# Introducción

Conceptos:

* Frontend: se encarga de la parte visual e interactiva del sitio.
* Backend: el backend se encarga del procesamiento y despliegue de la información, configuración y administración tanto de la base de datos como el servidor en el que se ejecuta la aplicación.

Cosas a aprender:

* Lenguaje de programación
* Motor de base de datos para consultar información
* Se verán los fundamentos del backend

# Como funciona un servidor web

Servidor: se puede referir a la computadora donde se coloca nuestro código o a alas peticiones que viene del navegador, es decir tanto el software como el hardware que se necesitan para enviar una página web

Cliente: es una referencia al programa o dispositivo para acceder una página web.

El cliente y el servidor se comunican usando un protocolo http (Protocolo de Transferencia de Hipertexto) (Protocolo de Transferencia de Hipertexto).

Esta computadora esta especializada para recibir estas peticiones a través de puertos de la red, el protocolo http utiliza el puerto 80 y el 443 para conectar un cliente con el servidor web.

# NodeJS

Es un ambiente de ejecución, además de permitirnos ejecutar el programa un ambiente de desarrollo provee al programa de objetos, API y librerías correspondientes a cada entorno.

Ofrece acceso al sistema lo que quiere decir es que podemos manipular archivos, peticiones por la red, conectar a base de datos y más. Es por eso que notejs se usa para crear aplicaciones web por que provee las herramientas necesarias para que escribamos código de java script que pueda hacer uso de funciones del sistema que son necesarias para aplicaciones web

# Express

Es una librería de desarrollo web que ofrece el código base para desarrollar una aplicación web. Tiene motores de vista, analizador de los datos, manejo de sesiones etc.

# HTTP

Es la comunicación entre cliente y servidor para visitar una página web.

# Verbos Http

Una parte de la petición que se envía hacia el servidor corresponde al método Http o verbo Http como también conocemos a este elemento.

Comúnmente este método puede ser POST o GET, que son los verbos más populares, de hecho, quizás los identifiques si antes has trabajado con formularios HTML.

Estos verbos indican qué acción queremos realizar sobre el servidor y son GET, POST, PUT, PATCH, DELETE, HEAD, CONNECT, OPTIONS y TRACE. Cada uno indica una acción diferente a la que el servidor debe responder.

* POST: envía datos al servidor.
* GET: solicita una representación del recurso especificado. Las solicitudes que usan GET solo deben usarse para recuperar datos (no deben incluir datos).
* PUT: crea un nuevo elemento o reemplaza una representación del elemento de destino con los datos de la petición.
* PATCH: aplica modificaciones parciales a un recurso.
* DELATE: elimina el recurso especificado.
* HEAD: solicita los encabezados que se devolverían si la dirección URL de la solicitud se solicitara con el método HTTP GET. Por ejemplo, si una URL puede producir una descarga grande, una solicitud podría leer su encabezado Content-Length para comprobar el tamaño del archivo sin descargar realmente el archivo.
* CONNECT: inicia la comunicación en dos caminos con la fuente del recurso solicitado. Puede ser usado para abrir una comunicación tunel.
* OPTIONS: solicita opciones de comunicación permitidas para una dirección URL o servidor determinados. Un cliente puede especificar una dirección URL con este método o un asterisco () para hacer referencia a todo el servidor.
* TRACE: efectúa una prueba de bucle de mensaje por el camino al recurso objetivo proporcionando un útil mecanismo de debugging.

# Direcciones web

Una dirección web para hacer una solicitud con Http puede verse de alguna de las siguientes maneras:

* localhost:8080
* google.com
* web.facebook.co
* codigofacilito.com/cursos

Todas son direcciones válidas y cada una tiene elementos distintos. Lo que aparece al principio, justo antes de los dos puntos en el primer ejemplo, y en el caso de google.com, se trata del nombre de dominio.

El nombre de dominio nos permite identificar la computadora en la que se encuentra el recurso que estamos solicitando, nota como en lugar de página estamos usando el concepto de recurso, porque, así como podemos solicitar una página, también puede ser otro tipo de archivo como una imagen.

Luego del nombre de dominio viene el puerto, en el primer ejemplo el puerto es el 3000, es el que aparece luego de los dos puntos. En el resto de los ejemplos se asume el puerto por defecto, porque no hay otro especificado. El puerto por defecto para el protocolo Http es el puerto 80, y es el que precisamente se usa cuando no se especifica otro, como en los ejemplos mencionados.

Además del dominio y el puerto, una dirección web contiene la ruta o el path en donde se encuentra el recurso, dentro del servidor. Piensa en que si el dominio fuera una colonia, el path sería la ruta para llegar a una casa en específico, ya que, si bien el dominio representa al servidor, el path representa el recurso en específico que queremos.

