

Introducción a Express (Parte I)

¿Qué es Express?

Competencia

- Reconocer las características de Express y su utilidad para el desarrollo de servidores web.

Introducción

Express es un famoso framework escrito en JavaScript, utilizado por muchas empresas de reconocimiento mundial y recomendado por expertos en la materia. Desde el lado del servidor, aparece en el mundo del desarrollo con el objetivo de facilitar y agilizar la creación de aplicaciones web, trabajando en el backend y creando servidores, middlewares, enrutadores, capas de seguridad, API REST para consumo de aplicaciones Front-end, junto con gestionar contenido de forma dinámica, comunicaciones con bases de datos, entre otras cosas.

Podemos encontrar mucha variedad de tecnologías y lenguajes de programación, así como también de frameworks y librerías en la demanda laboral, no obstante, la gran mayoría de las ofertas que dedican su contenido a JavaScript en el lado del servidor, solicitan conocimientos en Express como requisito base para el desarrollo backend y full stack, por lo que, aprender sobre este framework elevará tus probabilidades de éxito para conseguir trabajo como Full Stack Developer en JavaScript.

¿Por qué aprender Express?

En el mundo de la programación siempre tendremos el dilema de qué tecnologías aprender para poder crear aplicaciones cada vez más avanzadas, eficaces y eficientes.

Los frameworks son creados con la principal idea de facilitar el desarrollo de aplicaciones, conteniendo miles de líneas de código con funciones y algoritmos complejos que ofrecen por medio de un simple llamado, soluciones rápidas a problemas complejos, como pueden ser las siguientes:

- La creación de servidores.
- Configuración de rutas.
- Creación y aplicación de middlewares.
- Seguridad de la data que viaja por protocolo HTTP.
- Creación de API REST.
- Manejo de errores devueltos por el servidor.
- Manejo de sesiones de usuarios, entre otros.

Uno de los impactos que ha generado este framework en la industria, es la creación de un servidor con métodos minimalista en solo 3 instrucciones sencillas que veremos más adelante. Pero realmente ¿Quién utiliza Express? Express es usado por grandes compañías como PayPal, Uber, Fox Sports, IBM y muy probablemente por tu aplicación web favorita, por lo que hoy en día, se puede encontrar en casi todos los desarrollos backend que utilicen Node.

En la siguiente imagen se aprecia que Express es una de las piezas fundamentales dentro de una aplicación, ya que sostiene la infraestructura para conectar las fuentes de datos, con la lógica de negocio y sus clientes móviles o Web.

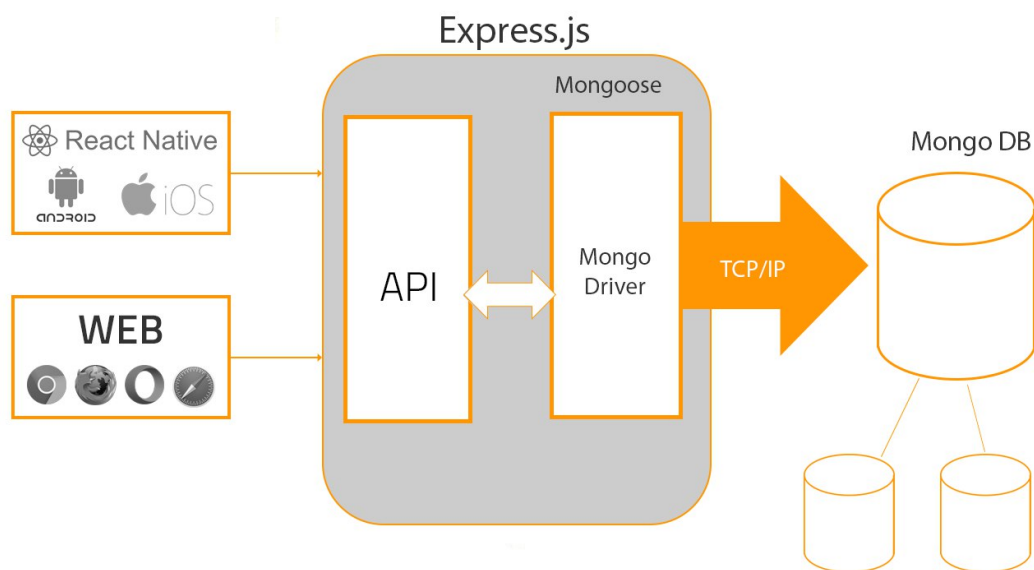


Imagen 1. Ejemplo de una arquitectura basada en Express.

Fuente: <https://github.com/osandadeshnan/expressjs-restful-apis-demo>

Express y su origen

La página web oficial lo define de la siguiente manera:

"Express es una infraestructura de aplicaciones web Node.js mínima y flexible que proporciona un conjunto sólido de características para las aplicaciones web y móviles."

Fuente: [Express js](https://expressjs.com/)

Express fue desarrollado por TJ Holowaychuk y lanzado al mercado en el 2010, es el framework de backend más popular en Node junto con los frameworks de JavaScript del lado del cliente como Angular, React Js o Vue js, y MongoDB (Gestor de bases de datos no relacional) formamos los famosos stack's de JavaScript MEAN, MERN y MEVN. En la siguiente imagen podrás ver que la tercera letra define la tecnología frontend del stack.



Imagen 2. Stacks de JavaScript.

Fuente: Desafío Latam

Características de Express

A continuación, te invitamos a revisar las principales características de express que se relacionan directamente con las ventajas de su utilización.

- **Infraestructura web rápida:** El rendimiento de express es envidiable, porque al final se traduce en puro código JavaScript y la velocidad de interpretación/compilación destaca en comparación con sus homólogos.
- **Minimalista:** Permite con simples llamados de sus métodos obtener funciones complejas, sin necesariamente conocer sus procesos internos.
- **Flexible:** Permite configurar su propia estructura a voluntad, definiendo tu propia estructura de desarrollo basada en Express.
- **Servidores:** La creación de un servidor en Express con solo 3 instrucciones toma menos de 1 minuto.
- **Entorno de Node:** Ya que su creación fue hecha bajo este entorno, se pueden aprovechar todas las potencialidades de Node para sumar a su versatilidad.
- **Creación de APIs REST:** De los usos más comunes, es su enrutamiento interno compatible con el protocolo HTTP y sus métodos, además de middlewares que ofrecen a tiempo de leopardo la creación de APIs y APIs REST.
- **Renderización de contenido dinámicos:** La compatibilidad con motores de plantillas como Pug, Ejs, Handlebars, entre otros.
- **Documentación:** En su página oficial se puede encontrar la documentación escrita por su equipo de desarrollo, la cual está disponible en varios idiomas.
- **Comunidad:** Para comienzos del año 2020, solo en NPM se registran más de 12 millones de descargas, sin contar con sus subpaquetes creados por desarrolladores en todo el planeta.

