«Talento Tech»

Back-End

Node JS

Clase 09

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 9 - Creando un Servidor Web

Temario:

1. Servidor Web Node Nativo
2. Express JS
3. MIddlewares

Objetivos de la Clase

Los objetivos de esta clase se centran en proporcionar a los estudiantes una comprensión sólida y práctica del desarrollo de servidores en entornos JavaScript, utilizando tanto Node.js nativo como el marco de trabajo Express.js. Se espera que al finalizar la misma, los estudiantes puedan configurar y ejecutar un servidor básico en Node.js sin depender de librerías externas, entendiendo el funcionamiento interno de las solicitudes y respuestas HTTP. Además, aprenderán a instalar, configurar y utilizar Express.js, explorando herramientas como Express Generator para agilizar la creación de proyectos. Un enfoque clave será implementar un servidor que sirva contenido desde una carpeta pública, integrando conceptos básicos de enrutamiento y gestión de recursos estáticos. Finalmente, se introducirá el concepto de middlewares en Express.js, destacando su importancia en la personalización y el manejo de flujos de datos dentro de las aplicaciones web.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Servidor Web Node Nativo

Introducción a Node.js

Node.js es un entorno de ejecución para JavaScript que permite construir aplicaciones del lado del servidor. Una de sus múltiples funcionalidades incluye la capacidad de crear servidores web utilizando módulos nativos como `http`.

Paso a paso para crear un servidor

Una vez tengamos nuestra configuración inicial para proyectos de Node, habiendo inicializado NPM y teniendo nuestro archivo de punto de entrada llamado `index.js` podemos comenzar a trabajar en nuestro servidor:

1. **Importar el módulo http**: Para crear un servidor, debemos utilizar el módulo http nativo de Node.js. Este módulo proporciona herramientas para manejar solicitudes y respuestas HTTP.

```
const http = require('http');
```

2. **Crear el servidor:** Usamos el método `createServer` del módulo `http`, que recibe una función callback. Esta función tiene dos parámetros principales: `req` (con los datos de la solicitud del cliente) y `res` (la respuesta del servidor que configuramos).

```
const server = http.createServer((req, res) => {
  // Código de estado HTTP
  res.statusCode = 200;
  // Configuramos el tipo de contenido
  res.setHeader('Content-Type', 'text/plain');
  // Enviamos la respuesta
  res.end('Hola, mundo!');
});
```

En el código anterior configuramos el `statusCode` con el valor **200** para indicarle al navegador que hizo la petición que la solicitud se procesó correctamente. Además el `setHeader` nos permite avisarle al receptor que estamos enviando contenido en texto plano y finalmente mediante `end` indicamos el cuerpo de la respuesta que es lo que se terminará mostrando al usuario final. En esta ocasión es una cadena de texto pero podría ser sin problemas código HTML.

3. **Escuchar en un puerto:** Finalmente, el servidor necesita escuchar en un puerto específico para poder recibir solicitudes. Comúnmente se utiliza el **puerto 3000** para desarrollo pero podría ser cualquier otro. Recordemos que los puertos de nuestra PC sirven como canal de entrada a los distintos servicios que se comunican a través de la red, en este caso para llegar a nuestro servidor.

```
const PORT = 3000;

server.listen(PORT, () => {
  console.log(`Servidor corriendo en http://localhost:${PORT}`);
});
```

4. **Ejecutamos el código:** mediante el comando `node index.js` o en caso de haber configurado un script llamado **start** en nuestro archivo `package.json` podemos utilizar el comando `npm run start` para iniciar nuestro servidor. Finalmente, entrando a la URL http://localhost:3000 desde el navegador, vamos a poder leer el mensaje **Hola, mundo!** configurado como respuesta.

Este servidor básico nos brinda una base para entender cómo funciona un servidor web, pero presenta limitaciones cuando intentamos ampliarlo. Por ejemplo, el código actual responde únicamente a la ruta por defecto "/" (https://miservidor.com/), lo que complica la configuración de rutas adicionales como "/productos" o "/productos/1". Además, este servidor solo responde con contenido estático, sin capacidad para generar respuestas dinámicas o realizar operaciones adicionales antes de responder.

