# 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 las 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 está 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.

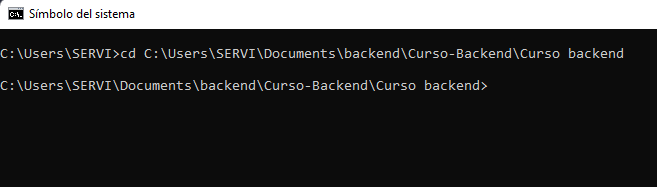
El path por defecto es la /, misma que puede omitirse por lo que asumimos que si no hay path, estamos tratando de acceder al inicio, la /. En nuestros ejemplos de direcciones, el path es siempre / con excepción del último donde se indica que buscamos el recurso con la ruta /articulos.

## 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”

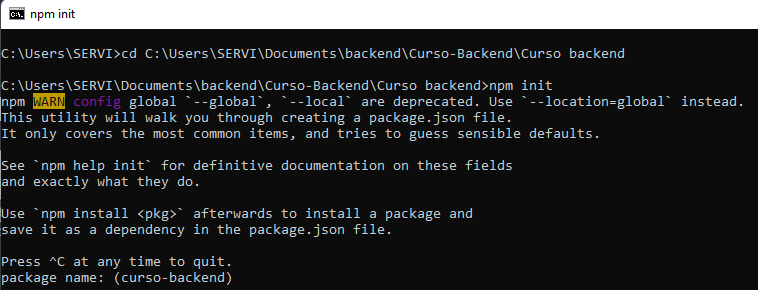


Listo ahora nos encontramos en la carpeta que acabamos de generar.

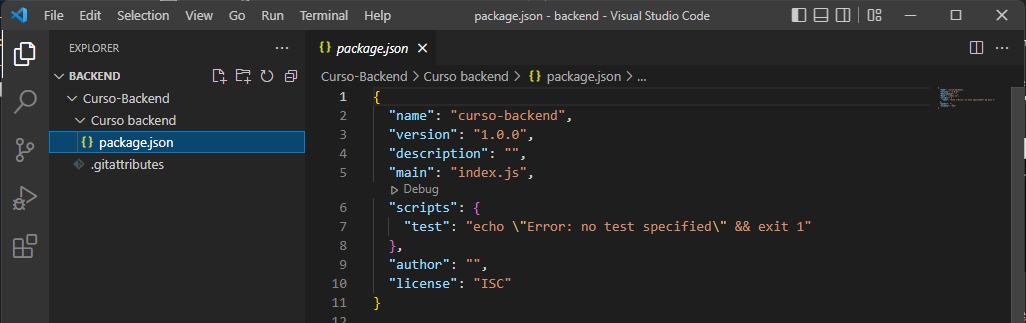
Para iniciar nuestro proyecto usaremos npm, el cual usaremos para instalar express.

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 extensión .js)

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

const http = require('http');

Para crear el servidor usaremos el sig. código:

let server = http.createServer();

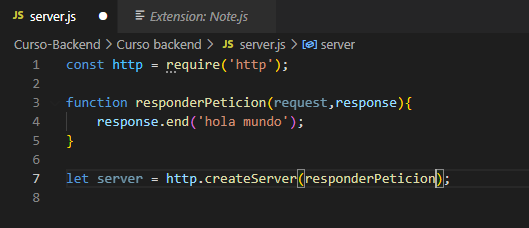
A este método le pasaremos una función que se ejecutara cada vez que recibamos una petición del cliente.

function responderPeticion(request,response){

response.end('hola mundo');

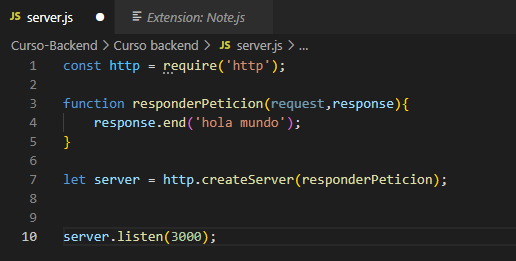
}

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);



Después hay que guardarlo.

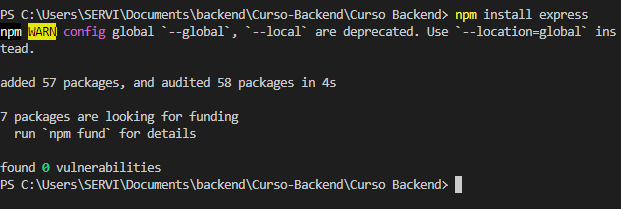
Ahora vamos a ejecutar nuestro código para ello, regresaremos a la terminal, escribiremos el sig. comando:

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 podemos 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:

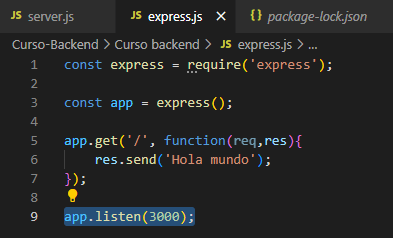
res.send('Hola mundo');

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

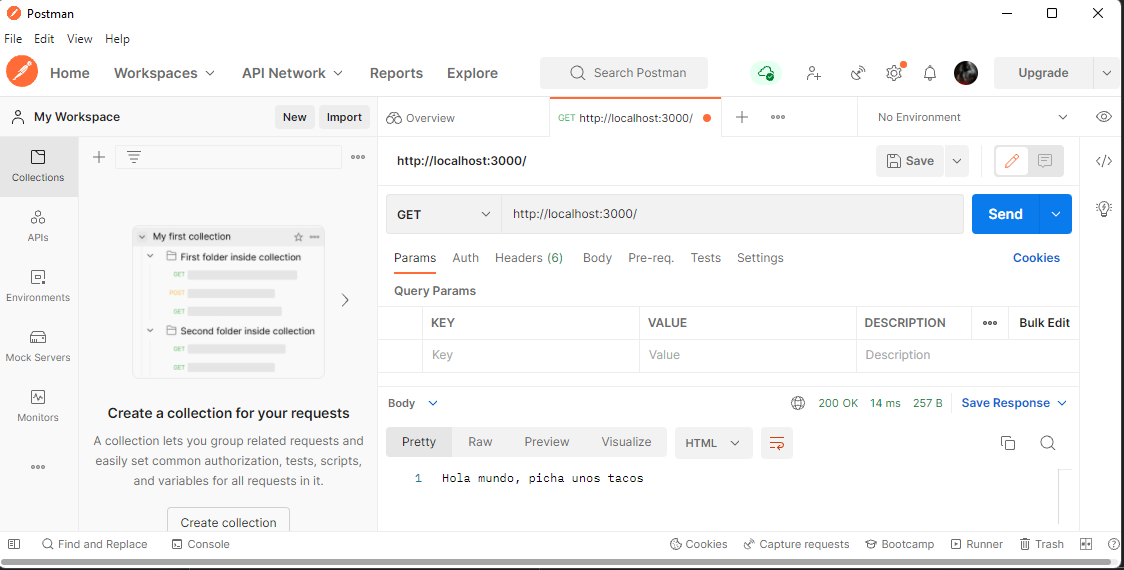


Luego vamos al navegador y ejecutamos:

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.

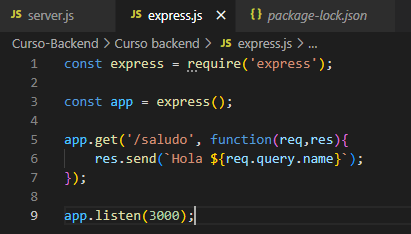
En el código cambiaremos:

res.send('Hola mundo ');

Por:

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

La diferencia con el otro es que 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)

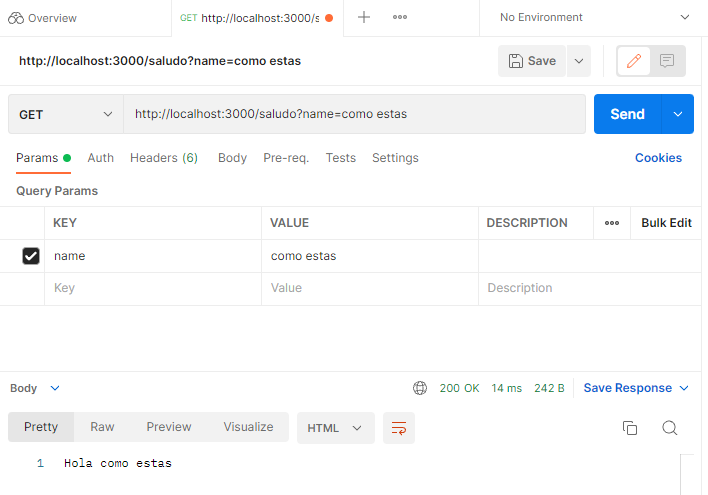
luego le colocamos el dato name que queremos enviar,una de las formas es colocarlos como parte de la dirección, los cuales son llamados como Query Params o parámetros de consulta donde colocamos el nombre del quary y su valor, o como ya lo habíamos dicho como parte de la dirección, usando la sig. estructura

<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

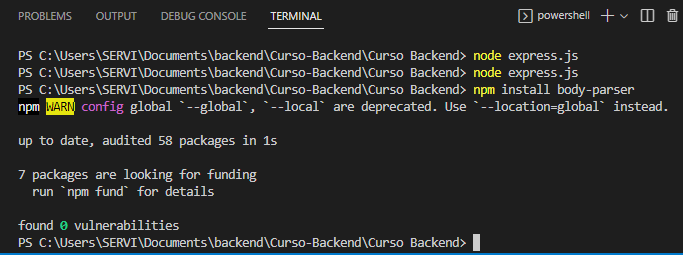
¿Qué es la librería body-parser de npm?

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>

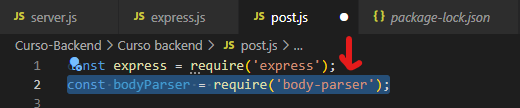
Lo instalaremos con npm de la sig. manera:

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}`);

});

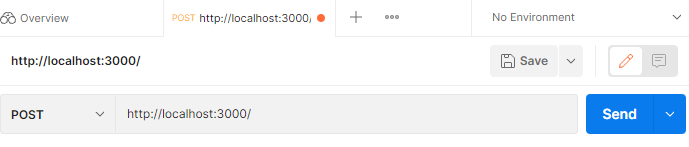


Luego guardamos e iniciamos el servidor ahora con post.js.

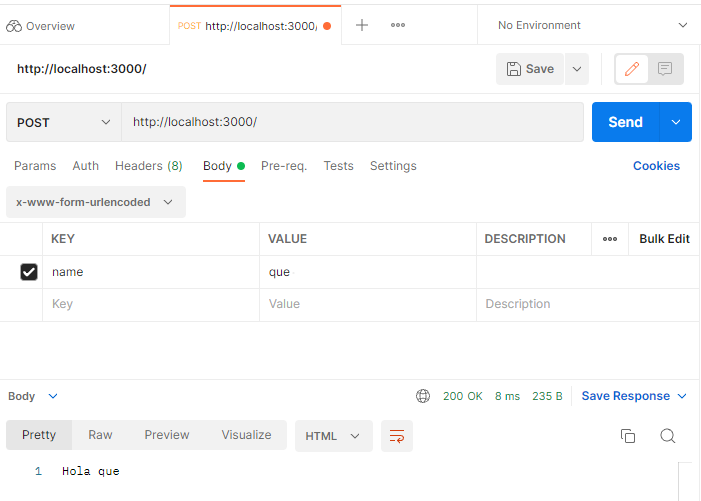
Luego vamos a postman, colocamos la dirección que en este caso es:

<http://localhost:3000/>

Luego cambiamos de GET a POST.



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

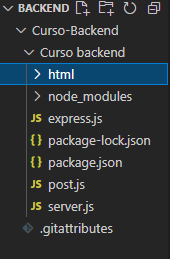


Como ya sabemos usamos la clave name y luego el valor que asignamos, lo enviamos y recibiremos una respuesta.

## 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 a través 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.

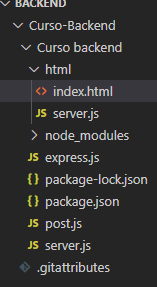
Luego definiremos una ruta inicial para que responda con un documento html, usando el sig. código:

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:

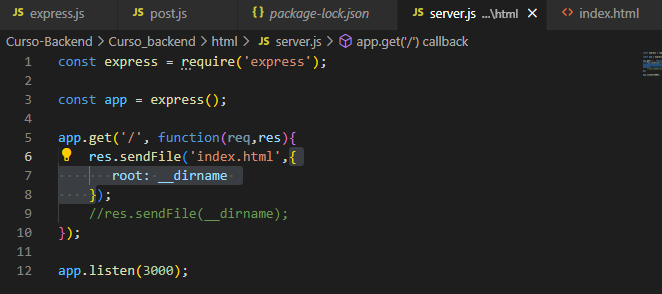
res.sendFile(‘archivo html’);

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.