Ejercicio propuesto (1)

Debate con tus compañeros la siguiente pregunta ***¿Podremos con Express hacer todo lo que hemos hecho con Node en los módulos anteriores?***

Express vs Node puro

Competencia

- Reconocer las características de Express para establecer la diferencia en el desarrollo de un servidor con Node puro.

Introducción

Cada vez que empezamos a aprender a usar un framework o una nueva librería, es importante notar las diferencias entre el desarrollo de aplicaciones usando el lenguaje puro y el uso de librerías o frameworks, siendo reconocidos estos últimos por disminuir las dificultades en el desarrollo y logrando los mismos resultados. Por ello, en este capítulo aprenderás de forma práctica las principales diferencias de Express y Node puro.

La mejor manera para entender Express viniendo de conocimiento de Node, es haciendo prácticas comparativas entre ambos, aplicando uno de sus usos más conocidos; y el principal para el caso de Express, la creación de servidores. Una vez que tengas claras las diferencias, estarás listo para empezar a desarrollar ejercicios usando el framework más popular de Node.

Creación de servidores con Node puro y Express

¿Sientes que ha sido suficiente teoría por ahora? hemos llegado al momento más esperado, es hora de hacer código.

Instalación de Express

Lo primero será instalar Express a través de NPM, así que crea una carpeta y ábrela en tu editor de código favorito. En esta muestra, usaré Visual Studio Code aprovechando su terminal integrada. Ocupa la siguiente línea de comandos para la instalar Express.

```
npm install express --save
```

Con la carpeta creada y nuestro framework instalado, procederemos a crear 2 archivos, el primero llamado "serverExpress.js" y el segundo "serverNode.js", ¿Cuál será el objetivo de esto? Ver la diferencia entre crear un servidor con Node puro y llegar a lo mismo usando Express.

En la siguiente imagen puedes ver como debería estar tu árbol de archivos actualmente.

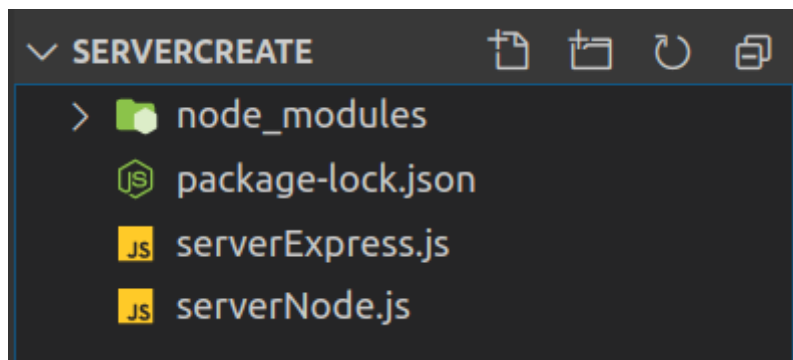


Imagen 3. Árbol de archivos para la creación de servidores
Fuente: Desafío Latam

Considerando que en el mundo de la tecnología todo puede cambiar en cualquier momento y tenemos actualizaciones consecutivamente, te recomiendo que visites siempre la página oficial de Express, en donde encontrarás todas las actualizaciones que sus desarrolladores puedan lanzar. Acá te dejo el [link](#) oficial y en español a la sección de instalación para que verifiques este proceso.

Creación de servidores

Es momento de ver el versus que estabas esperando, iniciemos con el código de Node puro, ¿Recuerdas qué necesitábamos para esto? Por supuesto, el módulo `http` y sus métodos `createServer`, que recibe una función con los parámetros correspondientes al request y response, `listen` para definir el puerto que usaremos al comunicarnos con el servidor. Además, crearemos una ruta GET **/Inicio** con el que devolveremos un String saludando e indicando que estamos usando Node puro.

El código para iniciar un servidor con Node puro y una ruta inicial es el siguiente:

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

http
  .createServer((req, res) => {
    if (req.url == "/Inicio" && req.method == "GET") {
      res.end("Hola mundo! Servidor con Node js puro");
    }
  })
  .listen(3000, () => {
    console.log("El servidor está inicializado en el puerto 3000");
  });
```

Muy bien, nada nuevo por aquí, ahora veamos el código para iniciar un servidor con la misma ruta GET **/Inicio** con Express, para esto te muestro el siguiente código:

```
const express = require('express')
const app = express()

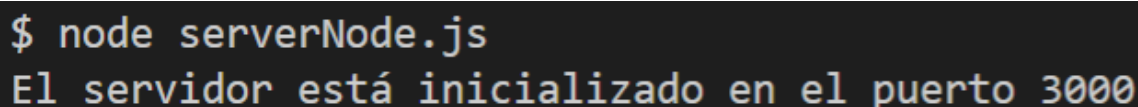
app.listen(3000, () => {
  console.log('El servidor está inicializado en el puerto 3000')
})

app.get("/Inicio", (req, res) => {
  res.send("Hola mundo! Servidor con Express js &#128526;")
})
```

Tomate unos minutos para intentar identificar las semejanzas y diferencias entre ambos códigos, y descubrirás el porqué Express se considera un framework minimalista. No te asustes con la combinación de caracteres al final del mensaje, es solo un emoji que podremos ver una vez que consultemos la ruta del servidor creada. Te dejo el siguiente [link](#) para que puedas revisar los diferentes emoticones que puedes ocupar en HTML y hacer tu desarrollo aún más entretenido. Ahora que tenemos ambos archivos escritos, levantemos los servidores empezando con Node, para esto, ocupa la siguiente línea de comando en la terminal:

```
node serverNode.js
```

Deberás ver el mensaje que te muestro en la siguiente imagen por consola:

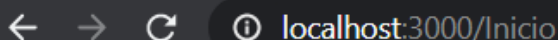


```
$ node serverNode.js
El servidor está inicializado en el puerto 3000
```

Imagen 4. Mensaje por consola al levantar un servidor con Node puro.

Fuente: Desafío Latam

Ahora, abre tu navegador y consulta la siguiente dirección: <http://localhost:3000/Inicio> y deberás ver lo que te muestro en la siguiente imagen:



← → ↻ ⓘ localhost:3000/Inicio

Hola mundo! Servidor con Node js puro

Imagen 5. Respuesta recibida en el navegador al consultar una ruta en el servidor con Node.

Fuente: Desafío Latam

Muy bien, ahora es el turno de Express, cancela la terminal en donde levantaste el servidor con Node puro y ocupa la siguiente línea de comando para levantar el servidor:

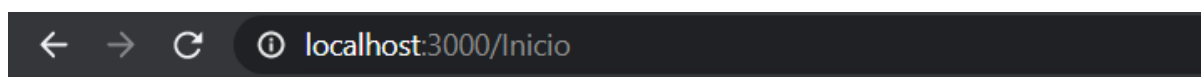
```
node serverExpress.js
```

Deberás ver el mensaje que te muestro en la siguiente imagen por consola, notarás que es el mismo que obtuvimos al levantar el servidor con Node:


```
$ node serverExpress.js  
El servidor está inicializado en el puerto 3000
```

Imagen 6. Mensaje por consola al levantar un servidor con Express.
Fuente: Desafío Latam

Dirígete nuevamente a tu navegador, consulta la misma dirección <http://localhost:3000/Inicio> y deberás ver lo que te muestro en la siguiente imagen:



Hola mundo! Servidor con Express js 😎

Imagen 7. Respuesta recibida en el navegador al consultar el servidor con Express.
Fuente: Desafío Latam

Ahí lo tenemos, mismo resultado pero desarrollado con el framework Express ocupando métodos minimalistas para crear un servidor y una ruta GET.

Consideraciones para el desarrollo inicial con Express

Si intentas abrir el servidor en el puerto 3000 desde el navegador, obtendrás la siguiente respuesta: "Cannot GET /", no te debes preocupar, esta es solo la forma de Express para decirte que aun no se ha configurado la respuesta para una consulta a la ruta raíz, es muy común cuando empezamos a utilizar Express creer que este mensaje significa que algo salió mal, pero no es así, al contrario por ser un mensaje genérico de Express, estamos confirmando que fue instalado correctamente y está respondiendo a nuestra consulta.

Otra consideración importante, es la instancia del paquete Express en el código, como puedes notar es almacenada en una constante llamada "app", y su método "listen" es invocado pasándole como parámetro el puerto y la función callback. Esta instancia por supuesto contiene muchos métodos que están muy bien descritos dentro de la documentación oficial del framework, te dejo el [link](#) para que lo revises.

Ejercicio propuesto (2)

Debate con tus compañeros la siguiente pregunta: En base a lo visto hasta ahora, **¿Te parece que el desarrollo de un servidor con Express tiene un código más limpio y cómodo de escribir que con Node puro?**

Creación de rutas

Competencias

- Reconocer el uso del enrutador de Express para la creación de rutas en el servidor.
- Construir un servidor con Express que disponibiliza una ruta por defecto para la devolución de una respuesta personalizada.

Introducción

Así como en el desarrollo con Node puro revisado en el capítulo anterior, Express puede crear rutas ocupando la instancia de Express, la cual comúnmente creamos con el nombre “app”. Esta instancia nos permite a través de sus métodos crear los diferentes tipos de rutas que ocupamos en el desarrollo de una API REST, como si no fuera poco, disponemos de los parámetros “request” y “response” en la función callback de cada ruta, para obtener las propiedades enviadas en una consulta desde el cliente y definir las propiedades para la respuesta de la misma petición.

En este capítulo, utilizaremos el utilitario POSTMAN para probar una de las rutas que crearemos en nuestro servidor, además de generar una ruta por defecto, en caso de ser consultada una dirección que no está definida en nuestro código. Ocupando esta metodología para probar rutas, conseguirás un desarrollo más estable, puesto que podrás testear tus rutas en todo momento y comprobar que la lógica que creas como respuesta de cada una, esté siendo recibida correctamente.

Enrutamiento (Routing)

Antes de empezar a crear rutas, partamos con la definición de enrutamiento. El enrutamiento se refiere a determinar cómo una aplicación responde a una solicitud del cliente a un punto final particular, que bien puede ser una URI o una URL, además de poseer algún método de solicitud HTTP específico (GET, POST, PUT, DELETE, etc.).

La definición de ruta en Express toma la siguiente estructura:

```
app.METHOD(PATH, CALLBACK)
```

Donde:

- “app” es una instancia de Express.
- “METHOD” es un método de solicitud HTTP en minúscula.
- “PATH” es el complemento de una ruta en el servidor.
- “CALLBACK”, también conocido como “handler”, es la función que se ejecuta cuando la ruta es consultada.

Ejercicio guiado: Toc toc

Desarrollar un servidor que disponibiliza una ruta **GET /TocToc** que al ser consultada devuelva el mensaje “¿Quién es?”, sigue los siguientes pasos para el desarrollo de este ejercicio:

- **Paso 1:** Crear una instancia de Express con una constante “Express”.
- **Paso 2:** Guardar en una constante “app” la ejecución de la instancia que creaste en el paso anterior. ¿Cómo es posible esto? Resulta que Express inicia como una función que al ser ejecutada devuelve un objeto con los diferentes métodos que ocuparemos a lo largo de este módulo.
- **Paso 3:** Ocupar el método “listen” de la constante “app” para definir en el primer parámetro el puerto 3000 y como segundo parámetro, una función callback que imprima por consola que el servidor fue inicializado en el puerto 3000.
-

- **Paso 4:** Ocupar el método “get” de la constante “app” para crear una ruta **/TocToc** que utilice como callback el parámetro “res” (response) y su método “send”. Este método nos servirá para devolver un contenido de texto, no obstante también se podrá interpretar en los navegadores como HTML.

```
// Paso 1
const express = require("express");

// Paso 2
const app = express();

// Paso 3
app.listen(3000, () => {
  console.log("El servidor está inicializado en el puerto 3000");
});

