.txt'  
at Error (native)
```

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

DIRECCIONAMIENTO Y RUTAS

El direccionamiento de Express.js permite asociar una acción a una ruta determinada.

```
app.get("/users", function(request, response) {  
    // Acciones a realizar cuando se realice una petición  
    // de tipo GET sobre la URL /users.  
});
```

Existen funciones para los demás tipos de peticiones HTTP:

- app.post()
- app.put()
- app.delete()
- app.options()
- app.head()
- etc.

Ejemplo:

```
app.post("/nuevo_usuario", (request, response) => {
    // Se ha realizado una petición de tipo POST
    // sobre la URL /nuevo_usuario
});

app.put("/modificar_usuario", (request, response) => {
    // Se ha realizado una petición de tipo PUT
    // sobre la URL /modificar_usuario
});
```

RUTAS PARAMÉTRICAS

Es posible especificar marcadores dentro de una ruta.

De este modo, la ruta se convierte en una plantilla a la que pueden ajustarse distintas rutas.

Por ejemplo: `/usuarios/:id`

Esta plantilla comprende las siguientes rutas:

- `/usuarios/34`
- `/usuarios/pep`
- `/usuarios/43pep`
- `/usuarios/gerardo-hernandez`

Podemos acceder a los valores concretos de la ruta paramétrica mediante los atributos de `request.params`.

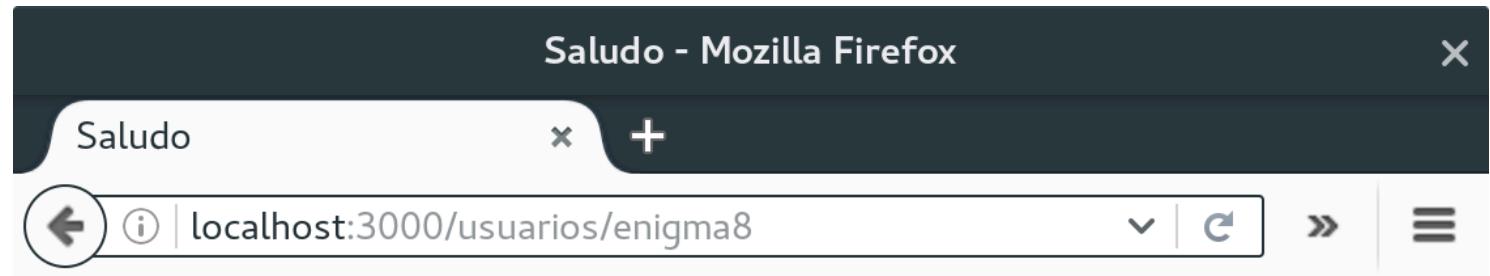
```
app.get("/usuarios/:id", function(request, response) {
  response.status(400);
  response.render("usuario", { ident: request.params.id });
});
```

views/usuario.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Saludo</title>
    <meta charset="utf-8">
  </head>
  <body>
    <h1>¡Bienvenido, <%= ident %>!</h1>
  </body>
</html>
```



¡Bienvenido, david!



¡Bienvenido, enigma8!

Otros ejemplos de rutas paramétricas:

/usuarios/:id/perfil

/usuarios/:id/editar_perfil.html

/libros/:idAutor/:libro

Más información:

<https://github.com/pillarjs/path-to-regexp>

MIDDLEWARE ESPECÍFICO DE RUTA

Es posible incluir una secuencia de middlewares específicos para una determinada ruta.

```
app.get(ruta, middleware_1, middleware_2, ..., manejador);  
app.post(ruta, middleware_1, middleware_2, ..., manejador);  
...
```

Estos middlewares solamente se tendrán en cuenta cuando la petición ajuste con la ruta.

El middleware específico resulta útil para aquellas rutas que requieren pasos previos (por ejemplo, autenticación):

```
function identificacionRequerida(request, response, next) {
  if (usuario_identificado) {
    next();
  } else {
    response.redirect("/login.html");
  }
}

app.get("/secreto.html", identificacionRequerida,
         (request, response) => { ... } );
app.get("/otro_secreto.html", identificacionRequerida,
         (request, response) => { ... } );
app.get("/publico.html", (request, response) => { ... } );
```

SUBAPLICACIONES (ROUTERS)

Un **router** puede considerarse como una mini-aplicación web con sus propias rutas y su propia cadena de middleware.

Los routers se crean mediante `express.Router()`

El router creado tiene los métodos `use()`, `get()`, `post()`, etc, que permiten incorporar middleware y manejo de rutas en la subaplicación.

Módulo: miRouter.js

```
export const miRouter = express.Router();

miRouter.get("/crear_usuario.html", (request, response) => {
  console.log("Creando usuario.");
  response.end();
});

miRouter.get("/buscar_usuario.html", (request, response) => {
  console.log("Buscando usuario");
  response.end();
});
```

Es posible incorporar un router en otra aplicación indicando la ruta sobre la que se quiere montar:

`app.use(ruta, router)`

```
const app = express();
import { miRouter } from "./miRouter.js";
app.use("/usuarios", miRouter);
```

Montar sobre la ruta /usuarios

En este caso, las rutas definidas en el router estarán accesible a través de las siguientes URLs:

`http://localhost:3000/usuarios/crear_usuario.html`

`http://localhost:3000/usuarios/buscar_usuario.html`

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

PETICIONES Y FORMULARIOS

Supongamos un formulario HTML:

```
<form method="método HTTP" action="url">  
...  
</form>
```

El método HTTP puede ser GET o POST.

Al enviar el formulario el navegador saltará a la url indicada. Podemos procesar la información del formulario mediante la ruta correspondiente:

```
app.get("url", (request, response) => {  
    // Tratar información del formulario.  
});  
// o bien: app.post("url", (request, response) => { ... });
```

Trabajaremos con el siguiente formulario:

Formulario de tipo GET - Mozilla Firefox

Formulario de tipo GET x +

← i | localhost:3000/form_get.html | ⟳ 🔍 Buscar » ≡

Nombre: Germán

Edad: Hernando Coello

Hombre Mujer

Fumador/a

Enviar

Enviar

```
<form method="GET" action="procesar_get.html">
  <div>
    <label for="formNombre">Nombre:</label>
    <input type="text" name="nombre" id="formNombre">
  </div>
  <div>
    <label for="formEdad">Edad:</label>
    <input type="text" name="edad" id="formEdad">
  </div>
  <div>
    <input type="radio" name="sexo" value="H" id="formHombre">
    <label for="formHombre">Hombre</label>
    <input type="radio" name="sexo" value="M" id="formMujer">
    <label for="formMujer">Mujer</label>
  </div>
  <div>
    <input type="checkbox" name="fumador" value="ON"
           id="formFumador">
    <label for="formFumador">Fumador/a</label>
  </div>
  <div>
    <input type="submit" value="Enviar">
  </div>
</form>
```