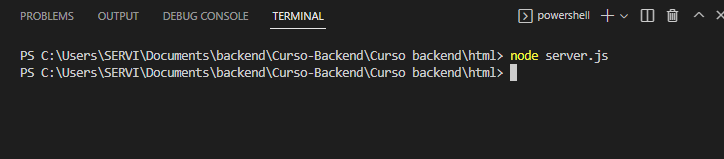
Si llegáramos a mover la carpeta y no quisiéramos modificar la ruta podemos usar la variable llamada “\_\_dirname”.

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

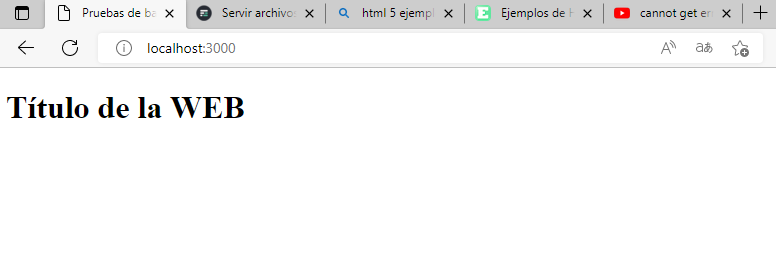
{root: \_\_dirname}



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



Resultado:

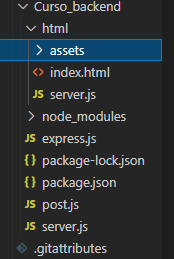


## Servir archivos estáticos

Ahora que podemos enviar documentos html, podemos enriquecer nuestra página 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.

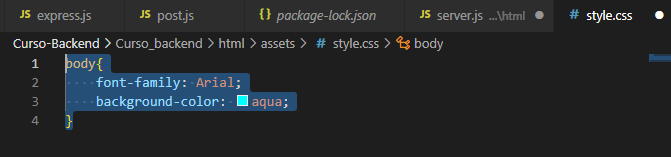
Luego colocaremos en la carpeta “assets” un nuevo archivo “style.css” donde le colocaremos lo sig.:

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

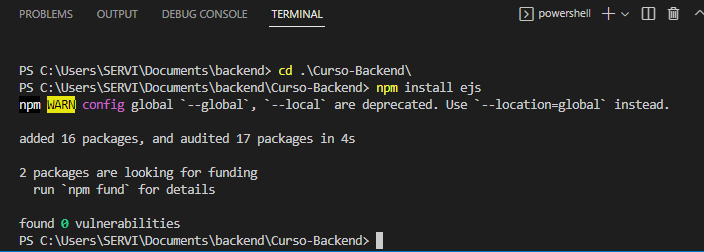
La información de un documento html es dinámica, es decir, que es probable que cambie cada que hacemos una petición web por dicho recurso, normalmente para construir un documento html se consideran datos almacenados en una base de datos, si el usuario inicio sesión o no, entre muchas cosas más.

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



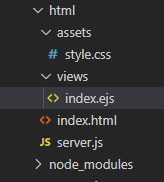
Luego creamos una nueva carpeta en “html” llamada “views”. Para indicar el uso de un motor de vistas tenemos que usar el método “set” del objeto “app” que va de la sig. manera:

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

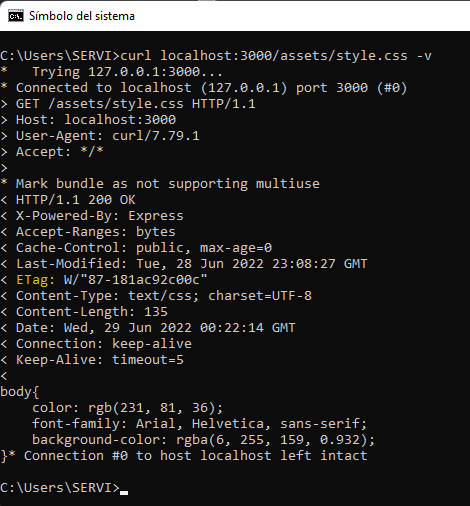
Por defecto los archivos que se envían usando nuestro servidor de archivos estáticos se almacenan en el navegador, como desarrolladores del proyecto podemos desactivar este comportamiento o modificar la forma en como el ordenador sabe que debe actualizar las copias que tiene.

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 información 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:

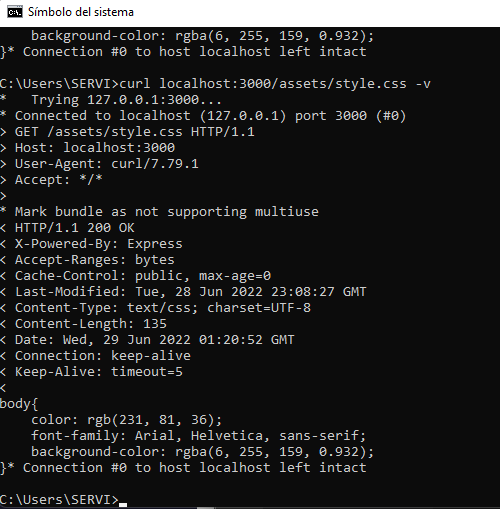
app.use('/assets',express.static('assets', {

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.

## Qué son las cookies y las sesiones.

Una de las características del protocolo Http es que es stateless, es decir, no conserva un estado para ninguna petición que recibe.

Esto significa que el servidor no tiene memoria, es una persona que saludas, te presentas, le dices tu nombre y lo olvidará para la siguiente vez que lo saludes. El servidor no recuerda nada entre peticiones, por lo que no sabe si esa petición ya vino, qué usuario es, cuáles son sus datos, etc.

Por otro lado, nosotros usamos un internet que siempre sabe quiénes somos, sabe qué publicaciones debe mostrarnos en facebook, qué cuenta de correo estamos usando y mucho más. La pregunta es, si el protocolo http es stateless, cómo saben todas las páginas que yo estoy usando su sitio, la respuesta es con cookies y sesiones.

Las cookies y sesiones son mecanismos a través de los cuales podemos identificar una petición, con estas estrategias podemos almacenar información de nuestros usuarios que pueda ayudarnos como los productos que ha agregado a un carrito de compras, sus preferencias, si inició sesión o no, y mucho más.

Las cookies son datos que se almacenan en tu navegador, y son enviados al servidor en cada petición que haces del cliente al servidor mismo. Esto significaría, por ejemplo, que cada que llegas con el servidor sin memoria te presentes y le digas toda la información que posiblemente requería de ti.

Las cookies tienen varias limitaciones, por un lado la cantidad de información que puedes almacenar en cookies está limitada por el navegador, por otro lado, el usuario es libre de alterar las cookies como él prefiera, alterando así el funcionamiento de tu aplicación. Por último, si algo sale mal con las cookies, no está en tu control arreglarlo, tendrías que pedirle a tus usuarios que ellos mismos arreglen el problema, a final de cuentas, los datos están en su computadora.

Las sesiones por otro lado, guardan la información en el servidor y no en el cliente, y lo que se envía entre cada petición es un identificador de sesión para cada usuario, a través del cuál puedes obtener los datos que guardaste en el servidor. Esto significaría, por ejemplo, que cuando llegues con el servidor sin memoria le entregues una identificación tuya y él busque entre sus datos toda la información que necesitas saber, en lugar de que tú tengas que decirle todo.

La ventaja de las sesiones es que se almacenan en tu servidor, por lo que puedes guardar mucha más información que en las cookies, además, puedes usar uno de múltiples almacenes para sesiones, desde guardarlas en archivos, usar la memoria RAM, hasta guardar las sesiones en una base de datos especial para este tipo de información.

Además, las sesiones no pueden modificarse desde el cliente, ya que aunque el identificador de una sesión es guardado en una cookie, modificarlo significa que la información del usuario y el usuario ya no están conectados, y esto por sí mismo no presenta un riesgo de seguridad. Por otro lado, almacenar información sensible como el ID del usuario en una cookie puede prestarse a que se modifique y entonces sí exponga a tu servidor a una brecha de seguridad.

Por otro lado, es común que las aplicaciones web encripten las cookies para que estas no puedan ser leídas o modificadas en texto plano, para eso también hay distintas implementaciones con algoritmos de encriptación.

En general, la regla es, no almacenes información sensible o de importancia para tu aplicación en cookies, usa sesiones. Para todo lo demás, usa cookies.

## Cookies

Cada framework tiene estrategias distintas para procesar las cookies e implementaciones de código también diferentes. Las cookies tienen limitantes como que los navegadores tienen una limitante de cookies de 4kb de información para las cookies de cada dominio, no se comparten entre dominios.

Para la practica crearemos una nueva carpeta llamada “cookies-sessiones” y dentro creamos un archivo llamado “server.js”, luego moveremos nuestro manejo de módulos a esa carpeta con los sigs. comandos en la terminal:

npm init

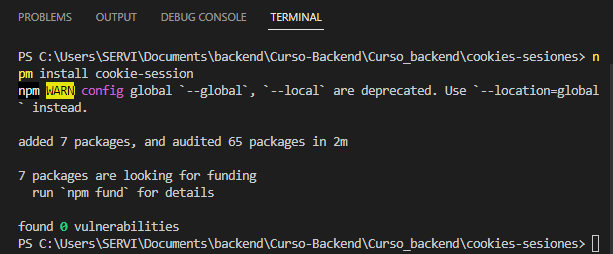
Seguido le damos enter a todo y después instalamos express con el sig. comando:

npm install express

Express por defecto no tiene como manejar las cookies, la implementación para el framework esta separa en una dependencia llamada “cookies-session” que tenemos que instalar. El framework esta separado en multiples dependencias y solo se selecciones aquellas que la aplicación web necesitara.

Lo instalamos de la sig. manera:

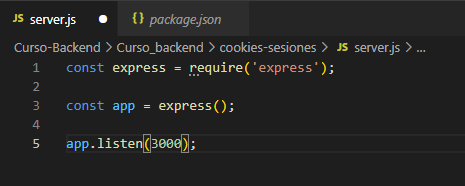
npm install cookie -session



Esta dependía esta únicamente pensada para almacenar cookies.

Ya con las dependencias instaladas vamos a configurar nuestro servidor. Realizamos la misma configuración

* importamos express.
* generamos el objeto app.
* ponemos a la escucha nuestro proyecto con el puerto 3000



Para configurar las cookie-session, importamos la librería “cookie-session”.

const cookieSession = require('cookie-session')

El manejador de cookie-session es también un middleware que se encarga de encriptar las cookies para que las guarde el navegador, de extraerlas en cada petición http y desencriptarlas para que podamos manejarlas.

Para pedirle a express que use cookie-session, tenemos que pasar el resultado de ejecutar la librería. Importamos como si fuera una función hacia el método “use” del objeto app usando el sig. código:

app.use(cookieSession());

Como parte de la ejecución de cookie-sessión tenemos que enviar un objeto “json” de configuración en el que especifiquemos información relevante para la forma en que se guardara en las cookies.

El primero será el nombre de la cookie donde se guardará la información, por defecto el nombre es sesión, pero podremos cambiarlo con el parámetro “name” de la sig. forma:

name: 'cookita', (xD)

Luego pasaremos un arreglo con claves para poder firmar y verificar los valores que se guardaron en la cookie con el parámetro “keys” de la sig. forma:

keys: [''], (Dentro podemos poner lo que queramos e incluso en diferentes srings)

Se usar para firmar las cookies mismas, se firman usando la primera cadena que especifiquemos el resto de las claves que especifiquemos se usan para verificar las cookies y rotar las llaves haciendo mas complicado que alguien lore hallar la llave, vulnerando nuestra aplicación, las cookies se guardan en el cliente, por eso nunca debemos guardarlas sin encriptarlas y podría alguien venir y utilizar el navegador de otra persona para extraer la información que guardemos en las cookies.

Otro parámetro es el “maxage” como en el cache que es un numero que en milisegundos tendrán las cookies antes de expirarlas, además tenemos una opción “expire” para colocarle una fecha de expiración.

