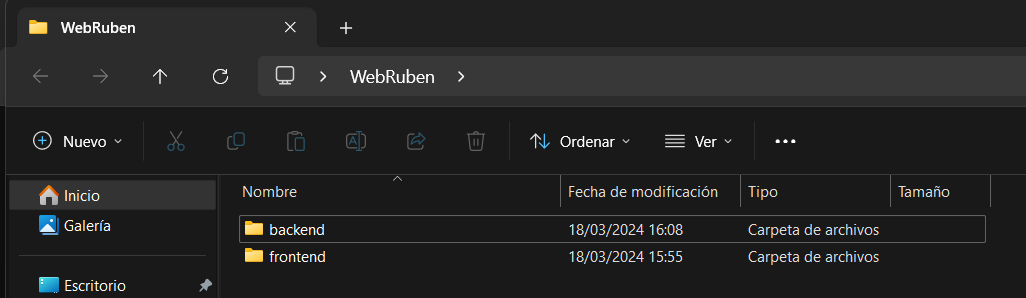
**TUTORIAL CREACION DE PROYECTO CON NODE JS – PHASE I**

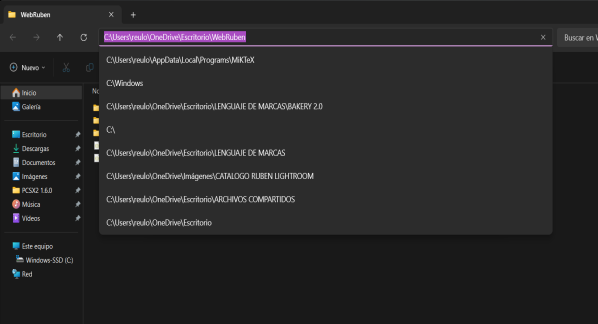
**1º CREAR LA CARPETA CONTENEDORA DEL SERVIDOR**

Creamos una carpeta con el nombre del proyecto que hagamos, en mi caso se llama:**WebRuben**.

Dentro, crearemos las carpetas **frontend** y **backend.**



**2º INICIALIZAR EL SERVIDOR E INSTALAR EXPRESS Y MORGAN**



Tenemos que indicar la ruta en la que se encuentra la carpeta en la que vamos a tener el proyecto que vamos a desarrollar.

Truco: Si haces click derecho sobre la carpeta, puedes abrir el cmd y te ubicara directamente en el directorio de la carpeta.

Otra forma es irte a la carpeta, buscar la URL y pegarla en el CMD de la siguiente forma:

**cd aqui\pones\la\ruta**

Una vez ubicados en la carpeta, instalamos los comandos: **npm init –yes** **npm i express morgan**

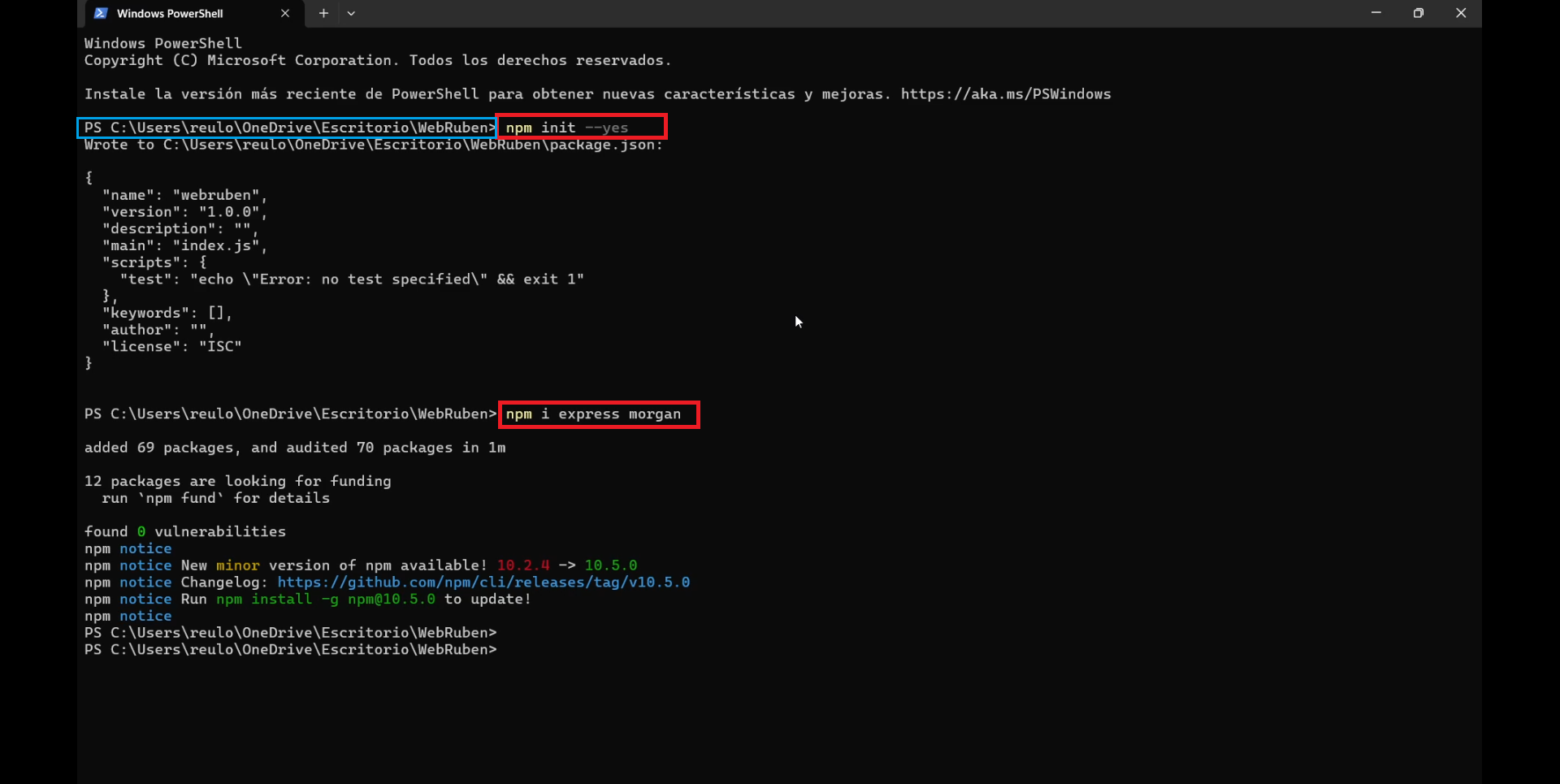
-El comando npm init --yes se utiliza para inicializar un nuevo proyecto de Node.js con valores predeterminados sin necesidad de realizar una serie de preguntas interactivas. Al agregar --yes al comando npm init, estás indicando a npm que acepte todas las opciones predeterminadas sin solicitar confirmación.

-El comando npm i express morgan se utiliza para instalar los paquetes express y morgan en un proyecto de Node.js utilizando npm (Node Package Manager). Aquí tienes una breve descripción de cada uno de estos paquetes:

+Express: Es un marco de aplicación web para Node.js que simplifica el proceso de creación de aplicaciones web y API. Express proporciona una serie de características útiles para manejar rutas, manejar solicitudes y respuestas HTTP, administrar sesiones, y mucho más. Es uno de los marcos web más populares en el ecosistema de Node.js debido a su simplicidad y flexibilidad.

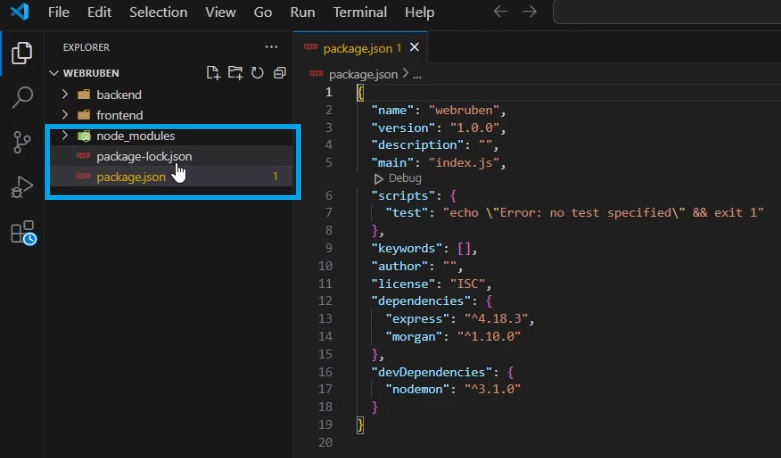
+Morgan: Es un middleware de registro de solicitudes HTTP para Node.js. Morgan simplifica la tarea de registrar las solicitudes HTTP en la consola o en un archivo de registro. Proporciona varios formatos de registro predefinidos y también permite personalizar el formato de registro según tus necesidades.

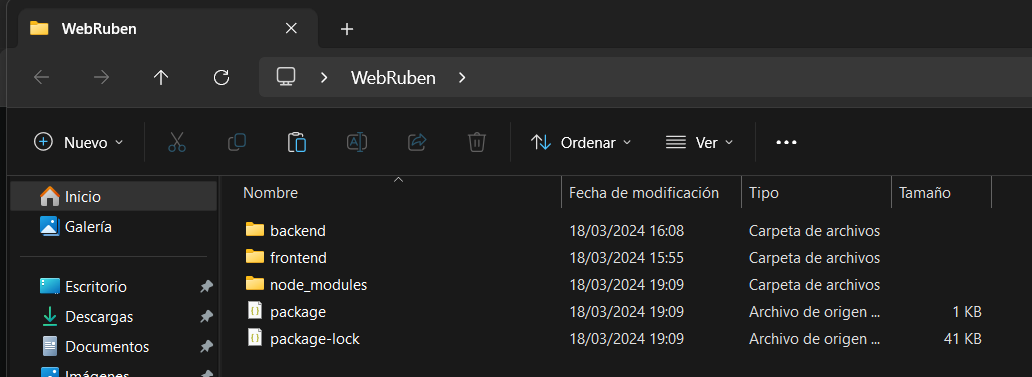
Cuando ejecutas npm i express morgan, npm instalará ambos paquetes en el directorio de tu proyecto y los agregará como dependencias en tu archivo package.json. Una vez instalados, puedes utilizarlos en tu código de Node.js requeridos como módulos.



**3º COMPROBAR QUE LA INSTALACION ES CORRECTA**

Cuando hayamos instalado los anteriores comandos, se nos creara en la carpeta lo siguiente marcado en la foto. Son todos los archivos y configuraciones iniciales necesarios para empezar nuestro servidor en NODE JS.