Aunque es posible implementar todas estas funcionalidades de manera nativa con Node.js, hacerlo puede ser tedioso y propenso a errores. Por esta razón, es común utilizar frameworks como Express.js o Nest.js, que ofrecen herramientas y soluciones que simplifican y optimizan el desarrollo de aplicaciones web.

Express JS

Express.js es un framework minimalista y flexible para Node.js que permite construir servidores web y APIs de forma rápida y sencilla. Es ampliamente utilizado por su eficiencia en el manejo de rutas y middlewares, facilitando el desarrollo backend.

Una vez tengamos nuestro proyecto de node inicializado como vimos en módulos anteriores, es necesario que instalemos el framework para comenzar a utilizarlo.

Instalación de Express.js

1. El primer paso es instalar Express como una dependencia de nuestro proyecto.

   ```
   npm install express
   ```

   Al finalizar, verás que la carpeta `node_modules` y la dependencia correspondiente a Express en el archivo `package.json` se han agregado a tu proyecto.

   Recuerda colocar la propiedad `"type": "module"` en el `package.json` para indicar que utilizaremos esa sintaxis para importar y exportar código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Paso a paso para crear un servidor

Ahora que tenemos Express instalado, veamos cómo crear un servidor y qué diferencias encontramos con la opción nativa.

1. En `index.js` importa el módulo de Express.

```
import express from 'express';
```

2. Utiliza `express` para crear una nueva instancia en nuestra aplicación.

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

3. Ahora para definir la respuesta a una ruta podemos llamar al método `.get()` de `app`.

```
app.get('/', (req, res) => {
  res.send('Hola, mundo desde Express!');
});
```

Este método recibe dos (2) parámetros, en el primero definimos la ruta a la que responderá nuestra aplicación y el segundo es una función de callback que captura la request (petición) y entrega una *response* (respuesta).

La palabra **get** del método puede resultar un tanto conocida y es que responde a peticiones HTTP realizadas a través del método GET que conocimos anteriormente. Además de **get**, podemos utilizar otros como **post**, **put**, **delete**, etc.

4. Una vez definida nuestra primera ruta debemos indicarle al servidor en que puerto estará escuchando peticiones. Para ello utilizaremos el método `.listen();` disponible en nuestra variable `app`.

```
const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}`);
});
```

De esta manera, logramos montar nuestro primer servidor web con Express. Si ejecutamos el comando `npm run start` e ingresamos a la ruta **http://localhost:3000** podremos ver el mensaje *"Hola, mundo desde Express!"*.

Sin embargo esto no es todo el potencial que posee Express, del mismo modo que configuramos una ruta que escuche las peticiones del tipo **GET** realizadas a "/", podemos configurar otras, por ejemplo:

```
app.get('/productos', (req, res) => {
   res.send('Bienvenid@ a la página de productos');
});

app.get('/productos/14', (req, res) => {
    res.send('Estás viendo el producto N° 14.');
});
```

Más adelante aprenderemos el concepto de ruteo y cómo devolver información estática y dinámica frente a las diferentes peticiones.

Ventajas de usar Express.js

- **Simplicidad:** Proporciona una API sencilla para manejar rutas y solicitudes HTTP.
- **Flexibilidad:** Permite integrar middlewares y otras librerías con facilidad.
- **Ecosistema:** Cuenta con una amplia comunidad y múltiples extensiones.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Express Generator

Express Generator es una herramienta proporcionada por el equipo de Express.js que permite crear la estructura inicial de un proyecto de manera automática y estandarizada. Su objetivo principal es agilizar el proceso de configuración inicial al generar los archivos y directorios básicos necesarios para un proyecto de Express.

¿Para qué sirve?

Express Generator simplifica la creación de aplicaciones al proporcionar una base sólida con las configuraciones iniciales más comunes. Incluye:

- **Estructura de carpetas predefinida:** Organiza los archivos del proyecto en carpetas como `routes`, `views` y `public`.
- **Plantillas integradas:** Ofrece compatibilidad con motores de plantillas como Pug, lo que facilita la generación de vistas dinámicas.
- **Archivos de configuración inicial:** Configura módulos y middlewares esenciales desde el inicio.