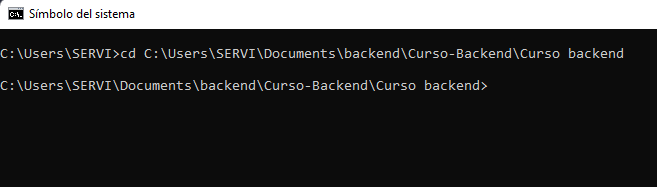
El path por defecto es la /, misma que puede omitirse por lo que asumimos que si no hay path, estamos tratando de acceder al inicio, la /. En nuestros ejemplos de direcciones, el path es siempre / con excepción del último donde se indica que buscamos el recurso con la ruta /articulos.

# Creando nuestro primer servidor

Primero crearemos una carpeta, única y exclusiva para el proyecto, esto nos permitirá mantener organizados los archivos del proyecto, luego otra carpeta para el proyecto.

Lugo tenemos que posicionar nuestra terminal con la carpeta para que cuando se ejecuten los comandos, se ejecuten sobre la carpeta de nuestro proyecto, para ello usaremos el sig. comando:

cd “ruta a donde nos queremos mover”

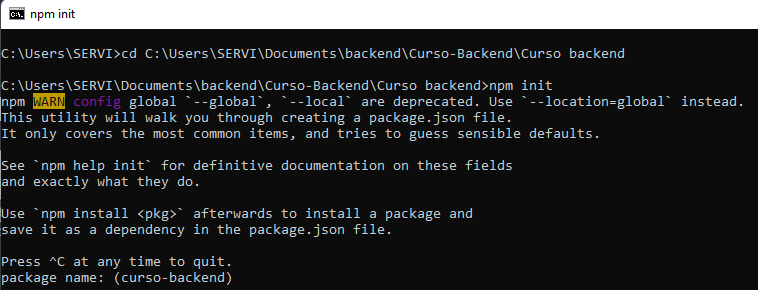


Listo ahora nos encontramos en la carpeta que acabamos de generar.

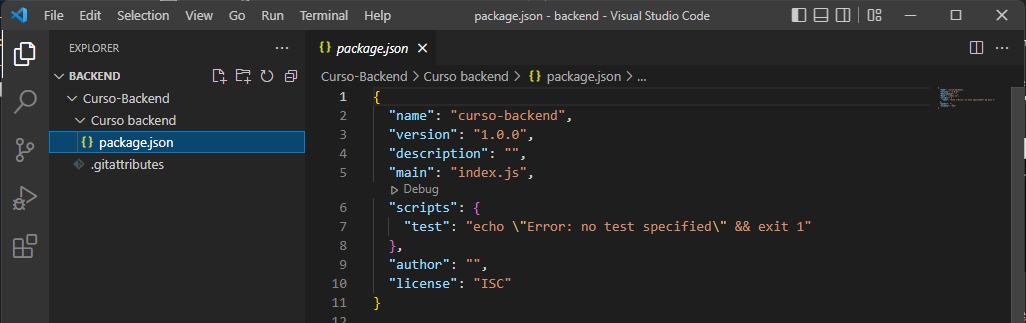
Para iniciar nuestro proyecto usaremos npm, el cual usaremos para instalar express.

Primero ejecutmos el comando:

npm init (pedirá datos pero por el momento no son relevantes, le daremos enter a tod)



Esto hará que nos generara una carpeta llama “package.json”



Crearemos un nuevo archivo llamado “server.js” (todos los archivos de notejs llevan la extension .js)

Notejs ofrece una librería con la que podemos hacer un programa que acepte mensajes via http

const http = require('http');

Para crear el servidor usaremos el sig. código:

let server = http.createServer();

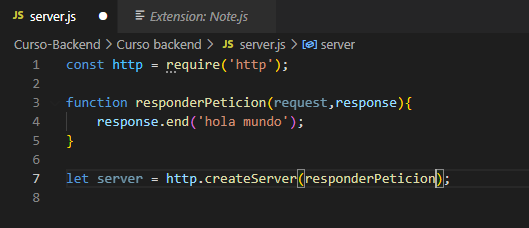
A este método le pasaremos una función que se ejecutara cada vez que recibsmos una petición del cliente.

function responderPeticion(request,response){

response.end('hola mundo');

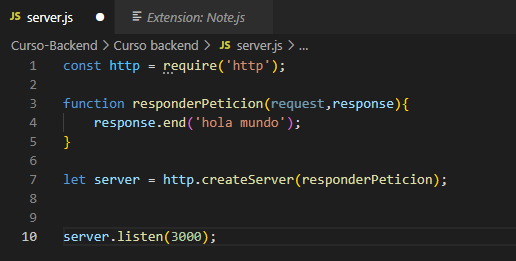
}

En la parte “let server = http.createServer();” en los paréntesis le agregaremos la “funcion responderPeticion” lo que hace es mandar llamar a la peticion



Ahora colocamos el puerto por donde escuchara.

server.listen(3000);



Despues hay que guardarlo.