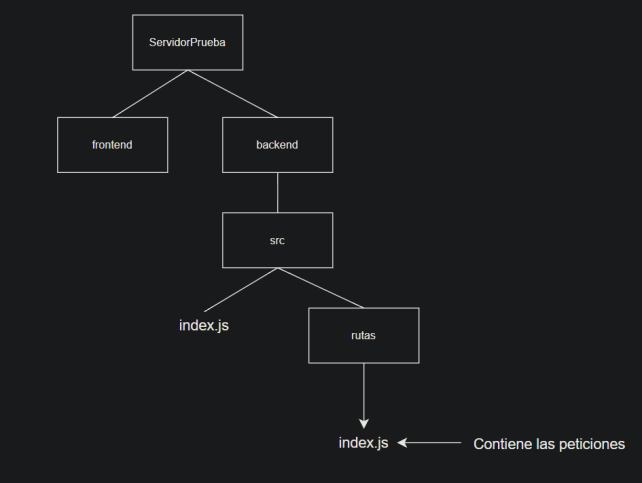
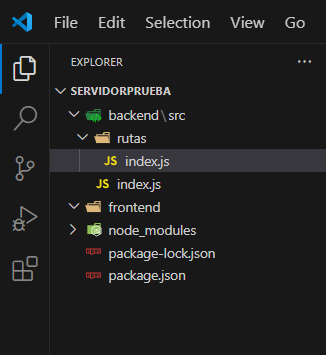




**4º CREAR LA ESTRUCTURA DE CARPETAS**

Ahora vamos a crear una carpeta llamada **src** dentro de la carpeta backend y dentro de src un archivo **index.js**

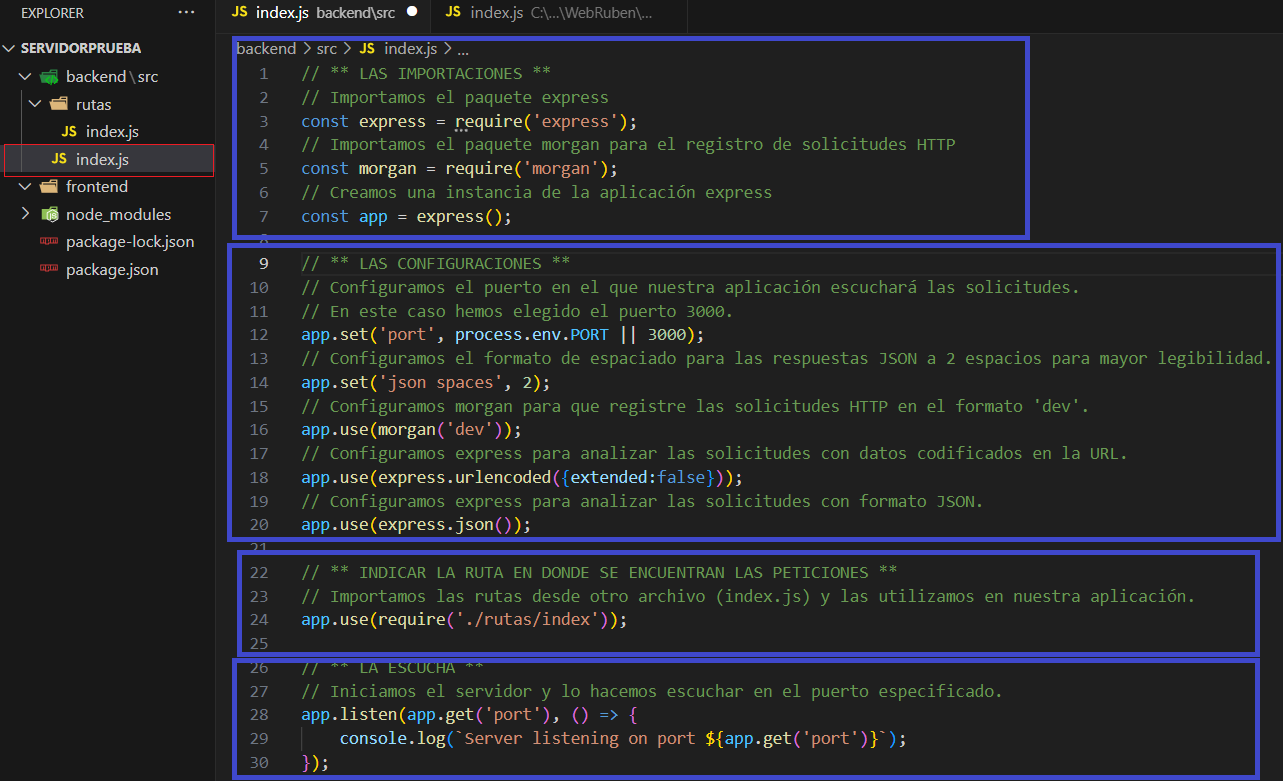
Dentro de src creamos otra carpeta llamada **routes** y dentro de esta el mismo archivo otra vez, **index.js**

****

**5º CREAR EL INDEX PRINCIPAL: IMPORTACIONES Y CONFIGURACIONES**

En el primer index.js que hemos creado (el que esta fuera de la carpeta rutas) vamos a escribir el código inicial para su configuración, que va a contener 4 cosas: las importaciones, la configuración, la ruta de las peticiones y el puerto de escucha. Escribimos lo siguiente:

Nota: Todas las request (peticiones) NUNCA se almacenan en este index, sino en el de rutas que veremos más adelante. En este lo que haremos será escribir la ruta (el otro index de la carpeta rutas) a la que tiene que dirigirse.



CODIGO PARA COPIAR

// \*\* LAS IMPORTACIONES \*\*

// Importamos el paquete express

const express = require('express');

// Importamos el paquete morgan para el registro de solicitudes HTTP

const morgan = require('morgan');

// Creamos una instancia de la aplicación express

const app = express();

// \*\* LAS CONFIGURACIONES \*\*

// Configuramos el puerto en el que nuestra aplicación escuchará las solicitudes.

// En este caso hemos elegido el puerto 3000.

app.set('port', process.env.PORT || 3001);

// Configuramos el formato de espaciado para las respuestas JSON a 2 espacios para mayor legibilidad.

app.set('json spaces', 2);

// Configuramos morgan para que registre las solicitudes HTTP en el formato 'dev'.

app.use(morgan('dev'));

// Configuramos express para analizar las solicitudes con datos codificados en la URL.

app.use(express.urlencoded({extended:false}));

// Configuramos express para analizar las solicitudes con formato JSON.

app.use(express.json());

// \*\* INDICAR LA RUTA EN DONDE SE ENCUENTRAN LAS PETICIONES \*\*

// Importamos las rutas desde otro archivo (index.js) y las utilizamos en nuestra aplicación.

app.use(require('./routes/index'));

// \*\* LA ESCUCHA \*\*

// Iniciamos el servidor y lo hacemos escuchar en el puerto especificado.

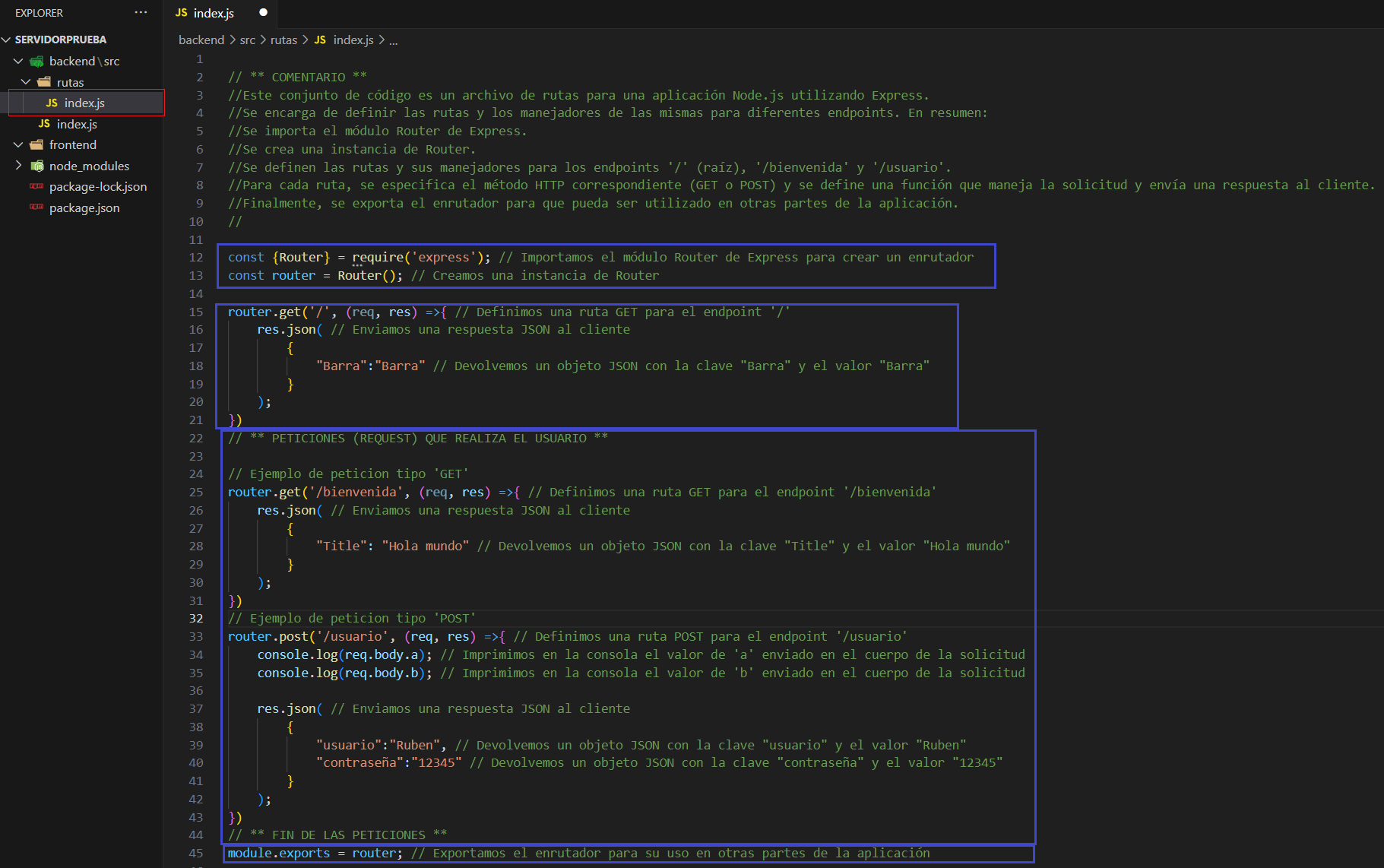
app.listen(app.get('port'), () => {

    console.log(`Server listening on port ${app.get('port')}`);

});

**6º CREAR EL INDEX DE PETICIONES**

Ahora nos dirigimos al index.js ubicado dentro de la carpeta rutas. Y escribimos el siguiente código.



CODIGO PARA COPIAR

// \*\* COMENTARIO \*\*

//Este conjunto de código es un archivo de rutas para una aplicación Node.js utilizando Express.

//Se encarga de definir las rutas y los manejadores de las mismas para diferentes endpoints. En resumen:

//Se importa el módulo Router de Express.

//Se crea una instancia de Router.

//Se definen las rutas y sus manejadores para los endpoints '/' (raíz), '/bienvenida' y '/usuario'.

//Para cada ruta, se especifica el método HTTP correspondiente (GET o POST) y se define una función que maneja la solicitud y envía una respuesta al cliente.

//Finalmente, se exporta el enrutador para que pueda ser utilizado en otras partes de la aplicación.

//

const {Router} = require('express'); // Importamos el módulo Router de Express para crear un enrutador

const router = Router(); // Creamos una instancia de Router

// \*\* PETICIONES (REQUEST) QUE REALIZA EL USUARIO \*\*

// Ejemplo de peticion tipo 'GET'

router.get('/bienvenida', (req, res) =>{ // Definimos una ruta GET para el endpoint '/bienvenida'

res.json( // Enviamos una respuesta JSON al cliente

{

"Title": "Hola mundo" // Devolvemos un objeto JSON con la clave "Title" y el valor "Hola mundo"

}

);

})// \*\* FIN DE LAS PETICIONES \*\*

module.exports = router; // Exportamos el enrutador para su uso en otras partes de la aplicación

--------------------------------------------------------------------------------------------------------------------------------------------------------------

**PETICIONES ‘GET’ vs ‘POST’**

Las peticiones GET y POST son dos de los métodos HTTP más comunes utilizados para comunicarse con los servidores web.

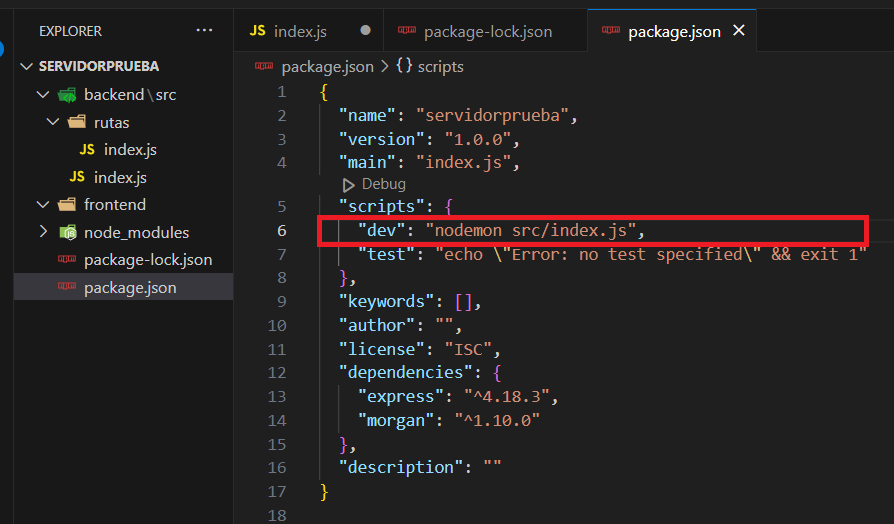
1. **GET**:
   * Una solicitud GET se utiliza para solicitar cualquier contenido específico del servidor.
   * Los datos se envían en la URL a través de parámetros de consulta (query parameters).
2. **POST**:
   * Una solicitud POST se utiliza para enviar datos al servidor para su procesamiento. Cuando se tienen que enviar al servidor web paquetes grandes de datos, como imágenes o datos de formulario de carácter privado, por ejemplo, el método GET se queda corto, porque todos los datos que se transmiten se escriben en abierto en la barra de direcciones del navegador.
   * Los datos se envían en el cuerpo (body) de la solicitud HTTP.

