Introducción a Express.js

Objetivos

- ¿Qué es Express.js?
- ¿Qué es un servidor web?
- Crear nuestro primer servidor web
- Arquitectura Cliente – Servidor
- El objeto request(req) y response(res)
- Devolviendo un JSON en una ruta
- Recibir un payload
- ¿Qué es un middleware?
- ¿Qué es Cors?

¿Qué es Express.js?

Express.js es un marco (framework) de desarrollo web para Node.js, un entorno de ejecución de JavaScript del lado del servidor. Express simplifica el proceso de construir aplicaciones web al proporcionar una serie de utilidades y características que facilitan la creación de servidores y el manejo de rutas, solicitudes y respuestas HTTP.

Características de Express.js

- Express.js permite crear rápidamente un servidor web con solo unas pocas líneas de código.

- Express facilita la definición de rutas para manejar diferentes solicitudes HTTP (GET, POST, etc.).

- Express está diseñado para ser minimalista y flexible, proporcionando solo las funciones esenciales para construir aplicaciones web.

- Gracias a su diseño modular y sencillo, Express es escalable y se adapta tanto a proyectos pequeños como a aplicaciones empresariales más grandes.

¿Qué puedo hacer con Express.js?

- **Desarrollo de Aplicaciones Web**: Puedes construir aplicaciones web completas utilizando Express para manejar rutas, renderizar vistas, y gestionar solicitudes y respuestas HTTP.

- **APIs RESTful**: Express es ampliamente utilizado para construir APIs RESTful, facilitando la creación de servicios web que siguen principios REST para la comunicación entre clientes y servidores.

- **Servidores HTTP Personalizados**: Express te permite crear servidores HTTP personalizados con configuraciones específicas y características adaptadas a tus necesidades.

- **Manejo de Formularios y Datos**: Puedes gestionar fácilmente la recepción de datos de formularios HTML y procesarlos utilizando middleware como body-parser.

- **Autenticación y Autorización**: Implementar sistemas de autenticación y autorización es más sencillo con Express, y puedes integrar fácilmente bibliotecas como Passport.js.

- **Manejo de Errores**: Implementar manejo de errores es directo en Express. Puedes definir middleware específico para manejar errores y personalizar las respuestas en consecuencia.

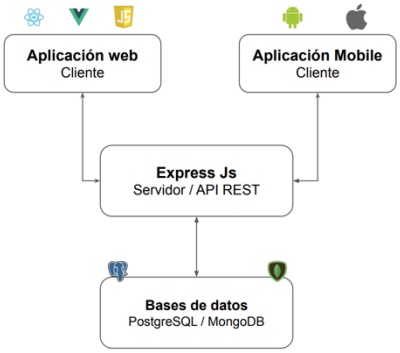
- **Protección contra Ataques**: Express incluye funciones integradas y middleware para ayudar a proteger tu aplicación contra ataques comunes, como la inyección de SQL, ataques de CSRF, entre otros.

- **Integración con Bases de Datos**: Puedes integrar Express con diversas bases de datos, como MongoDB, MySQL, PostgreSQL, etc., utilizando bibliotecas como Mongoose, Sequelize, etc.

Servidores web

Un servidor web es un programa informático que acepta solicitudes a través del protocolo HTTP (Hypertext Transfer Protocol) o HTTPS (HTTP Secure) de los clientes (navegadores web) y entrega respuestas en forma de páginas web u otros recursos. Los servidores web gestionan el flujo de datos entre el cliente y el servidor, permitiendo la comunicación y el intercambio de información.

En el contexto de Node.js y Express, cuando se habla de "servidor web", nos referimos a una aplicación construida con Node.js que utiliza el marco Express para gestionar las solicitudes y respuestas HTTP. En este escenario, el servidor web está específicamente diseñado para ejecutarse del lado del servidor y manejar la lógica de la aplicación web.



