Desarrollo de Front-END

Tema 11

Express

César Andrés Sánchez

Universidad Camilo José Cela
Escuela Politécnica Superior de Tecnología y Ciencia

Curso 2025 - 2026

Express

Express.js

Express.js, conocido simplemente como *Express*, es un *framework* para el desarrollo de aplicaciones web en el lado del servidor.

- Permite crear aplicaciones web y servicios REST usando el mismo lenguaje del cliente: **JavaScript**.
- Está construido sobre la plataforma **Node.js**.
- Ofrece una alternativa ligera y flexible frente a frameworks como **Django** (Python) o **Ruby on Rails** (Ruby).
- Es software libre, gratuito y con una comunidad muy activa.
- Fue creado en 2010 por **TJ Holowaychuk**. Más tarde pasó a StrongLoop, actualmente integrada en IBM.

Instalación de Express

- Para utilizar Express.js es necesario disponer de una **versión reciente de Node.js** (recomendado: LTS 18 o 20). La versión incluida por defecto en Ubuntu 20.04 (v10) es obsoleta.
- Para comprobar la versión instalada de Node.js:

```
node -v
```

- Instalación de Node.js LTS en Ubuntu 20.04 (requiere privilegios de administrador):

```
curl -fsSL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
sudo apt-get install -y nodejs
```

- Instalación de Express.js en macOS o Linux (no requiere privilegios de root):

```
npm install express
```

- Para comprobar la versión instalada de Express.js:

```
npm list express
```

Diseño de las URL

El interfaz de una aplicación web —las URL que utilizarán cliente y servidor—, tanto en Express.js como en cualquier otra tecnología, debe seguir principios de calidad y estabilidad.
Una URL bien diseñada debería poder mantenerse durante años, independientemente de los cambios en tecnologías internas (framework, lenguaje, servidor, etc.).
Para lograrlo:

- **No exponer detalles técnicos.** Las URL no deben mostrar extensiones o tecnologías internas como *php*, *asp*, *jsp* o *js*.
  - **Correcto:** http://www.ejemplo.com/busqueda
  - **Incorrecto:** http://www.ejemplo.com/busqueda.php

- **Evitar palabras innecesarias.** Las URL deben ser breves y cada segmento debe aportar significado. Si todas comienzan por */home/*, entonces esa palabra sobra.
  - **Correcto:** http://www.ejemplo.com/busqueda
  - **Incorrecto:** http://www.ejemplo.com/home/app/auto/busqueda

- **Mantener un estilo consistente.** Se pueden usar guiones, barras bajas o notación tipo CamelCase, siempre que se utilice el mismo criterio en toda la aplicación. Por claridad, se suele recomendar guiones y minúsculas.
  - **Correcto:** http://www.ejemplo.com/user-id/:user-id/app-id/:app-id
    http://www.ejemplo.com/userId/:userId/appId/:appId
  - **Incorrecto:** http://www.ejemplo.com/userId/:userId/app-id/:app-id
- **No utilizar espacios ni caracteres especiales.** Las URL deben emplear únicamente caracteres ASCII y sin espacios.
  - **Correcto:** http://www.ejemplo.com/spain
  - **Incorrecto:** http://www.ejemplo.com/españa

APIs REST con Express.js

- Express.js permite construir de manera sencilla servicios web basados en arquitectura **REST** o **ROA** (Resource-Oriented Architecture).
- Estos servicios se implementan como un servidor que responde a las peticiones **GET**, **POST**, **PUT** y **DELETE**, siguiendo los principios REST.
- La petición más habitual es **GET**, mediante la cual un cliente solicita un recurso identificado por su URL. Dicho recurso suele ser generado dinámicamente por una aplicación.
- Para modificar información, se utiliza normalmente el método **POST**, con el que el cliente envía datos al servidor dentro del cuerpo de la petición. Estos datos serán procesados por la lógica de la aplicación en el servidor.

Configuración de Express

Para ejecutar una aplicación en Express, creamos un fichero `.js` y lo lanzamos con Node.js. Ejemplo:

```
node api_rest.js
```

Si el puerto elegido ya está ocupado (por ejemplo, porque se dejó un servidor ejecutándose previamente), Node mostrará un mensaje de error poco descriptivo:

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

Para probar nuestro servidor:

- Podemos utilizar *Postman API Platform*, una herramienta libre y muy popular.
- También podemos emplear *RESTer*, una extensión para navegadores.
- Las peticiones **GET** pueden probarse directamente desde cualquier navegador web.