PETICIONES DE TIPO GET

Recordemos que, en las peticiones de tipo `GET`, la información del formulario se incluye en la URL de destino.

```
/procesar_get.html ?nombre=Germ%C3%A1n&edad=Hernando+Coello&sexo=H
```

El objeto `request.query` contiene los datos del formulario:

```
app.get("/procesar_get.html", (request, response) => {
  console.log(request.query);
  // → { nombre: 'Germán',
  //       edad: 'Hernando Coello',
  //       sexo: 'H' }
  request.end();
});
```

Ejemplo de uso:

```
app.get("/procesar_get.html", (request, response) => {
  let sexoStr = "No especificado";
  switch (request.query.sexo) {
    case "H": sexoStr = "Hombre"; break;
    case "M": sexoStr = "Mujer"; break;
  }
  response.render("infoForm", {
    nombre: request.query.nombre,
    edad: request.query.edad,
    sexo: sexoStr,
    fumador: (request.query.fumador === "ON" ? "Sí" : "No")
  });
});
```

Plantilla infoForm.ejs:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Información proporcionada</title>
    <meta charset="UTF-8">
  </head>
  <body>
    <h1>Formulario procesado</h1>
    <table>
      <tr>
        <th>Nombre:</th>
        <td> <%= nombre %> </td>
      </tr>
      <tr>
        <th>Edad:</th>
        <td> <%= edad %> </td>
      </tr>
      <tr>
        <th>Sexo:</th>
        <td> <%= sexo %> </td>
      </tr>
      <tr>
        <th>Fumador:</th>
        <td> <%= fumador %> </td>
      </tr>
    </table>
  </body>
</html>
```

Resultado:



Formulario procesado

Nombre: Germán

Edad: Hernando Coello

Sexo: Hombre

Fumador: No

PETICIONES DE TIPO POST

```
<form method="POST" action="procesar_post.html">  
...  
</form>
```

Al contrario que en las peticiones **GET**, la información del formulario no forma parte de la URL. Se incluye en el **cuerpo** de la petición **POST**:

```
POST http://localhost:3000/procesar_post.html HTTP/1.1  
Host: localhost:3000  
User-Agent: Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:49.0)  
Accept: text/html  
...
```

```
Content-Type: application/x-www-form-urlencoded  
Content-Length: 46
```

Cuerpo de la petición

```
nombre=Germ%C3%A1n&edad=Hernando+Coello&sexo=H
```

Existen métodos de lectura en el objeto `request` que permiten acceder al cuerpo de la petición.

Sin embargo, estos métodos devuelven la cadena con la información del formulario «en bruto»:

```
nombre=Germ%C3%A1n&edad=Hernando+Coello&sexo=H
```

Nuestro programa tendría que analizar sintácticamente esta cadena para extraer cada uno de los valores del formulario.

Por suerte, existe un middleware que hace esto:

`body-parser`

EL MIDDLEWARE body-parser

```
npm install body-parser
```

Este middleware obtiene el cuerpo de la petición HTTP, interpreta su contenido y modifica el objeto `request`, añadiéndole un nuevo atributo (llamado `body`) con la información del formulario.

```
import bodyParser from "body-parser";
```

Cada uno de los siguientes métodos que exporta `body-parser` devuelve un middleware:

- `bodyParser.urlencoded(options)`

Supone que el cuerpo de la petición contiene información en formato URL (como el mostrado anteriormente). El atributo `req.body` contiene un objeto con dicha información.

- `bodyParser.json(options)`

Igual que el anterior, pero supone que la petición contiene un objeto en formato JSON.

En nuestro caso:

```
// ...  
  
// Incluimos el middleware body-parser en la cadena de middleware  
app.use(bodyParser.urlencoded({ extended: false }));  
  
// ...  
  
app.post("/procesar_post.html", function(request, response) {  
    console.log(request.body);  
    // → { nombre: 'Germán',  
    //      edad: 'Hernando Coello',  
    //      sexo: 'H' }  
    response.end();  
});
```

El middleware `body-parser` debe situarse antes de la ruta `/procesar_post.html`, para que esta última tenga la propiedad `request.body` disponible.

EJEMPLO: LA OPCIÓN `extended`

```
<form method="POST" action="procesar_post.html">  
    ...  
    <input type="text" name="datosPersonales[nombre]"  
           id="formNombre">  
    ...  
    <input type="text" name="datosPersonales[edad]"  
           id="formEdad">  
    ...  
    <input type="radio" name="sexo" value="H" id="formHombre">  
    ...  
    <input type="radio" name="sexo" value="M" id="formMujer">  
    ...  
    <input type="checkbox" name="fumador" value="ON"  
           id="formFumador">  
    ...  
</form>
```

```
app.post("/procesar_post.html", (request, response) => {
  console.log(request.body);
  response.end();
});
```

Con extended:false

```
{
  "datosPersonales[nombre]" : "Germán",
  "datosPersonales[edad]" : "Hernando Coello",
  sexo : "H"
}
```

Con extended:true

```
{
  datosPersonales : {
    nombre : "Germán",
    edad : "Hernando Coello"
  },
  sexo : "H"
}
```

FORMULARIOS CON CAMPOS OCULTOS

Hemos visto que dentro de una etiqueta `<form>` podemos tener distintos tipos de componentes:

- `<input type="text">`
- `<input type="submit">`
- `<input type="radio">`
- `<input type="checkbox">`
- `<select>...</select>`

Veamos uno nuevo:

```
<input type="hidden">
```

Los elementos `hidden` permiten incorporar información adicional al enviar un formulario.

Estos elementos no serán visibles al usuario, pero su valor será enviado junto con el resto de los campos del formulario cuando el usuario haga clic en el botón `submit` del mismo.