Para poder almacenar u obtener una sesión usamos el objeto session que ahora forma parte del objeto request con el que hemos trabajado para demostrar su uso, haremos un servidor web que cuente las funciones que hemos ingresado, colocamos la estructura get del objeto app de la sig. forma:

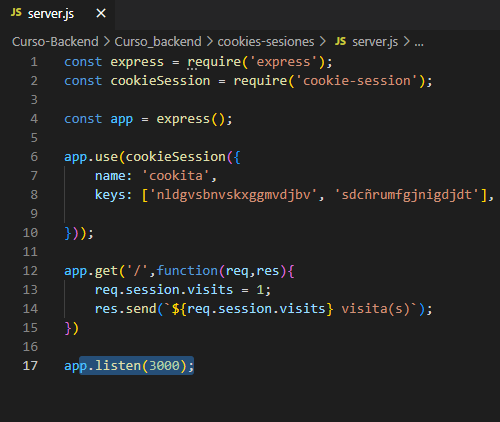
app.get('/',function(req,res){

})

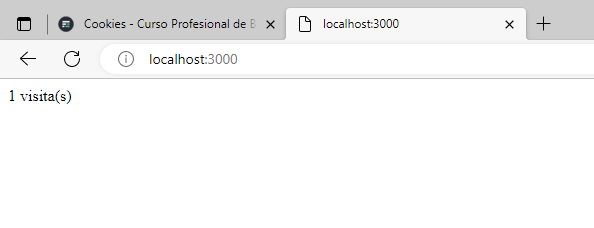
Lo primero que haremos será almacenar en la cookie las sigs. propiedades:

req.session.visits = 1;

res.send(`${req.session.visits} visita(s)`);



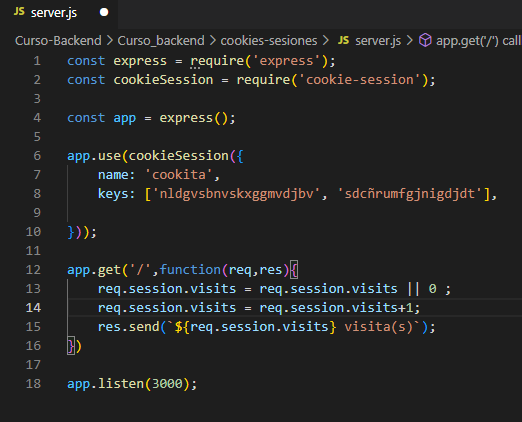
Con esto logramos contar la cantidad de visitas que se ha entrado. Luego guardamos e iniciamos el servidor de la carpeta “cookies-sesiones”.



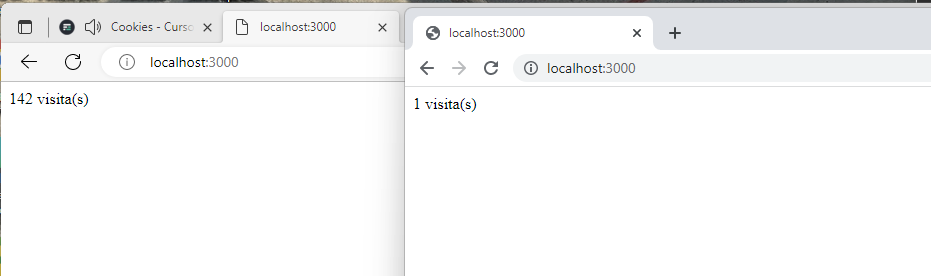
Como se ve es un valor estático, ahora lo configuraremos de modo que el valor se sume cada vez que visitemos la pág. cambiando el “req” de la siguiente manera:

req.session.visits = req.session.visits || 0 ; (si nunca se ha visitado dará error, debido a que no tiene valor, con expresión le daremos el valor de 0)

req.session.visits = req.session.visits+1;



Si accedemos desde otro navegador tendrá otro valor diferente.



Esto nos indica que cada navegador tiene su almacén de cookies y este no se comparte

# Bases de datos

## El lenguaje SQL

Structured Query Language o SQL como comúnmente lo conocemos, es el lenguaje a través del cual realizamos consultas a un motor de base de datos.

SQL se ubica en la categoría de los que conocemos como lenguajes de dominio específico.Lenguajes diseñados especialmente para realizar tareas dentro de un dominio bien definido, el contraste de estos lenguajes son los de propósito general que nos sirven para la solución de tareas en dominios distintos.

El dominio del lenguaje SQL es por supuesto el de las bases de datos. Originalmente fue conocido como Sequel, pero algunas restricciones con el registro de dicho nombre hicieron que se cambiara SQL aunque algunas personas conservaron la pronunciación, es por eso que a veces nos referimos a SQL como Sequel o Esciuiel o Esecuele.

SQL está compuesto por 3 sublenguajes, el de la definición de datos, con el que definimos el esquema de nuestra información, el lenguaje de manipulación de datos, y el lenguaje de declaración de consultas, con el que precisamente obtenemos la información almacenada en nuestro motor.

Las instrucciones SQL que escribamos eventualmente serán procesadas por un procesador de consultas SQL en el motor de nuestra base de datos, este procesador generará un plan de ejecución de la consulta que pueda ejecutarse sobre el motor de base de datos, este plan está especialmente optimizado para que la información sea generada lo más rápido posible,

Como desarrolladores Backend vamos a usar SQL para escribir consultas que nos permitan hacer esas 3 operaciones, definir nuestro esquema, insertar datos y consultarlos, para eso usamos las reglas del lenguaje y las palabras reservadas que se han definido.

SQL es un lenguaje complejo y para entenderlo a fondo necesitamos no sólo saber qué operaciones podemos definir, si no cómo se ejecutarán. En conjunto, una comprensión clara del lenguaje y de su ejecución, nos permitirán obtener información de manera rápida y eficiente.

Es por eso que sería imposible integrar ese conocimiento en este curso, por lo que voy a limitarme a enseñarte las consultas para ejecutar las operaciones más comunes de una aplicación web, y aunque estoy seguro de que con lo aprendido en este curso podrás empezar a crear aplicaciones web con bases de datos relacionales usando el lenguaje SQL para consultas.

## Configurando SQLite

El motor de base de datos que vamos a usar para este curso es SQLite, es un motor ligero y fácil de instalar, que está pensado para almacenar cantidades de datos pequeñas y es ideal para el desarrollo de aplicaciones.

Dicho esto, también es importante considerar que para cuando subamos nuestra aplicación al internet, seguramente vamos a querer utilizar un motor de base de datos más robusto como mySQL o Postgres, y ojo, aunque todos estos motores usan SQL como lenguaje de consultas, cada uno de ellos agrega o cambia ciertas cosas al lenguaje, así que ten cuidado con qué operaciones vas a utilizar y considera si están disponibles en todos los motores de base de datos en los que tu aplicación vaya a ejecutarse.

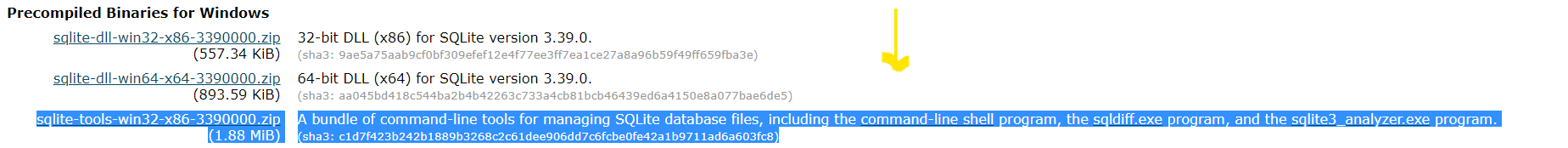
INSTALACIÓN EN WINDOWS

Para instalar sqlite3 en Windows necesitas, en primer lugar, dirigirte al sitio web cahí te dirigirás a la sección Precompiled Binaries for Windows, en la que encontrarás los binarios para Windows, de ahí tienes que descargar los archivos:

sqlite-tools-win32-x86-3390000.zip

(1.88 MiB)

A bundle of command-line tools for managing SQLite database files, including the command-line shell program, the sqldiff.exe program, and the sqlite3\_analyzer.exe program.



Una vez que descargues los archivos y los descomprimas, ya que originalmente vienen como un .zip, tienes que moverlos a la carpeta system32 para agregarlos C:\WINDOWS\system32

Una vez que hayas instalado y configurado sqlite deberás poder ejecutar el siguiente comando en la Terminal o el CMD para poder visualizar la versión instalada de sqlite

sqlite3 –version

## Que son las bases de datos y por qué las usamos.

Un base de datos es cualquier almacén de archivos.

Un motor de base de datos es el algoritmo o programa que se encarga de almacenar y obtener datos de ese almacén, nos permite manejar mucha información, consultarla, actualizarla, eliminarla, modificarla, cálculos etc. Donde existes dos tipos, los de bases de datos relacionales y no relacionales.

## Drivers de conexión

Cuando vamos a trabajar con algún motor de bases de datos, existe una brecha entre nuestro programa y nuestro entorno de ejecución, y el motor de bases de datos. Esta brecha se cierra utilizando un Driver.

Este driver es el que nos permite comunicarnos con el motor de base de datos, internamente estos programas, librerías o dependencias como quieras llamarlos, implementan los protocolos de comunicación necesarios para ejecutar operaciones sobre la base de datos.

Por supuesto cada motor de base de datos tiene necesidades distintas, por lo que el funcionamiento interno de cada driver depende de con qué base de datos estamos buscando comunicarnos.

Estos drivers a su vez exponen una API, es decir una serie de métodos, objetos y funciones con los que nos podemos comunicar con la base de datos.

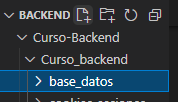
Cuando busques una librería que haga este trabajo debes de buscar considerando el lenguaje de programación o entorno de ejecución y el motor de base de datos. Por ejemplo, driver para Go y mySQL, o driver para Python y Postgres, etc.

En el caso de nuestro conjunto de tecnologías, necesitamos algo que nos permita comunicar NodeJS con Sqlite, precisamente hay una dependencia para eso en NPM que se llama sqlite3, así que procedamos a configurarla.

## Crear una tabla (Primer consulta)

Lo primero que haremos será configurar el driver Sqlite3 para que nuestros programas ejecutados con nodejs se puedan comunicar con Sqlite.

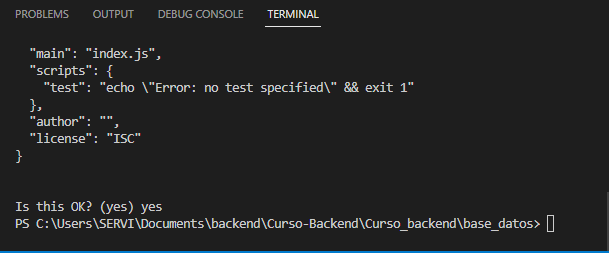
Para ello crearemos un nuevo proyecto el cual llamaremos “base\_datos”.



Luego en la terminal nos ubicamos en la capeta y le insertamos el sig. comando:

npm init

Y a todos los valores nomas le damos enter.



Luego toca instalar las dependencias

* Express: npm install express
* Sqlite3: npm install express

Luego crearemos el archivo “server.js” aunque en esta ocasión no será un server como tal.

Luego en el archivo colocaremos la librería y una instrucción para conectar con la base de datos que sería el sig. código:

const sqlite3 = require('sqlite3');

let db = new sqlite3.Database('Nombre de la base de datos');

Para esta práctica llamaremos “proyecto-backend” a la base de datos.

También podemos crear una base de datos anónima en la memoria RAM con el sig. código:

:memory (sirve para pruebas iniciales pero por ahora no lo haremos)

Luego se procede a realizar/correr una consulta, para ello el objeto de la base de datos tiene un método “run”.

db.run();

Luego dentro de el vamos a colocar un string que contenga una sentencia en el lenguaje SQL vamos a comenzar con aquella que nos deja crear una tabla a la cual llameremos en este caso “task”que sería la sig.:

db.run('CREATE TABLE task()');

Dentro de los paréntesis colocaremos los sigs. campos:

* Identificador del registro: ID (int)
* Texto: description (varchar)

Cada campo debe de tener su tipo de dato, ya sea varchar, int, boolean etc. Para ello insertaremos el sig. código:

db.run('CREATE TABLE task(ID int, Description varchar(255))');

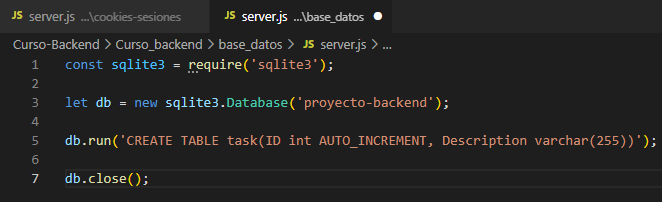
“255” es la cantidad de caracteres que puede almacenar.

Para no tener que llevar el control del identificador único colocaremos la palabra reservada “AUTO\_INCREMENT” que indica que la base de datos lo llenara automáticamente tomando el valor anterior y aumentándolo en 1.

db.run('CREATE TABLE task(ID int AUTO\_INCREMENT, Description varchar(255))');

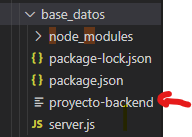
Por último, se colocará una instrucción de cierre de la conexión para evitar fugas de memoria que afecten el rendimiento de la computadora.

db.close();



Luego guardamos y ejecutamos el servidor con “node server.js”.