// Paso 4
app.get("/TocToc", (req, res) => {
  res.send("¿Quién es?");
});
```

Levanta el servidor y consulta la ruta <http://localhost:3000/TocToc> utilizando POSTMAN y deberás recibir lo que te muestro en la siguiente imagen.

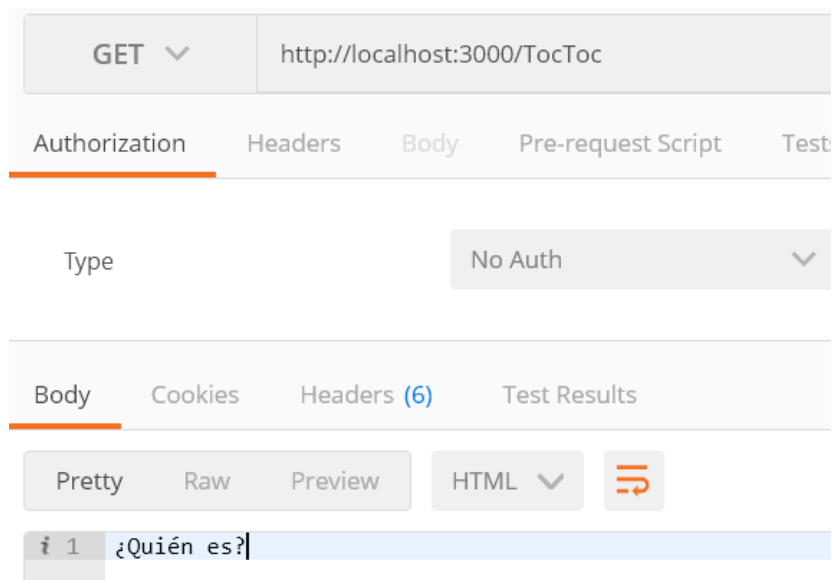


Imagen 8. Respuesta recibida al consultar la ruta **/TocToc** desde POSTMAN.
Fuente: Desafío Latam

Excelente, al recibir “¿Quién es?” como respuesta en POSTMAN, hemos comprobado el funcionamiento correcto de nuestra ruta **/TocToc**.

Ejercicio propuesto (3)

Desarrollar un servidor con Express que disponibilice una ruta **GET /QuienSoy**, que al ser consultada devuelva un string con tu nombre.

Ruta por defecto

Express evalúa cada una de las rutas (routes) que nosotros escribimos en el código, en el mismo orden en el que han sido escritas. ¿Pero qué pasa si un cliente envía una petición, por ejemplo <http://localhost:3000/contacto>? Express devuelve el siguiente mensaje:



Imagen 9. Respuesta recibida al consultar una ruta que no existe en el servidor.

Fuente: Desafío Latam

Esto debido a que nuestro sitio no posee esta ruta o existe un error en la URL. No obstante, es necesario manejar este tipo de situaciones e informar a los usuarios lo que está sucediendo con un mensaje personalizado. Para esta ocasión, se debe por lo mínimo informar al usuario que la página que busca no existe.

Para ello, se pueden capturar todas las rutas que no estén predefinidas en nuestro sitio Web, enviando lo que se conoce como un error genérico a los usuarios, por ejemplo, un código HTML que diga "Esta página no existe."

Ejercicio guiado: Sorry, aquí no hay nada :/

Incluir al servidor creado en la sección anterior una ruta genérica que devuelva el mensaje "Sorry, aquí no hay nada :/", al recibir una petición a una ruta que no coincida con ninguna de las rutas creadas. ¿Cómo definimos una ruta genérica? Al igual que en otros lenguajes de programación, el carácter asterisco (*) nos sirve para definir un "Todo" o un "Cualquier cosa", así que definiremos como "PATH" de esta ruta, un string con un asterisco, su contenido se ejecutará entonces al consultar cualquier ruta indefinida por el servidor, en este caso la probaremos consultando la ruta: <http://localhost:3000/MoteConHuesillo>.

Sigue los siguientes pasos para el desarrollo de este ejercicio guiado:

- **Paso 1:** Agregar una ruta GET y ocupa como primer parámetro lo siguiente: "*"
- **Paso 2:** Enviar como respuesta de la ruta genérica un encabezado centrado con HTML que diga "Sorry, aquí no hay nada :/"

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

app.listen(3000, () => {
  console.log("El servidor está inicializado en el puerto 3000");
});

app.get("/TocToc", (req, res) => {
  res.send("¿Quién es?");
});

// Paso 1
app.get("*", (req, res) => {
  // Paso 2
  res.send("<center><h1>Sorry, aquí no hay nada :/ </h1> </center>");
});
```

Ahora, abre tu navegador y consulta la ruta: <http://localhost:3000/MoteConHuesillo>, deberás recibir lo que te muestro en la siguiente imagen:



Imagen 10. Respuesta recibida al consultar la ruta /MoteConHuesillo.
Fuente: Desafío Latam

Como lo puedes notar, al consultar una ruta cualquiera estamos recibiendo el mensaje genérico escrito e interpretado por el propio navegador en formato HTML, de esta manera, podremos personalizar cualquier respuesta a nuestros usuarios al momento de intentar entrar a rutas inexistentes.

Consideraciones para las rutas genéricas

Las rutas genéricas deben ser ubicadas luego de la definición de todas las rutas, ya que al final, si Express no encuentra un handler adecuado para la petición recibida, usará este nuevo código de forma genérica para gestionar todas las peticiones que no coincidan con sus rutas preestablecidas.

Ejercicio propuesto (4)

Desarrollar un servidor con Express que contenga una ruta genérica y devuelva un párrafo de color rojo diciendo "La dirección que estas consultando no existe".

Objetos Request y Response

Competencias

- Construir una ruta que utilice el objeto request para extraer los parámetros de una consulta.
- Construir una ruta que utilice el objeto response para redireccionar a un usuario con el método redirect.

Introducción

Cuando creamos rutas en nuestros servidores bien sea con Node puro o con Express, disponemos de los objetos request y response que se presentan como parámetros de una función callback, que se ejecuta cuando una ruta determinada es consultada, por lo que gracias a ellos, podremos obtener y manipular las propiedades de la consulta HTTP.

Tenemos varias ventajas al poder ocupar el objeto request, ya que nos permite manipular los parámetros, cabeceras y método HTTP que se están usando para hacer esta consulta, entre otras cosas. En el caso del objeto response, podemos definir a través de sus métodos el código de estado HTTP, emitir una función que descargue un archivo alojado en el servidor y/o redireccionar al cliente que realizó la petición, lo cual implica que con este objeto nos comunicaremos con el cliente que consulta nuestro servidor.

En este capítulo, aprenderás a utilizar estos objetos elevando tus habilidades y capacidades en el desarrollo de servidores, por el simple hecho de ampliar las funcionalidades y tener acceso a las propiedades que acompañan una consulta HTTP.

Objeto Request

Dentro de las funciones de las rutas, se contempla el uso del objeto request, el cual corresponde a la instancia, que contiene las propiedades definidas en una consulta emitida por una aplicación cliente. Con este objeto, podemos acceder a los parámetros, cuerpo o encabezados y es el primer parámetro de la función que se ejecuta, al coincidir con una ruta en el servidor, comúnmente denominado como "req".

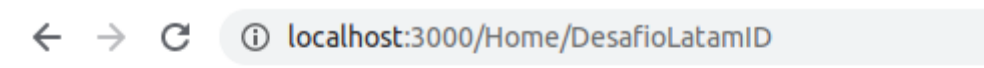
Este objeto contiene varios métodos y propiedades interesantes, pero los más utilizados son:

- **req.params:** Manejo de los parámetros recibidos por URL.
- **req.header():** Recibe como argumento el nombre de una propiedad y devuelve el valor de esta.
- **req.secure:** Confirma si el sitio de la consulta tiene el sello de seguridad https.
- **req.method:** Devuelve el método HTTP con el que se está haciendo la consulta.