Objeto app

Las primeras líneas típicas de un programa en Express son:

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

const app = express();
// Esta función devuelve un objeto, que guardamos en 'app'
```

A partir de ese momento, utilizamos métodos como:

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

En JavaScript, las funciones son objetos, por lo que pueden poseer métodos. Este comportamiento es habitual en este lenguaje y facilita la programación funcional.

Routing

- Una vez creado el objeto app, definimos el *routing*: qué respuesta se dará a cada tipo de petición.
- Mediante `get`, `post`, `put` y `delete` indicamos cómo responder a las peticiones HTTP.
- El primer parámetro de estos métodos es la URL.
- El segundo parámetro es el manejador de la petición, una función que se ejecutará cuando llegue una solicitud a dicha URL.

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

Aunque se utiliza habitualmente la *arrow function*, es posible usar funciones tradicionales sin problema.

Objeto req y objeto res

El manejador de una ruta recibe dos parámetros:

1. `req` — Objeto que contiene los detalles de la petición (*request*).
2. `res` — Objeto que representa la respuesta que enviaremos al cliente (*response*).

Algunos métodos útiles:

- `res.send()` envía texto u otros contenidos.
- `res.json()` envía un objeto JSON de forma directa.

Parámetros en la URL

Podemos capturar valores dentro de la propia URL mediante *parámetros*. Se indican precedidos por dos puntos y se guardan en `req.params`. 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('Coordenadas recibidas: ' + x + ' ' + y);
});
```

Una petición a `/api/coords/150/300` permitirá capturar los valores 150 y 300.

Parámetros avanzados

Es posible mezclar parámetros y segmentos fijos:

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

Los nombres de los parámetros pueden contener letras, números y guiones bajos.
Express los almacena automáticamente en `req.params`.

Peticiones PUT

Para leer el cuerpo (*body*) de una petición PUT o POST, debemos activar el middleware integrado:

```
app.use(express.json());
```

Antes de Express 4.16 se utilizaba la librería *body-parser*, pero ya no es necesario.

```
app.put('/api/add', (req, res) => {
  console.log('Objeto recibido:');
  console.log(req.body);
  res.json(req.body);
});
```

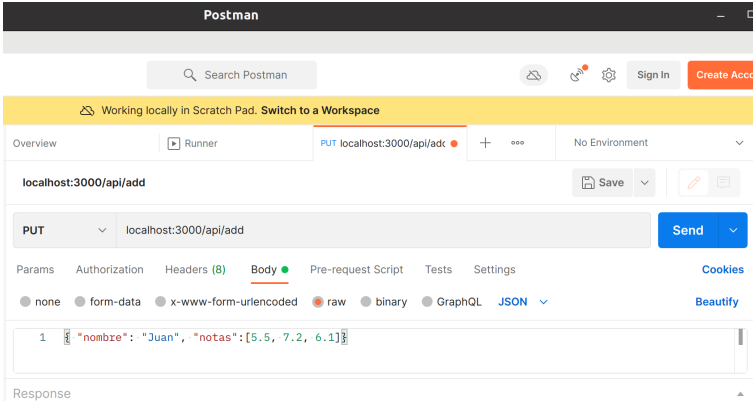
- El cuerpo de la petición está disponible en `req.body`.
- Debe ser un objeto JSON válido.

Postman

- Para probar peticiones PUT y POST utilizamos herramientas como *Postman*.
- Puede instalarse con:

  ```
  snap install postman
  ```

- En los laboratorios de la ETSIT se encuentra disponible en: `/opt/Postman/postman`
- Para las pruebas básicas no es necesario crear una cuenta.



- Seleccionamos el método `PUT`.
- Especificamos la URL del recurso.
- En el *body*, elegimos *raw* y escribimos el JSON correspondiente.
- Pulsamos *Send*.

Manejo de errores

Tras definir todas las rutas, añadimos los manejadores de error. Es importante colocarlos al final del fichero:

```
// Error 404 - Recurso no encontrado
app.use((req, res) => {
  res.type('text/plain');
  res.status(404);
  res.send('404 - Not Found');
});

// Error 500 - Error interno del servidor
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 atendida por Express.js debe devolver un *status code* siguiendo los criterios habituales de HTTP: errores del servidor, errores del cliente o peticiones correctas.

- **Error de servidor catastrófico**
  Indica un fallo grave en el servidor, normalmente debido a una excepción no gestionada. Suele requerir reiniciar el servidor. Código típico:
  `500 (Internal Server Error)`.
- **Error de servidor recuperable**
  Fallos que pueden ser puntuales, como un problema con un fichero o un error temporal en la base de datos. También utilizan 500 como código de estado.

- **Errores del cliente**
  Se producen cuando el cliente usa incorrectamente el API: solicita un recurso inexistente, envía datos inválidos o no tiene permisos. Desde el punto de vista del servidor, no son fallos internos.
  Códigos habituales: `404 (Not Found)`, `400 (Bad Request)`, `401 (Unauthorized)`.
- **Peticiones correctas**
  Cuando la solicitud es válida, el servidor responde con `200 (OK)`.
  - Incluso si el cliente pide una lista y está vacía, es una respuesta correcta: 200, no un error.

Método listen()

Después de definir todas las rutas, iniciamos el servidor con `app.listen()`. Este método recibe dos argumentos:

- El **puerto TCP** en el que el servidor escuchará peticiones.
- Una **función callback** que se ejecutará al arrancar el servidor, normalmente para mostrar un mensaje informativo.

```
const puerto = process.env.PORT || 3000;
// Si existe la variable de entorno PORT, la usamos.
// Si no, el servidor se inicia en el puerto 3000.

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

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.`));
```

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.
```

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