Veremos que en nuestra carpeta de “base\_datos” se creo un nuevo archivo que es nuestra base de datos.



## Qué es un CRUD

En programación solemos usar el término CRUD para referirnos a las operaciones básicas que puedes realizar sobre un conjunto de datos y por sus siglas son:

Crearlos, ya sabes, nuevos registros, cuando hablamos de bases de datos esto quiere decir insertar información.

Leerlos, r por Read, esto quiere decir consultar esa información, ya sea un registro o una colección de estos registros.

Actualizarlos, u por Update, que significa tomar un registro que ya existe en la base de datos y modificar alguna de las columnas.

Por último eliminar registros, d por Delete, que significa tomar un registro y quitarlo del almacén.

Los CRUDs son programas que involucren todas estas operaciones sobre una entidad, como en nuestro caso en el que trabajaremos con tareas pendientes, por eso nombré a la tabla tasks, o tareas.

## Insertar registros en la base de datos.

Como la tabla ya fue creada borraremos la sentencia de crear tabla, importante una tabla ya creada no puede volver a ser creada, es decir, si ejecutamos de nuevo la sentencia recibiremos un error.

Primero ejecutamos la estructura básica de un servidor express.

* Importamos la librería express: const express = require('express');
* Obtenemos el objeto app: const app = express();
* Ponemos en escucha un puerto para el servidor: app.listen(3000);

Luego definimos una ruta post, que será “/pendientes” y la función manejadora para esta petición con la sig. sentencia:

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

}));

Como es una ruta “post” para obtener los campos necesitamos traducir el cuerpo del mensaje para lo cual necesitaremos body parse. Por lo cual instalamos la dependencia en la carpeta “base\_datos” con el sig. comando:

npm install body-parser

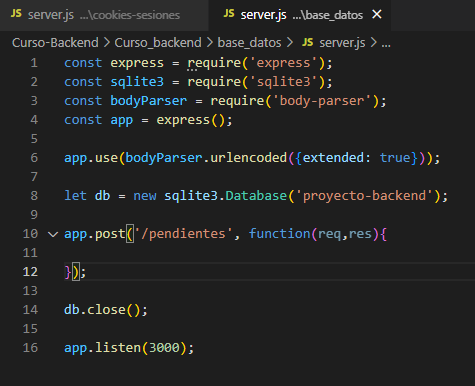
Luego configuramos el middleware que nos sirve para leer los campos que vienen en la petición, por lo cual primero importamos body-parser y luego lo colocaremos como un middleware de nuestro servidor web que ayude al análisis de la información que viene en el mensaje http, por lo cual agregaremos las sigs. sentencias:

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

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

Ya que la aplicación esta previamente configurada, conservaremos la línea con la que abrimos la base de datos.

let db = new sqlite3.Database('proyecto-backend');



Ahora nuestro desafío será donde debemos de cerrar la conexión a nuestra base de datos, no podemos colocarla como una instrucción en el script por que se cerraría antes de escuchar peticiones, pero tampoco podemos dejarla abierta por puede afectar el rendimiento de nuestro equipo, queremos que la conexión sea cerrada cuando el servidor sea terminado, por lo cual lo haremos de la sig. manera.

process.on('',function () {

})

Este objeto nos permite escuchar eventos relacionados con el proceso, es decir, cada que ejecutamos nuestro programa un nuevo proceso del sistema operativo es creado para nuestro programa, este proceso está en ejecución y se encarga del control de nuestro código, a la vez este proceso puede recibir mensajes del exterior como el que recibe cuando presionamos ctrl + c para cerrar el servidor, con el objeto “process” podemos escuchar estos mensajes y ejecutar código acorde, en este caso vamos a escuchar al mensaje “SIGINT” que es el que envia el proceso cuando colocamos ctrl +c y después cerraremos el servidor.

process.on('SIGINT',function () {

console.log('El servidor no esta disponible regrese mañana y con un buen lonche de bisteck para el administrador');

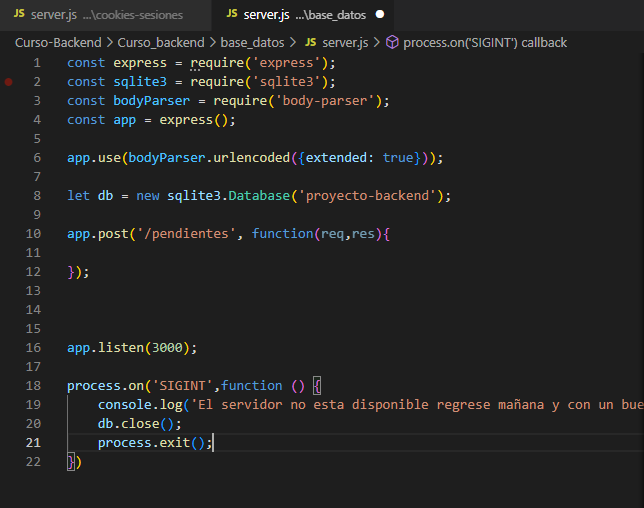
db.close();

})

Para evitar interrumpir el flujo de cerrar el servidor le agregaremos:

process.exit();

Que es lo que finalmente cierra el servidor de node independientemente de donde se ejecute. Con eso ya cerraremos la conexión a la base de datos cuando queramos.



Ahora ya podemos trabajar con la base de datos, vamos a hacer la inserción en la petición post a la ruta “pendientes”, para ello tenemos que correr una nueva consulta SQL de la sig. manera:

db.run('');

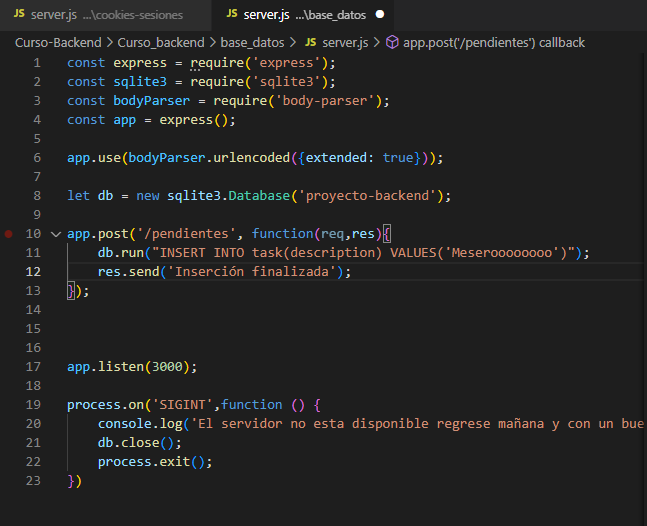
La consulta para hacer una inserción es “INSERT INTO”, luego colocamos el nombre de la tabla “task” en este caso, luego la instrucción VALUES seguido de paréntesis de los datos que queremos insertar. Como no queremos insertar un valor para el campo ID, tendremos que especificar uno por uno a que campos queremos asignar un valor, por lo cual en task o nombre de la tabla, escribiremos el campo y separado por coma los campos a los que queremos insertar un valor, por ejemplo haremos lo sig,:

db.run("INSERT INTO task(description) VALUES('Meseroooooooo')");

Nota: se cambian por dobles comillas por que dentro de la declaración de la consulta usaremos las comillas para identificar una cadena.

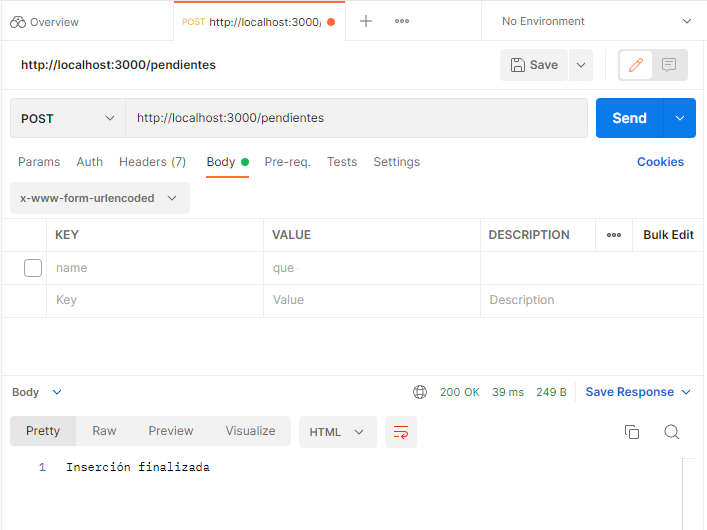
Una ves listo vamos a enviar un mensaje que diga inserción finalizada con la sig. sentencia:

res.send('Inserción finalizada');

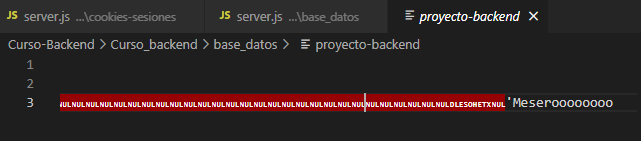


Luego guardamos y ejecutamos el servidor y después abrimos POSTMAN.

En POSTMAN indicamos que queremos hacer una petición “Post” a /pendientes, sin argumentos y le damos a “send”.



Luego abrimos nuestra base de datos “proyecto-backend” y veremos que en algún punto aparece el mensaje que acabamos de mandar.



## Limpiar parámetros (sanitize).

Para este punto nosotros solo estamos insertando cadenas estáticas, por ello vamos a hacerlo mas interesante insertando los elementos que vengan desde una petición http. En el ejemplo previo configuramos toda nuestra petición para recibir peticiones http vía POST y también el analizador del cuerpo de una petición (app.use(bodyParser.urlencoded({extended: true}));) para obtener los datos que ese nos envía en este caso con el metodo urlencoded.

Tomaremos un campo “description” del cuerpo de la petición para indicar que eso vamos a insertar en la base de datos,

Cambiaremos esto:

db.run("INSERT INTO task(description) VALUES('Meseroooooooo')")

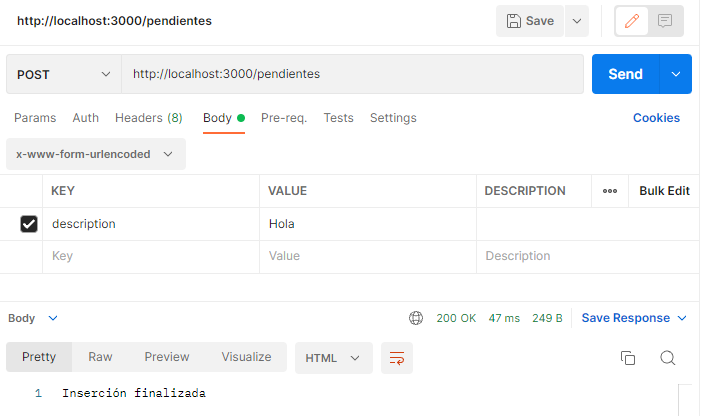
Por esto:

db.run(`INSERT INTO task(description) VALUES('${req.body.description}')`);

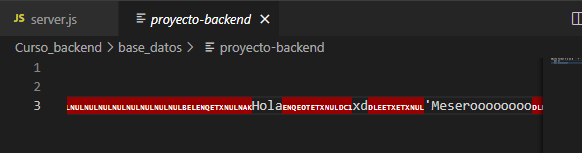
Dejamos las comillas simples para indicar que se sigue tratando de un string y dentro se coloca “${}” y ahí por así decirlo se va a sustituir esto por lo que se nos envié en el campo “description” en el cuerpo a la petición http, después de ello volvemos a ejecutar el servidor con el comando “node server.js”.



Usando postman enviaremos un nuevo valor con el campo “description” a la tabla, en la ruta <http://localhost:3000/pendientes>, en body, en key colocaremos el campo “Description” y “Value” el valor que le queremos dar.



Para ver que el dato llego, nos dirigimos a la base de datos y verificamos que el valor fue insertado.



Funciona, pero este método contiene un error y es que puede exponer nuestra aplicación a toda clase de ataque SQL Injection, este tipo de ataques se presentan cuando no limpiamos los parámetros que enviamos a una consulta como es el caso de este ejemplo. Justo ahora una persona podría insertar su propio código SQL como valor del campo “description” y al hacerlo nosotros parte de la sentencia durante la interpolación nos exponemos a que efectivamente el atacante obtenga resultados que nosotros no deseamos que obtenga, usando su conocimiento SQL el atacante puede obtener tablas, contraseñas, usuarios etc.