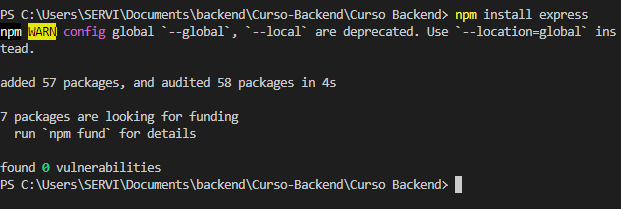
Ahora vamos a ejecutar nuestro código para ello, regresaremos a la terminal, escribiremos el sig. comando:

node server.js

# Express

Lo primero que hay que hacer es instalar express, para ello insertamos el sig, comando:

npm install express



Después creamos un nuevo archivo llamado “express.js” donde le insertaremos la sig. librería:

const express = require('express');

Esto retorna una función que al ejecutarse nos entrega un objeto con el cual podmos configurar nuestra aplicación, después le escribimos la sig. función:

const app = express();

Este objeto app puede especificar rutas que puede responder y que respuesta enviara, ejemplo:

app.get('/', function(req,res){}

);

En la ruta vamos a pasar una función que va a responder a las peticiones que vengan precisamente a esta ruta, tenemos dos objetos “req” y ”res”, “res” lo usamos para mandar un mensaje como respuesta de la petición. Despues de esto escribimos el sig. código:

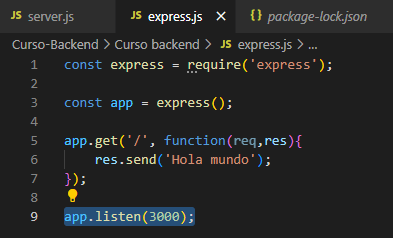
res.send('Hola mundo');

Antes de ejecutarlo necesitamos que la app se escuche en algún puerto por lo cual le colocamos el sig. código:

app.listen(3000);

Después lo guardamos y luego lo ejecutamos en la terminal de la sig. manera:

node express.js

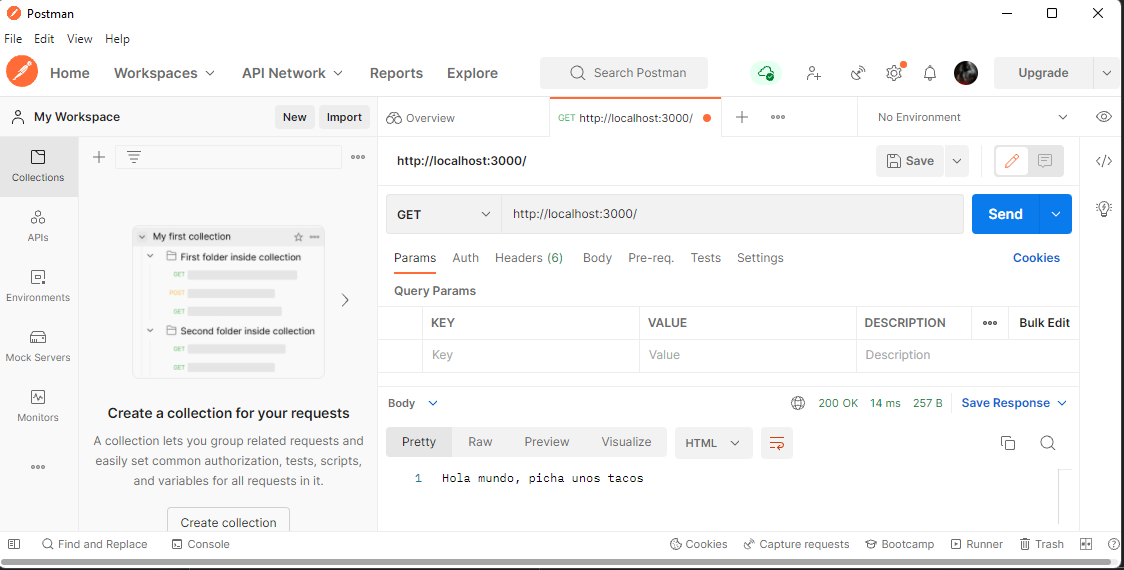


Luego vamos al navegador y ejecutamos:

localhost:3000

# Postman

Nos permite crear peticiones http usando una interfaz gráfica, nos permite enviar información en distintos formatos, crear peticiones con cada método http disponible, modificar los encabezados de una petición y visualizar las respuestas que se obtuvo desde el servidor.