**Diferencias**:

1. **Datos**: En una solicitud GET, los datos se envían en la URL a través de parámetros de consulta, mientras que en una solicitud POST, los datos se envían en el cuerpo de la solicitud HTTP.
2. **Uso**: GET se utiliza para solicitar datos del servidor, mientras que POST se utiliza para enviar datos al servidor para su procesamiento.
3. **Visibilidad de datos**: Los datos enviados a través de GET son visibles en la URL, mientras que los datos enviados a través de POST no son visibles en la URL, lo que los hace más seguros en términos de privacidad.

--------------------------------------------------------------------------------------------------------------------------------------------------------------

**6.5 º AUTOMATIZACION DE REINICIO TRAS CAMBIOS**



nodemon se encargará de ejecutar tu servidor desde el archivo src/index.js, y se reiniciará automáticamente cada vez que detecte cambios en los archivos del proyecto. Esto es útil durante el desarrollo, ya que te permite ver los efectos de tus cambios en tiempo real sin tener que reiniciar manualmente el servidor cada vez.

**Nota: Cuando finalices una petición html, usa ctrl + c para finalizarla.**

**7º COMPROBAR PETICIONES TIPO ‘GET’**

Vamos a comprobar que las peticiones funcionan correctamente. En este caso tenemos dos peticiones:

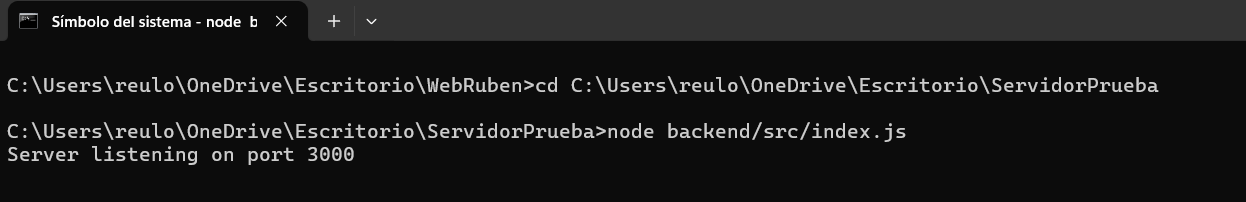
-Una petición tipo GET /bienvenida

-Otra petición tipo POST /usuario

Para ello tenemos que ir al CMD de Windows y ubicarnos en la carpeta en donde tenemos nuestro servidor. (explicado más en detalle anteriormente).

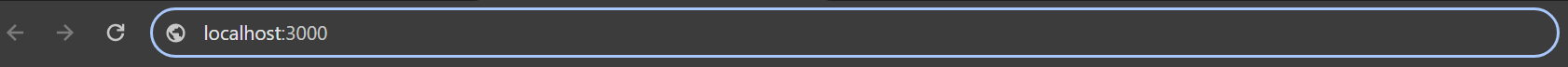
Alli tenemos que escribir el siguiente comando: node ruta/de/la/carpeta/contenedora/de/las/peticiones

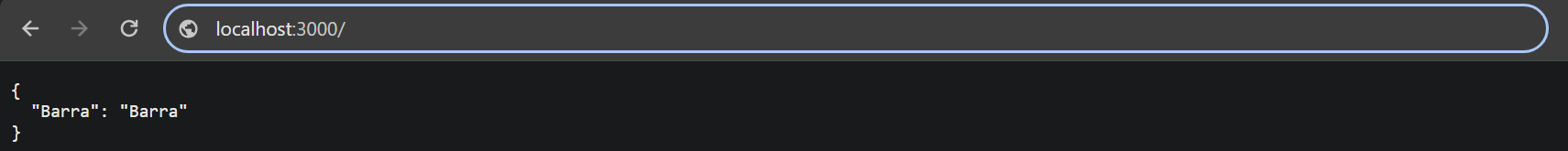
Si esta todo correcto, nos indicara que el servidor se encuentra a la escucha en el puerto que le indicamos antes, el 3000.

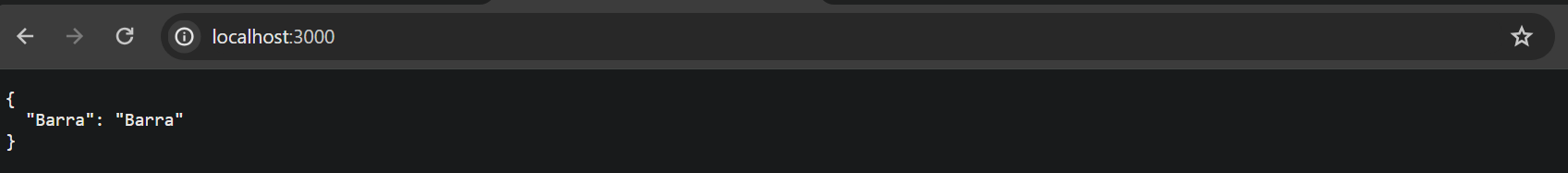


Esto quiere decir que si el usuario hace una petición en el buscador del navegador nuestro servidor mostrara en el navegar su contenido para esa petición si es que la hemos programado. Si escribimos localhost:3000 o localhost:3000/ encontramos que nos muestra lo programado que en este caso es lo mostrado, a modo de prueba.

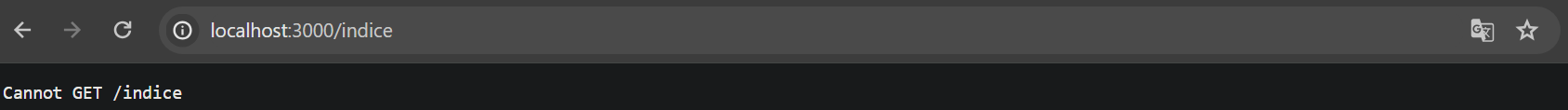
Nota:Ten en cuenta que escribir localhost:3000 o localhost:3000/ es lo mismo.



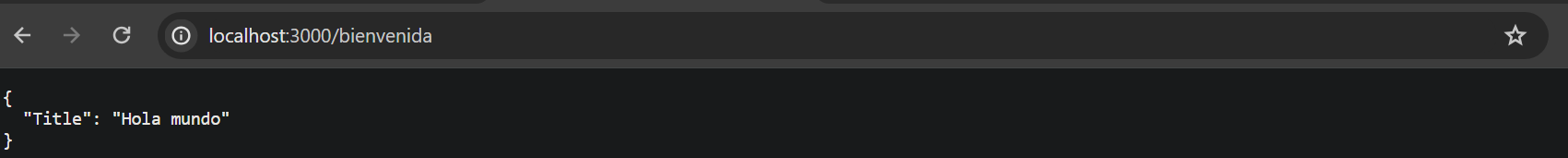




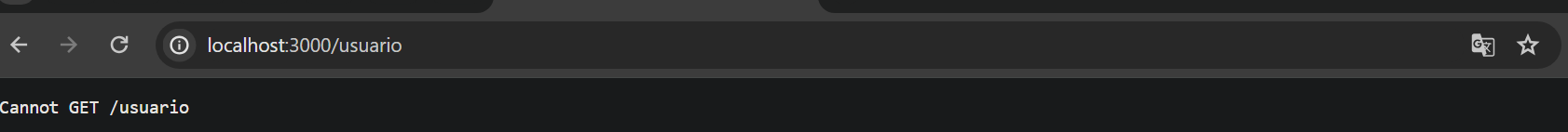
En cambio, si buscamos algo que no esté programado como /índice, podemos ver que nos devuelve un mensaje de error porque no existe.



Pero si buscamos algo que si hemos programado, como /bienvenida, aquí si vemos que nos muestra lo que le hemos programado.



En cambio, si hacemos lo mismo con /usuario, pese a estar programado veremos que nos da error, porque la petición es tipo post y no get, y por lo tanto no puede mostrarse en el navegador.



A su vez, podemos ver que en el CMD se nos muestran datos relacionados con las peticiones realizadas.

Esto podemos verlo gracias a los paquetes instalados de express y morgan y nos brindan ayuda para entender y buscar fallos.

Son registros de solicitudes HTTP recibidas por el servidor y sigue un formato común:

**MÉTODO |RECURSO |CÓDIGO DE ESTADO| TIEMPO DE RESPUESTA |TAMAÑO DE RESPUESTA**

Aquí puedes ver las categorías por tipos de estado de los códigos:  
 -Códigos de estado **1xx**: Información.

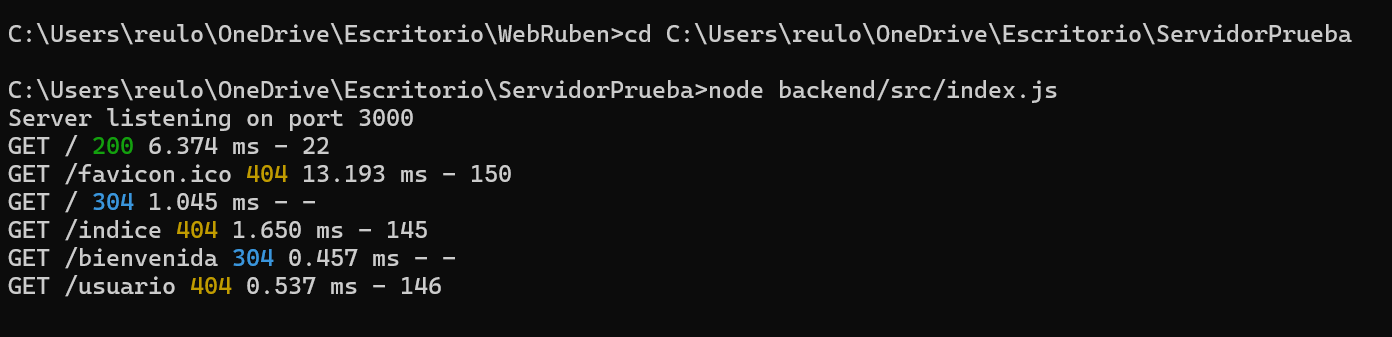
-Códigos de estado **2xx**: Éxito.

-Códigos de estado **3xx:** Redirección.

-Códigos de estado **4xx:** Error del cliente.

-Códigos de estado **5xx:** Error del servidor.

Aquí puedes ver que nos da el error 404 cuando las peticiones no se correspondían con nada que estuviese escrito.