En el siguiente código te muestro un ejemplo de cómo se vería una ruta GET **/Home** que recibe como subrecurso un parámetro "id":

```
app.get('/Home/:id', function (req, res) {  
  res.send(req.params.id)  
});
```

Como puedes ver, se está devolviendo el valor del mismo parámetro especificado en la ruta. Por ejemplo, si la ruta fuese <http://localhost:3000/Home/DesafioLatamID>, obtendremos lo siguiente:



DesafioLatamID

Imagen 11. Respuesta en el navegador al consultar una ruta que devuelve el parámetro ID.
Fuente: Desafío Latam

Ejercicio guiado: Número de la suerte

Crear un servidor con Express que escuche el puerto 3000. Una vez creado el servidor sigue los siguientes pasos para agregar una ruta **GET /azar/:numero** donde se obtenga el parámetro "numero" de la consulta y a la vez generar un número random entre el 1 y el 3. El objetivo será responder al cliente un mensaje de éxito o fracaso en el caso que coincida o no con el número aleatorio:

- **Paso 1:** Agregar una ruta GET **/azar/:numero**.
- **Paso 2:** Ocupar el objeto Math para generar un número entero aleatorio entre 1 y 3.
- **Paso 3:** Utilizar la propiedad "params" del objeto request para guardar en una constante el parámetro "numero".
- **Paso 4:** Utilizar un operador ternario para evaluar que el número generado de forma aleatoria, sea igual al recibido en la ruta como parámetro.

En caso de ser "true" la condición deberá devolver el mensaje "Hoy estás de suerte :)", en caso de ser false debes devolver el mensaje "Buena suerte para la próxima..."

```
// Paso 1
app.get("/azar/:numero", (req, res) => {
  // Paso 2
  const n = Math.floor(Math.random() * (4 - 1)) + 1;
  // Paso 3
  const numero = req.params.numero;
  // Paso 3
  numero == n
    ? res.send("Hoy estás de suerte :)")
    : res.send("Buena suerte para la próxima...");
});
```

Ahora intenta consultar desde el navegador varias veces la ruta que creaste definiendo un número como parámetro, y si tienes suerte recibirás lo que te muestro en la siguiente imagen:

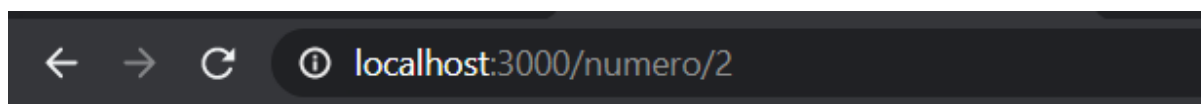


Imagen 12. Respuesta en el navegador al consultar una ruta que genera un número aleatorio
Fuente: Desafío Latam

Si quieres saber más sobre el objeto request, visita el siguiente [link](#) donde encontrarás toda su API en español según la documentación oficial de Express.

Ejercicio propuesto (5)

Desarrollar una ruta **GET /usuario/:nombre** que almacene el parámetro nombre en una constante y devuelva un mensaje de éxito en caso de que el nombre empiece con una vocal.

Objeto response

El objeto "response" representa la respuesta HTTP que enviamos desde el servidor cuando recibimos una solicitud HTTP. Por convención internacional, el objeto siempre se conoce como "res".

¿Cuáles son las cualidades de este objeto? Entre sus métodos más comunes están:

- **res.download():** Permite la descarga de un archivo dentro del servidor. Es importante destacar que esta es una cualidad muy potente del framework Express.
- **res.redirect():** Así como su nombre lo indica, se utiliza para redireccionar la consulta recibiendo como argumento la dirección en formato string.
- **res.sendStatus():** Recibe como argumento el código de estado que deseemos devolver como respuesta de la solicitud.

Ejercicio guiado: ¿Dónde puedo seguir estudiando?

Crear un servidor con Express que escuche el puerto 3000. Una vez creado el servidor sigue los siguientes pasos para agregar una ruta **GET /estudiar** que al ser consultada redirija al usuario usando el método "redirect" del objeto response. El objetivo será redirigirlo al sitio web de la academia [Desafío Latam](https://desafiolatam.com/).

Sigue los siguientes pasos para el desarrollo de este ejercicio:

- **Paso 1:** Agregar una ruta GET **/estudiar**.
- **Paso 2:** Ocupar el parámetro res y el método "redirect" para redireccionar al cliente al sitio web de Desafío Latam.

```
// Paso 1
app.get("/estudiar", function (req, res) {
  // Paso 2
  res.redirect("https://desafiolatam.com/");
});
```

Ahora levanta el servidor y consulta la ruta <http://localhost:3000/estudiar> y deberás ser direccionado a la página web de la academia, tal y como te muestro en la siguiente imagen.

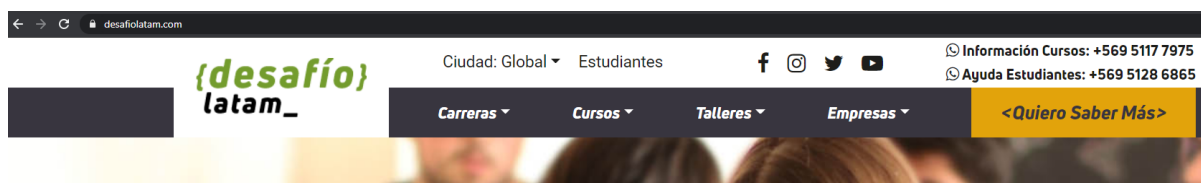


Imagen 13. Página web de la academia Desafío Latam

Fuente: Desafío Latam

Ejercicio propuesto (6)

Desarrollar una ruta **GET /musica** que redirija al cliente al sitio web de [Spotify](https://www.spotify.com/).

Middlewares

Competencias

- Reconocer el uso de los middlewares en el desarrollo de servidores con el framework Express.
- Construir un servidor que utilice middlewares para el procesamiento de una lógica antes de la ejecución de la ruta.

Introducción

En el capítulo anterior, aprendimos a crear rutas que al ser consultadas responden con un mensaje al cliente, pero ¿Qué pasa si necesitamos hacer algo antes de emitir la respuesta, como por ejemplo, validar los datos, parámetros o cabeceras de la consulta? Para esto tenemos los middlewares, los cuales se describen como funciones que se ejecutan entre la consulta de una aplicación cliente y la respuesta de esta petición devuelta por el servidor.