# Recibir datos de la petición

Una petición http incluye en el mensaje una serie de datos útiles para la misma. Para que podamos utilizar la información que viene del mensaje de la petición express genera un objeto con toda esa información, dicho objeto se recibe como primer argumento en la función que pasamos como manejador de una ruta, modificaremos nuestro código para ahora recibir un dato que ahora identificaremos como name y que el servidor use para saludarte cuando reciba una peticion por esta ruta.

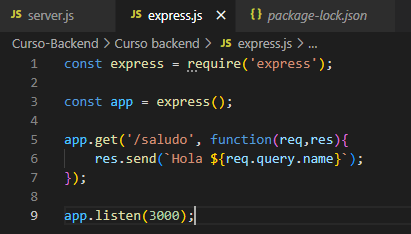
En el código cambiaremos:

res.send('Hola mundo ');

Por:

res.send(`Hola ${req.query.name}`);

La diferencia con el otro esqe este tipo de acento invertido, es un string cuya diferencia es que soporta interpolación con $ {}, donde dentro ira una expresión java script que se insertara dentro de la cadena.



Dentro del objeto “req” y que contiene toda la información de la petición, tenemos un objeto query, mismo que contiene una propiedad por cada dato que recibe en al petición, como desarrollador backend, nosotros decidimos como se identificara los datos que nos envían, por ejemplo en la propiedad name estamos asumiendo que el nombre del saludo será name, eso signfica que quien crea el mensaje de la petición tendrá que usar ese mismo identificador para que la respuesta sea la apropiada, luego reiniciamos el servidor para que los cambios tengan efecto.

Luego vamos a postman, le colocamos la dirección que en este caso es:

<http://localhost:3000/saludo> (el /saludo es un nombre que le dimos a la ruta nomas)

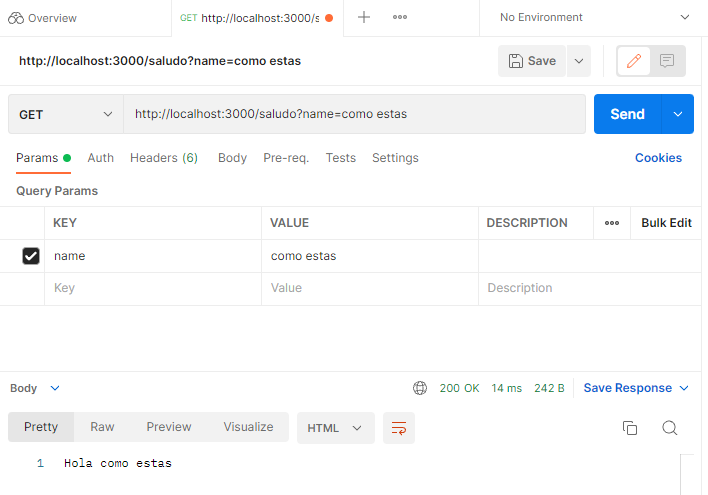
luego le colocamos el dato name que queremos enviar,una de las formas es colocarlos como parte de la dirección, los cuales son llamados como Query Params o parámetros de consulta donde colocamos el nombre del quary y su valor, o como ya lo habíamos dicho como parte de la dirección, usando la sig. estructura

<http://localhost:3000/saludo?(> Query-Params)

Asi que quedaia de la sig. manera:

http://localhost:3000/saludo?name=como estas

El resultado seria el sig.:



# Datos con POST

GET y POST son muy importantes, la primera diferencia está entre los parámetros que recibes de la petición, mientras que los datos que se envían al servidor con GET son parámetros de la consulta y son parte de la dirección, los parámetros POST son parte del cuerpo de la petición y no son visibles en la dirección.

Primero crearemos un nuevo archivo llamado post.js. tomamos la base del código anterior de express.js y le agregamos el sig. código:

app.post('/', function(req,res){

});

Luego usaremos una herramienta llamada body parse

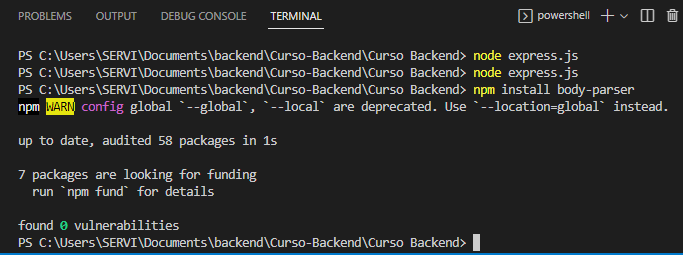
¿Qué es la librería body-parser de npm?