Ventajas de usar Express Generator

1. **Ahorro de tiempo:** Evita la necesidad de configurar manualmente la estructura del proyecto.
2. **Estandarización:** Garantiza que los proyectos sigan una organización comúnmente aceptada, facilitando la colaboración en equipos.
3. **Escalabilidad:** Proporciona una base modular que permite agregar funcionalidades sin desordenar el código.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Instalación de Express Generator

Para instalar esta herramienta, utiliza el siguiente comando:

```
npm install -g express-generator
```

Este comando instala Express Generator de manera global, es decir que podrás crear un proyecto nuevo con express desde la terminal, en cualquier momento y desde cualquier sitio de tu PC.

Creación de un proyecto con Express Generator

1. Para generar un nuevo proyecto, ejecuta el siguiente comando:

   ```
   express nombre-del-proyecto
   ```

2. Esto creará una nueva carpeta con la estructura inicial del proyecto.

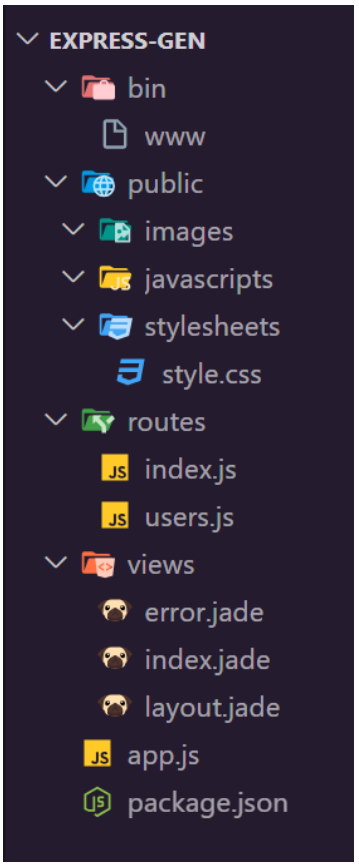
   ```
   cd nombre-del-proyecto
   ```

3. Una vez generado, navega al directorio del proyecto e instala las dependencias necesarias:

   ```
   npm install
   ```

4. Finalmente, ejecuta el proyecto con:

   ```
   npm start
   ```



Como se observa en la imagen, esta herramienta genera un entorno de trabajo con Express, listo para empezar a desarrollar.

Una de sus principales ventajas es su flexibilidad, ya que permite configurar el proyecto según las necesidades específicas. Por ejemplo, en una API Rest como la que estamos creando, la carpeta `views` destinada a vistas, común en aplicaciones basadas en el modelo MVC, no será necesaria, por lo que podemos removerla y continuar solamente con las capas que necesitamos y agregar las que falten.

Un aspecto clave del entorno generado es la carpeta `/public`. Esta carpeta juega un papel esencial en los servidores web, ya que aloja todos los archivos estáticos, es decir, aquellos que no cambian durante la ejecución de la aplicación. Aquí podemos colocar imágenes, archivos JavaScript destinados al navegador, HTML, CSS, entre otros recursos.

Si alguna vez usaste un servicio de hosting tradicional para un sitio web estático, habrás notado la existencia de una carpeta llamada `public_html`. Esa carpeta tiene la misma función que `/public` en Express: almacenar y servir los archivos públicos del servidor web.

Esta similitud subraya cómo Express Generator simplifica la gestión de estos recursos, creando una estructura consistente y eficiente desde el inicio.