Para solucionarlo la mayoría de los drivers que usamos para comunicarnos entre un lenguaje y una base de datos tienen la funcionalidad de hacer un proceso que en el inglés se llama “Sanitize” o limpiar, este proceso se aplica sobre aquello que el usuario nos allá enviado y evita que ese ejecute como código SQL y se interprete como un string común y corriente sin importar que haya colocado el atacante.

En el caso SQLite3 el driver de conexión tiene un proceso que se da de la siguiente manera:

* Sustituimos los valores que el usuario vayan a asignar con un signo de interrogación.

db.run(`INSERT INTO task(description) VALUES('${?}')`);

* Luego pasamos los valores que se reemplazaran por el signo de interrogación como signos adicionales al llamado de “run” aparte de la sentencia original que después van a ir sustituyendo los signos de interrogación.

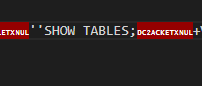
db.run(`INSERT INTO task(description) VALUES('?')`, req.body.description);

* La librería hará el trabajo de primero lavar los caracteres que puedan comprometer la consulta, luego ira reemplazando de izquierda a derecha los signos de interrogación por cada argumento que hayamos mandado como extra al método “run”, es decir se puede colocar tantos signos de interrogación queramos y tantos valores a sustituir, siempre un signo de interrogación por cada argumento.

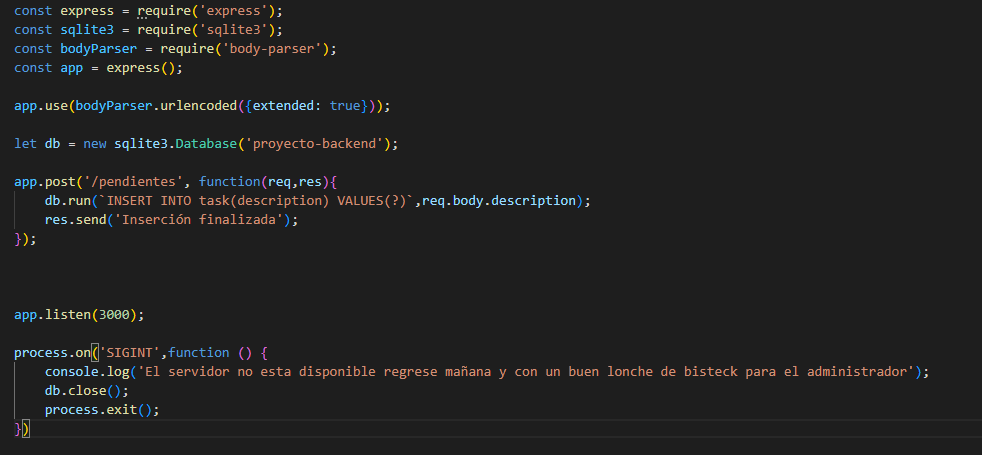
(Nota: durante el video se explica que es sin comillas, por eso todavía da error)

db.run(`INSERT INTO task(description) VALUES(?)`, req.body.description); (asi es correcto)

* Ahora llegan a la base de datos tal cual se escribe



La ventaja ahora es que nuestra aplicación ya no es susceptible a ataques SQL. La lección aquí es que nunca se debe insertar datos que el usuario envió en una instrucción SQL de ningún tipo sea inserción, consulta, actualización o lo que sea sin antes y previamente haber limpiado esta información (este proceso por lo general lo hacen los drivers de conexión, investigar como hacer en caso de que cambiamos de lenguaje de programación.)



## ¿Qué es un ORM?

Aunque podríamos desarrollar una aplicación web completa usando instrucciones SQL, estas pueden convertir nuestro código en uno muy verboso, complejo de entender y difícil de modificar.

Alternativamente, los desarrolladores backend usamos lo que se conocen como Object Relational Mapping, u ORM por sus siglas. Estas librerías se encargan de ofrecer clases y métodos para que podamos manipular la base de datos usando programación orientada a objetos.

Internamente, estos métodos se traducen a SQL para que las operaciones puedan ejecutarse sobre la base de datos. Además de eso, las instrucciones generadas por un ORM son usualmente seguras y optimizadas, esto porque los mejores ORM’s tienen años en desarrollo y son de código abierto, permitiendo que los mismos usuarios del ORM reporten y solucionen bugs, de modo que los resultados del ORM están en constante mejora.

Si aún no te imaginas cómo podrían verse las consultas SQL con objetos, considera este ejemplo con Sequelizejs, un ORM para NodeJS.

La inserción a nuestra tabla tasks pasaría de verse así:

db.run(`INSERT INTO tasks (description) VALUES (?)`,req.body.description);

A verse así:

Task.create({

description: req.body.description

});

Como puedes ver, el ejemplo con objetos y métodos es más legible y claro que la instrucción SQL, por otro lado, aprender SQL es mucho más complejo que aprender a usar un ORM, como podrás ver del ejemplo mismo.

Sin embargo, es importante mencionar que, para sacar el mayor provecho de un ORM, y poder optimizar las sentencias que se generan, sí debes saber SQL y fundamentos de bases de datos relacionales.

Por otro lado, uno de los grandes beneficios de usar un ORM, es que nos permite intercambiar el motor de base de datos, sin tener que modificar el código que utilizamos, simplemente le decimos, ahora usa sqlite, y ahora usa Postgres, y la librería se encargará de generar el código SQL para el motor indicado.

Este es un beneficio muy valioso considerando que ciertos motores están mejor diseñados para tareas específicas, sqlite para el desarrollo del sitio, Postgres para el entorno en producción, etc.

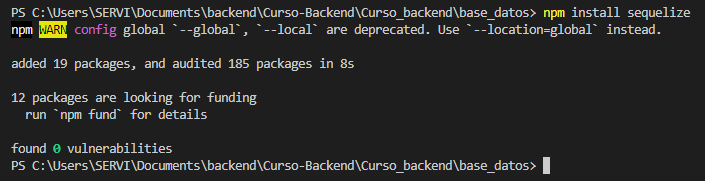
En general, te recomiendo que, a menos que tengas una razón especial para no usar un ORM, siempre los utilices, protegerán tu aplicación de posibles ataques por una base de datos mal configurada, harán del motor de base de datos prescindible, de manera que no casarás tu aplicación con uno de ellos, y además las operaciones serán más fáciles de comprender y razonar cuando ves tu código, que si usas SQL.

## Configurar SequelizeJS

Existe una gran disponibilidad de ORM´s disponibles para cada lenguaje de programación, en general aquel que sirve para bases de datos relacionales, esta disponible para varios motores de base de datos, uno de ellos es SequelizeJS y funciona bara bases de datos en MySQL, Postgress y Sequelize.

Para empezar, instalaremos la dependencia usando “NPM” y trabajaremos sobre el mismo proyecto que hemos trabajado, por lo que ejecutaremos el sig. comando de instalación en la carpeta de este proyecto:

npm install sequelize



Una vez instalado lo primero que haremos será importar la librería de sequelize:

const Sequelize = require('sequelize');

Luego tenemos que configurar la conexión sqlite con sequelize, para ello debemos instanciar un nuevo objeto de la clase sequelize.

Esto ya no lo vamos a necesitar así que lo borramos:

let db = new sqlite3.Database('proyecto-backend');

Lo cambiaremos por:

const sequelize = new Sequelize();

La clase/constructor recibe de tres a cuatro argumentos que son:

1. Nombre de la base de datos.
2. Nombre de usuario.
3. Contraseña.

En este proyecto nuestra base de datos no tiene ni usuario ni contraseña por lo que en esos dos argumentos le pondremos “null” y se escribirá así:

const sequelize = new Sequelize('proyecto-backend',null,null);

En el cuarto argumento pasamos un JSON de configuración en el que pondremos opciones mas específicas para la conexión con un objeto de la sig. manera:

const sequelize = new Sequelize('proyecto-backend',null,null, {

dialect: 'sqlite', (Se especifica el motor de la base de datos con el que vamos a conectar sequelize usando “dialect”)

storage: './proyecto-backend', (Con “storage” haremos referencia a la ubicación del archivo)

});

Con esto podremos eliminar el código con el que previamente nos conectábamos a sqlite ya que no lo vamos a necesitar, por lo tanto, ya no nos tendremos que desconectar por lo que eliminamos el sig. código:

process.on('SIGINT',function () {

console.log('El servidor no esta disponible regrese mañana y con un buen lonche bisteck para el administrador');

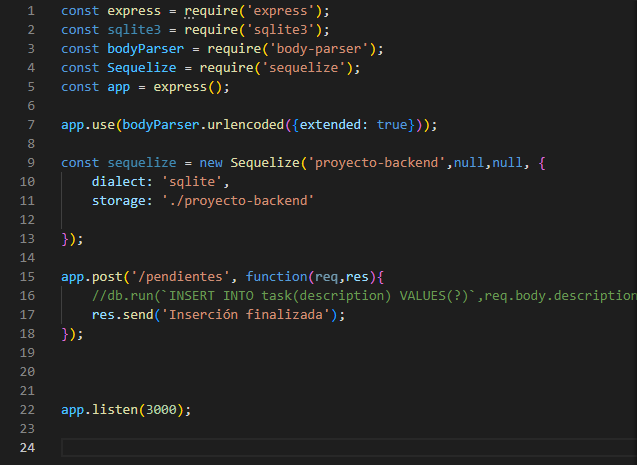
db.close();

process.exit();

})

Comentamos el sig. código:

db.run(`INSERT INTO task(description) VALUES(?)`,req.body.description);



Después guardamos y procederemos a reiniciar el servidor.

## Base de datos relacionales y no relacionales

En el mercado de las bases de datos podemos distinguir dos principales grupos, las bases de datos relacionales y las no relacionales.

Cada que inicies una nueva aplicación web, vas a enfrentarte con la decisión de si debes usar un tipo u otro, por eso en este vídeo voy a explicarte las diferencias y te daré algunos consejos para decidir entre un tipo u otro.

Vamos a empezar por darnos una idea visual de la diferencia entre un tipo y otro. Las bases de datos relacionales suelen estar representadas por un conjunto de tablas, estas tablas tienen una estructura definida de campos, como las tablas, podemos pensar en estas entidades como estructuras rígidas, nuestros registros entonces sólo pueden tener la forma de una fila que llena los campos de la tabla.

Las bases de datos no relacionales se componen de colecciones de documentos, algunas bases de datos representan estos documentos como objetos JSON. Estas bases de datos son más flexibles, te permiten guardar los campos que quieras en cada registro que vas guardando, como en un objeto, tú puedes decidir qué campos guardar para cada uno de los objetos que irán al almacén.

Este primer panorama nos dice que, si nuestros datos tienen una estructura bien definida, usaremos bases de datos relacionales y si no, usaremos no relacionales.

Si vamos al nombre, la diferencia está en las relaciones, unas, las relacionales las tienen, y otras, las no relacionales, simplemente no.

Usualmente los esquemas de datos que pensamos para nuestra aplicación tienen elementos que están relacionados entre sí, por ejemplo si estamos creando una red social, las publicaciones tienen una relación de propietario con el usuario que las creó, porque son SUS publicaciones, le pertenecen a él, las fotos por ejemplo, le pertenecen a alguna persona, en otros casos las relaciones son más complejas, por ejemplo en una aplicación como CódigoFacilito tenemos cursos, que son tomados por muchas personas y esas mismas personas pueden tomar uno o varios cursos.

Estas relaciones suelen trasladarse a nuestro esquema de base de datos de manera que sea fácil poder hacer consultas basado en estas relaciones, como, por ejemplo, obtener todas las publicaciones que son propiedad de un usuario, o todos los cursos que una persona está tomando, o al revés, qué personas están tomando un curso.

Algunas de estas operaciones son más complejas aún, qué cantidad de cursos estás tomando, cuál es el promedio de los exámenes que has tomado, etc.

Para eso las bases de datos relacionales no sólo ofrecen la posibilidad de establecer estas relaciones, si no que, además, ofrecen los mecanismos para hacer operaciones aprovechando las relaciones de nuestro esquema, desde cálculo de datos, agrupación y obtener datos de múltiples entidades que estén relacionadas entre sí.

Por otro lado, algunas de estas operaciones son muy lentas o requieren de un consumo importante de memoria.

Es por esto que las bases de datos no relacionales ofrecen una alternativa en la que las operaciones entre elementos relacionados se sustituyen por otras estrategias, cada base de datos ofrece algunas distintas, en algunas de ellas por ejemplo duplicamos información o almacenamos resultados de un cálculo para no tener que estarlo realizando cada que necesitemos esa información.

Esta es la principal diferencia entre ambos tipos de bases de datos, las relaciones. Mi recomendación es que utilices las bases de datos relacionales cuando estas relaciones representan parte importante de tu aplicación, ya que aunque sí podrás replicarlo con las estrategias no relacionales, el modelo es más complejo de razonar y puede llevar pasos extra que eventualmente se recompensan con un mejor rendimiento.