Usualmente el cuerpo de una peticion (payload), contiene información desde una petición tipo POST cuando un cliente desea crear una nueva entidad/registro o actualizar uno existente mediante PUT.

<https://apuntes.de/nodejs-desarrollo-web/body-parser/#gsc.tab=0>

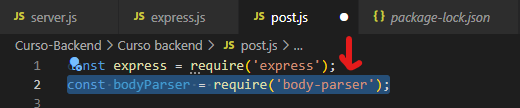
Lo instalaremos con npm de la sig. manera:

npm install body-parser



En el archivo, agregamos la librería de la sig. forma:

const bodyParser = require('body-parser');



El nombre debe ser el mismo con el que se instaló.

Después para usarlo agregamos la sig. línea de código:

app.use(bodyParser.urlencoded({extended: true}));

En esta línea le pedimos a express que como parte del proceso de dar respuesta a una petición use el analizador body parser para leer el cuerpo de datos de una petición, particularmente le pedimos que use este analizador para leer el cuerpo de una petición que venga con el formato urlencoded.

La opción extended sirve para que se pueda enviar parámetros anidados y así poder enviar objetos con propiedades en la petición.

Ahora nuestro servidor hace la lectura de los datos que viene en el cuerpo de la petición usaremos esta información.

En el objeto “res” la información viene en el objeto body.

Como ejemplo usaremos el res de app.get, siendo ahora app.post y lo modifcaremos de la sig. manera:

app.post('/', function(req,res){

res.send(`Hola ${req.body.name}`);

});

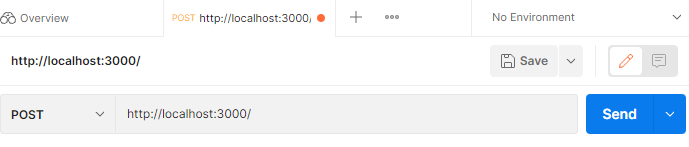


Luego guardamos e iniciamos el servidor ahora con post.js.

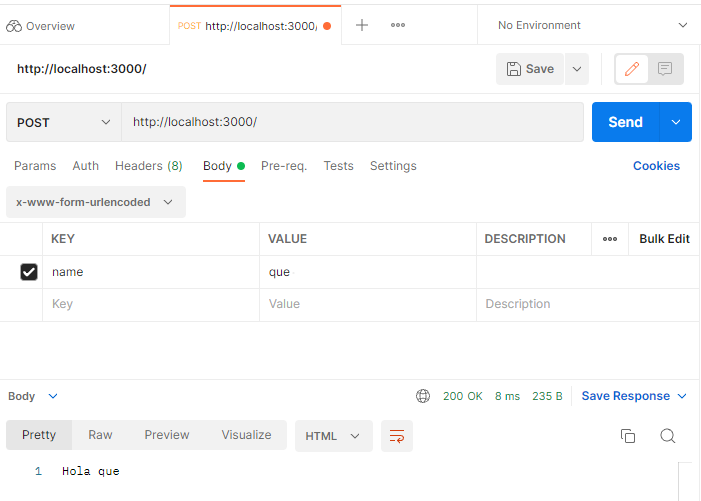
Luego vamos a postman, colocamos la dirección que en este caso es:

<http://localhost:3000/>

Luego cambiamos de GET a POST.



Luego seleccionamos la parte de “body” y seleccionamos la opcion que tiene “urlencoded”

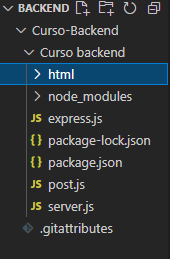


Como ya sabemos usamos la clave name y luego el valor que asignamos, lo enviamos y recibiremos una respuesta.

# Enviar HTML

Como desarrollador backend, vamos a pensar en nuestro servidor como un proveedor de recursos ya sean imágenes, pdf, archivos de script de código, css y más. El principal recurso que se envia es un archivo o documento html.

Lo primero que haremos será crear un nuevo servidor, además de eso crearemos una nueva carpeta la cual llamaremos “html” y en ella iremos colocando el código.



En “html” crearemos un nuevo servidor express y se llamara “server.js”. Primero importaremos la librería express con el sig. código:

const express = require('express');

Después vamos a ejecutar la función para obtener el objeto app que es atraves del cual definimos las rutas:

const app = express();

Por último, ponemos a escuchar nuestro servidor:

app.listen(3000);

Esta sería la estructura básica para levantar un servidor de peticiones http con express.