```
<form action="actualizar" method="POST">
  <input type="text" name="nombre">
  <input type="hidden" name="ident" value="23">
  <input type="submit" value="Enviar">
</form>
```

```
<form action="actualizar" method="POST">
  <input type="text" name="nombre">
  <input type="hidden" name="ident" value="23">
  <input type="submit" value="Enviar">
</form>
```

Si el usuario introduce el valor `Alberto` en el cuadro de texto y pulsa el botón enviar, se adjuntará la siguiente información a la petición POST `actualizar`:

```
nombre=Alberto&ident=23
```

El manejador de ruta `/actualizar` podrá acceder a este valor `23` mediante `request.body.ident`.

Si quieres saber para qué sirven los atributos ocultos, conviene realizar el ejercicio 2 de la hoja de ejercicios 5.

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

COOKIES Y SESIONES

El protocolo HTTP es un protocolo sin estado

Cada petición es una transacción independiente que no guarda relación con ninguna otra anterior.

En otras palabras, HTTP es un protocolo «sin memoria»

Cuando recibe una petición, la atiende, y se «olvida» del cliente que realizó la petición.

Si un cliente realiza varias peticiones consecutivas, desde el punto de vista del servidor es como si cada una de las peticiones proviniese de un cliente distinto.

Sin embargo, hay numerosas ocasiones en las que un cliente desearía ser «recordado» por un servidor web:

- **Identificación de usuarios**: cuando un usuario se *loguea* a una página web, no debería tener que hacerlo cada vez que quiera visitar una página que requiera identificación.
- **Carros de la compra en una tienda web**: el usuario añade productos a su carro de la compra a medida que navega por las páginas del sitio web, y el servidor debería «recordar» el contenido del carro de la compra entre una página y otra
- **Operación en fases**: si una determinada operación (registro en un portal web, pago en tienda web, etc.) requiere visitar varias páginas en secuencia, el servidor debería recordar la información introducida en las fases previas.

Podrían utilizarse variables globales:

```
let carroCompra = [];  
...  
app.get("/añadirProducto/:id", (request, response) => {  
  let producto = buscarProducto(request.params.id);  
  carroCompra.push(producto);  
  ...  
});
```

¡Esto sería un **gran error!**

Una variable global tiene el **mismo valor para todos los clientes**. Con esta implementación, todos los usuarios de la web tendrían el mismo carro de la compra.

Necesitamos un mecanismo que:

- Permita guardar información y mantenerla entre distintas peticiones de un usuario/a.
- Mantenga información individual para cada usuario/a de la aplicación.

Existen dos mecanismos para esto:

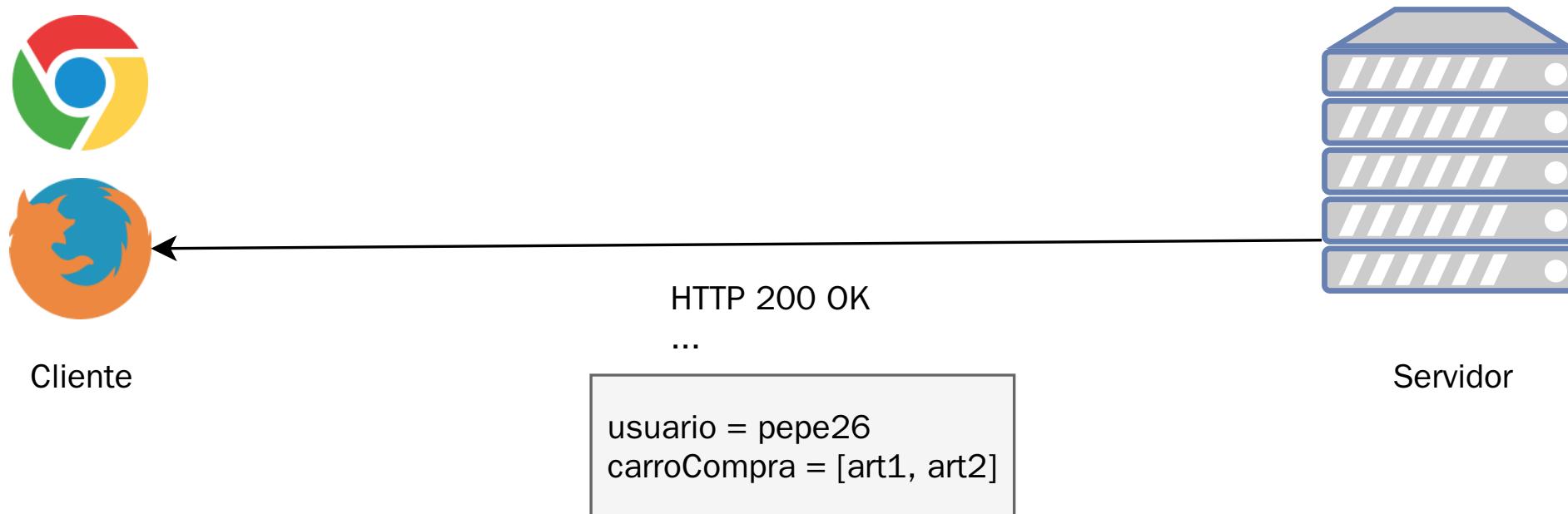
- **Cookies**
- **Sesiones**

COOKIES

Funcionan del siguiente modo:



El cliente realiza una petición HTTP al servidor.

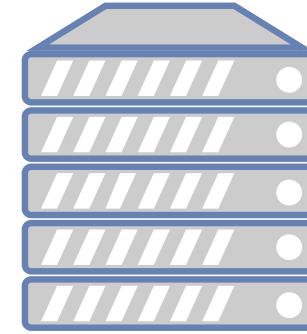


El servidor responde al cliente, e incorpora en la respuesta una o varias **cookies** con la información que debe ser «recordada» de una petición a otra.



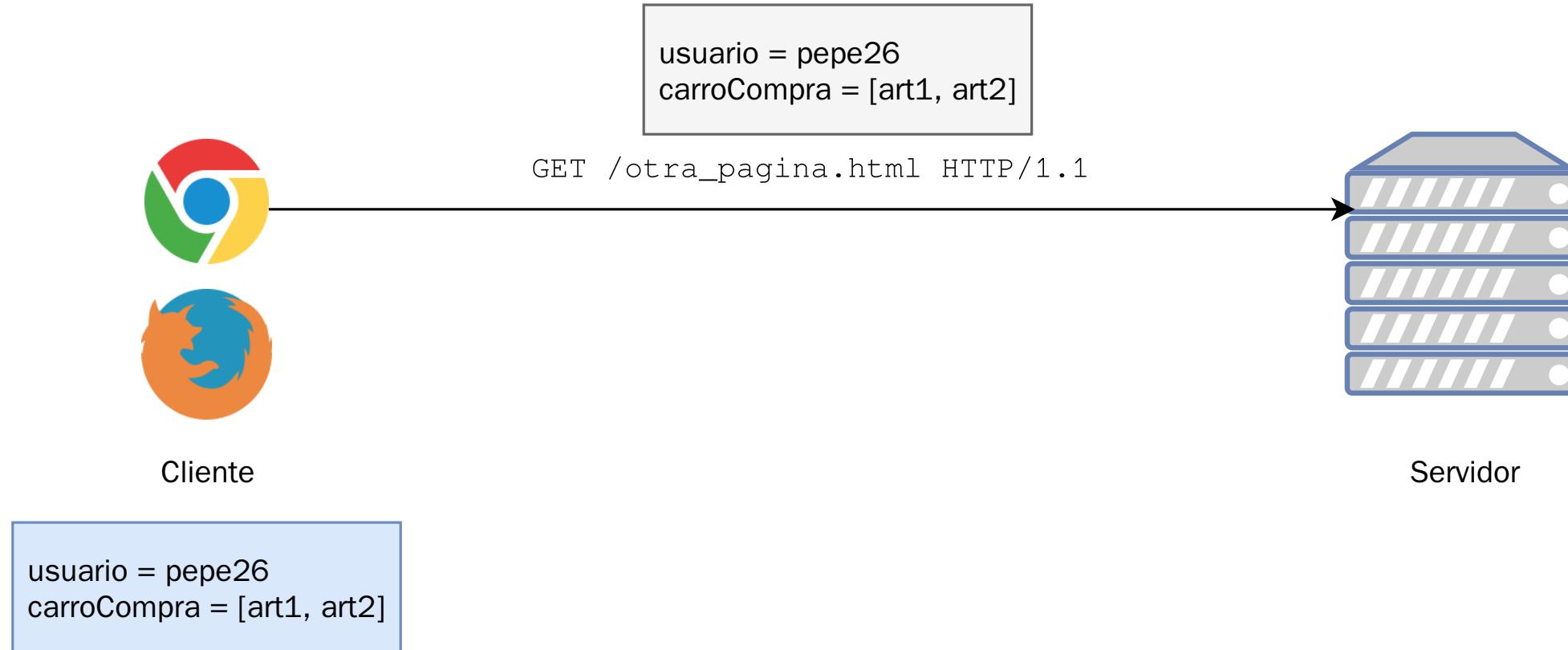
Cliente