Creando nuestro primer servidor web con Express.js

Para empezar a utilizar Express js primero hay que instalarlo por npm, para esto crea una nueva carpeta, dentro de la carpeta ejecuta la siguiente línea de comando para iniciar un proyecto NPM:

```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-express → npm init -y
```

Una vez ejecutado este comando se creará un archivo en la base de nuestro proyecto llamado package.json, este es un archivo de configuración fundamental en proyectos Node.js. Este archivo es utilizado para gestionar las dependencias del proyecto, scripts de ejecución, metadatos y otra información importante.

```
{
  "name": "introduccion-a-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Una vez creada la base del proyecto comenzaremos con la creación de nuestro servidor web para eso ejecutaremos por la terminal el siguiente comando:

```
Node.js Version: v16.20.2 📦 Ruby Version: 3.2.2 💎 Desktop/introduccion-a-express → npm i express
```

Una vez ejecutado se creará la carpeta node modules esta carpeta contiene las dependencias del proyecto. Cada paquete instalado tiene su propia carpeta dentro de node_modules, y esta estructura se crea automáticamente para organizar las dependencias del proyecto.

Ahora crearemos el archivo que contendrá la lógica de nuestro servidor, para eso crearemos un archivo llamado server.js en la raíz de nuestro proyecto y tendrá el siguiente código

server.js > ...

```
import express from "express"
const app = express()
const PORT = process.env.PORT || 3000


app.get("/home", (req, res) => {
  res.send("Hello World Express Js")
})


app.listen(PORT, console.log("¡Servidor encendido!"))
```

Como se puede apreciar, primero importamos el paquete de express usando ECMAScript 6, para lograrlo no olvides especificar lo siguiente en tu archivo package.json.

```
{
  "name": "introduccion-a-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  ▷ Debug
  "scripts": {
```

Siguiendo con la explicación del código podemos describir lo siguiente:

Si bien en el código de ejemplo la parte donde está el puerto, esta con una variable(se explicará más adelante), se debe entender que esta parte solo recibe el número de puerto donde se va a iniciar nuestro servidor.

Para crear una ruta usamos app que es una instancia de express y con ella podemos crear las rutas que necesitemos solo debe cumplir con lo siguiente:

```
app.method( path, callback )
```

En donde:

- **app:** Es una instancia de Express.
- **method:** Es un método de solicitud HTTP en minúscula.
- **path:** Es la ruta que será consultada por la aplicación cliente, por ejemplo **/home**
- **callback:** Es la función que se ejecutará cuando la ruta sea consultada.

Para iniciar nuestro servidor ingresamos la siguiente instrucción en la terminal