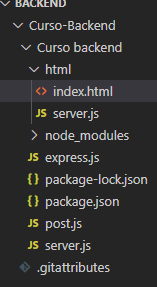
Luego definiremos una ruta inicial para que responda con un documento html, usando el sig. código:

app.get('/', function(req,res){

});

Como siempre responde con dos argumentos uno con la información de la petición y otro con lo necesario para la respuesta.

Antes crearemos nuestro documento html en un archivo llamado “index.html” (Le daremos una configuración básica).



Para enviar un documento html podemos aprovechar el mentodo “sendfile” identificado por “res” de la sig, manera:

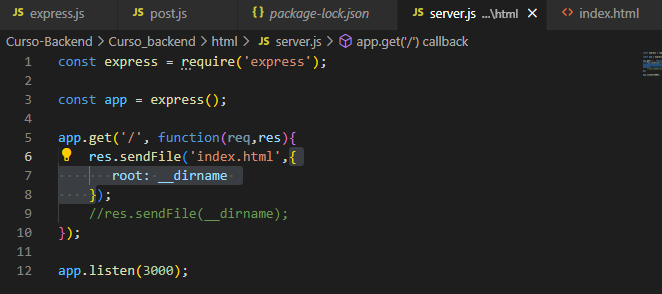
res.sendFile(‘archivo html’);

Para que “sendFile” pueda encontrar el archivo tenemos dos opciones, por un lado seria colocar una ruta absoluta, es decir una ruta donde está la carpeta del proyecto o podemos especificar un “json” de opciones la ruta absoluta hacia la ruta del proyecto.

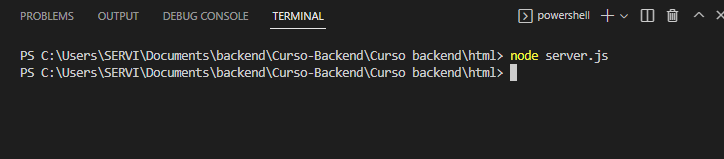
Si llegáramos a mover la carpeta y no quisiéramos modificar la ruta podemos usar la variable llamada “\_\_dirname”.

Le haremos saber al servidor que debe responder con un mensaje que contenga la ruta absoluta del proyecto.

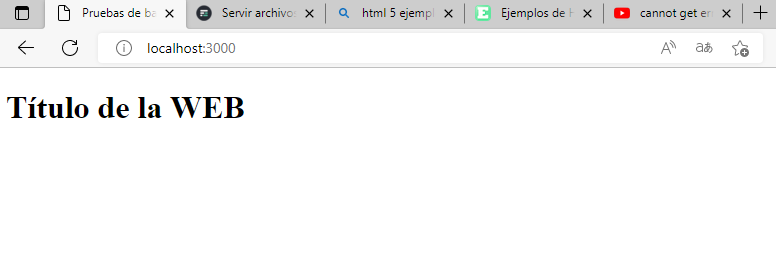
{root: \_\_dirname}



Luego vamos a iniciar nuestro servidor situados en la carpeta “html”.



Resultado:

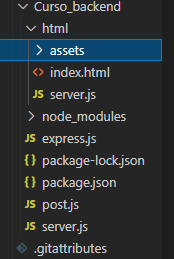


# Servir archivos estáticos

Ahora que podemos enviar documentos html, podemos enriquecer nuestra pagina con otro tipo de recursos ya sean imágenes entre otros. Cuando nuestro servidor envía este tipo de archivos decimos que es un servidor de archivos estáticos, se llama así debido a que este tipo de archivos no se modifican cuando son solicitados por algún cliente.

Los frameworks web incluyen un módulo para archivos estáticos que permite especificar una carpeta en la que todos los archivos incluidos deberán poder ser servidos por el servidor web, es importante señalar que se deben separa los archivos estáticos y los que conforman el código del proyecto. Los archivos colocados aquí pueden descargarse, es por eso que el código del proyecto no debe estar aquí.

En “html” crearemos una carpeta llamada “assets”.



En nuestro código debemos configurar esta carpeta. La librería express que importamos contiene un método “static” que define un servidor de archivos estáticos para una carpeta, el código sería el sig.:

express.static('assets')

Usaremos el método “use” para integrar nuestro servidor de archivos estáticos a nuestro servidor web.

app.use(express.static('assets'));

El metodo “use” lo que hace es insertar un nuevo middleware en el stack de middleware.

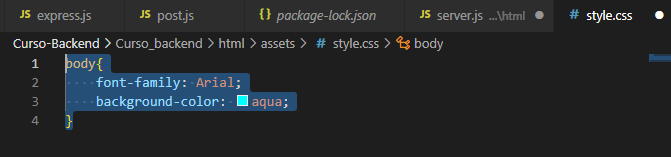
Luego colocaremos en la carpeta “assets” un nuevo archivo “style.css” donde le colocaremos lo sig.:

body{

font-family: Arial;

background-color: aqua;

}



Luego en el “use” le agregaremos “’ ’/assets’ ”, esto es el middleware de archivos estáticos, esto quiere decir que para usar los assets estarán detrás de la ruta /assets/style.css por decir un ejemplo.