```
usuario = pepe26  
carroCompra = [art1, art2]
```



Servidor

El navegador almacena esta información en su depósito de cookies.



Cada vez que el cliente realiza una petición al mismo servidor, adjunta las cookies correspondientes.

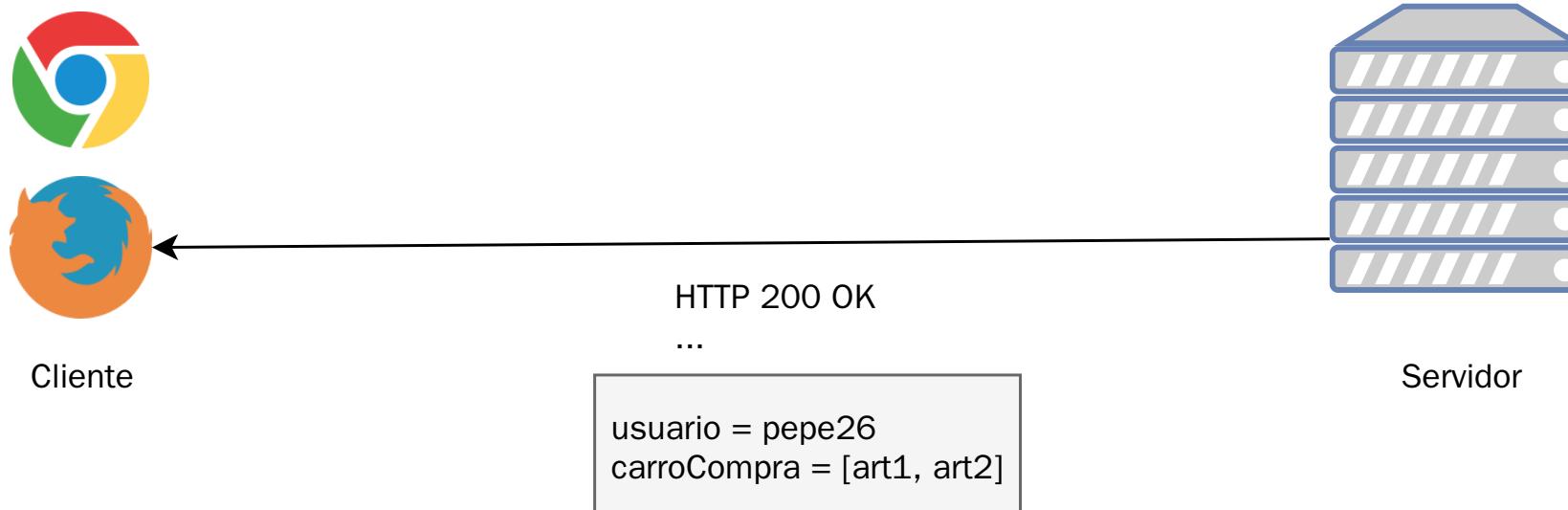
De este modo, el servidor dispone de la información, específica para ese usuario, que debía ser recordada de una petición a otra.

¿CÓMO ENVÍA EL SERVIDOR LAS COOKIES AL CLIENTE?

Mediante las cabeceras de la respuesta HTTP.

```
HTTP/1.1 200 OK
Set-Cookie: usuario=pepe26; carroCompra=[...]; Path=/
Content-Type: text/html; charset=utf-8
...
<html>
  <head>
    ...