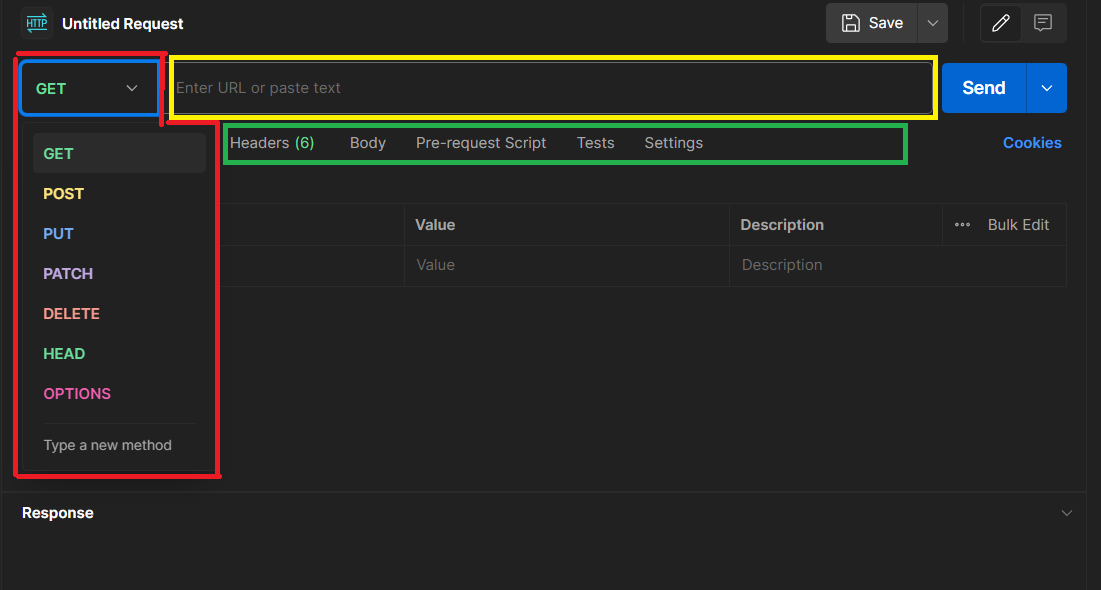
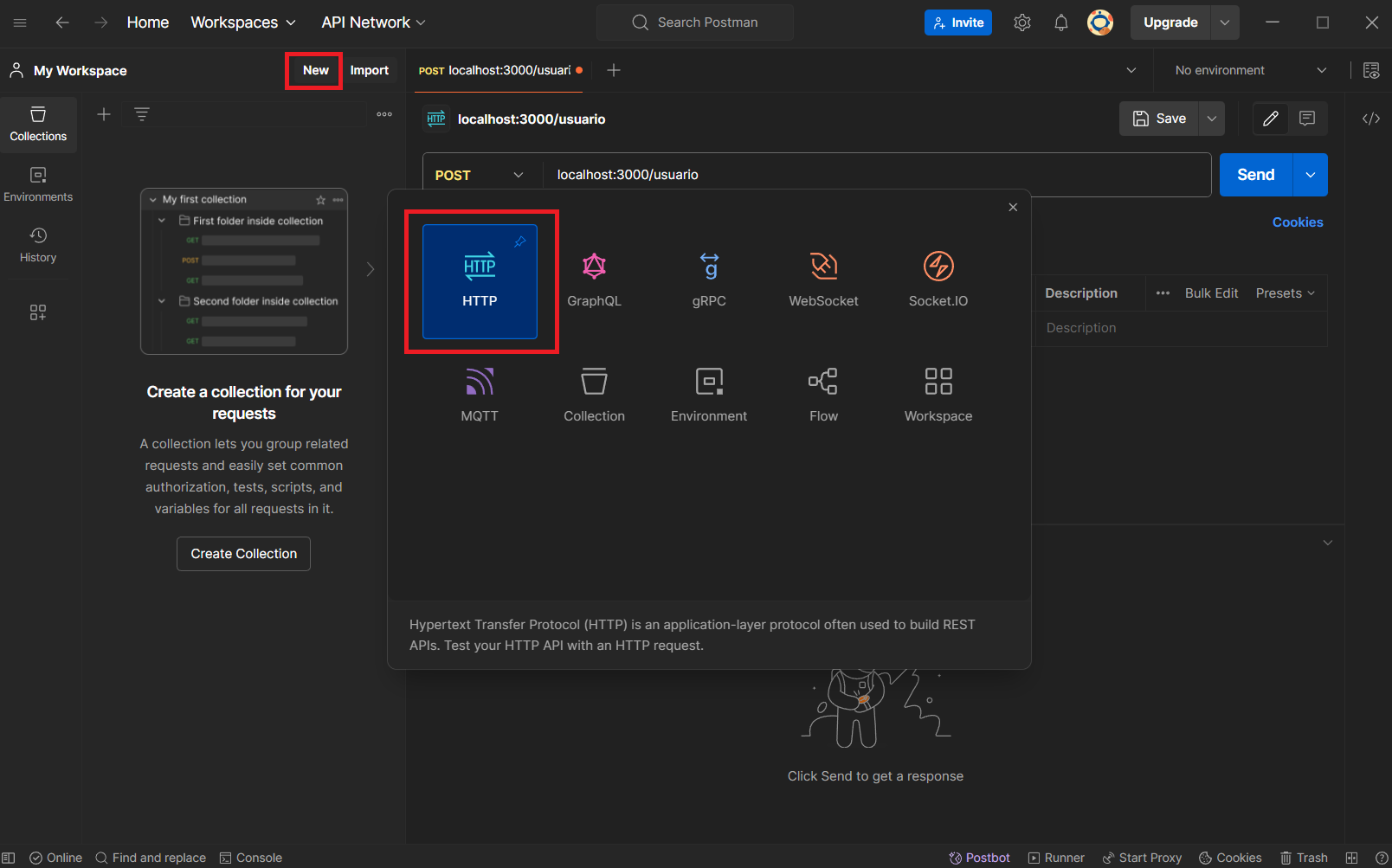


--------------------------------------------------------------------------------------------------------------------------------------------------------------

**POSTMAN – HERRAMIENTA PARA ANALISIS DE RESPUESTAS – SOLICITUDES HTML**

Hasta ahora hemos visto las peticiones HTML con GET y como se mostraban por el navegador. Bien, pues para poder trabajar de una forma practica con peticiones tipo POST, como no podemos comprobar si los datos se envían y reciben correctamente, al no mostrarse en el navegador por las propias propiedades de este método, necesitamos usar la herramienta POSTMAN.

Hay que descargarla, instalarla y crear una cuenta como cualquier programa externo normal.



--------------------------------------------------------------------------------------------------------------------------------------------------------------

**9º COMPROBAR PETICIONES TIPO ‘POST’**

Ahora vamos a hacer una petición tipo POST. Para ello en POSTMAN, indicamos:

·Que la petición es tipo POST

·La petición en sí: localhost:3000/usuario

·Indicamos que la información viajara en el body.

·Indicamos que queremos que sea tipo RAW y en formato JSON

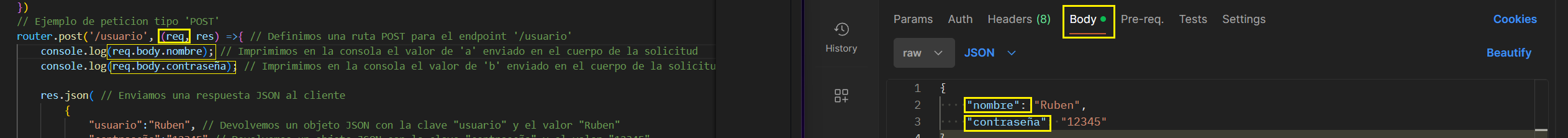
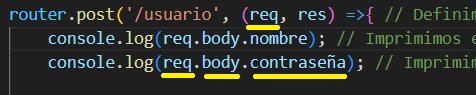
·Escribimos un JSON con el nombre y la contraseña en este caso.

Después tenemos que ir al CMD y pedir que el servidor se encuentre a la escucha con: **node /ruta/concreta**

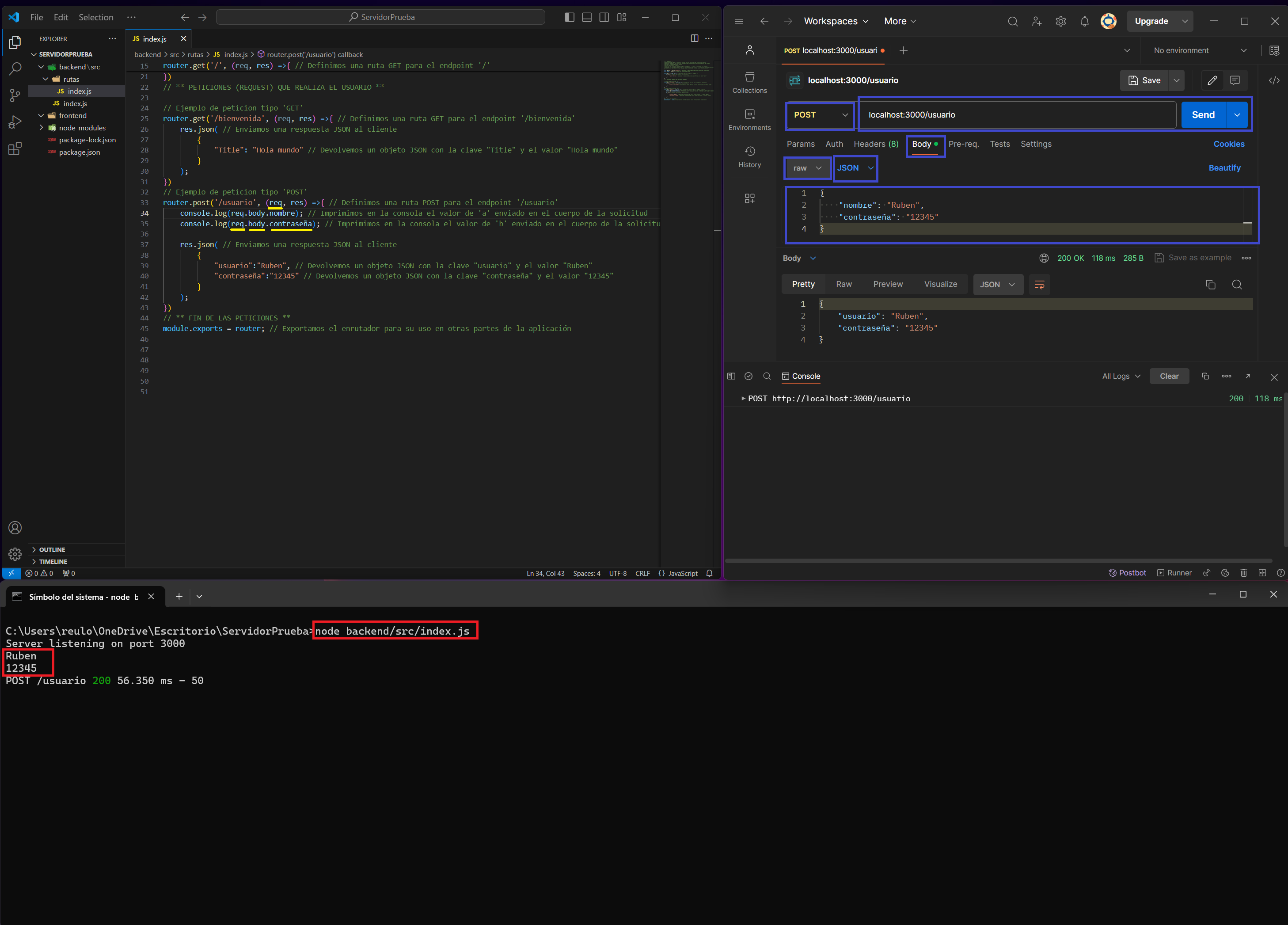
En el código de VISUAL STUDIO CODE escribiremos console.log () para mostrar por pantalla (en el CMD) lo que se reciba desde POSTMAN.

Para ello tienes que entender que la instancia router recibe una petición (req) y una respuesta (res).

Piensa en ello con una variable, tenemos que darle a req un valor, en este caso el del JSON creado en POSTMAN. Para ello tendremos que indicarle la ruta, que en este caso sería req.body.nombre y req.body.nombre.



Pulsamos enviar en POSTMAN y se ejecutara todo el proceso. Si todo es correcto, nos mostrara en el CMD la información recibida, verificando que el sistema funciona pues NODE JS recibe la información enviada desde POSTMAN.