Aprendiendo a programar estas funciones, podrás controlar de mejor manera el flujo funcional que sucede en la interacción de una aplicación, construida bajo la arquitectura cliente-servidor, ¿Por qué? Porque la utilidad principal de los middlewares en el desarrollo de servidores es poder procesar diferentes validaciones utilizando los datos de la consulta, ¿Cómo? gracias al acceso que tienen con el objeto "request" y el objeto "response", con los cuales podremos extraer los parámetros, el payload, las cabeceras y/o el método utilizado en la consulta.

Middlewares

Entendiendo que las rutas creadas en nuestro servidor contienen una función que se ejecuta en el momento que son consultadas, por su parte, los middlewares son funciones que se activan antes que nuestras rutas y nos sirven como filtro o estación previa, en donde podemos definir diferentes validaciones. Su sintaxis es muy parecida a la creación de una ruta, pero el método que ocupamos es diferente, como puedes ver en el siguiente código de ejemplo:

```
// Middleware
app.use(<PATH>, (req, res, next) => {
  // Validaciones
  next()
});
```

En este ejemplo, se logra apreciar que hay un par de diferencias:

1. El método para la instancia “app” se llama “use”, a diferencia de las rutas que se definen según el método HTTP que deseemos disponer en nuestro servidor.
2. Ocupamos un tercer parámetro llamado “next”, que será una función que al ejecutarse permitirá la continuación de la consulta bien sea a otro middleware o a la ruta que se está consultando.

En base a lo anterior ¿En qué momento se ejecuta un middleware? Su ejecución sucede en el momento en que el servidor recibe una consulta y en dependencia del “PATH”, se ejecutará el middleware correspondiente primero. Cabe destacar que no es obligatorio programar middleware por cada ruta, normalmente creamos estas funciones “filtro” antes de ejecutar alguna ruta que devuelva contenido restringido o protegido.

Ejercicio guiado: El tiempo secreto

Validar la propiedad Authorization dentro de las cabeceras de una consulta realizada a la ruta **GET /Tiempo**. El objetivo será simular la verificación de tokens que se realizan en servicios web cuyos recursos están restringidos con verificación de usuarios basada en tokens. ¿Y cómo vamos a evaluar si existe o no la propiedad Authorization? Por supuesto, primero la vamos a extraer con el objeto request y a través de un operador ternario condicionamos la respuesta del servidor.

Nuestro condicional evaluará el valor de la propiedad Authorization y en caso de devolver “true”, se procederá a ejecutar la función “next”, de esa forma, el cliente podrá pasar a la

lógica que tenemos en la propia ruta **GET /Tiempo**, en donde se estará devolviendo un objeto con el tiempo del servidor. Sigue los pasos para realizar el siguiente ejercicio:

- **Paso 1:** Agregar un middleware para la ruta **/Tiempo**.
- **Paso 2:** Utilizar el parámetro "request" y su método "header" para almacenar en una constante el valor de la propiedad "Authorization".
- **Paso 3:** A través de un operador ternario evalúa la constante creada en el paso anterior y en caso de ser "true", ejecutar el parámetro "next()", de lo contrario responder con un mensaje "¿Quién es?".
- **Paso 4:** Crear una ruta **GET /Tiempo**.
- **Paso 5:** Crear y devuelve un objeto que incluya la fecha actual. Para esto, ocupa el `Date.now()`

```
// Paso 1
app.use("/Tiempo", (req, res, next) => {
  // Paso 2
  const Auth = req.header("Authorization");
  // Paso 3
  Auth ? next() : res.send("¿Quién es?");
});

// Paso 4
app.get("/Tiempo", (req, res) => {
  // Paso 5
  const tiempo = { time: Date.now() };
  res.send(tiempo);
});
```

Tomate unos minutos para leer el código anterior e identificar las diferencias que te mencioné entre un middleware y una ruta.

¿Qué pasaría si se consultara entonces a la ruta **/Tiempo**? Utiliza POSTMAN para realizar esta consulta sin especificar nada en las cabeceras, deberás obtener lo que te muestro en la siguiente imagen.

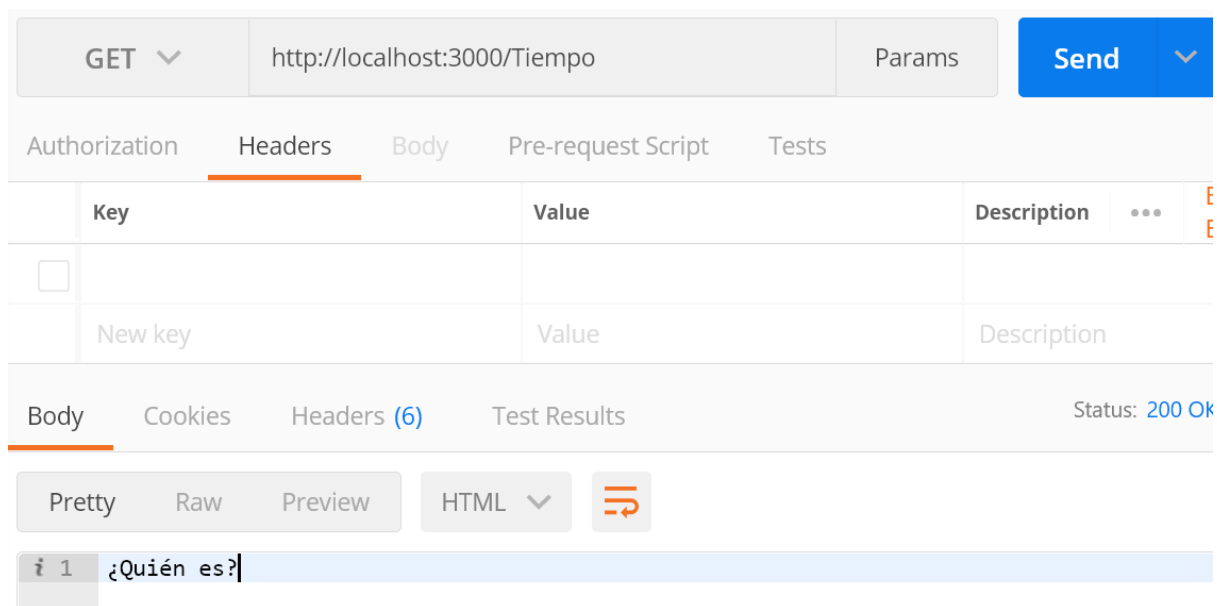


Imagen 14. Respuesta recibida al consultar la ruta **/Tiempo**.
Fuente: Desafío Latam

Como puedes notar, recibimos el mensaje "¿Quién es?", ¿Por qué crees que sucedió esto? Porque no estamos especificando en las cabeceras la propiedad "Authorization".