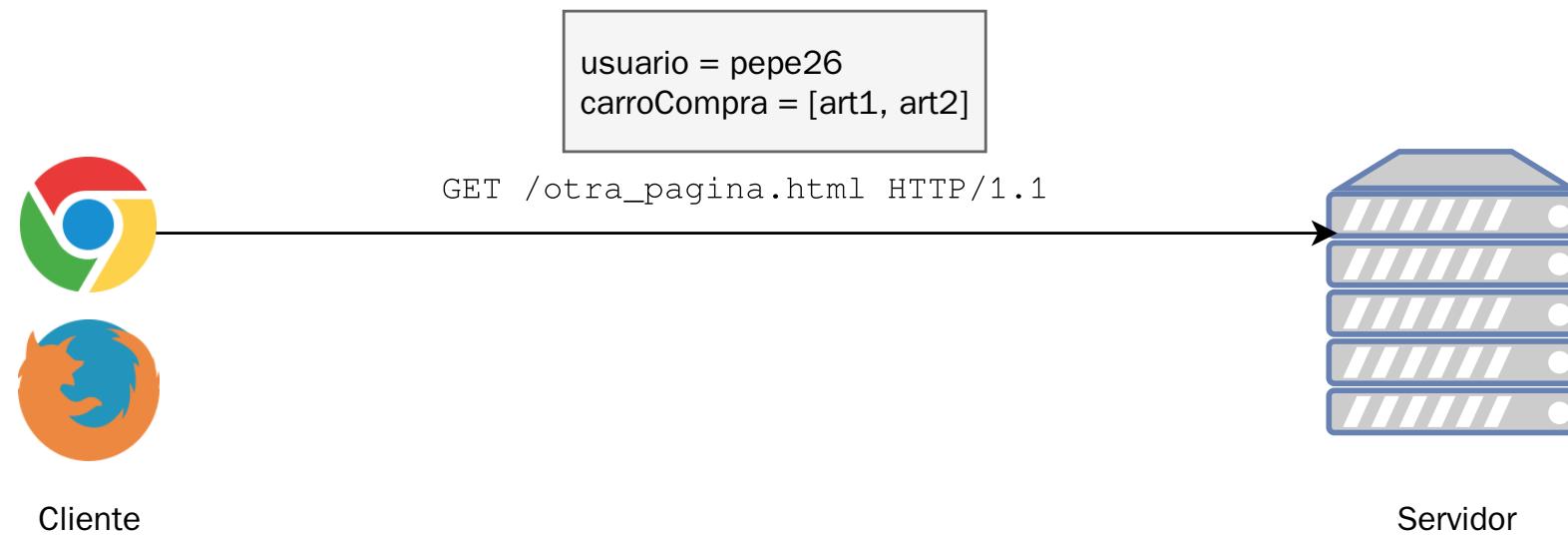
```



¿CÓMO RECUERDA EL CLIENTE LAS COOKIES AL SERVIDOR?

Mediante las cabeceras de la petición HTTP.

```
GET /pagina6.html HTTP/1.1
Host: localhost:3000
User-Agent: Mozilla/5.0 (X11; Fedora; Linux x86_64; rv:49.0) ...
Accept: text/html,application/xhtml+xml
...
Cookie: usuario=pepe26; carroCompra=[...]
...
```



```
usuario = pepe26
carroCompra = [art1, art2]
```

El navegador almacena las cookies de los distintos sitios web que visita.

Cada cookie tiene un ámbito, que viene determinado por:

- El **dominio** del que procede
Por ejemplo: `foo.com`, `domain.org`, etc.
- Una **ruta (path)** dentro del dominio
Por ejemplo: `/admin`, `/user`, etc.

ESTABLECER COOKIES EN NODE

El objeto `response` tiene un método para establecer una cookie en el cliente:

```
response.cookie(nombre, valor[, opciones])
```

```
app.get("/reset.html", (request, response) => {
  response.status(200);
  response.cookie("contador", 0);
  response.type("text/plain");
  response.end("Has reiniciado el contador");
});
```

Mozilla Firefox

X

http://localhost:3000/reset.html × +



| localhost:3000/reset.html



Has reiniciado el contador

```
HTTP/1.1 200 OK
X-Powered-By: Express
Set-Cookie: contador=0; Path=/
Content-Type: text/plain; charset=utf-8
Date: Thu, 10 Nov 2016 19:53:13 GMT
Connection: keep-alive
Content-Length: 26
```

OPCIONES DE cookie

- `domain`, `path`

Determinan el ámbito de la cookie.

- `expires`

Determina la fecha de caducidad de la cookie. A partir de esta fecha, el cliente ya no la enviará al servidor.

- `maxAge`

Alternativa a `expires`, pero indicando el tiempo de vida de la cookie (en ms).

```
// La siguiente cookie se almacena durante 24h = 24*60*60*1000
response.cookie("contador", 0, { maxAge: 86400000 } );
```

ACCESO A LAS COOKIES ENVIADAS POR EL CLIENTE

El navegador envía las cookies en la cabecera de cada petición HTTP.

Podría utilizarse el método `request.get()` para acceder a la línea correspondiente de la cabecera, pero esta función nos devuelve el contenido de texto en formato de texto «bruto», que tendríamos que analizar sintácticamente:

```
app.get("/increment.html", (request, response) => {
  console.log(request.get("Cookie"));
  // → "contador=0"
  response.end();
});
```

Pero el middleware `cookie-parser` nos ayuda con esto.

EL MIDDLEWARE `cookie-parser`

```
npm install cookie-parser
```

Recupera la información de la cabecera HTTP `Cookie`, realiza su análisis sintáctico y añade al objeto `request` un atributo `cookies` con el contenido de la cookie.

```
// ...

import cookieParser from "cookie-parser";

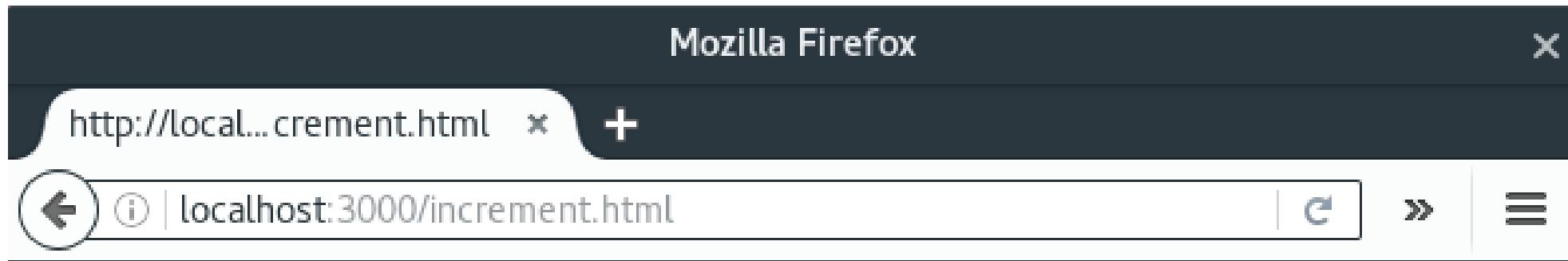
// ...

app.use(cookieParser());