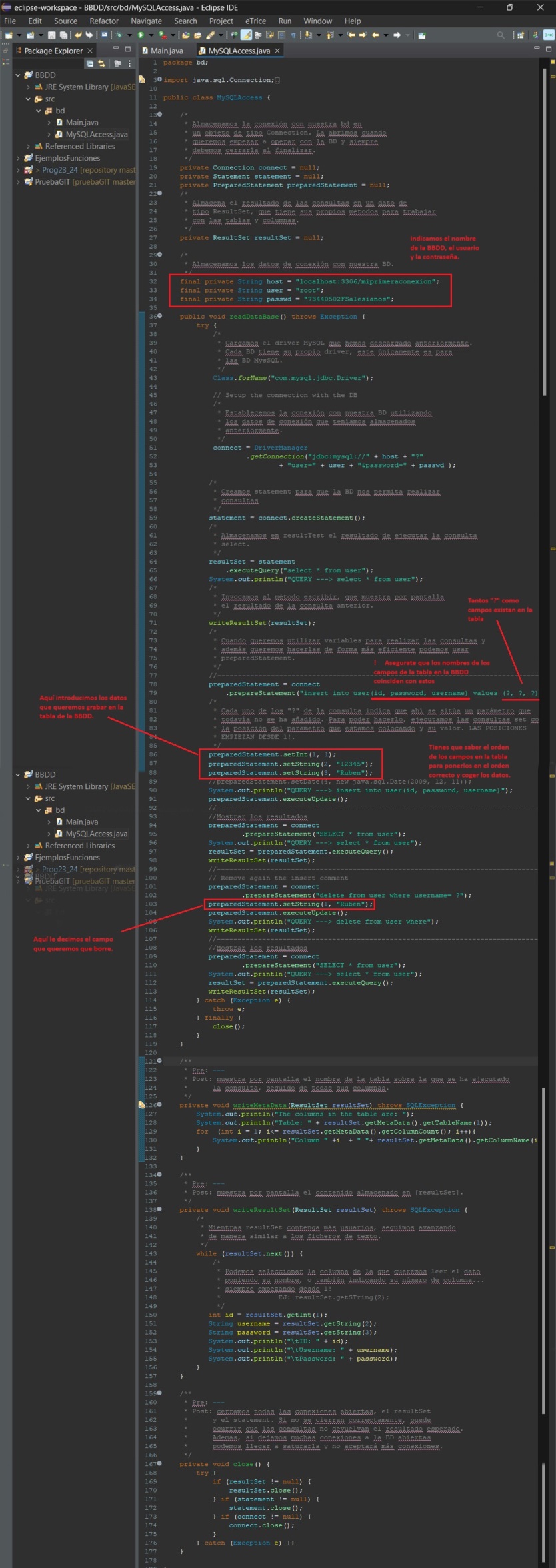


**PROCESO PASO A PASO QUE SIGUE LA PETICION TIPO POST DESDE POSTMAN**

El proceso que sigue la información desde que se escribe en Postman y se envía tipo POST, RAW, JSON hasta que se muestra por pantalla en el CMD (línea de comandos) en Visual Studio Code puede describirse de la siguiente manera:

1. **Escritura en Postman**: El usuario escribe un objeto JSON en el cuerpo de la solicitud en Postman. Este objeto JSON puede contener cualquier información que el usuario desee enviar al servidor.
2. **Envío de la solicitud**: El usuario envía la solicitud desde Postman al servidor. La solicitud se envía como una solicitud HTTP POST, con el tipo de contenido RAW y el formato JSON.
3. **Recepción en el servidor**: El servidor Node.js está escuchando en un puerto específico (en este caso, el puerto 3000). Cuando recibe la solicitud POST en la ruta "/usuario", el servidor utiliza el middleware Body Parser (que no está explícitamente mostrado en tu código, pero se asume que está configurado) para analizar el cuerpo de la solicitud JSON y hacerlo accesible a través de **req.body**.
4. **Procesamiento en el servidor**: El servidor imprime en la consola los valores de "nombre" y "contraseña" recibidos en el cuerpo de la solicitud. Luego, responde al cliente con un objeto JSON que contiene un usuario y contraseña predefinidos.
5. **Respuesta al cliente**: La respuesta JSON se envía de vuelta al cliente que hizo la solicitud (Postman en este caso). Esta respuesta contiene los datos del usuario y la contraseña predefinidos.
6. **Visualización en la consola (CMD)**: En la consola donde se ejecuta el servidor Node.js (Visual Studio Code), se muestra un registro indicando que se recibió una solicitud POST en la ruta "/usuario", seguido de los valores de "nombre" y "contraseña" impresos en la consola. Finalmente, se muestra un registro de que se envió una respuesta con el código de estado 200 y el tiempo que tomó procesar la solicitud.
7. **Finalización del servidor**: El servidor sigue ejecutándose y escuchando nuevas solicitudes hasta que el usuario lo detiene manualmente. En este caso, el usuario presionó **Ctrl + C** en la consola, lo que detuvo la ejecución del servidor y mostró el prompt de comando nuevamente.

En resumen, este proceso muestra cómo se recibe, procesa y responde a una solicitud POST en un servidor Node.js utilizando Express, así como la visualización de registros en la consola (CMD) durante este proceso.

**CONECTAR JAVA (ECLIPSE) CON NUESTRA BBDD (MY SQL)**

**PASO PREVIO – INSTALAR EL DRIVER**

Lo primer para iniciar la conexión a una BBDD necesitamos instalar el driver correspondiente.

En este caso, descargamos el JDBC Driver, esto es un archivo tipo .jar necesario para el correcto funcionamiento que tenemos que tener.

El .jar lo importamos en un sitio fijo, no se puede borrar bajo ningún concepto, sino fastidiaremos la conexión.

Nota: Este paso solo es necesario hacerlo una vez.

En mi caso, el archivo se encuentra aquí: C:\Users\reulo\eclipse-workspace\BBDD\archivos

C:\Users\reulo\eclipse-workspace

Pasos para importar librería .jar:

* Botón derecho sobre el proyecto Java y seleccionamos "Properties".
* En el menú de la izquierda seleccionamos "Java Build Path".
* Dentro seleccionamos "Libraries".
* Elegimos la opción Modulepath y le damos al botón "Add External Jars".
* Buscamos los .jar y los añadimos.

Con esto ya tenemos el driver instalado en Eclipse.

**ENLAZAR JAVA A MYSQL**

Alvaro nos ha pasado el codigo básico para enlazar Java con la BBDD que tengamos en MySQL para usarlo como plantilla.

**ENTENDIENDO LA CONEXIÓN – GRABAR Y BORRAR DATOS DESDE JAVA EN MY SQL**

Se crean 2 instancias de dos objetos diferentes:

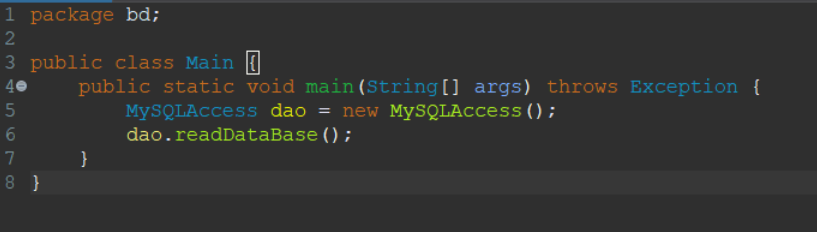
- connect representa la conexión establecida entre la aplicación Java y la base de datos MySQL, permite enviar o recibir datos (consultas).

-resultSet almacena los datos obtenidos.

Tenemos 3 atributos: host, user y password en los que tendremos que indicar que el nombre y puerto de la BBDD a la que queramos acceder, el usuario (root en mi caso) y la contraseña.

**Podemos grabar y borrar información desde Java en la BBDD si conocemos el nombre de las tablas y columnas y su posición en la tabla.**

**Advertencia**, después de hacer cambios recuerda hacer un COMMIT manualmente en la BBDD porque si no no veras los cambios y te volverás loco.

****

**CONECTAR NODE JS (Visual Studio Code) CON NUESTRA BBDD (My SQL) – PHASE II**

**1º INSTALAR LOS MODULOS MY SQL Y EL MODULO DOTENV EN NODE JS**

Abrimos la carpeta backend de nuestro proyecto en el CMD. Después usamos el siguiente comando: **npm i mysql dotenv --save-dev**

Esto instalara dos cosas:

Nota: El dot de dotenv significa .env porque DOT en ingles es punto .

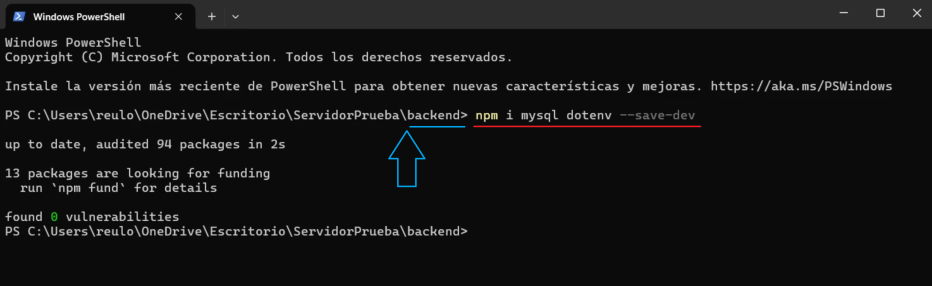
Explicacion del comando:

-mysql 🡪 Este paquete proporciona una forma de conectarse a una base de datos MySQL desde una aplicación de Node.js. Facilita la ejecución de consultas SQL, la gestión de conexiones y otras operaciones relacionadas con la base de datos MySQL. Después de instalar este paquete, puedes utilizarlo en tu aplicación de Node.js para interactuar con tu base de datos MySQL.

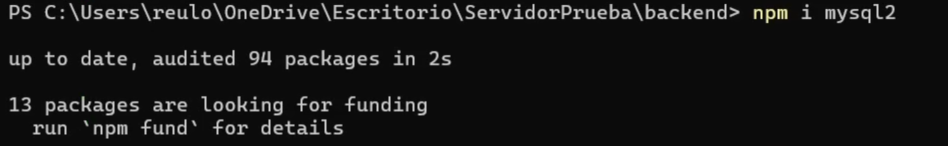
-dotenv: 🡪 Este paquete ayuda a cargar variables de entorno desde un archivo .env en tu proyecto de Node.js. Es común utilizar variables de entorno para almacenar información sensible como credenciales de base de datos, tokens de acceso, etc.

El paquete dotenv permite cargar estas variables de entorno de forma segura en tu aplicación desde un archivo .env. Esto facilita la gestión de configuraciones sensibles y ayuda a mantener separadas las variables de entorno del código fuente.

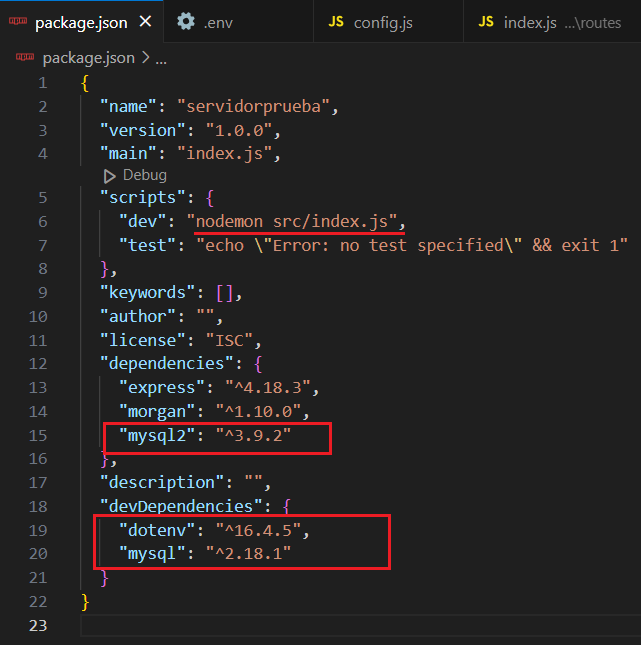
--save-dev: Esta opción le indica a npm que debe agregar estos paquetes como dependencias de desarrollo (devDependencies) en el archivo package.json de tu proyecto. Las dependencias de desarrollo son paquetes que se utilizan durante el desarrollo de la aplicación, pero no son necesarios para su funcionamiento en producción. En este caso, dotenv podría ser considerado una dependencia de desarrollo, mientras que mysql probablemente sería una dependencia regular utilizada tanto en desarrollo como en producción.



Nota: ASEGURATE de que la instalación de los paquetes se produce en la carpeta backend, es decir, POSICIONATE en esa carpeta en la terminal y aplica el paquete o lo instalaras en otro lugar y no funcionara. Tal y como en la imagen debes hacerlo.