En el “index.html” colocaremos el “style.css” de la sig. manera:

<link rel="stylesheet" href="/assets/style.css">



Luego reiniciamos el servidor

Como dato, cada vez que queramos realizar modificaciones sobre los archivos estáticos no es necesario reiniciar el servidor

# Motores de vistas

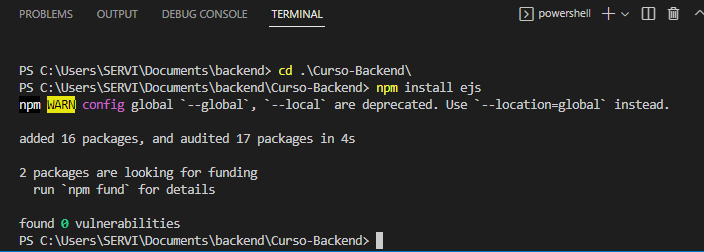
La información de un documento html es dinámica, es decir, que es probable que cambie cada que hacemos una petición web por dicho recurso, normalmente para construir un documento html se consideran datos almacenados en una base de datos, si el usuario inicio sesión o no, entre muchas cosas más.

En el backend se usa lo que conoce como un motor de vistas para generar estos recursos de manera dinámica, existe una gran variedad de motores de vista disponible para nuestro entorno con nodejs.

El trabajo de estos motores de vista es, por un lado, permitir enviar datos de nuestro código del servidor a los documentos html, a quienes llamamos vistas, por otro lado, agregar sentencias y operaciones a la vista para que podamos usar ciclos, condiciones y más dentro de los archivos de las vistas mismas.

El motor de vistas en una dependencia adicional, por lo cual habrá que instalarla desde la terminal “Curso\_Backend” usando el sig. comando:

npm install ejs



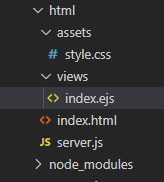
Luego creamos una nueva carpeta en “html” llamada “views”. Para indicar el uso de un motor de vistas tenemos que usar el método “set” del objeto “app” que va de la sig. manera:

app.set();

El primer argumento a enviar es “view engine” para que express sepa que es un motor de vista el que vamos a montar, el segundo es el motor de vista que vamos a usar.

app.set('view engine', 'ejs');

Por defecto los motores de vista, esperan que los archivos de vista estén en una carpeta llamada “views” que es precisamente la que acabamos de crear. Dentro de esa carpeta crearemos un archivo llamado “index.ejs” (la extensión dependerá del motor que se esté utilizando).



Luego copiaremos el código de nuestro “index.html” en el “index.ejs” como ejemplo.



Después de haber configurado nuestro motor de vista, vamos a indicarle al servidor web que debe de responder con esta vista a la petición “get” que ya teníamos, por lo cual borraron el “res.sendFile” y en su lugar colocaremos lo sig.:

res.render.();

Que es el método para mostrar una vista y como argumento colocaremos el nombre de la vista a mostrarse que es este caso será “index”.



Después de eso en la terminal nos situamos en la carteta “html” y reiniciamos el servidor con un “node server.js” en la terminal.

# Qué es el caché

Este concepto hace referencia a un programa que almacena copia de las respuestas que envía un servidor, a fin de que no sea necesario volver a consultar al servidor por estos archivos de ahí nace el termino caching y significa almacenar en el cache. Este varia, puede ser desde guardar una consulta a una base de datos hasta guardar un archivo que sabemos que no cambiara en un tiempo definido. Entre los beneficios que tiene esta, reduce la carga al servidor, incrementa el rendimiento de la aplicación web, reduce el tiempo de carga a la página. El proxy es un intermediario entre el la solicitud y la respuesta, el proxy puede tomar una petición, buscar en el cache para ver si hay una copia previa y entregar dicha copia para que la petición no llegue hasta el servidor. Un contratiempo seria la actualización de copias cuando un archivo es modificado o actualizado.

# Tipos de cache

De acuerdo con la Mozilla Developer Network, en general podemos identificar dos tipos de cache, los locales y los compartidos.

Prácticamente todos los usuarios tienen un caché local que viene incluido por defecto con su navegador, este puede servir para que las páginas que visitamos carguen más rápido o bien para que podemos ver un sitio incluso cuando no tengamos internet.

Este es un buen ejemplo de un caché local, lo que lo diferencía de un caché compartido es que este caché sólo sirve para un usuario, aquél que accede al sitio web vía dicha computadora.