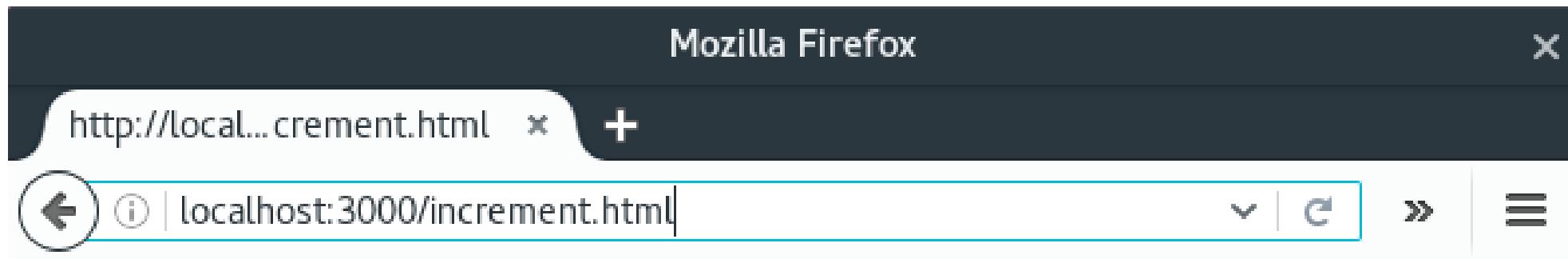
app.get("/increment.html", (request, response) => {
  console.log(request.cookies);
  // → { contador: '0' }
  console.log(Number(request.cookies.contador));
  // → 0
  response.end();
});
```

EJEMPLO

```
app.use(cookieParser());  
  
app.get("/reset.html", (request, response) => {  
  response.status(200);  
  response.cookie("contador", 0, { maxAge: 86400000 } );  
  response.type("text/plain");  
  response.end("Has reiniciado el contador");  
});  
  
app.get("/increment.html", (request, response) => {  
  if (request.cookies.contador === undefined) {  
    response.redirect("/reset.html");  
  } else {  
    let contador = Number(request.cookies.contador) + 1;  
    response.cookie("contador", contador);  
    response.status(200);  
    response.type("text/plain");  
    response.end(`El valor actual del contador es ${contador}`);  
  }  
});
```



El valor actual del contador es 1



El valor actual del contador es 2

OBSERVAR COOKIES EN FIREFOX

Es necesario activar la pestaña de *Almacenamiento* dentro de las herramientas del desarrollador.

Esto puede hacerse a través del botón de configuración  en la barra de herramientas del desarrollador.



OBSERVAR COOKIES EN GOOGLE CHROME

Dentro de las herramientas para desarrolladores, utilizar la pestaña *Almacenamiento*.

Explorar cookies en Chromium



LIMITACIONES DE LAS COOKIES

- **Su contenido es accesible al cliente**

El usuario de la aplicación web puede alterarlas mediante las herramientas del desarrollador. Esto conlleva problemas de seguridad.

Las cookies firmadas (*signed cookies*) pueden prevenir la manipulación por parte del usuario. [\[+\]](#)

- **Limitaciones de almacenamiento**

Existe un número máximo de cookies que un servidor puede enviar a un cliente, y un tamaño máximo para cada una de ellas.

SESIONES

Solucionar las limitaciones de las cookies delegando su almacenamiento en el **servidor**.

El servidor guarda una base de datos con las cookies de cada uno de los clientes. Cada entrada de la BD está identificada con una clave, llamada **session id** (SID).

De este modo, cada cliente tan solo tiene que almacenar su SID y enviárselo al servidor en cada petición.

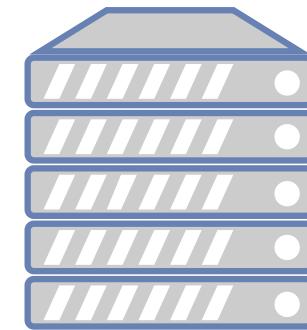
Para el almacenamiento del SID se utiliza una cookie.



Cliente

HTTP 200 OK

sid=24G4EM



Almacenamiento sesión



Servidor

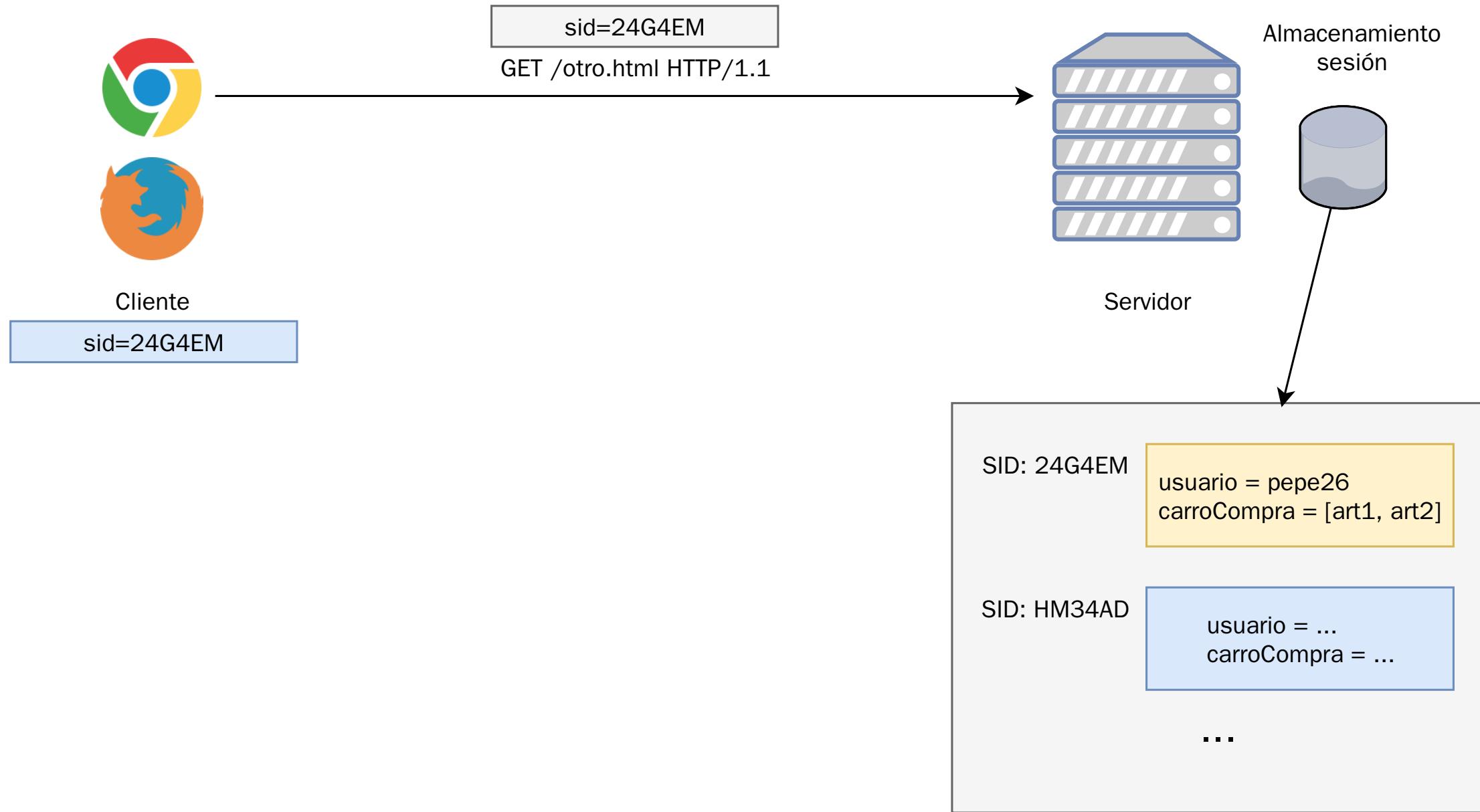
SID: 24G4EM

usuario = pepe26
carroCompra = [art1, art2]

SID: HM34AD

usuario = ...
carroCompra = ...

...



El navegador envía su SID con cada petición

EL MIDDLEWARE `express-session`

Gestiona el almacenamiento de sesiones.