```
Node.js Version: v16.20.2 Ruby Version: 3.2.2 Desktop/introduccion-a-express → node server.js
¡Servidor encendido en el puerto! 3000
```

Para probar la ruta creada podemos usar postman y enviar una solicitud a la siguiente url

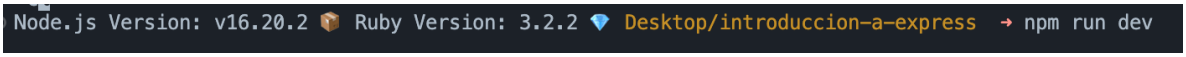
Y obtendremos la respuesta `Hello World Express Js`

Cuando levantamos un servidor de Express, este queda a la espera de solicitudes bloqueando la terminal

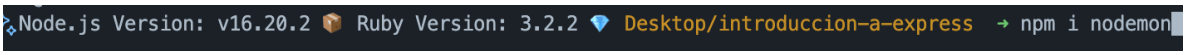
Para que los cambios que hagamos en el código del servidor sean aplicados es necesario apagar el servidor y volverlo a levantar. Esto supone un problema por que debemos apagar y encender el servidor por cada cambio que realicemos, para evitar esto podemos usar la nueva instrucción nativa watch (disponible desde la versión 18 de node), o si estas usando una versión anterior de node puedes instalar el paquete nodemon. Estas dos alternativas nos dan una ventaja ya que reinician el servidor de manera automática por nosotros manteniendo el servidor actualizado después de cada cambio.

Para usar watch solo debemos ir a nuestro package.json y en el apartado de scripts agregaremos lo siguiente.

Para iniciar nuestro servidor ahora usaremos la siguiente instrucción en la terminal



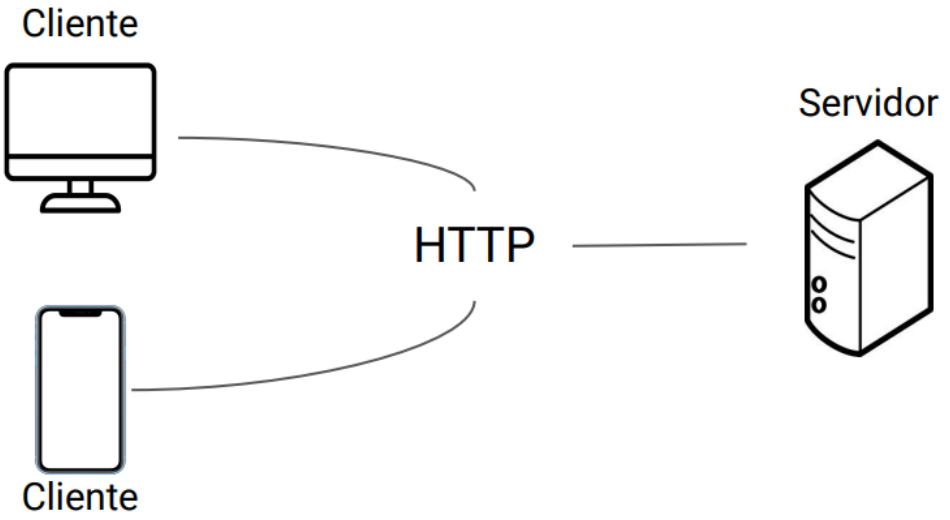
Para usar nodemos tenemos que instalar el paquete ingresando lo siguiente en nuestra terminal:



Y en nuestro package.json escribir lo siguiente en el apartado de scripts

Arquitectura Cliente – Servidor

La creación de servidores consiste en permitir la comunicación entre 2 aplicaciones que requieren compartir información o la posibilidad de ejecutar funciones a partir de consultas HTTP. Esto mismo se conoce como la arquitectura Cliente – Servidor



El objeto request(req) y response(res)

En Express el request y el response son objetos que podemos utilizar para obtener el detalle de la consulta y poder dar respuesta:

req (Request): El objeto req representa la solicitud HTTP que el servidor web recibe del cliente. Contiene información sobre la solicitud, como los encabezados, los parámetros de la URL, los datos del cuerpo (en el caso de solicitudes POST o PUT), y otros detalles relacionados con la solicitud del cliente.

Algunos ejemplos de propiedades en el objeto req incluyen:

- **req.params**: Contiene parámetros de la URL.
- **req.query:** Contiene los parámetros de la cadena de consulta.
- **req.body:** Contiene los datos del cuerpo de la solicitud (generalmente utilizado en solicitudes POST y PUT).
- **req.headers:** Contiene los encabezados de la solicitud.
- **req.method:** Contiene el método HTTP de la solicitud (GET, POST, etc.).

res (Response): El objeto res representa la respuesta que el servidor enviará al cliente después de procesar la solicitud. Se utiliza para enviar datos de vuelta al cliente, establecer encabezados de respuesta, y gestionar el estado de la respuesta.

Algunos métodos y propiedades comunes en el objeto res incluyen:

- **res.send():** Envía una respuesta al cliente.
- **res.json():** Envía una respuesta JSON al cliente.
- **res.status():** Establece el código de estado de la respuesta.
- **res.setHeader():** Establece un encabezado de respuesta.
- **res.redirect():** Redirige la solicitud a otra URL.

Devolviendo un JSON en una ruta

Frecuentemente, crearemos una API REST para entregar información a partir de una consulta bajo el método GET.

Realicemos este ejercicio creando un archivo productos.json en la misma carpeta donde creamos el proyecto