Por otro lado, un caché compartido puede estar compartido entre múltiples usuarios, existen servicios que ofrecen la posibilidad de funcionar como intermediarios entre los usuarios y el servidor, un buen ejemplo de ello es Cloudflare.

Los cachés compartidos son bastante interesantes porque pueden ofrecer beneficios de performance y velocidad incluso a usuarios que visitan por primera vez nuestro sitio web. Algunos de estos servicios replican las copias que se han generado en múltiples ubicaciones del mundo, acercándolas más a los usuarios que eventualmente requerirán estas copias.

# Caché de archivos estáticos

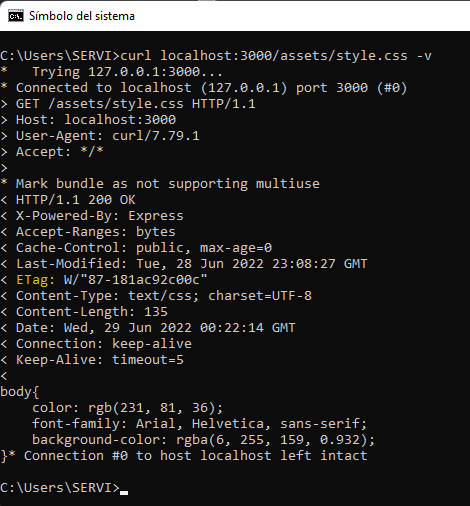
Por defecto los archivos que se envían usando nuestro servidor de archivos estáticos se almacenan en el navegador, como desarrolladores del proyecto podemos desactivar este comportamiento o modificar la forma en como el ordenador sabe que debe actualizar las copias que tiene.

Para el cache de archivos estáticos usaremos dos estrategias que el servidor tiene disponibles “etag” y el “maxage”.

Etag es la abreviatura de entity tag, para ver cómo funciona ejecutamos nuestro servidor y en el cmd ejecutaremos el sig. comando:

curl localhost:3000/assets/style.css -v

Gracias a curl que recordemos, podemos hacer peticiones web y visualizar la respuesta de información adicional, en este caso buscaremos la informacion del “style.css”



Una vez recibida la petición, veremos el contenido del css y los encabezados correspondientes, entre ellos el “ETag”, el cual es una representación del documento, cada servidor la obtiene de manera distinta, lo importante es saber para que se está utilizando, el navegador almacena este dato junto con el archivo estático del cache, la siguiente vez que el navegador solicite este archivo lo enviara junto con la petición

ETag: W/"87-181ac92c00c"

Con esto se puede saber si el archivo se actualizo o no, si el archivo no se actualizo en lugar de enviar el documento al servidor responderá con un status http 304 que significa que el archivo no ha sido modificado. Así sabe el servidor que su copia sigue vigente y la utiliza como respuesta para la petición del usuario y reducirá el consumo de la red.

Para desactivarlo tenemos la opción “etag false” al declarar el servidor de archivos estitacos, agregamos el argumento después de la ubicación de la carpeta:

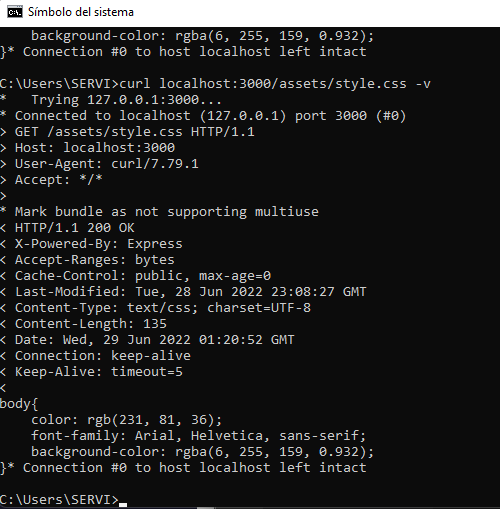
app.use('/assets',express.static('assets', {

etag: false,

}));



Como pueden observar después de cargar el servidor ya no aparece “etag”.



Con la opción “maxage” es una alternativa, no se trata de un encabezado como si lo hace “etag”, sino que es un dato que forma parte de otro encabezado llamado “catch control”, con esto le indicamos al navegador cuanto tiempo es valida la copia y si dicho tiempo no expira el navegador no solicitara una nueva versión de esa copia. Eso si debemos de estar seguros del tiempo de respuesta que asignemos, ya que de lo contrario no surgirán cambios.  
Para agregarlo lo haremos en la misma ubicación del etag, colocando lo sig.:

app.use('/assets',express.static('assets', {

etag: false,

maxAge: '5h'

}));

Puede ser un valor en string o en int representado en milisegundos.