Las bases de datos no relacionales son excelentes para aplicaciones donde la consulta es constante y debe hacerse rápido, los chats por ejemplo son un buen ejemplo, constantemente estamos escribiendo datos y consultándolos.

Para la mayor parte de los casos, y especialmente ahora que estás comenzando, vas a necesitar una base de datos relacional, es una de este tipo la que usaremos en lo que resta del curso. Sin embargo, en CódigoFacilito mismo tenemos otros cursos con bases de datos no relacionales, así que más adelante podrás aprender a trabajar con este otro tipo de motores.

# Buenas prácticas de desarrollo

## Presentación del bloque

Cuando escuchamos el término “buenas prácticas de desarrollo” normalmente se refiere a estrategias que los desarrolladores con más experiencia utilizan al programar a fin de buscar algunos de los sigs. beneficios:

1. Mejor organización del código: los proyectos grandes suelen involucrar miles de archivos distintos por eso es importante saber donde tenemos que buscar cada cosa, porque es imposible recordar donde se deja cada archivo.
2. Código claro y legible: el buen código se entiende a la primera vista, no solo por quien desarrollo el proyecto sino también por otros miembros del equipo o personas nuevas que se integre al proyecto.
3. Código modular: el buen código además es reemplazable, en un proyecto bien organizado y programado, las piezas que lo componen pueden reemplazarse, mejorarse o actualizarse sin afectar al resto del proyecto.
4. Código fácil de corregir: las aplicaciones fallan constantemente, el buen código nos dice dónde están estos errores, donde han sucedido y principalmente hace que sea fácil corregir estos problemas, si hay varios archivos y sale un error, encontrarlo es muy difícil.

En este capitulo aprenderemos sobre el MVC una arquitectura muy popular en el desarrollo de proyectos web y por otro lado sobre REST que habla acerca de como aprovechando http debemos de definir el código y su funcionalidad.

## ¿Qué es el MVC?

El concepto de arquitectura en programación es muy usado, hablamos de arquitectura en muchos conceptos, el MVC por su parte es un tipo de arquitectura que se utiliza en el desarrollo de aplicaciones de software, se puede decir que una arquitectura es una solución previamente diseñada para problemas en común.

Esta solución define guías del problema, estas guías pueden decirnos como debemos separar nuestro código como en el caso del MVC, el nombre de esta arquitectura proviene:

* M: modelo
* V: vistas
* C: controladores

Esta arquitectura define que nuestro proyecto debe estar organizado en estas tres capas:

Presentación: vistas que es todo aquello que ve el usuario.

Datos: guardarlos, consultarlos, modificarlos, estructurarlos etc. Todo lo que tenga que ver con los datos va con los modelos.

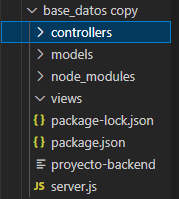
Comunicación: son los controladores que están entre el modelo y los vistas. Comunica ya sea las vistas al modelo y del modelo a las vistas.

## Organizar un proyecto MVC

Tomaremos de base del proyecto del capítulo pasado ya que tendremos configurado sequekize y sqlite para la base de datos.

Para lo cual haremos un duplicado.

Luego crearemos una carpeta llamada “views”, esta carpeta es en donde los motores de vista de express buscan por defecto las vistas, por eso es importante tenerla. Adicionalmente habrá una carpeta llamada “models” donde colocaremos todo lo relacionado al manejo de datos, aquí es donde trabajaremos con un ORM que en este caso es “sequelize” ya que todas las consultas se hacen con clases, el mismo ORM nos permite generar estas clases, estas clases se convertirán en nuestros modelos, porque tienen funcionalidad para generar consultas a nuestra base de datos. La parte del controlador particularmente nodejs se presta para mas opciones, por ahora nuestro archivo server.js podría hacer la tarea del controlador, en aplicaciones grandes donde tenemos que manejar datos de muchas tablas es conveniente utilizar un archivo controlador por cada modelo y un archivo controlador para cada tabla, así si mi aplicación fuera a manejar una tabla usuarios y una tabla tareas tendría dos controladores, uno para usuarios y uno para tareas por lo que dentro del proyecto también añadiremos una carpeta llamada “controllers”.



## ¿Qué son las migraciones?

Administrar una base de datos es una tarea delicada que requiere de un control preciso sobre las modificaciones que se hacen a las bases de datos, un mal cambio puede alterar la información de nuestros usuarios o incluso peor hacer que las perdamos.

Las migraciones son archivos de código que definen los cambios que serán sobre el esquema de la base de datos, es decir, si vamos a crear una tabla, eliminarla, crear un nuevo campo, si vamos a modificar el tipo de dato que se va a guardar etc.

Las migraciones no solo nos permiten llevar un control de que cambios se van realizar, también ejecutan estos cambios, usan un lenguaje propio de la librería de migraciones que les permite definir los cambios y generar el SQL apropiado.

Una de las ventajas de que las migraciones ejecuten los cambios y no nosotros manualmente es que estas librerías ofrecen la funcionalidad de revertir estos cambios, que es extremadamente útil cuando hacemos un cambio que afecta nuestra aplicación o si nos equivocamos en un dato o si definimos mal el tipo de un campo. Eventualmente las migraciones se convierten en un historial de los cambios hechos sobre la base de datos, una manera de revertirlos y de tener control sobre el esquema de la información.

Las migraciones son utilizadas en muchos frameworks, reils, jango y sequelize ofrece su propio modulo para crear y ejecutar y revertir migraciones.

## CLI de Squelize

Existe una herramienta para la terminal que nos permite generar archivos y administrar un proyecto que contenga sequelize, esta misma utilidad nos ayuda a generar las migraciones para nuestro proyecto asi que vamos a instalarlo de usando el sig. comando:

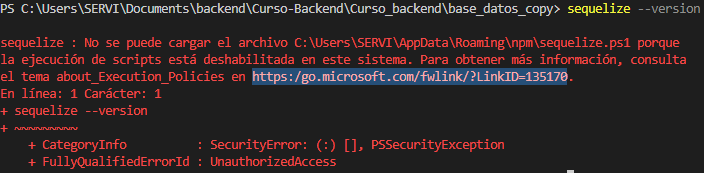
npm install -g sequelize-cli

Con la opción “-g” hará que el paquete se instale de manera global para que podamos tener acceso a este comando con el cual podremos generar y administrar nuevos proyectos de sequelize (tener acceso significa que el comando se puede utilizar en cualquier punto de la terminal y queda disponible tanto para este proyecto como para muchos otros).

Una vez instalado debemos tener acceso a un comando sequelize con el que vamos a ejecutar operaciones de administración del proyecto. Con el sig. comando podemos observar la versión con la que contamos:

sequelize –version

Puede que al inicio te salga el siguiente error:



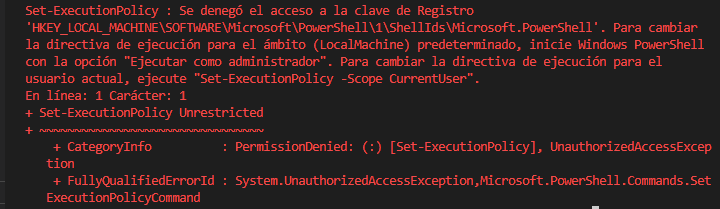
Para solucionarlo escribiremos el sig. comando:

Get-ExecutionPolicy

Con este comando nos va a mostrar si estamos restricted o unrestricted. Si sale restricted es por eso que no puedes ejecutar scripts y deberemos cambiarlo con el sig. comando:

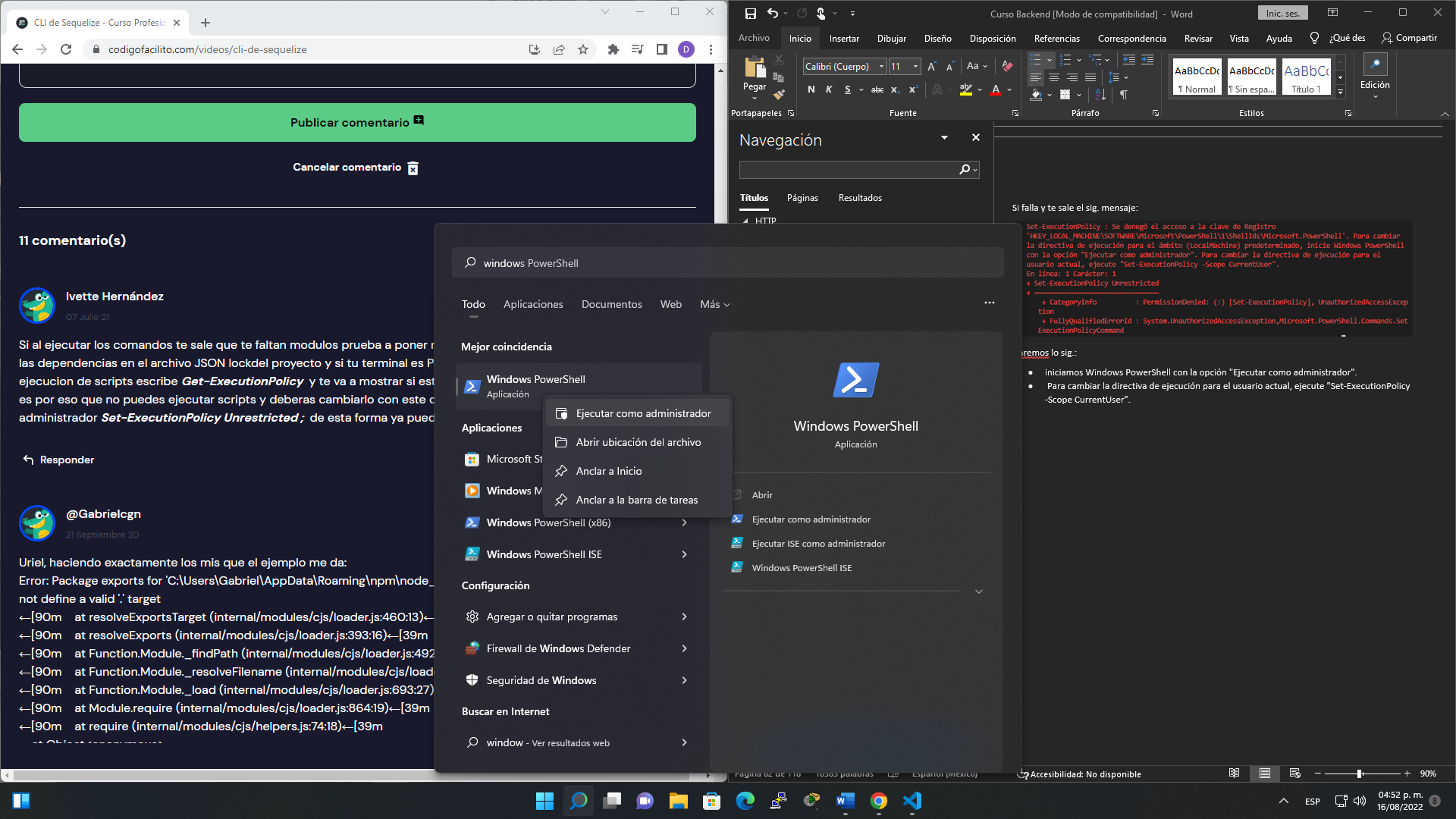
Set-ExecutionPolicy Unrestricted

Si falla y te sale el sig. mensaje:

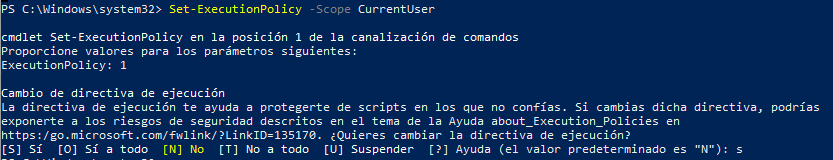


Haremos lo sig.:

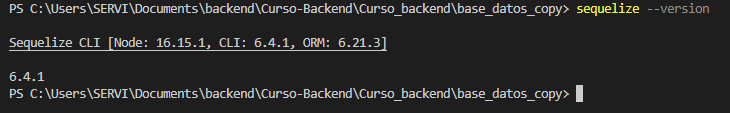
* iniciamos Windows PowerShell con la opción "Ejecutar como administrador".



* Para cambiar la directiva de ejecución para el usuario actual, ejecute "Set-ExecutionPolicy -Scope CurrentUser".
* Después escribiremos 1 y luego S



Ahora si con esto podremos ejecutar scripts para powershell que lo que hace los comandos de sequelize.