```
npm install express-session
```

Exporta una única función que devuelve un middleware.

Este middleware añade un atributo `session` al objeto `request` que contiene los datos de sesión correspondientes al cliente que se está atendiendo.

El atributo `request.session` también se utiliza para añadir o modificar información de la sesión. Al utilizar `response.end()` o similar, el servidor guarda toda la información de `request.session` en el almacenamiento de sesiones del servidor.

EJEMPLO

Implementamos el ejemplo anterior utilizando sesiones:

```
import session from "express-session";
...
const middlewareSession = session({
  saveUninitialized: false,
  secret: "foobar34",
  resave: false
});
app.use(middlewareSession);
app.get("/reset.html", (request, response) => {
  response.status(200);
  request.session.contador = 0;
  response.type("text/plain");
  response.end("Has reiniciado el contador");
});
...

```

Opciones del middleware

```
...
app.get("/increment.html", (request, response) => {
  if (request.session.contador === undefined) {
    response.redirect("/reset.html");
  } else {
    let contador = Number(request.session.contador) + 1;
    request.session.contador++;
    response.status(200);
    response.type("text/plain");
    response.end(`El valor actual del contador es ${contador}`);
  }
}) ;
...
```

OPCIONES DEL MIDDLEWARE

- `saveUninitialized : false`
Indica que no se cree ninguna sesión para los clientes que no estén en la BD de sesiones, a menos que durante la petición se añada algún atributo a `request.session`.
- `resave : true`
Fuerza a que se guarde el contenido en la sesión en la BD de sesiones al final de la petición, aunque no se haya modificado ningún atributo de `request.session`.
- `secret`
Cadena que se utilizará para firmar el SID que se envía al cliente.

ALMACENAMIENTO DE DATOS DE SESIÓN

Las sesiones se almacenan, por defecto, en un objeto `MemoryStore` alojado en memoria.

Problema: al reiniciar o finalizar el servidor, la información del almacén de sesiones desaparece.

Un `MemoryStore` puede ser útil en la fase de desarrollo, pero para servidores en producción es mejor utilizar un método de almacenamiento con persistencia, p.ej. un SGBD.

<https://github.com/expressjs/session#compatible-session-stores>

El módulo `express-mysql-session` permite almacenar la información de sesión en una base de datos MySQL.

```
const mysqlSession = require("express-mysql-session");
const MySQLStore = mysqlSession(session);
const sessionStore = new MySQLStore({
  host: "localhost",
  user: "root",
  password: "",
  database: "miBD"
});
```

El objeto `sessionStore` creado puede pasarse a la función `session` como parámetro:

```
const middlewareSession = session({
  saveUninitialized: false,
  secret: "foobar34",
  resave: false,
  store: sessionStore
});
app.use(middlewareSession);
```

De este modo la información de las sesiones se guarda en una tabla llamada sessions) en la BD:

session_id	expires	data
NXtGYmmTT3JTzPCITQ3PuqdP3XJTNfCV	1478957299	{"cookie":{"originalMaxAge":null,"expires":null,"h...
u5JlfZe_KRM_I1y2Nhjz4pggYJ-sJpvb	1478957277	{"cookie":{"originalMaxAge":null,"expires":null,"h...

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

DISEÑO AVANZADO DE PLANTILLAS

Delimitadores EJS vistos hasta ahora:

- Delimitadores de sentencias <%, %>
- Delimitadores de expresión <%=, %>

```
<% for (let i = 0; i < 10; i++) { %>
  <li>Opción <%= i+1 %></li>
<% } %>
```

Las plantillas se traducen a código Javascript antes de ser ejecutadas:

```
try {
  var __output = [], __append = __output.push.bind(__output);
  with (locals || {}) {
    ; for(let i = 0; i < 10; i++) {
    ; __append("<li>Opción ")
    ; __append(escape(i + 1))
    ; __append(" </li>")
    ;
  }
  return __output.join("");
} catch (e) {
  rethrow(e, __lines, __filename, __line);
}
```

INCLUSIÓN LITERAL DE CONTENIDO

Los delimitadores `<%=` y `%>` interpretan los caracteres especiales HTML (`<`, `>`, etc) y los converten en las entidades correspondientes (`<`, `>`, etc.)

Si no se desea realizar esta transformación, se puede utilizar otro par de delimitadores: `<%-` y `%>`.

```
<p> <%= msg %> </p>
<p> <%- msg %> </p>
```

Si la variable `msg` toma el valor
"Esto es **importante**"
tenemos:

```
<p> Esto es &lt;b&gt;importante&lt;/b&gt; </p>
<p> Esto es <b>importante</b> </p>
```

Si el contenido de una plantilla proviene de la entrada proporcionada por el usuario, es recomendable utilizar <%= y %> para evitar que el usuario de la aplicación inyecte HTML en nuestras páginas.

SUBPLANTILLAS

La función `include` sirve para insertar el contenido de una plantilla dentro de otra.