Ahora, desde el mismo POSTMAN incluye la propiedad Authorization en las cabeceras enviando un token cualquiera y deberás obtener lo que te muestro en la siguiente imagen:

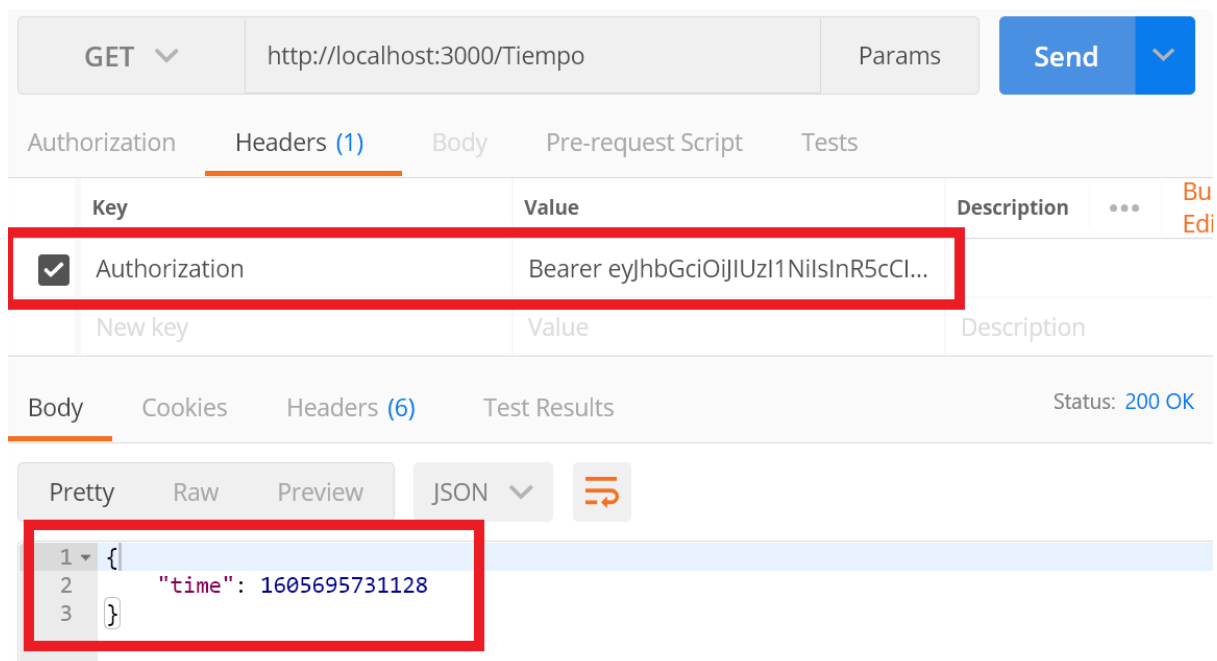


Imagen 15. Respuesta recibida al consultar la ruta **/Tiempo** definiendo la propiedad.
Fuente: Desafío Latam

Ahí está, estamos obteniendo el objeto con el tiempo emitido desde el servidor, lo cual significa que logramos pasar por la validación del middleware. Es importante que sepas que este proceso de ejemplo no es aplicable 100% en el mundo laboral, puesto que para validar un token se necesita ocupar el método “verify” de la tecnología JWT (JSON WEB TOKEN), si por curiosidad quieres adelantarte a este contenido y saber un poco más sobre las validaciones de tokens desde servidores, te dejo el [link](#) del paquete que permite trabajar con JWT en un entorno Node.

Ejercicio propuesto (7)

Desarrollar un middleware para la ruta **GET /colores** que evalúe si el parámetro “color” es igual a “Azul”, de ser correcto devolver un mensaje de éxito, de lo contrario un mensaje de fracaso.

Devolución de sitios web estáticos

Competencia

- Construir un servidor utilizando Express para servir un sitio web estático

Introducción

Ahora que sabes crear servidores con Express, usar su enrutador y construir middlewares, es momento de aprender a servir un sitio web estático, es decir, que a través de una ruta creada en el servidor devolverás un HTML y contenido estático.

En el desarrollo puro con Node hicimos esto sin mayor problemas gracias al módulo File System, sin embargo, Express nos simplifica bastante el desarrollo, al permitir a través de un middleware definir un directorio como público y “liberar” de esta manera todos los archivos dentro. ¿Qué utilidad tiene esto en el desarrollo backend? Una muy importante, es el no tener que crear una ruta por cada archivo estático, sino que podremos acceder a través de la base URL del servidor a todos los documentos alojados en este directorio público.

Aprendiendo esto, podrás empezar a integrar tus sitios web desarrollados en el lado del cliente con tus servidores contruidos con Express. Esta será la base de las próximas sesiones en donde sacaremos provecho del poder de Express mezclado con desarrollo frontend.

Publicando directorio

Cuando desarrollamos servidores con Node en módulos pasados, tuvimos la necesidad de crear una ruta por cada archivo estático que deseábamos importar en nuestro documento HTML, o simplemente disponer un fichero o archivo estático al cliente. Con Express, podemos a través de un middleware especificar un directorio y hacer público o “liberar” su contenido, ¿Cómo? Utilizando el método “static” de la instancia de Express.

La instrucción que debemos ocupar es la siguiente:

```
app.use(express.static("<Nombre o dirección de la carpeta o archivo>"));
```

Si desglosamos esta instrucción tendremos lo siguiente:

- Estamos creando un middleware para incluir un comportamiento en nuestro servidor.
- Se está usando la instancia directa del paquete Express.
- En la instancia de Express se está ocupando un método llamado “static”, el cual permite definir a través de un String, la dirección de un directorio de archivos estáticos, sin embargo, también puede ser usado para devolver un único archivo, por ejemplo un index.html.

Con esta instrucción escrita en la lógica de nuestro servidor, podremos acceder libremente a todos los archivos que tengamos dentro la ruta que definamos como argumento del método “static”.

Sirviendo un sitio web estático

Una de las principales funcionalidades de los servidores web, es servir aplicaciones clientes, cuyo contenido esté alojado en el servidor. De esta manera podemos mezclar ambos mundos (frontend y backend) en un mismo entorno de desarrollo. ¿Qué ventaja consigo con esto? El poder que tenemos a nivel operativo en el backend no lo podemos conseguir desde el frontend y las interacciones con los usuarios que logramos en las aplicaciones clientes no lo podemos programar desde el backend, por lo que la unión de estos 2 mundos nos aumentan inmensamente las posibilidades y amplían nuestro rango de funcionalidades.

