Express.js

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universidad Rey Juan Carlos

gsyc-profes (arroba) gsyc.urjc.es

Abril de 2022

Express.js

Express.js, también llamado simplemente *Express* es un *Web Application Framework*

- Permite desarrollar aplicaciones web en el servidor usando el mismo lenguaje que en el cliente: JavaScript
- Basado en Node.js
- Alternativa a Django o Ruby on Rails
- Aplicación libre y gratuita, muy popular
- Desarrollado por TJ Holowaychuk en 2010. Vendido a StrongLoop, que actualmente pertenece a IBM

Instalación de Express

- Para usar Express.js necesitamos una versión reciente de node.js. Con Ubuntu 20.04 se distribuye node.js v10, que no cumple este requisito
- Para saber qué versión de node.js tenemos instalada:

```
node -v
```

- Para instalar node.js v16 en Ubuntu 20.04 (si tenemos privilegios de root)

```
cd
curl -sL https://deb.nodesource.com/setup_16.x | sudo bash -
sudo apt-get install -y nodejs
```

- Para instalar express en macOS o Linux (no son necesarios privilegios de root)

```
npm install express
```

- Para saber qué versión de express.js tenemos instalada

```
npm list express
```

Diseño de las URL

El interface de una aplicación (las URL que usarán cliente y servidor) en Express.js o en cualquier otra herramienta, debe seguir los mismos princios de calidad. Una URL bien diseñada debería permanecer un número indefinido de años (*toda la vida*), sin importar que cambien las tecnologías: framework, lenguaje, sistema operativo etc. Para ello:

- No exponer detalles técnicos. No se debería ver palabras como *php asp* o incluso *js*
  - Bien:
    http://www.ejemplo.com/busqueda
  - Mal:
    http://www.ejemplo.com/busqueda.php

- Evitar palabras irrelevantes. Una URL debe ser corta, cada palabra debe tener significado. P.e. si todas las URL empiezan por */home/*, esa palabra sobra
  - Bien:
    http://www.ejemplo.com/busqueda
  - Mal:
    http://www.ejemplo.com/home/app/auto/busqueda

- Ser consistente con la separación de palabras. Se pueden usar guiones, barras bajas, NotacionCamello o notacionDromedario, con tal de que siempre se use la misma. Generalmente se recomiendan los guiones y todas las palabras en minúsculas, aunque es opinable.
  - Bien:
    http://www.ejemplo.com/user-id/:user-id/app-id/:app-id
    http://www.ejemplo.com/userId/:userId/appId/:appId
  - Mal:
    http://www.ejemplo.com/userId/:userId/app-id/:app-id
- No usar nunca espacios ni caracteres no ingleses
  - Bien:
    http://www.ejemplo.com/spain
  - Mal:
    http://www.ejemplo.com/españa

APIs REST con Express.js

- Con Express.js podemos preparar de forma muy sencilla servicios web con interface REST / ROA
- Esencialmente consiste en un servidor web que responde peticiones GET, PUT, POST y DELETE siguiendo la metodología REST

`http://ortuno.es/rest.pdf`

- Las más habituales son las peticiones GET: el cliente web solicita un recurso al servidor, indicando su URL. Normalmente será un fichero generado dinámicamente (por una aplicación)
- Para actualización de datos, mediante el método POST el cliente web envía datos al servidor en el cuerpo de la petición. Estos datos serán usados por una aplicación en el servidor

Configuración de Express

Para lanzar Express, preparamos un fichero `.js` y lo ejecutamos con node. Ejemplo:

```
node api_rest.js
```

Atención, si el puerto ya está ocupado (típicamente porque, por error, hemos lanzado un servidor sin cerrar el anterior) el mensaje no es demasiado claro

```
events.js:174
      throw er; // Unhandled 'error' event
```

Para probar el servidor

- Podemos usar *Postman API platform*, una herramienta libre y gratuita, muy popular
- Podemos user *RESTer*, una extensión para Chrome
- Las peticiones GET podemos probarlas, además, desde cualquier navegador web