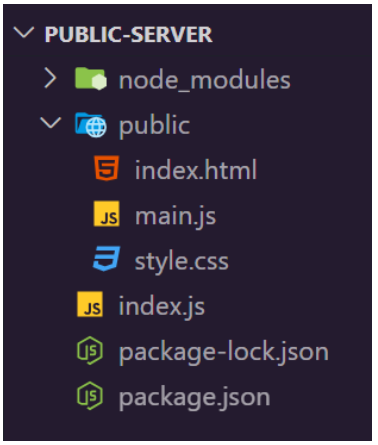
Primer servidor de archivos estáticos

Si bien nuestro objetivo principal es enfocarnos en la creación de servicios del tipo API Rest, en esta ocasión vamos a aprender a configurar un servidor con Express.js que sirva archivos estáticos directamente desde una carpeta designada, comúnmente llamada `public`. Este enfoque es fundamental para proyectos que requieren servir contenido como imágenes, estilos CSS, archivos JavaScript y otros recursos accesibles al cliente de manera directa.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Paso a paso para crear un servidor de archivos estáticos

1. Crea la carpeta **`public`:** Dentro de tu proyecto, crea un directorio llamado `public`. Aquí es donde alojarás los archivos estáticos.



2. Una vez realizada la configuración básica, en el archivo `index.js` usamos el middleware **`express.static`:** que se utiliza para servir archivos estáticos. Es importante que apunte al directorio `public` por lo que nos ayudamos del módulo nativo `path` junto con `__dirname` para ubicar correctamente la referencia a la carpeta dentro del servidor.

```
import express from 'express';
import { join, dirname } from 'path';
import { fileURLToPath } from 'url';

const __filename = fileURLToPath(import.meta.url);
const __dirname = dirname(__filename);

const app = express();

// Configurar middleware para servir archivos estáticos
app.use(express.static(join(__dirname, 'public')));

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Servidor en http://localhost:${PORT}`);
});
```

3. Finalmente agregamos contenido a nuestros archivos estáticos

```html
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="UTF-8">
   <title>Servidor de archivos estáticos</title>
   <link rel="stylesheet" href="style.css">
 </head>
 <body>
   <h1>Hola desde nuestro servidor de archivos estáticos</h1>
 </body>
 <script src="main.js"></script>
</html>
```

4. Ahora, si corremos nuestro servidor y nos dirigimos a la ruta `http://localhost:3000` desde el navegador, nos encontraremos con nuestro sitio web estático:

*Tip: podés probar volcando todos los archivos de un sitio web estático que hayas creado en el pasado dentro de la carpeta **public** y deberías poder acceder y navegar localmente desde el navegador.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Middlewares

En la sección anterior, al momento de crear nuestro servidor estático mencionamos un concepto hasta el momento desconocido: **middleware**. Particularmente utilizamos `express.static` dentro de `app.use()` lo que nos permitió indicarle al servidor que antes de consultar cualquier ruta, debía buscar dentro de la carpeta de archivos públicos.

¿Qué es un Middleware?

Un middleware es una función que se ejecuta entre el momento en que un cliente realiza una solicitud y el momento en que el servidor envía una respuesta. Cada middleware tiene la capacidad de modificar los objetos de solicitud y respuesta, finalizar el ciclo de solicitud/respuesta, o pasar el control a otro middleware utilizando la función `next()`.

Tipos de middlewares

1. **Middlewares de aplicación:** Se ejecutan para todas las rutas o rutas específicas en tu aplicación.
2. **Middlewares de ruta:** Se utilizan para manejar rutas específicas.
3. **Middlewares de terceros:** Son librerías externas que extienden las capacidades de Express, como `body-parser` o `cors`.
4. **Middlewares integrados:** Express incluye algunos middlewares listos para usar, como `express.static` para servir archivos estáticos.

En el caso de los dos primeros, considerados como middlewares que se crean de forma manual, reciben tres parámetros: `req`, `res` y `next`, donde el único que no conocemos es **next**. Este parámetro nos permitirá indicarle al servidor cuando termina la lógica de nuestro **middleware** para que continúe con la ejecución de nuestro programa.

Una forma sencilla de ver un **middleware** es como una función que intercepta las solicitudes a nuestro servidor, realiza determinada lógica o acción y continúa con la ejecución. Cabe destacar, que en determinadas ocasiones aprovecharemos este recurso para evitar que nuestro programa siga su curso establecido.