****

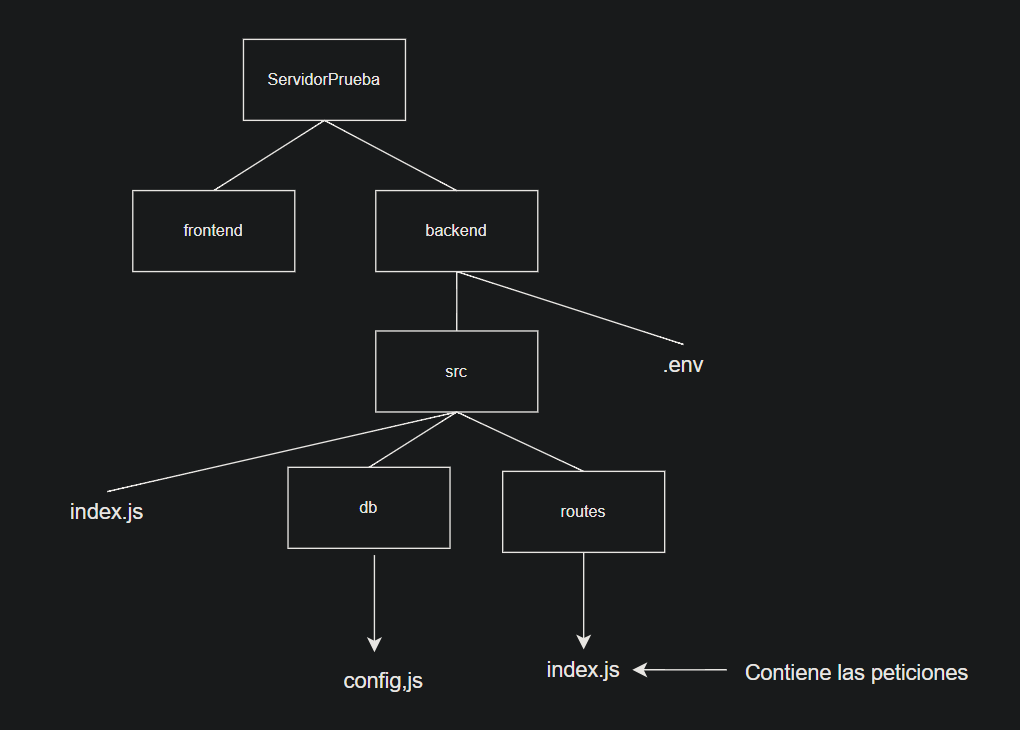
Haz lo mismo ahora con otro paquete con este comando: **npm i mysql2**

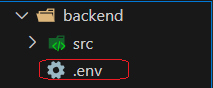


Si en Visual Studio Code nos vamos al archivo package.json podemos ver que se han instalado las dependencias correctamente, en este caso: mysql, dotenv y mysql2.

Podemos ver también la versión que se ha instalado.

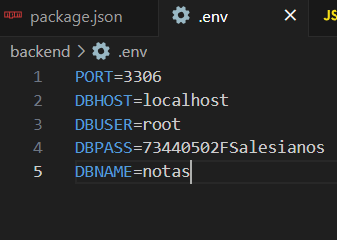
**ESTRUCTURA DEL PROYECTO**

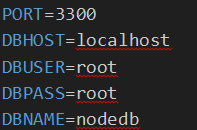
****

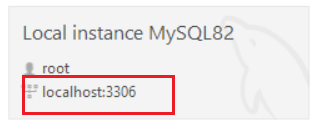


**2º CREAR .env Y SU CODIGO**

**Tenemos que crear a la altura de la carpeta src el archivo .env.**

**Y escribir el siguiente código.**

**Este es el código por defecto, en tu caso deberás escribir el puerto de tu BBDD que puedes verlo en MySQL, el usuario y contraseña y el nombre de la BBDD que vayas a emplear.**

 **A la derecha tienes un ejemplo 🡪**

**COPIAR CODIGO**

PORT=3306

DBHOST=localhost

DBUSER=Ruben

DBPASS=73440502FSalesianos

DBNAME=servidorFP

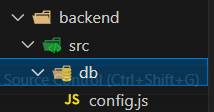
El paquete **dotenv** que instalamos antes nos permite crear ficheros ocultos que son los que comienzan por un punto . es decir todo fichero cuyo nombre comienzo por . automáticamente se vuelve oculto y solamente la persona que tenga privilegios para abrirlos y verlos.

En Java anteriormente escribíamos sobre el código tanto el valor de usuario como la propia contraseña y por privacidad eso NUNCA de debe hacer porque resulta vulnerable, en cambio, lo que se hace es guardar esa información en ficheros ocultos dentro del proyecto que solo el creador o aquellos a los que designe con privilegios puedan verlos y acceder a ellos. De esta forma solamente pueden llegar a ver que estás trabajando con un variable que contiene la contraseña o el usuario pero no pueden ver su contenido. Si alguien nos robase el código nunca podrían acceder al valor de la variable y conocer la información porque tendría el código compilado.

Es muy útil porque si necesitas acceder a esta informacion, en vez de ponerlo cada vez, con referenciar este archivo que contiene los datos ya lo tienes.

Pues bien eso es lo que hemos creado, un fichero oculto que almacene los datos sensibles del puerto, usuario, contraseña etc.

**3º CREAR CARPETA db Y config.js**



En el fichero config ira la configuración que hicimos en Java el otro dia. Ponemos el siguiente código:

Esto permite conectar con nuestra BBDD.

//Exporto el objeto que me permite entrar a las variables ocultas.

const dotenv = require ("dotenv");

//Reojo ese fichero exportado.

dotenv.config();

//Exporto el mysql

const mysql = require ('mysql2');

//Creamos una variable conexion que de momento esta vacia.

let connection;

try{

    //Creamos una variable connection que valdra los valores ocultos del archivo que hemos configurado previamente .env.

    connection = mysql.createConnection({

        host:process.env.DBHOST,

        user: process.env.DBUSER,

        password: process.env.DBPASS,

        database:process.env.DBNAME

    });

} catch (error){

    console.log("Error al conectar con la base de datos");

}

module.exports = {connection};//Exporta la conexion para que sea accesible desde otros ficheros del proyecto.

**4º LA CRUZ: INSERTAR, LISTAR (VER LO QUE HAY), ELIMINAR Y ACTUALIZAR (CAMBIAR ALGO)**

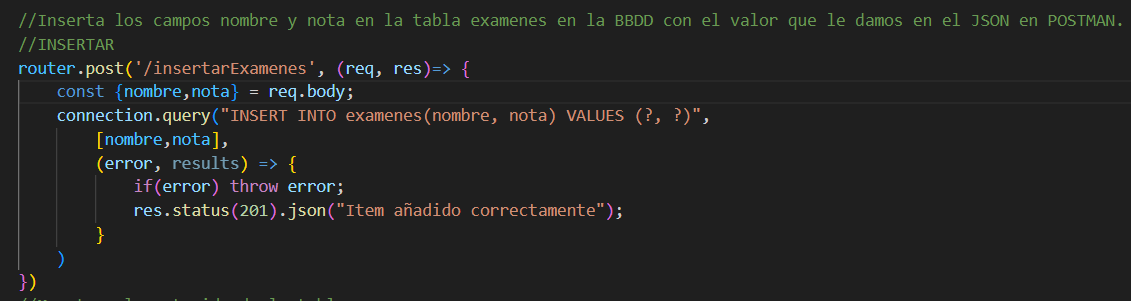
Un cruz es un programa informatico que permite:insertar, listar, eliminar y editar algo, se usa como herramienta para dotar de funcionalidad.

**NOTA:**  Siempre que vayamos a enviar algún tipo de dato lo hacemos de tipo POST y para verificarlo deberemos de usar POSTMAN.

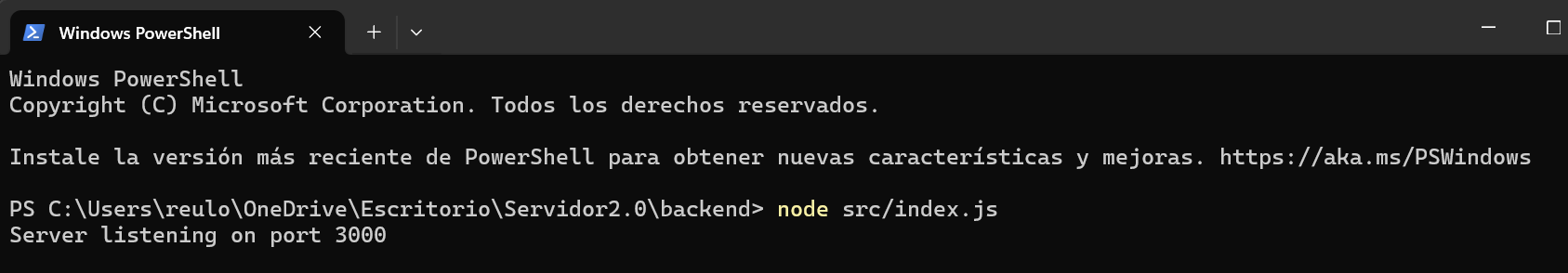
**FUNCIONAMIENTO DE LA RUTA S**

**Nota**: Recuerda que tenemos que usar el comando: **node backend/src/index.js** para aponer el servidor a la escucha en el puerto 3000 y en el navegador poder la ruta de petición que queramos probar. Recuerda tras cada petición hacer ctrl + c para salir y generar otra nueva.

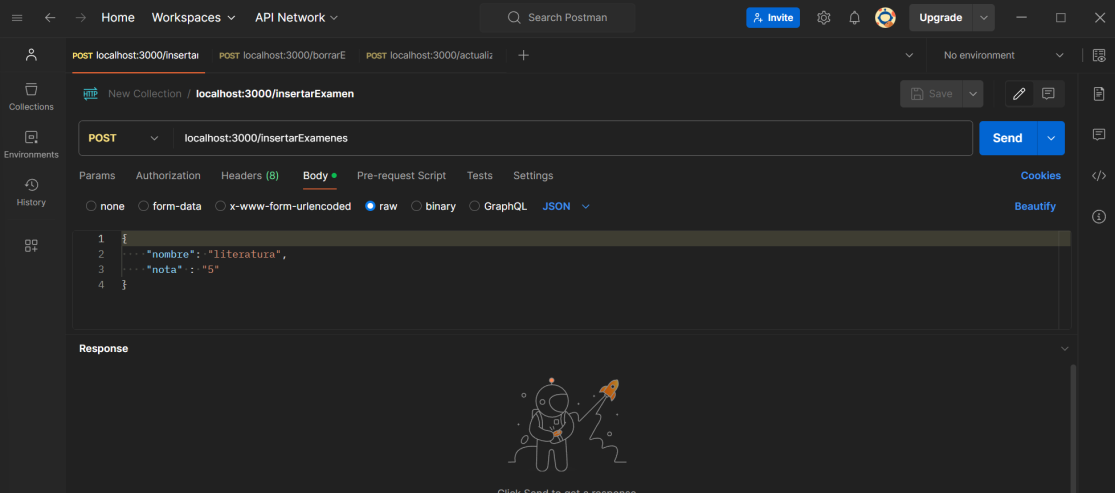
**EJECUCION DE PASOS: ESCRIBES EL CODIGO 🡪 PONER A LA ESCUCHA 🡪 EJECUTAR POSTMAN 🡪 CTRL + C**

****

**1º**

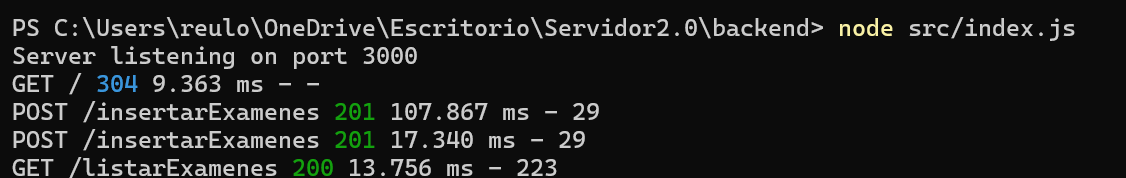
****

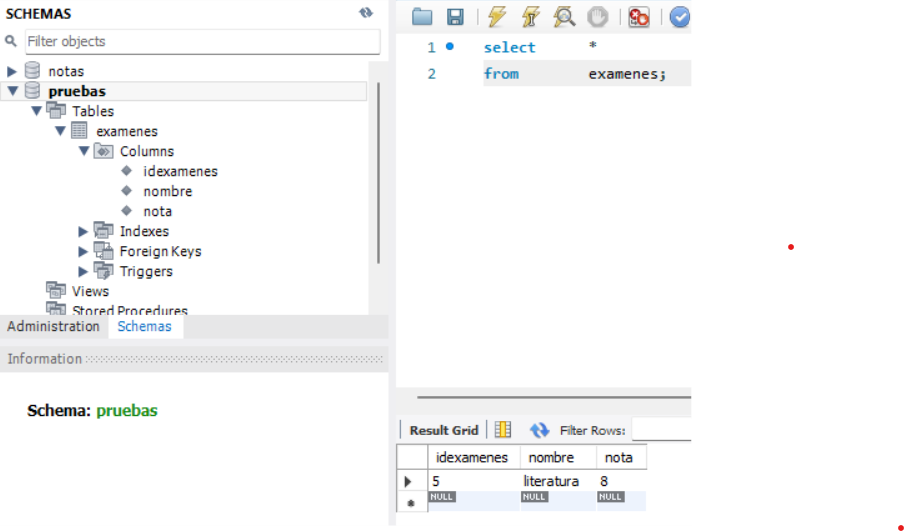
**2º**



**3º**

**ATENCION!** Recuerda tras ejecutar POSTMAN que si no realizas un commit NO veras reflejado el cambio en la BBDD lo que te puede llevar a pensar que no se ha hecho correctamente. Fijate tambien en los datos que te chiva la terminal, si muestra estados 200 es que se ha ejecutado correctamente