Objeto app

Las dos primeras líneas de un programa en express suelen ser

```
const express = require('express');
//  Importamos el módulo express, que por omisión
//  exporta una función

const app = express();
//  Asignamos esa función al objeto app
```

En el resto del programa, invocamos a métodos como

```
app.get()  app.put()
```

Esto es algo normal en JavaScript pero raro en otros lenguajes: una función es un caso particular de objeto, que a su vez puede tener métodos

Routing

- Una vez listo el objeto app, nos ocupamos del *routing*, esto es, para cada petición del cliente, indicar qué respuesta se le dará
- Con los métodos `get`, `put`, `post`, `delete` indicamos qué hacer con las peticiones GET, PUT, POST, DELETE
  - El primer parámetro indicamos la URL
  - El segundo parámetro es el manejador de la petición, la función que se disparará cuando llegue una petición a la URL

```
app.get('/about', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Esto es una prueba de Express');
})
```

Se suele usar *notación flecha*, pero nada impide emplear notación tradicional

El manejador tiene dos parámetros, normalmente llamados

1. `req`
   Objeto con todos los detalles de la petición (*request*)
2. `res`
   Objeto con todos los detalles de la respuesta (*response*)
   - `res.send()` permite responder texto, indicando previamente la codificación con `res.type()`
   - `res.json()` seponde al cliente un objeto json

Parámetros

Podemos usar *parámetros* en la dirección: segmentos de la URL que capturan los valores especificados en esa posición. Se indican anteponiendo el carácter *dos puntos*. Los parámetros se guardan en el objeto *params* del objeto *req*. Ejemplo:

```
app.get('/api/coords/:x/:y', (req, res) => {
  let x = req.params.x
  let y = req.params.y
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido las coordenadas '+ x + ' ' + y);
})
```

El cliente podrá hacer una petición GET a la dirección `/api/coords/150/300`, y capturaremos los valores 150 y 300 para x e y, respectivamente

No hay inconveniente en alternar parámetros con segmentos fijos
Ej:

```
/user/:userId/subject/:subjectId
```

Para nombrar estos parámetros podemos usar letras inglesas mayúsculas o minúsculas, números y la barra baja

Peticiones PUT

Para acceder al cuerpo de una petición PUT, ejecutamos `app.use(express.json())` [1]. El método *use* permite añadir capas de *middleware*, esto es, código intermedia que procesa todas las peticiones