```
[
    {
        "id": 1,
        "nombre": "Audifonos",
        "precio": 18990
    },
    {
        "id": 2,
        "nombre": "Aro de Luz",
        "precio": 21990
    }
]
```

Ahora creemos una ruta GET /productos en nuestro servidor y utilicemos el objeto response de la función callback para devolver el JSON creado.

Para esto será necesario importar el módulo File System para leer el archivo productos.json y posteriormente devolverlo al cliente. El código del servidor quedaría así:

```
app.get("/productos", (req, res) => {
  try {
    const productos = JSON.parse(fs.readFileSync("productos.json"));
    res.json(productos);
  } catch (error) {
    res.status(500).json({ error: "Error al procesar la solicitud" });
    console.error("Error al procesar la solicitud:", error);
  }
});
```

Ahora realicemos una consulta HTTP con la extensión Thunder Client de VSC o postman consultando la siguiente ruta: http://localhost:3000/productos

La respuesta que veremos será la siguiente:

Status: 200 OK Size: 92 Bytes Time: 13 ms

Response Headers 6 Cookies Results Docs


```
[
  {
    "id": 1,
    "nombre": "Audifonos",
    "precio": 18990
  },
  {
    "id": 2,
    "nombre": "Aro de Luz",
    "precio": 21990
  }
]
```


Recibir un payload

Hasta ahora solo hemos devuelto información a un cliente que consulta una de nuestras rutas. Sin embargo, en muchas ocasiones necesitaremos recibir datos de parte del cliente para recibir estos datos usamos un payload que se entiende como la **carga** dentro de una consulta. En express esta carga se encuentra en el atributo body del objeto request.

```
Request: {
  body: {
    "id": 3,
    "nombre": "Monitor",
    "precio": 59990
  }
}
```

Al recibir datos de parte del cliente podemos utilizarlos para por ejemplo registrar un nuevo producto en nuestra base de datos (por ahora solo un json local).

Para poder leer el payload es necesario realizar una pequeña configuración en nuestra instancia app. Esta configuración consiste en ejecutar una función antes de cada ruta para parsear el contenido enviado desde el cliente. Para entender bien lo que vamos a hacer tenemos que conocer el concepto de middleware.

¿Qué son los middlewares?

Un middleware en el contexto de desarrollo web, y específicamente en el marco de Express.js para Node.js, es una función que tiene acceso a los objetos de solicitud (req), respuesta (res), y la siguiente función en el ciclo de solicitud-respuesta. Los middlewares son esenciales para realizar tareas específicas durante el procesamiento de una solicitud HTTP antes de que esta llegue a su destino final.

En Express, las funciones middleware se pueden utilizar para realizar diversas tareas, como la autenticación, la manipulación de encabezados, el registro, el manejo de errores, la compresión de respuesta, el análisis de cuerpos de solicitud, y muchas otras.

Un middleware puede tener una o más de las siguientes características:

- **Ejecución de código antes de llegar a la ruta final:** Un middleware se coloca en la cadena de manipulación de solicitudes antes de llegar a la ruta final. Esto permite ejecutar código antes de que se ejecute el controlador de ruta principal.

- **Modificación de la solicitud (req) o la respuesta (res)**: Un middleware puede realizar modificaciones en el objeto de solicitud o respuesta, como agregar propiedades, cambiar encabezados, o realizar otras transformaciones.

- **Finalización de la cadena de middleware o paso a la siguiente función**: Un middleware puede decidir finalizar la cadena y enviar la respuesta al cliente o pasar la solicitud y la respuesta a la siguiente función middleware en la cadena llamando a next().

El siguiente diagrama nos ayudará a entender cómo funciona

En Express Js los middlewares se declaran con el método .use() de express, pasando como argumento el middleware.

Para eso agregamos la siguiente línea en nuestro servidor

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

express.json(): es un middleware en el marco de desarrollo web Express.js que se utiliza para analizar datos JSON enviados en el cuerpo de las solicitudes HTTP. Este middleware se encarga de interpretar el contenido del cuerpo de la solicitud cuando este está en formato JSON y lo convierte en un objeto JavaScript para que sea más fácilmente manejable en las rutas y controladores.

Cuando se utiliza express.json(), Express asume que las solicitudes con el encabezado Content-Type: application/json contienen datos en formato JSON. El middleware se encarga de procesar esos datos y asignarlos a la propiedad req.body del objeto de solicitud (req).

Ahora que ya sabemos el concepto de middleware crearemos una ruta POST que sea utilizada para agregar un nuevo producto en nuestro JSON local.

Nuestro código quedara así:

```
app.post("/productos", (req, res) => {
  try {
    const producto = req.body;
    const productos = JSON.parse(fs.readFileSync("productos.json"));
    productos.push(producto);
    fs.writeFileSync("productos.json", JSON.stringify(productos));
    res.send("Producto agregado con éxito!");
  } catch (error) {
    res.status(500).json({ error: "Error al procesar la solicitud" });
    console.error("Error al procesar la solicitud:", error);
  }
```

```
const producto = req.body
```

Accedemos al payload de la consulta por medio del atributo body del request

```
productos.push(producto)

fs.writeFileSync("productos.json", JSON.stringify(productos))
```

Sobreescribimos el archivo productos.json luego de agregar el nuevo producto

Probemos nuestra ruta POST /productos realizando una consulta HTTP con la extensión Thunder Client pasando como payload lo siguiente:

```
{
    "id": 3,
    "nombre": "Monitor",
    "precio": 59990
}
```

Ahora si revisamos nuestro archivo local productos.json notaremos que el Monitor fue agregado.

¿Qué es Cors?

CORS significa "Cross-Origin Resource Sharing" (Compartir Recursos a través de Orígenes Cruzados, en español). Es un mecanismo de seguridad implementado en los navegadores web para controlar cómo las páginas web en un dominio pueden solicitar y acceder a recursos (como datos, imágenes, estilos, scripts, etc.) en otro dominio.

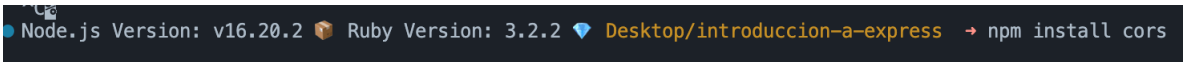
La política de mismo origen (Same-Origin Policy) es un principio de seguridad que restringe las solicitudes de recursos a un dominio diferente al de la página que realiza la solicitud. CORS se introdujo para permitir que los servidores indiquen a los navegadores si permiten que una solicitud de origen cruzado tenga acceso a sus recursos.

Cuando un navegador detecta que una página está tratando de hacer una solicitud a un dominio diferente al de la propia página, realiza una solicitud de "preflight" (vuelo previo) para obtener permisos antes de realizar la solicitud real. El servidor del dominio al que se realiza la solicitud debe responder con los encabezados CORS adecuados indicando si permite o no la solicitud.

Esto ayuda a prevenir ataques de seguridad al limitar qué recursos pueden ser solicitados desde orígenes diferentes. La implementación de CORS es crucial en el desarrollo web moderno, especialmente cuando se trabajan con aplicaciones web que consumen recursos de servicios externos.

Los servidores que creamos con Express Js bloquean las consultas de aplicaciones externas gracias a que los CORS están deshabilitados por defecto, no obstante podemos habilitarlos instalando un paquete de NPM llamado cors.

Para hacerlo solo debemos instalar el paquete a través de la terminal



Y por último lo integramos en nuestro servidor usando la instancia de app

El paquete Cors también recibe más de un origen si así lo necesitáramos, un ejemplo de cómo agregar más orígenes seria pasar un arreglo con los dominios autorizados a realizar consultas a nuestro servidor.

Nota: llegamos al fin de esta unidad no olvides crear tu archivo .gitignore e ignorar el archivo node modules