Ejemplo básico de un middleware

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

// Middleware de aplicación
app.use((req, res, next) => {
    console.log(`Datos recibidos: ${req.method} ${req.url}`);
    next(); // Pasa el control al siguiente middleware o ruta
});

// Ruta principal
app.get('/', (req, res) => {
    res.send('Hola desde Express con middlewares!');
});

const PORT = 3000;
app.listen(PORT, () => {
    console.log(`Servidor en http://localhost:${PORT}`);
});
```

En el ejemplo anterior, vemos que el middleware intercepta TODAS las peticiones al servidor, imprime por consola el método HTTP utilizado y la URL requerida y luego mediante el uso de `next();` continúa la ejecución, llegando a `app.get(...);` donde si la ruta solicitada encaja con la allí definida, enviará la respuesta preestablecida.

Más adelante, aprenderemos a trabajar con **middlewares** de ruta, que resultan más específicos y también aprenderemos sobre los casos de uso más tradicionales, donde destacan el manejo de errores, autenticación, rutas no encontradas, entre otros.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicio Práctico.

Ejercicio 1 - Configurando el Proyecto Base

Matías te observa con una sonrisa intrigada mientras sostiene una taza de café. Está contento de lo que has logrado hasta el momento y aún más por tu reciente incorporación al equipo.

“Bueno, esto está tomando forma, pero necesitamos establecer una base sólida para nuestro proyecto integrador”, comenta mientras señala tu laptop.

Sabrina asiente. “Crear un proyecto bien estructurado es como construir los cimientos de un edificio. Queremos que este proyecto sea el punto de partida para algo más grande.”

Misión:

1. Crea un nuevo proyecto con el comando `npm init -y` para generar un archivo `package.json` básico.
2. Inicializa un repositorio de Git en el proyecto con `git init`.
3. Crea un archivo `.gitignore` que excluya la carpeta `node_modules`.
4. Configura el archivo `package.json` para que soporte el estándar de imports ES Modules. Modifica el archivo añadiendo `"type": "module"`.
5. Define un script en `package.json` para ejecutar el proyecto con Node.js usando `"start": "node index.js"`.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejercicio 2 - Tu Primer Servidor Web

Impresionados por tu habilidad para establecer las bases del proyecto, Matías y Sabrina deciden aumentar el desafío.

"Ahora es momento de que entres al mundo real del desarrollo web con Express," dice Matías, colocándose frente a la pizarra. "Queremos que configures un servidor básico que pueda manejar solicitudes."

Sabrina toma la palabra. "Recuerda, la clave está en lo simple pero funcional. Con una ruta bien configurada, demostraremos que nuestro servidor está listo para crecer."

Misión 2:

1. Usa la estructura creada en el ejercicio anterior para iniciar un servidor web.
2. Instala Express con `npm install express`.
3. Configura un servidor básico que corra en el puerto `3000` usando Express.
4. Agrega una ruta `/ping` que responda con el texto plano "/pong" cuando sea visitada desde un navegador.

Matías concluye: "Queremos ver un servidor funcional, código limpio y bien comentado. Este servidor será el corazón de nuestro proyecto integrador, así que pon tu mejor esfuerzo."

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Materiales y Recursos Adicionales:

Documentación oficial de Node.js para profundizar en el uso del módulo `path`.

Documentación de Express.js para explorar sus características avanzadas.

Entornos para probar API REST, como Postman y Insomnia.

Preguntas para Reflexionar:

- ¿Qué ventajas y desventajas encuentras en configurar un servidor nativo con Node.js en comparación con usar un framework como Express.js?
- ¿Por qué es importante organizar los archivos estáticos en una carpeta como `public`, y cómo mejora esto la seguridad y accesibilidad de un servidor?
- Al usar herramientas como Express Generator, ¿en qué casos prefieres configurar manualmente un proyecto y en cuáles optarías por su uso?
- ¿Qué elementos considerarías al estructurar un servidor web que escale para manejar miles de usuarios simultáneamente?

Próximos Pasos:

- **Modelando una API Rest**: Nos iniciaremos en la metodología para crear nuestra primera API Rest.
- **Request y Response**: llegó el momento de aplicar de forma práctica todo nuestro conocimiento sobre la comunicación web.
- **Capa lógica**: controlando la respuesta de nuestra aplicación.

Buenos Aires Ciudad