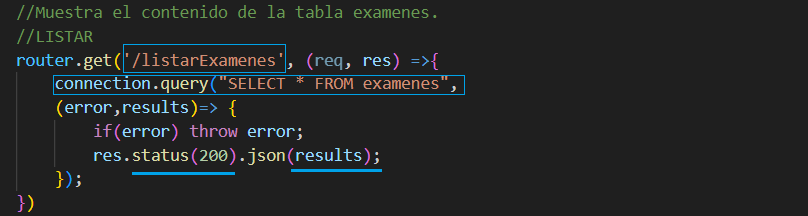
Para estos ejemplos tenemos una BBDD llamada pruebas y contiene una tabla examenes con los atributos:

· idexamenes

·nombre

.nota

**LISTAR**



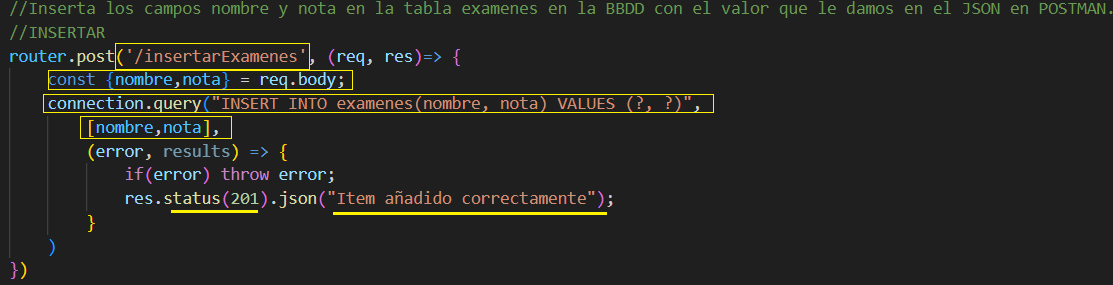
Tenemos la función list que se encarga de mostrar todas el contenido de la tabla examenes de la BBDD. Como no enviamos datos sino que los recogemos, usamos un tipo GET.

· La ruta va a ser /listarExamenes que es lo que deberemos poner en host:3000/listarExamenes para que el servidor nos devuelva esta petición.

· Contiene una petición (request) (req) y una respuesta (response)(res).

· Vemos que connection va a valer lo que una consulta (query) en SQL que es lo que queremos obtener de la BBDD.

· Por ultimo si la consulta se genera correctamente y se obtienen resultados (results) esto se transforma a un json y es lo que contiene res (response), marcando que se ha validado correctamente al servidor con el código 200.

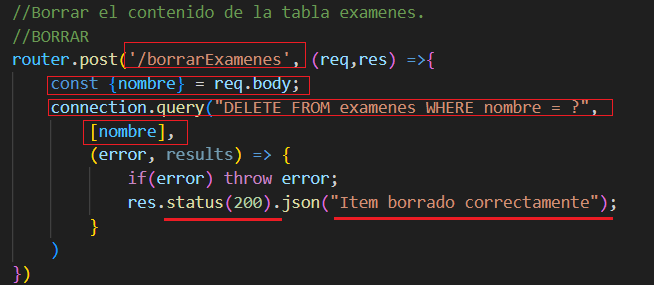
**INSERTAR**

La función insert que permite agregar un nuevo registro de examen a la base de datos MySQL cuando se recibe una solicitud POST en la URL '/insertarExamenes', utilizando los datos proporcionados en el cuerpo de la solicitud desde, en este caso de prueba, POSTMAN y en la vida real por el usuario que realice la petición.

En este caso se crean dos variables que van a valer lo que se escriba en el JSON dentro del body en POSTMAN y que es lo que se va a grabar en la consulta de inserción que va a contener connection.

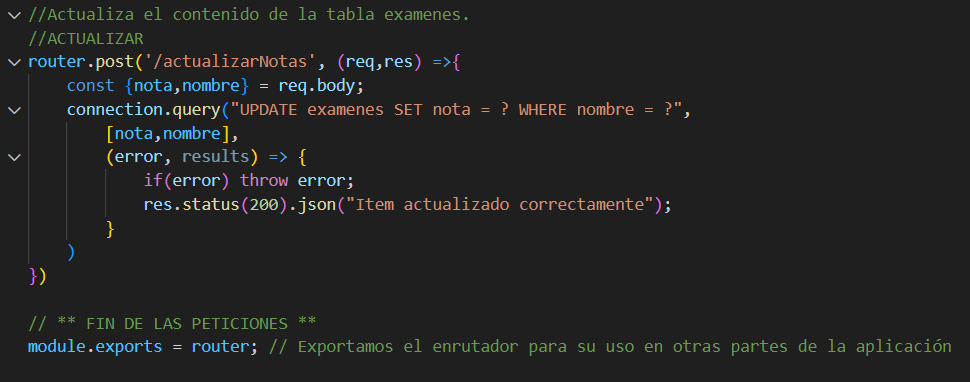
Si no da error la respuesta que devuelva al servidor sera la de que se a añadido correctamente para avisar de que se ha completado con éxito, en formato JSON y marcando al servidor un estado de éxito como es el 201.

**BORRAR**

La función borrar elimina el registro de la BBDD por el atributo que le indiques, en este caso el nombre y usarmos ‘/borrarExamenes’. El funcionamiento es el mismo, desde POSTMAN escribimos en el body un JSON que los que va a valer nombre en el código y connecition pasar a valer lo que la petición incluyendo la variable que queremos que use que será el nombre del examen escrito en POSTMAN.

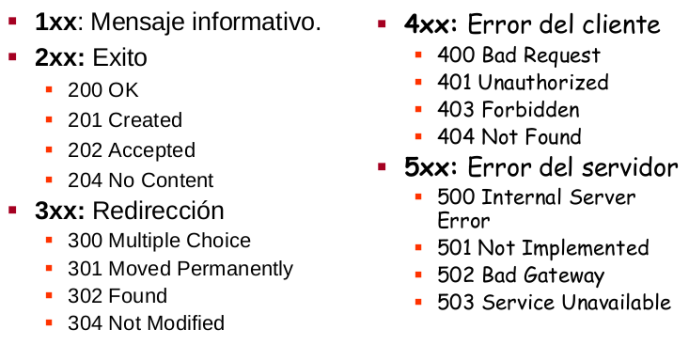
Despues mostrara un mensaje de que se ha borraro correcatamente y devolverá un estado 200 al servidor si es que ha tenido éxito.

**ACTUALIZAR**



Una vez más, con ‘/actualizarNotas’ editamos un registro ya escrito en la BBDD. Desde postman escribimos el registro a modificar y su modificación y el código siguie el mismo proceso anterioremente descrito.

**CODIGOS DE ESTADO DE RESPUESTA HTTP**

**· Código de estado 200 (OK): Este código se utiliza para indicar que la solicitud ha sido exitosa. Es el código de estado estándar para respuestas exitosas a solicitudes GET, POST, PUT, DELETE, etc. Indica que la solicitud se procesó correctamente y que el servidor está devolviendo el contenido solicitado.**

**· Código de estado 201 (Created): Este código se utiliza específicamente para indicar que una nueva entidad ha sido creada. Se utiliza comúnmente en respuestas a solicitudes POST para indicar que se ha creado exitosamente un nuevo recurso en el servidor. La respuesta generalmente incluirá una URL a la nueva entidad creada.**

ELANZAR DOS TABLAS DE SQL EN JAVA

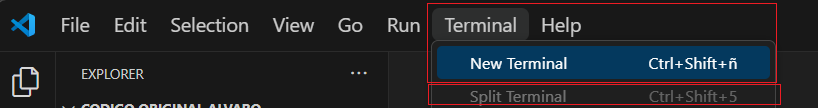
En Java, cuando para relacionar una asignatura a un alumno, el idalumno en SQL iria en una tabla de asignatura pero aquí tendrías un objeto alumno y otro asignatura, bueno pues en el atributo idalumno de asignatura iria un objeto alumno ruben por ejemplo, en vez de un idalumno. De esa forma es como se relacionan los datos.

**FRONTEND – CONOCIENDO REACT - PHASE III**

**INSTALAR REACT EN LA CARPETA FRONTEND**

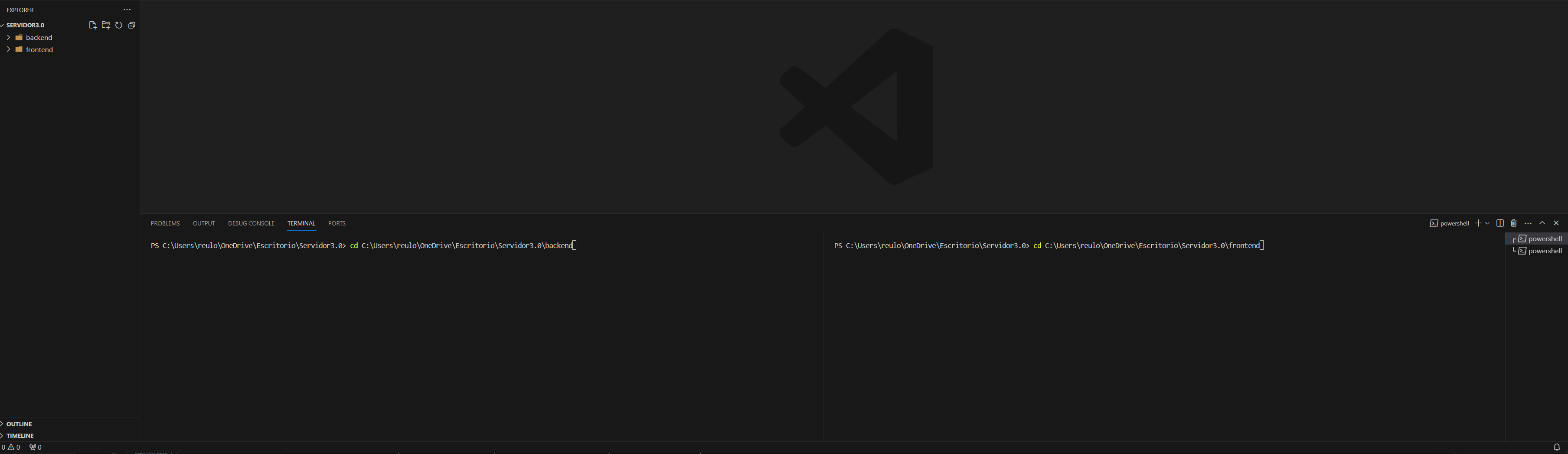
Para poder trabajar cómodamente vamos a usar dos terminales de Visual Studio Code. Vamos a trabajar con ambas separadas y localizadas, la de la izquierda en la carpeta backend y la de la derecha ubicada en la carpeta del frontend.

Creamos nueva terminal y después de decimos separar terminal, esto nos creara dos.



Copiamos las rutas de las carpetas en donde se encuentran ubicadas, en la terminal de la izquierda la del backend y en la derecha la de el frontend.