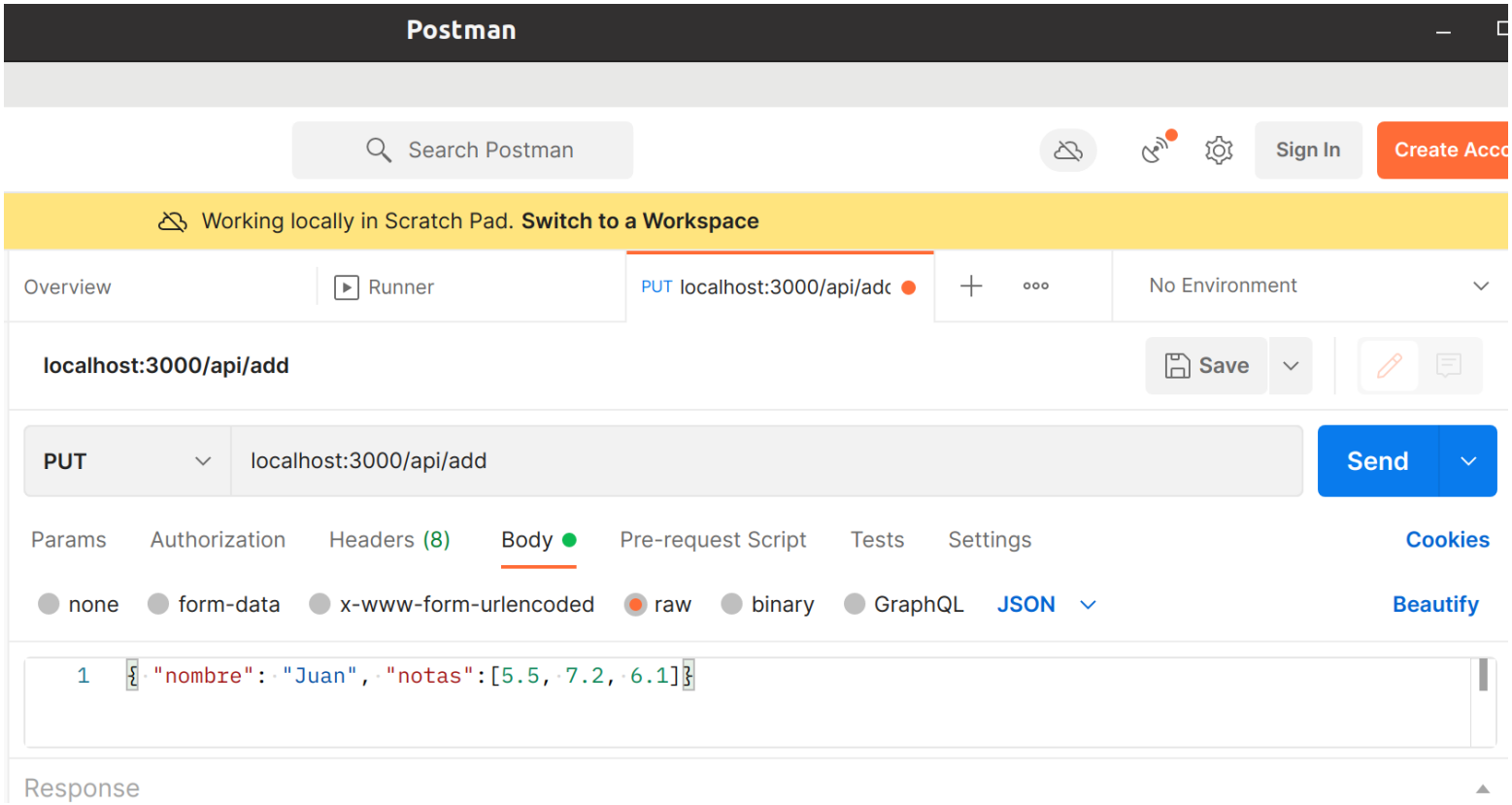
```
app.use(express.json());
[...]
app.put('/api/add',(req,res) => {
  console.log('Me has enviado este objeto:');
  console.log(req.body);
  res.json(req.body);
});
```

- El cuerpo de la petición está disponible en `req.body`
- Será un objeto JSON (no un valor cualquiera sino un objeto: una secuencia de pares clave-valor, entre llaves)

[1] La documentacióna antigua indica que es necesario instalar la librería body-parser, pero esto es obsoleto desde express 4.16 (año 2017)

postman

- Para probar las peticiones PUT, necesitamos una herramienta como *postman*
- Lo podemos instalar con
  `snap install postman`
- En el laboratorio de la ETSIT ya está instalado, en `/opt/Postman/postman`
- Para el uso básico de postman, no es necesario crear ninguna cuenta



- Elegimos `PUT` y escribimos la URL
- En el *body* escribimos una petición de tipo *raw* en JSON, con el objeto que corresponda
- Pulsamos *send*

Errores

Despues del *routing* de las peticiones previstas, añadimos los manejadores de los errores. Es importante hacerlo en este orden, para que se ejecuten solamente cuando no encaje ninguna ruta especificada previamente

```
//  Status Code 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Not Found');
})

// Status Code 500
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Server Error');
})
```

Status Code

Como en cualquier servidor web, cada petición a express.js debe recibir un *status code*, con los criterios habituales: error del servidor, error del cliente, petición sin errores

- Error de servidor catastrófico
  Error severo en el servidor, típicamente excepción sin manejar. Requieren iniciar el servidor. `Status code 500`
- Error de servidor recuperable
  Error en el servidor que tal vez no sea permanente, como fallo no previsto en un fichero o en la base de datos.
  `Status code 500`