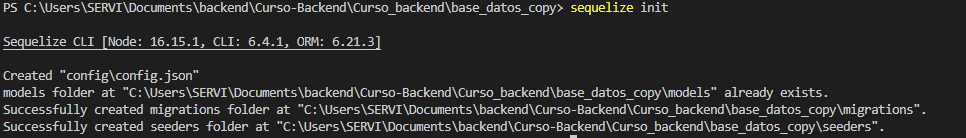
Una vez resuelto el problema y visto la versión de nuestro Sequelize CLI continuemos.

A este tipo de programas los llamamos CLI o Comando Line Interface y existen CLI para varias librerías, se llaman CLI por que se ejecutan en la línea de comandos a través de las secciones definidas nos permite generar o automatizar operaciones como la creación de archivos.

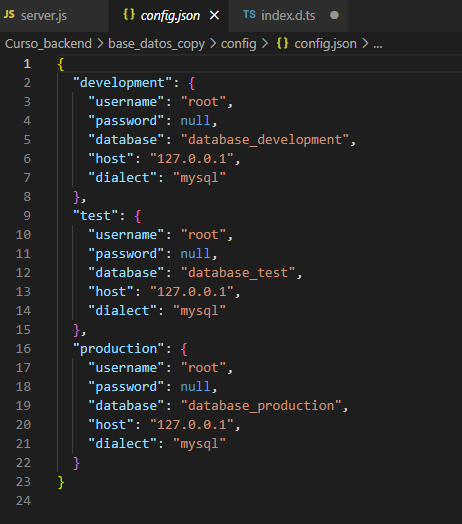
Antes de iniciar una migración vamos a ejecutar la operación “init” de la utilidad de sequelize para que esta configure nuestra carpeta como un proyecto de sequelize con el sig. comando:

sequelize init

Es importante que estemos en la carpeta del proyecto donde trabajaremos. Si se hizo correctamente se habran creado las carpetas "migrations" y “seeders” y tendremos el sig. mensaje:



En “migrations” irán las migraciones para modificar nuestra base de datos y la carpeta “seeders” sirve para llenar nuestra base de datos con datos de prueba. Tambien agrego la carpeta “config” donde tendremos un archivo config.json en el que vamos a colocar las credenciales que requeriremos para conectarnos a nuestra base de datos.

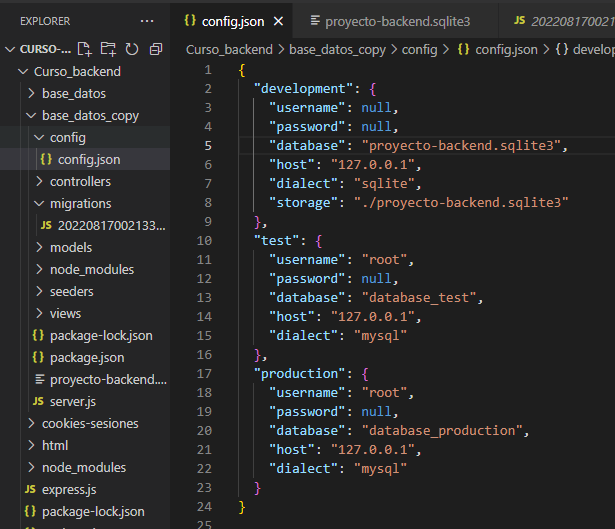


El cual tiene tres distintos grupos de credenciales:

* Test(pruebas)
* Development(desarrollo)
* Production(producción)

Vamos a modificar estos datos para que usen nuestra base de datos de desarrollo y el bloque que usaremos será “development”:

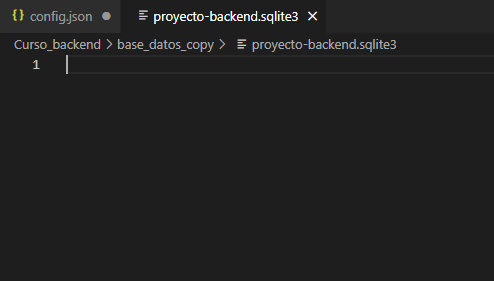
* En “username” y password colocaremos “null”.
* En “database” el nombre de nuestra base de datos que será en este caso para efectos de la práctica "proyecto-backend".
* En dialecto le colocaremos “sqlite” nuestro motor de base de datos.
* Le agregaremos una opción llamada “storage” donde especificaremos que la base de datos debe de guardarse en el archivo “proyecto-backend” (siguiendo las buenas prácticas al nombre de la base de datos le agregaremos la extensión .sqlite3 que es como se identifican los archivos con base de datos de sqlite) y quedaría de la sig. manera: “./proyecto-backend.sqlite3”.



## Generando migraciones

Los modelos y las migraciones suelen ir de la mano ya que para un modelo corresponde una tabla y para crear una tabla necesitamos una migración es por eso que sequelize crea tanto el modelo como la migración correspondiente, el CLI tiene una opción “model generate” que generan ambos. Crearemos una migración para nuestra tabla la cual llevara el nombre de “tareas”, para que esto funcione es necesario que eliminemos la base de datos anterior ya que ahí ya existe la tabla tareas. Vamos a recrearla para seguir este flujo más profesional de trabajo.

Primero borramos todo lo que hay dentro de la base de datos “proyecto-backend”.



Para generar el modelo y la migración primero colocamos el sig. comando:

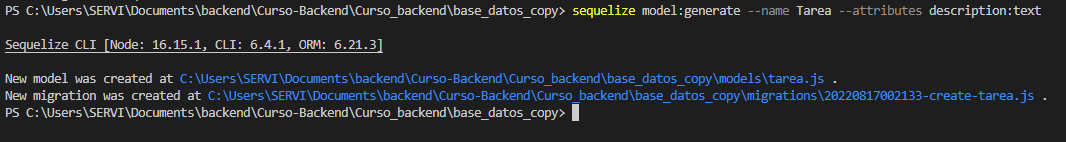
sequelize model:generate –name “NOMBRE DE LA BASE DE DATOS EN SINGULAR Y LA PRIMERA CON MAYUSCULA” –atributes “CAMPOS QUE SE GUADARAN EL LA TABLA INDICADA, ACOMPAÑADO DEL TIPO DE DATO”

sequelize model:generate --name Tarea --attributes description:text

<https://sequelize-mock.readthedocs.io/en/stable/api/data-types/>

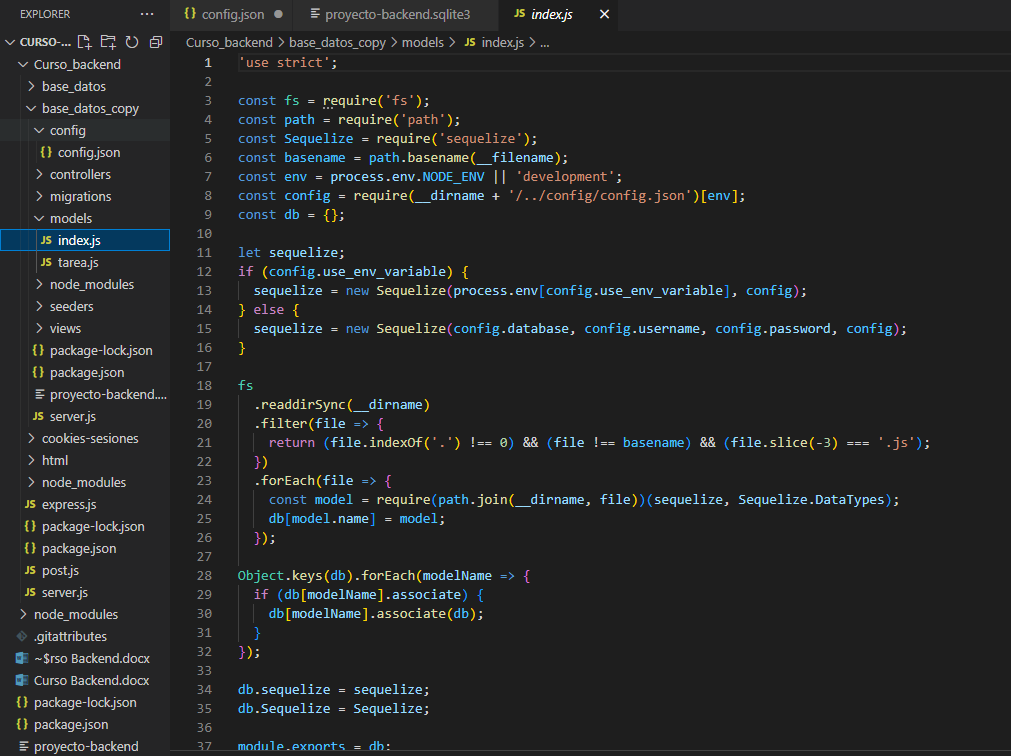
En este link encontraremos todos los tipos de datos aceptados para los campos que se pueden asignar a un modelo de sequelize usando el CLI.

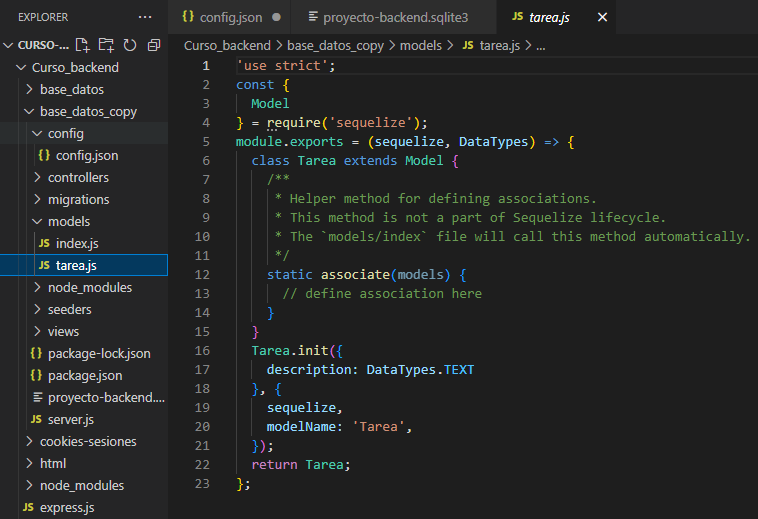
De momento nuestra tabla tendrá ese dato, en este caso no es necesario colocar un identificador único ya que la librería de la migración se encargará de crear ese campo y ejecutamos el comando.



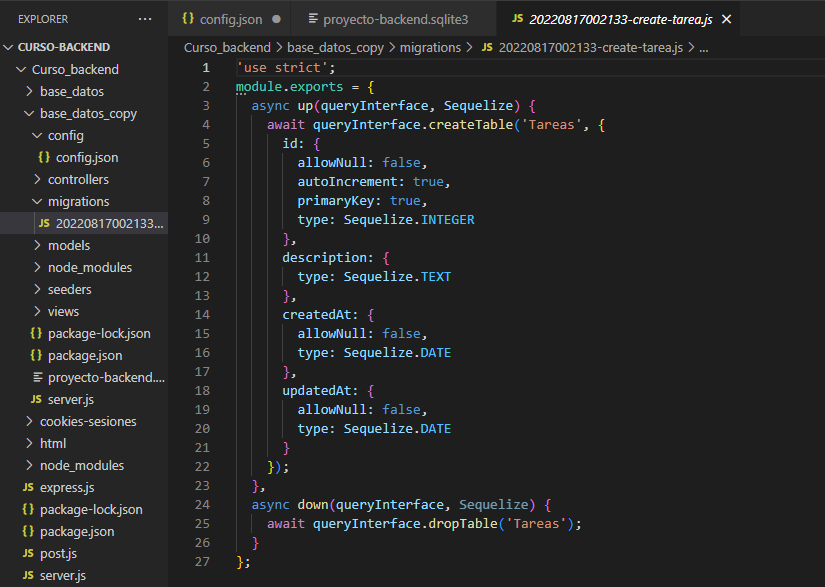
Esto nos generara dos archivos por un lado esta el modelo en la carpeta “models” y la migración en la carpeta “migrations”

* Modelo.



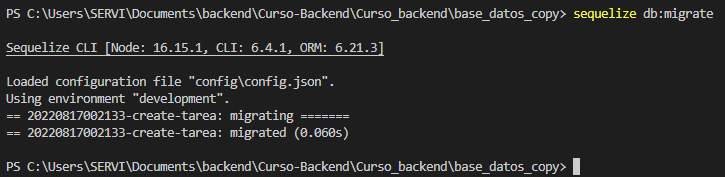


* Migracion.