```
<body>
    <%- include("header") %>
    <div>
        <ul>
            <% usuarios.forEach(u => { %>
                <%- include("userView", { usuario: u }) %>
            <% } ); %>
        </ul>
    </div>
    <%- include("footer") %>
</body>
```

En este caso tiene sentido utilizar `<%-` y `%>`, ya que se desea integrar las etiquetas HTML de la plantilla incluida dentro de esta plantilla.

views/header.ejs

```
<header>
    Cabecera de la página
</header>
```

views/footer.ejs

```
<footer>
    Pie de página
</footer>
```

views/userView.ejs

```
<li> <%= usuario.apellidos %> , <%= usuario.nombre %> </li>
```

En este caso hemos asignado el valor del objeto `usuario` en el `include` de la plantilla anterior.

```
var usuarios = [
  { nombre: "Francisco", apellidos: "Flores Blanco" },
  { nombre: "Elena", apellidos: "Nito del Bosque" },
  { nombre: "Germán", apellidos: "Gómez Gómez" }
]

app.get("/pag3", function(request, response) {
  response.status(200);
  response.render("view3", { usuarios: usuarios });
});
```

Cabecera de la página

- Flores Blanco, Francisco
- Nito del Bosque, Elena
- Gómez Gómez, Germán

Pie de página

VARIABLES GLOBALES DE PLANTILLA

Los objetos `app` y `response` tienen cada uno un atributo `locals` en el que podemos añadir variables, que serán accesibles desde cualquier plantilla.

```
app.locals.nombreApp = "Mi Aplicación";
app.locals.correo = "montenegro@fdi.ucm.es"
...
app.get("/pag4", function(request, response) {
    response.status(200);
    response.render("view4");
});
```

views/view4.ejs

```
<body>
    <h1>Bienvenido a <%= nombreApp %></h1>
    <p>Para más ayuda contactar con
        <a href="mailto:<%= correo %>"><%= correo %></a></p>
</body>
```

APLICACIÓN: MENSAJES *FLASH*

Los **mensajes flash** son notificaciones que se muestran en una página web para indicar si una operación ha tenido éxito o no.

Los mensajes de este tipo se muestran una sola vez por cada evento o notificación.

Cada página web incluye una sección para mostrar los mensajes flash, si los hay.
Esta sección se implementa mediante una subplantilla.

```
<body>
  <%- include("header") %>
  <%- include("flash") %>

  <!-- Contenido de la página -->

  <%- include("footer") %>
</body>
```

Fichero flash.ejs

```
<% let msg = getAndClearFlash(); %>
  if (msg) { %>
    <div class="flash"> <%= msg %> </div>
  <% } %>
```

Los mensajes flash suelen almacenarse como cookies o atributos de sesión, para que «sobrevivan» a las redirecciones, en caso de producirse.

```
app.use((request, response, next) => {
  response.setFlash = (msg) => {
    request.session.flashMsg = msg;
  };

  response.locals.getAndClearFlash = () => {
    let msg = request.session.flashMsg;
    delete request.session.flashMsg;
    return msg;
  };
}) ;
```

Ejemplo de uso:

```
app.post("/insertar_usuario", (request, response) => {
  /* ... insertar usuario en BD ... */
  response.setFlash(request, "Usuario insertado correctamente");
  response.redirect("/main");
});
```

Ver: <https://github.com/RGBboy/express-flash>

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

HERRAMIENTAS DE DESARROLLO COMPATIBLES CON EXPRESS.JS

- Monitorización de cambios: `nodemon`.
- Gestión de procesos: `forever`, `pm2` (pro).

MONITORIZACIÓN DE CAMBIOS

Durante el desarrollo de un servidor web con Node, cualquier cambio en los ficheros `.js` requiere el reinicio del servidor para que los cambios se hagan efectivos.

La herramienta `nodemon` realiza esto automáticamente.

<http://nodemon.io/>

```
npm install -g nodemon
```

Para ejecutar un programa:

```
nodemon fichero.js
```

Ejemplo:

```
# nodemon main.js
[nodemon] 1.11.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node main.js`
Escuchando en el puerto 3000
```

Al hacer cambio en el fichero `main.js`, o en alguno de los módulos de los que este depende, se reiniciara el programa automáticamente:

```
[nodemon] restarting due to changes...
[nodemon] starting `node main.js`
Escuchando en el puerto 3000
```

Puede forzarse el reinicio tecleando `rs ↵`.

GESTIÓN DE PROCESOS EN NODE

Un servidor web está pensado para ejecutarse de manera indefinida, aún en presencia de errores que impidan atender una petición concreta.

Los errores deben ser registrados en un fichero, pero no deben suponer la parada del servidor web.

Sin embargo, cuando en Node se lanza una excepción que no se captura en ningún sitio, el programa finaliza, y se ha de reiniciar manualmente.

La herramienta `forever` permite reiniciar un programa Node en el caso en que este aborte.

<https://github.com/foreverjs/forever>

```
# npm install -g forever
```

Esta herramienta lanza los programas Node en segundo plano y captura sus salidas, de modo que las llamadas a `console.log()`, `console.error()`, etc. se escriben en un fichero de registro en lugar de mostrarse por pantalla.

Para arrancar un programa:

```
# forever start fichero.js
```

`forever` reiniciará el programa en caso de error.

Es posible arrancar varios programas con `forever`. El comando `list` imprime los programas en ejecución:

```
# forever list
Forever processes running
uid  command  script  forever pid  id logfile  uptime
[0]  39Tw  node main.js  5813    5825    39Tw.log  0:0:0:4.627
```

Cada proceso Node se identifica con un UID, que es un número identificador que debe indicarse posteriormente para gestionar su parada, reinicio, etc.

OTRAS ACCIONES

- `forever stop UID`
- `forever restart UID`
- `forever logs UID`

Detiene el proceso correspondiente al UID dado.

Reinicia un proceso.

Imprime la salida (`console.log()`, etc.) de un proceso. Esta salida se

encuentra disponible en un fichero, cuyo nombre puede verse en el listado devuelto por `forever list`.

```
# forever logs 0
main.js:5825 - Escuchando en el puerto 3000

# forever stop 0
```

1. INTRODUCCIÓN
2. PRIMERA APLICACIÓN
3. PLANTILLAS CON EJS
4. MIDDLEWARE
5. DIRECCIONAMIENTO Y SUBAPLICACIONES
6. PETICIONES Y FORMULARIOS
7. COOKIES Y SESIONES
8. DISEÑO AVANZADO DE PLANTILLAS
9. HERRAMIENTAS DE DESARROLLO
10. BIBLIOGRAFÍA

BIBLIOGRAFÍA

- E.M. Hahn
[Express in Action](#)
Manning Publications (2016)
- A. Mardan
[Pro Express.js](#)
Apress (2014)
- [Express API Reference](#)
<http://expressjs.com/en/api.html>