- Errores del cliente
  El cliente usa mal el API: pide un recurso que no existe o no tiene los permisos adecuados. Desde el punto de vista del servidor, esto no es un error, es un comportamiento normal.
  Códigos más habituales:
  `404 (Not Found)`, `400 (Bad Request)`, `401 (Unauthorized)`
- Sin errores
  Cuando la petición es correcta, el servidor devolverá `Status Code 200`.
  - Si el cliente pide una lista de elementos, y la lista está vacía, esto también es un resultado correcto, no un error

Método listen()

Una vez configurado el *routing()*, ejecutamos `app.listen()` pasando dos argumentos

- Puerto TCP donde el servidor aceptará las peticiones
- Función a ejecutar en el servidor, típicamente para escribir mensaje en salida estándar

```
const puerto = process.env.PORT || 3000;
// Puerto indicado en la variable de entorno PORT, o 3000
// si la variable no está definida.

[...]

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

Ejemplo completo

```
const express = require('express');
// Importamos el módulo express, que exporta una función

const app = express();
// Función principal, que a su vez tiene métodos

const puerto = process.env.PORT || 3000;
// Puerto indicado en la variable de entorno PORT, o 3000
// si la variable no está definida.

app.use(express.json());
// middleware necesario para procesar las peticiones
// POST que incluyan un cuerpo json
```

```
// Direccionamiento para GET
app.get('/', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Bienvenidos a mi página de ejemplo');
})

app.get('/about', (req, res) => {
  res.type('text/plain; charset=utf-8');
  res.send('Esto es una prueba de Express');
})

app.get('/data', (req, res) => {
  res.json("['sota', 'caballo', 'rey']");
})
```

```
app.get('/api/carta/:id', (req, res) => {
  let id = req.params.id
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido la carta '+id);
})

app.get('/api/coords/:x/:y', (req, res) => {
  let x = req.params.x
  let y = req.params.y
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido las coordenadas '+ x + ' ' + y);
})
```

```
// Direccionamiento para PUT
app.put('/api/object/:id', (req, res) => {
  let id = req.params.id
  res.type('text/plain; charset=utf-8');
  res.send('Me has pedido crear el objeto '+ id );
})

app.put('/api/add',(req,res) => {
  console.log('Me has enviado este objeto:');
  console.log(req.body);
  res.json(req.body);
});
```

```
// Esta llamada debe ser la última, error 404
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Not Found');
})

// custom 500 page
app.use((err, req, res, next) => {
  console.error(err.message);
  res.type('text/plain');
  res.status(500);
  res.send('500 - Server Error');
})

app.listen(puerto, () => console.log(
  `Express iniciado en http://localhost:${puerto}\n` +
  `Ctrl-C para finalizar.`));
```

http://ortuno.es/api_rest.js.html

Cómo servir ficheros estáticos

- Aunque el propósito principal de Express.js es servir páginas web generadas dinámicamente, en ocasiones necesitaremos servir ficheros *estáticos*, esto es, páginas web que se correspondan con ficheros en el servidor *tal cual*
- Para esto, es fundamental tener claro qué significa *directorio raiz* de un sitio web, sin confundirlo con el directorio raiz del sistema de ficheros (disco duro) del ordenador que sirve el web

Directorio raiz de un sitio web

- El *directorio raiz* de un sitio web es el directorio que los clientes web percibirán como el directorio /
- Se corresponderá con cierto directorio en el servidor web, al que llamaremos así, *directorio raiz*. P.e. `/var/www/html/`
- No debemos confundirlo con el directorio raiz del sistema de ficheros (disco duro) del servidor web (/)
  - Que naturalmente será inaccesible via web, a menos que un atacante explote un fallo de seguridad muy severo

Ejemplo 1

Si el servidor web está en el puerto 3000 de `localhost` y `dir_raiz` vale