Ejercicio guiado: Devolviendo un sitio web

Servir un sitio web estático desde un servidor desarrollado con Express, en donde se solicita imprimir una cabecera y una imagen importada desde un directorio público definido en el servidor.

Consideraciones previas:

1. Crear un documento "index.html" con el siguiente código en tu árbol de archivos a la misma altura que el script donde tienes escrito tu servidor:

```
<h1>¡Soy un sitio web bebé =>!</h1>  

```

2. Crear una carpeta "assets" en tu servidor.
3. Dentro de la carpeta assets agrega alguna imagen que tengas guardada en tu sistema operativo. En mi caso usaré una imagen llamada "Paisaje.jpeg".
4. Express contiene un método llamado "sendFile" en el objeto response de una ruta. Este método permite devolver un archivo especificando su ruta como argumento, sin embargo, esta ruta se debe definir por medio de una concatenación con una variable de entorno llamada "__dirname", la cual contiene la referencia a la dirección dentro del árbol de archivos en donde está ubicado el archivo en el que se ocupa.

Con las consideraciones previas realizadas, sigue los pasos para el desarrollo de este ejercicio:

- **Paso 1:** Crear un servidor con Express que escuche el puerto 3000.
- **Paso 2:** Ocupar un middleware y el método "static" de Express para declarar la carpeta "assets" como directorio público del servidor.
- **Paso 3:** Crear una ruta GET raíz que devuelva el documento index.html.

```
// Paso 1
const express = require("express");
const app = express();

app.listen(3000, () => {
  console.log("El servidor está inicializado en el puerto 3000");
});

// Paso 2
app.use(express.static("assets"));

// Paso 3
app.get("/", (req, res) => {
  res.sendFile(__dirname + '/index.html')
})
```

Ahora, abre tu navegador y consulta la dirección <http://localhost:3000/>, deberás recibir lo que te muestro en la siguiente imagen:

← → ↻ ⓘ localhost:3000

¡Saludos desde la carpeta Assets!



Imagen 16. Sitio web estático recibido al consultar la ruta raíz del servidor.
Fuente: Desafío Latam

¡Excelente! Hemos logrado exitosamente la misión y ahora podemos servir sitios web estáticos que utilicen un directorio alojado en el mismo servidor como fuente de archivos estáticos.

Ejercicio propuesto (8)

Desarrollar un servidor con Express que sirva un sitio web estático y utilice un fichero .CSS alojado en un directorio local.

Resumen

En esta lectura aprendimos qué es Express y su diferencia con Node en la creación de servidores. Vimos además cómo crear rutas y middlewares para la manipulación y procesamiento del objeto request y response. Terminando con un ejercicio en el que servimos un sitio web estático que consulta una imagen alojada en el servidor gracias a la liberación de archivos estáticos que logramos hacer con el método “static” de express, hemos abordado los siguientes contenidos:

- Qué es Express, por qué utilizarlo y cuáles son sus características principales.
- Diferencias entre el desarrollo de servidores con Express o con Node, además de la demostración de cómo llegar al mismo resultado en ambos casos.
- Creación de rutas con Express para la comunicación con aplicaciones clientes que deseen consultar nuestro servidor.
- Definición y propiedades del objeto request y response como parámetros de una ruta creada en Express.
- Definición y uso de los middlewares. Su importancia en la validación o verificación de propiedades representativas de una consulta HTTP.
- Devolviendo un sitio web estático que importa archivos de un directorio público definido por el servidor ocupando el método “static” de la instancia de Express.
-

Solución de los ejercicios propuesto

1. Debate con tus compañeros la siguiente pregunta ¿Podremos con Express hacer todo lo que hemos hecho con Node.js en los módulos anteriores?

Sí, Express está desarrollado con Node, lo cual quiere decir que se están ocupando las mismas herramientas, pero estas son presentadas a través de métodos y sintaxis minimalista.

2. Debate con tus compañeros la siguiente pregunta: Con lo poco que has visto hasta ahora, ¿Te parece que el desarrollo de un servidor con Express tiene un código más limpio y cómodo de escribir que con Node puro?

Sí, por sus métodos y sintaxis minimalista y declarativa se vuelve más cómodo el desarrollo.

3. Desarrollar un servidor con Express que disponibilice una ruta **GET /QuienSoy** que al ser consultada devuelva un string con tu nombre.

```
const express = require('express')
const app = express()

app.listen(3000, () => {
  console.log('El servidor está inicializado en el puerto 3000')
})

app.get("/QuienSoy", (req, res) => {
  res.send("Chuck Norris &#128374;")
})
```

4. Desarrollar un servidor con Express que contenga una ruta genérica y devuelva un parrafo de color rojo diciendo "La dirección que estas consultando no existe"

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

app.listen(3000, () => {
  console.log("El servidor está inicializado en el puerto 3000");
});

app.get("/TocToc", (req, res) => {
```

```
res.send("¿Quién es?");
});

app.get("*", (req, res) => {
  res.send('<p style="color:red">La dirección que estás consultando no existe </p>');
});
```

5. Desarrollar una ruta **GET /usuario/:nombre** que almacene el parámetro nombre en una constante y devuelva un mensaje de éxito en caso de que el nombre empiece con una vocal.

```
app.get("/usuario/:nombre", (req, res) => {
  const nombre = req.params.nombre;
  const test = nombre.match(/^[aeiouAEIOU]/)
  test
    ? res.send("Si, tu nombre empieza con una vocal")
    : res.send("No, tu nombre no empieza con una vocal");
});
```

6. Desarrollar una ruta **GET /musica** que redirija al cliente al sitio web de [Spotify](https://www.spotify.com/cl/).

```
app.get("/musica", (req, res) => {
  res.redirect("https://www.spotify.com/cl/");
});
```

7. Desarrollar un middleware para la ruta **GET /colores** que evalúe si el parámetro "color" es igual a "Azul", de ser correcto devolver un mensaje de éxito, de lo contrario un mensaje de fracaso.

```
app.use("/colores/:color", (req, res, next) => {
  const { color } = req.params;
  color == "Azul" ? next() : res.send("No es azul");
});

app.get("/colores/:color", (req, res, next) => {
  res.send("Si, es azul");
});
```


8. Desarrollar un servidor con Express que sirva un sitio web estático y utilice un fichero con código CSS alojado en un directorio local.

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

app.listen(3000, () => {
  console.log("El servidor está inicializado en el puerto 3000");
});
app.use(express.static("assets"));

app.get("/", (req, res) => {
  res.sendFile(__dirname + '/index.html')
})
```

```
body{
  background: Green;
  color: white;
}
```

```
<link rel="stylesheet" href="estilos.css">
<h1>¡Color esperanza!</h1>
```

9.