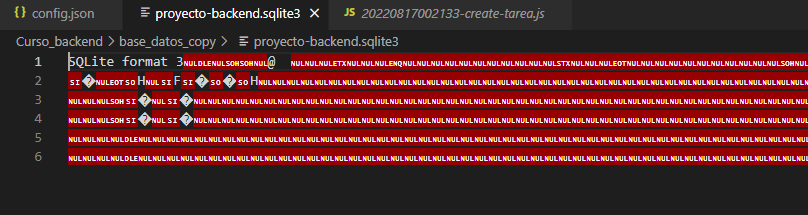


Decimos que ejecutamos una migración cuando precisamente ejecutamos este archivo de migración para que se apliquen las modificaciones que en la migración fueron definidas. Las librerías de las migraciones llevan un control interno de que migraciones se han ejecutado y cuales no se han ejecutado, así ofrecen un comando que se encarga de correr todas las migraciones que aun no han sido ejecutas y es el sig. comando:

sequelize db:migrate



Ejecutarlo nos mostrara un mensaje indicando que efectivamente la migración fue ejecutada y si abrimos el archivo “proyecto-backend.sqlite3” veremos que sufrió modificaciones.



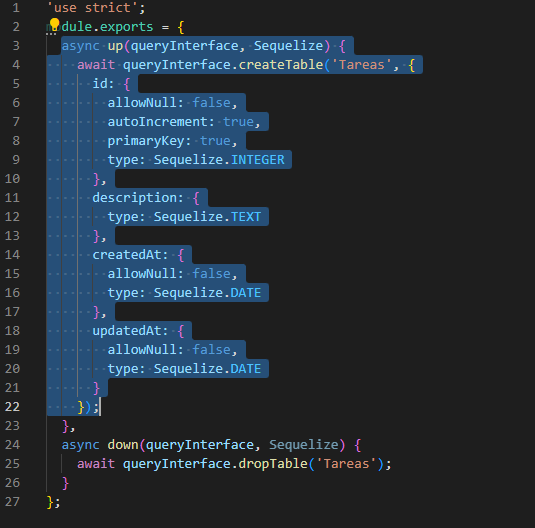
Debido a que creamos una nueva tabla con la migración.

Si exploramos la migración que aparece en la carpeta “migrations” el nombre se compone de un “timestamp” que es un valor numerico que presenta la fecha y hora en que fue creado la migración, este “timestamp” se acompaña de un descriptor de la migración

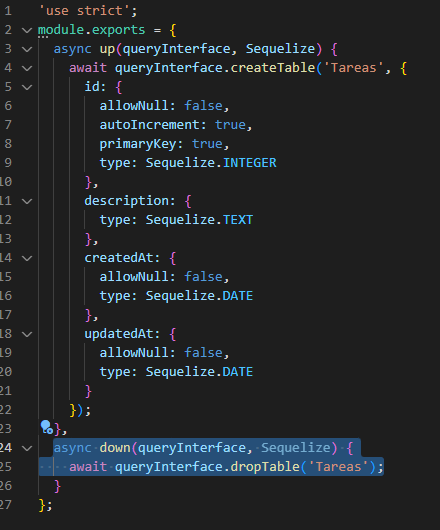


Como la mayoría de los manejadores de migraciones este archivo se divide en dos:

* Las instrucciones para el método up:



* Las instrucciones para el método down:

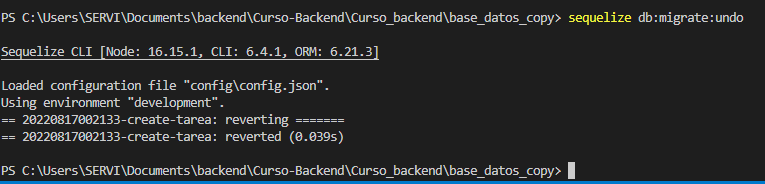


Las instrucciones del método “up” hacen referencia a los cambios que se harán a la base datos cuando corremos la migración.

Las instrucciones del método “down” hacen referencia a los cambios que se harán para revertir la migración misma.

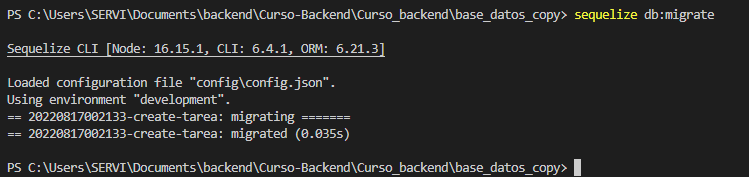
Si quisiéramos revertir la migración podríamos usar el sig. comado:

sequelize db:migrate:undo



Saldrá un mensaje donde se revertió la última migración, esta última hará que ya no tengamos una tabla “tarea” dentro de “proyecto-backend.sqlite3” y si la queremos volver a crear volvemos a insertar el comando:

sequelize db:migrate



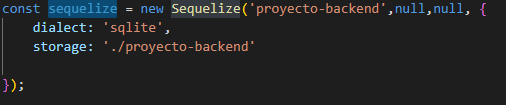
## Modelos

Cada ORM utiliza estrategias distintas para generar los modelos ya sea a través de herencias usando un método constructor, a final de cuentas el objetivo es generar una clase o un objeto con los métodos para generar las consultas SQL, es común que cada tabla se vea acompañada de modelo que haga las consultas sobre dicha tabla, sin embargo es importante saber que no todos los modelos corresponden a una tabla, si necesitáramos una clase que maneja datos y que no coincida con una tabla o que maneje información de varias tablas también podemos crear un archivo al que lo llamemos un modelo. Lo que significa que no es obligatorio que tengamos una tabla por cada modelo, podemos tener modelos que no coincida con una tabla.

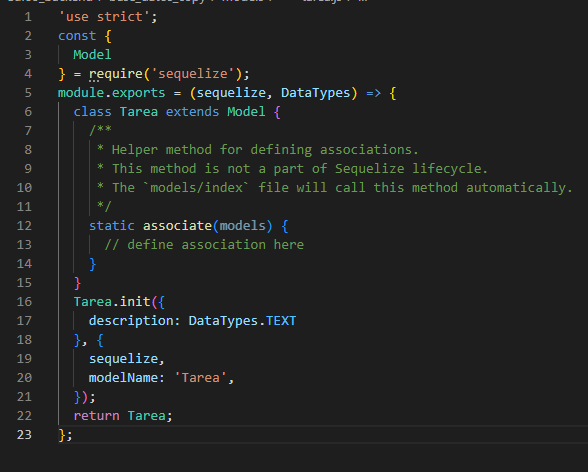
En los modelos es donde residirá la mayor parte de la lógica del código, uno de los consejos mas populares al trabajar con el MVC es que los modelos deben ser delgados, haciendo referencia a como gran parte del código de una aplicación web se escribirá en los modelos.

ANALIZANDO LOS MODELOS DE SEQUELIZE

Un modelo se define a partir del objeto de conexión que generamos cuando configuramos el ORM en nuestra aplicación mismo que aun se puede ver en el archivo server.js.



Este objeto tiene un método define que sirve para construir un nuevo modelo, a dicho método se le envía como primer argumento el nombre del modelo, es una convención que los modelos se escriban en singular y con la primera letra mayúscula justo como vemos en el archivo de nuestro modelo.



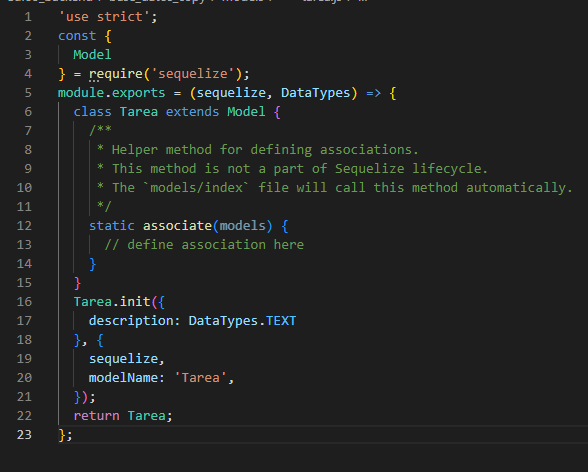
La tabla que corresponde a este modelo es el plural del modelo todo en minúsculas, si quisiéramos un modelo para la tabla “tareas” el nombre del modelo seria “Tarea”.

Como segundo argumento para la generación de un modelo en sequelize enviamos un objeto JSON en el que definimos la estructura que tendrá el modelo, es decir, los campos con los que cuenta la tabla correspondiente.

Hay que comenzar por decir que solo debemos conectarnos una vez a la base de datos, si instanciáramos varias veces el objeto como en server.js significaría que nos estuviéramos conectando múltiples veces a la base de datos, reduciendo el rendimiento de nuestra aplicación y abriendo la posibilidad de que algo salga mal.

Es por eso que el objeto sequelize tiene un método “import” que le permite importar un modelo que esta siendo definido en otro archivo diferente aquel en el que nos conectamos a la base de datos.

Para que nuestro modelo pueda ser importado por el método “import” tiene que exportar una función que reciba dos argumentos



Los dos argumentos son por un lado el objeto conectado a la base de datos y una clase DataTypes que contiene los diferentes tipos de datos disponibles para definir un modelo.

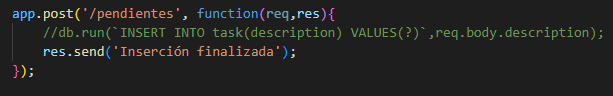
Eso es lo que genera sequelize, exporta una función que recibe dichos argumentos y que aprovechándolos exporta un nuevo modelo

## Controladores

Los controladores es otro de los tres componentes del MVC, son un intermediario entre las vistas y el modelo, entre lo que el usuario ve y la lógica de nuestra aplicación con los datos para formar la interfaz gráfica.

Generalmente un controlador recibe una petición http, consulta al modelo en caso de que sea necesario y hace el render de la vista con los datos que extrajo precisamente del modelo. Es el controlador entonces el que se encarga de recibir el mensaje http y de darle una respuesta.

Visto en temas anteriores hemos trabajado con manejadores de peticiones http, es decir, las funciones que reciben una petición y le dan una respuesta, cuando hablamos de controladores vamos a trabajar precisamente con código de este tipo:



Para obedecer la arquitectura MVC vamos a mantener separados los archivos modelo de los controladores, de las vistas, y de la configuración del servidor.

Los controladores los vamos a ir colocando en la carpeta “controllers”, luego de esto vamos a crear un archivo para manejar todas las peticiones http relacionadas con el modelo “Tarea”, es común que cada modelo tenga su controlador y que un controlador solo corresponda a un modelo (esta es una regla no escrita y en ocasiones habrá controladores que no empaten con ningún modelo).

Cuando un controlador corresponde a un modelo, la práctica común es nombrarlo con el plural del modelo, es decir nuestro modelo es “Tarea” y su controlador será “tareas”. Así que dentro de la “controllers” crearemos un archivo llamado “tareas.js”



Cuando nuestra aplicación se separa en múltiples archivos se recurre a la naturaleza modular de JavaScript para compartir información entre todos los archivos, las aplicaciones escritas con JavaScript son modulares, un archivo exporta piezas de funcionalidad como clases o funciones y otro archivo las importa

## Vistas

## Seeders

## Integrando todo

## ¿Qué es REST?

## REST en la práctica

## Verbos Http en REST

## Rutas REST en Express

## Crear nuevos registros

## Formularios

## Mostrar registros

## Vistas para todos los registros

## Identificadores únicos

## Consulta individual de recursos

## Actualizar Registros

## Formularios con PUT, PATCH y DELETE

## Eliminar registros

# Autenticación

## Explicando cómo funciona la autenticación de usuarios

## Formulario de registro

## Modelos y migración de usuarios

## Validaciones

## Validaciones en la base de datos y el modelo

## Encriptar el password

## Creación de usuarios

## Inicio de sesión

## Autenticación

## Manejo de sesiones

## ¿Qué es un middleware?

## Buscar usuario autenticado

## Cerrar sesión

## Middleware para protección de rutas

# Relaciones en la base de datos

## Fundamentos de relaciones en la base de datos

## Relaciones unos a muchos - El esquema

## Relaciones uno a muchos – Los modelos

## Nombres para asociaciones

## Guardar relación uno a muchos

## Relaciones en un proyecto real

## Eager loading (Cargado anticipado)

## Ejercicio – Implementa tu propio CRUD

## Relaciones mucho a muchos

## Integrando CRUD de categorías

## Tabla asociativa

## Definir relación muchos a muchos

## Crear relaciones muchos a muchos

## Mostrar objetos de una relación muchos a muchos

# Websockets (realtime)

## ¿Qué son los Websockets?

## Usuarios conectados en tiempo real

## Comunicar el servidor Http y el servidor de Websockets

## Enviar datos de la base de datos a los clientes

## Control de comunicación por usuario en Websockets

# Entorno de producción

## Conoce Heroku

## Instalacion y configuración de Git

## Adaptar el proyecto para Heroku

## Guardar sesiones en Postgress