```
/home/jperez/www/site01
```

Y el el servidor web tiene el fichero

```
/home/jperez/www/site01/holamundo.html
```

El cliente deberá pedirlo como

```
localhost:3000/holamundo.html
```

Observa que el nombre del directorio raiz no forma parte del path que debe pedir el cliente web

Ejemplo 2

Si el servidor está en

```
localhost:3000
```

y `dir_raiz` vale

```
/home/jperez/www
```

Y el el servidor tiene el fichero

```
/home/jperez/www/site01/holamundo.html
```

El cliente deberá pedirlo como

```
localhost:3000/site01/holamundo.html
```

Especificación del home

Muy importante: recuerda que (en ningún lenguaje) deberías escribir el directorio *home* como una cadena literal (*escribirla tal cual*, p.e. `/home/jperez`), sino leerlo desde la variable de entorno *home*

Al directoro *home* se le suele llamar de distintas formas, como

- `$HOME`
  Esta sintaxis es válida en la shell de linux, pero en ningún otro entorno
- `~/`
  Esta sintaxis es válida en la shell de linux y en algunas librerías de algunos lenguajes, pero raramente se puede escribir *tal cual* en un lenguaje de programación

En node.js, y por tanto en express.js, para construir el directorio

```
~/www/site01
```

Escribiríamos

```
path.join( process.env.HOME, "www/site01");
```

Esto generará el directorio que corresponda a nuestro usuario, nuestra máquina y nuestro sistema operativo
P.e. una cuenta de nuestro laboratorio, de nuestra máquina Linux o nuestra máquina macOS podría ser, respectivamente:

```
/home/alumnos/agarcia/www/site01
/home/jperez/www/site01
/Users/Ana
```

En este caso, no hay diferencia entre escribir

```
path.join( process.env.HOME, "www/site01");
```

o

```
path.join( process.env.HOME, "/www/site01");
```

Ya que

- El método `path.join()` concatena dos trayectos (sin importar si el último acaba en barra o no, o el primero empieza por barra o no)
- El segundo argumento de *join* no es ni un trayecto relativo ni un trayecto absoluto, sino un trayecto a concatenar con el trayecto especificado en el primer argumento

- Esto es lo que recomendamos aquí: especificar siempre el directorio raiz de un sitio web a partir de variables de entorno (ya sea *HOME*, como acabamos de ver, ya sea alguna otra variable que definamos en nuestro `~/.bashrc` o similar)
- Pero verás muchos libros y tutoriales que usan trayectos relativos, p.e

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

Esto significa *directorio llamado public, contenido dentro del directorio actual del proceso que hizo la llamada a express*.

- Es perfectamente válido, aunque suele resultar muy confuso para el principiante

Para servir los ficheros estáticos de un directorio, *tal cual*, basta con

```
app.use(express.static(dir_raiz))
```

Si queremos servir más de un directorio, hacemos varias llamadas a este método

```
app.use(express.static(dir_public))
app.use(express.static(dir_js))
```

Naturalmente, los subdirectorios están siempre incluidos, recursivamente

Ficheros estáticos, ejemplo completo

```
const express = require('express');
const app = express();
const puerto = process.env.PORT || 3000;

const path = require('path');
// Importamos el módulo path

dir_raiz = path.join( process.env.HOME, "www/site01");
// Construye ~/www/site01

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz

// Ahora haríamos las llamadas a app.use para tratar los
// errores 404 y 500, así como la llamada a app.listen(),
// de la forma habitual.
```

http://ortuno.es/estaticos01.js.html

En ocasiones querremos que cuando el cliente pida cierto fichero, reciba otro distinto. P.e, por omisión, / apunta a `index.tml`. Si quisiéramos que cuando pida / reciba `holamundo.html`

```
app.get('/', (req, res) => {
    res.sendFile(path.join(dir_raiz,'holamundo.html'));
});
// para '/', sirve ~/www/site01/holamundo.html

app.use(express.static(dir_raiz))
// Sirve todos los ficheros del directorio raiz
```

Recuerda que es importante el orden en que se llama a `app.use()`, la primera invocada tiene prioridad

`http://ortuno.es/estaticos02.js.html`