**Nota:** Recuerda usar el comando cd y añadir la ruta para ir a ese directorio



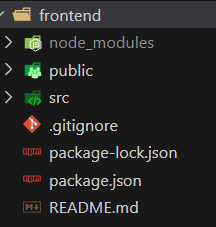


Ahora en la terminal de la izquierda, ubicados dentro de la carpeta frontend, usamos el siguiente comando: **npm init react-app .**

El comando **npm init react-app .** se utiliza para inicializar un nuevo proyecto de React utilizando Create React App, que es una herramienta oficialmente soportada por el equipo de React para generar rápidamente aplicaciones de una sola página (SPA).

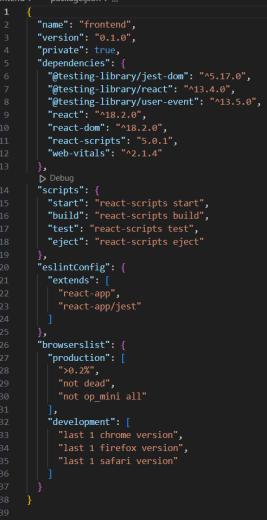
Desglosemos el comando:

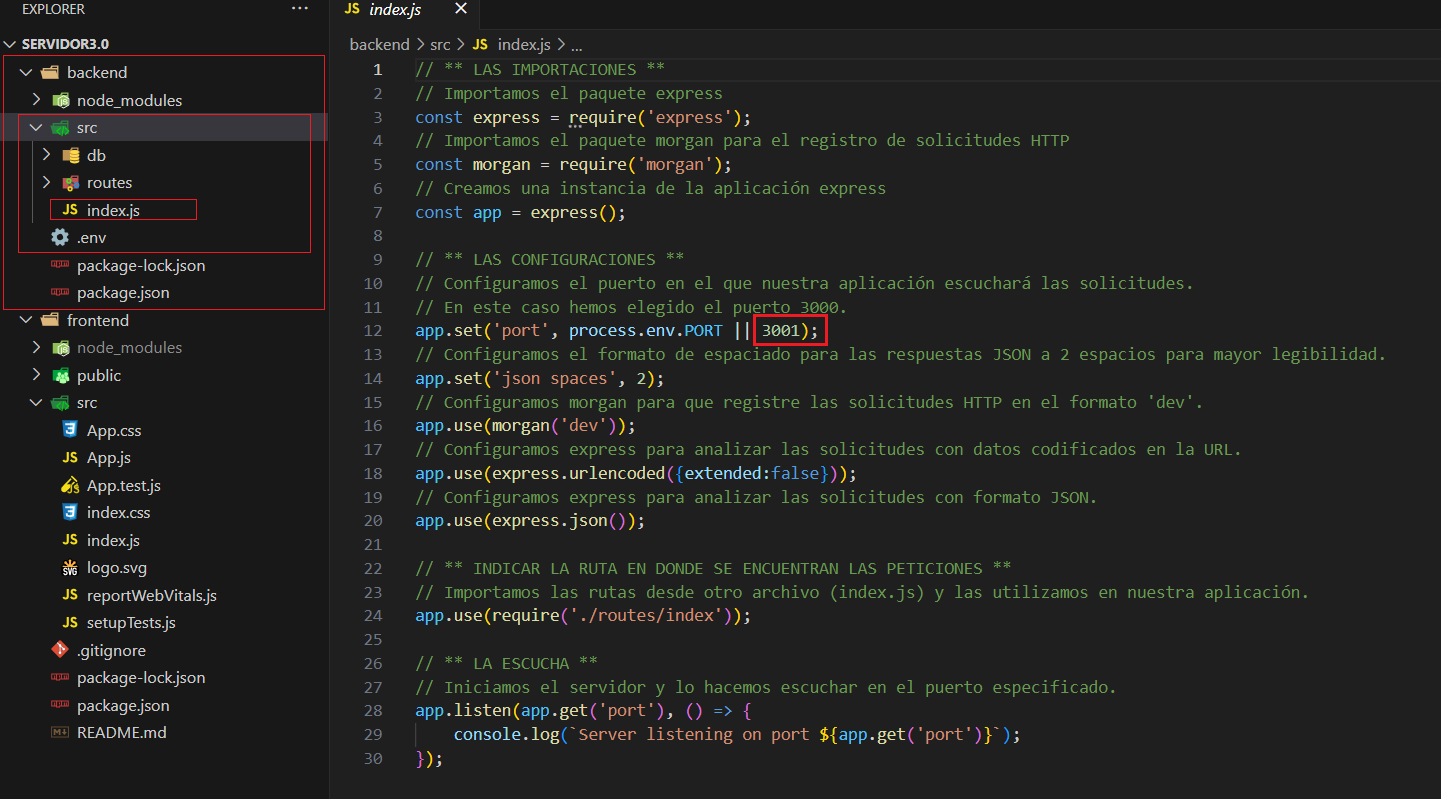
* **npm init**: Esto inicia el proceso de inicialización de un nuevo proyecto npm. **npm** es el administrador de paquetes de Node.js, y **init** es el comando para inicializar un nuevo proyecto.
* **react-app**: Esta es la plantilla o el paquete que deseamos usar para iniciar nuestro proyecto. En este caso, estamos utilizando **react-app**, que es la plantilla oficial para crear aplicaciones de React.
* **.**: El punto al final del comando indica que queremos inicializar el proyecto en el directorio actual. Esto significa que el proyecto se creará en el directorio en el que estamos ejecutando el comando.



Una vez nos aparezca el mensaje “Happy Hacking” habremos concluido la instalación y dentro de la carpeta frontend veremos todos estos archivos instalados.

**MODIFICAR EL PUERTO DEL SERVIDOR – DEL 3000 AL 3001**

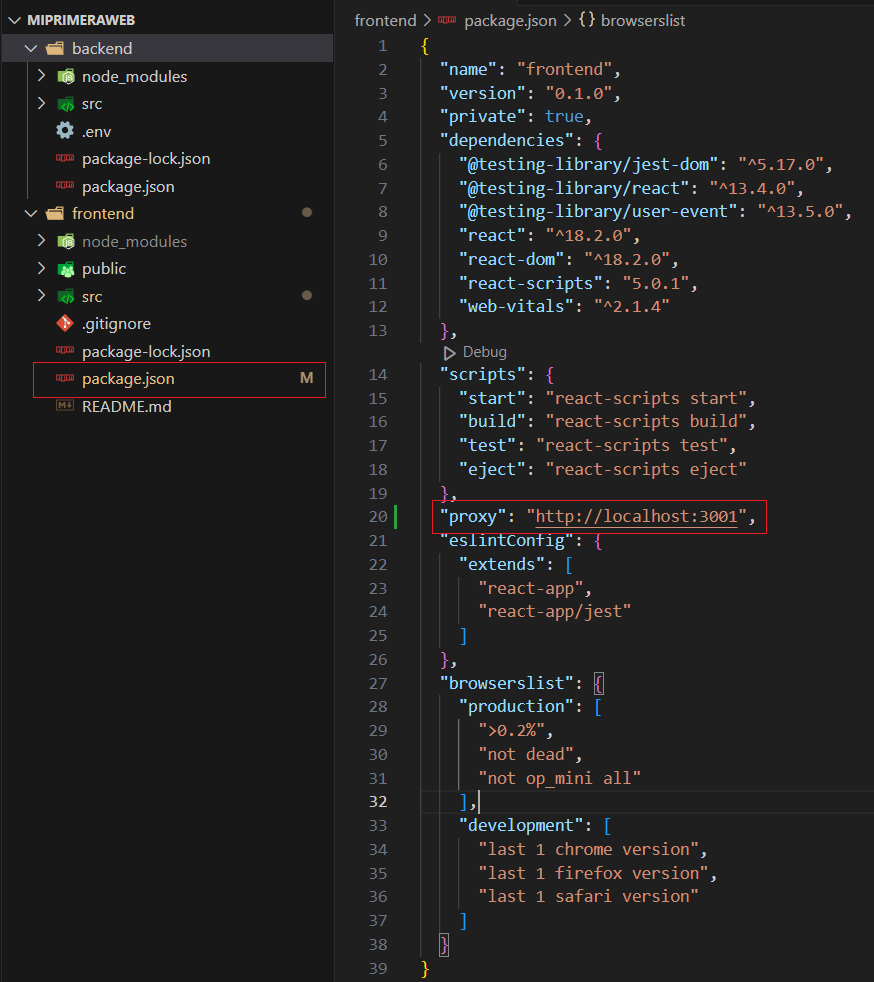
Si abrimos el package.json podemos ver como efectivamente tenemos módulos instalados de react y sus versiones.



El servidor como hemos visto en pasos anteriores funciona sobre el puerto 3000 que configuramos anteriormente.

React, al igual que el servidor también funciona en un puerto que, por defecto, resulta que es el 3000 también.

Pero, este puerto ya lo tenemos ocupado por el servidor.



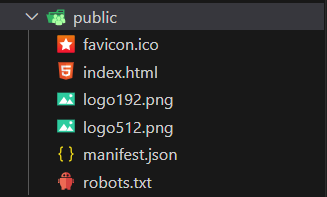
Por lo que vamos a ir a la carpeta de backend 🡪 src 🡪 index.js   
Aquí vamos a cambiar el puerto y en vez de 3000 vamos a decir que funcione en el 3001.

Ahora vamos a ir en frontend 🡪 package.json

Y ahí vamos a añadir: **“proxy”: ”http://localhost:3001”,**

Aquí estamos indicando que las solicitudes realizadas desde el frontend (React) se redirigirán automáticamente a un servidor backend que se ejecuta en el puerto que le indiquemos, en este caso el 3001.

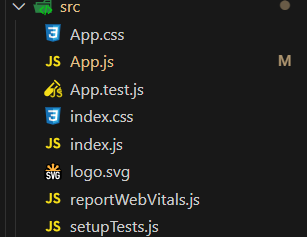
Es por eso que como el backend funciona en el puerto 3000 tendrías que asegurarte de que tu servidor backend esté escuchando en http://localhost:3001.

En la carpeta public, por defecto de la instalación, React te muestra sus propias fotos

Nota: Todas las páginas webs tienen el archivo robots.txt porque Google así lo decidió, dentro de este fichero aparece si tú quieres o no salir en google, si quieres esconder de google ciertos ficheros o que no te hagan webscrapping o poner palabras clave de tu página web.

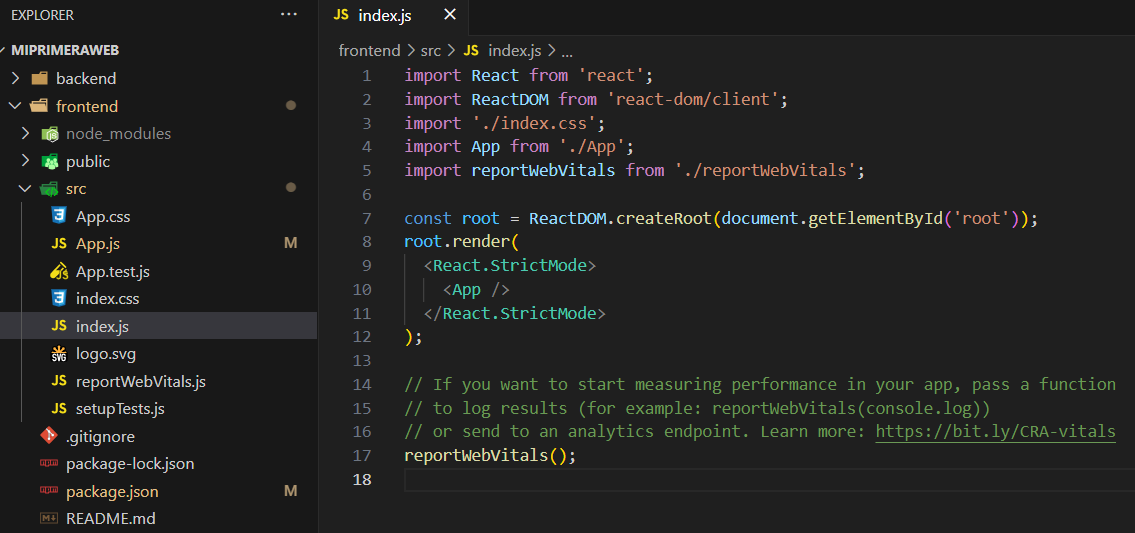
Lo que hace Google es leer este fichero para saber que tiene o que no tiene que hacer en función de lo que tu escribas aquí. No lo vamos a usar pero que lo sepamos.

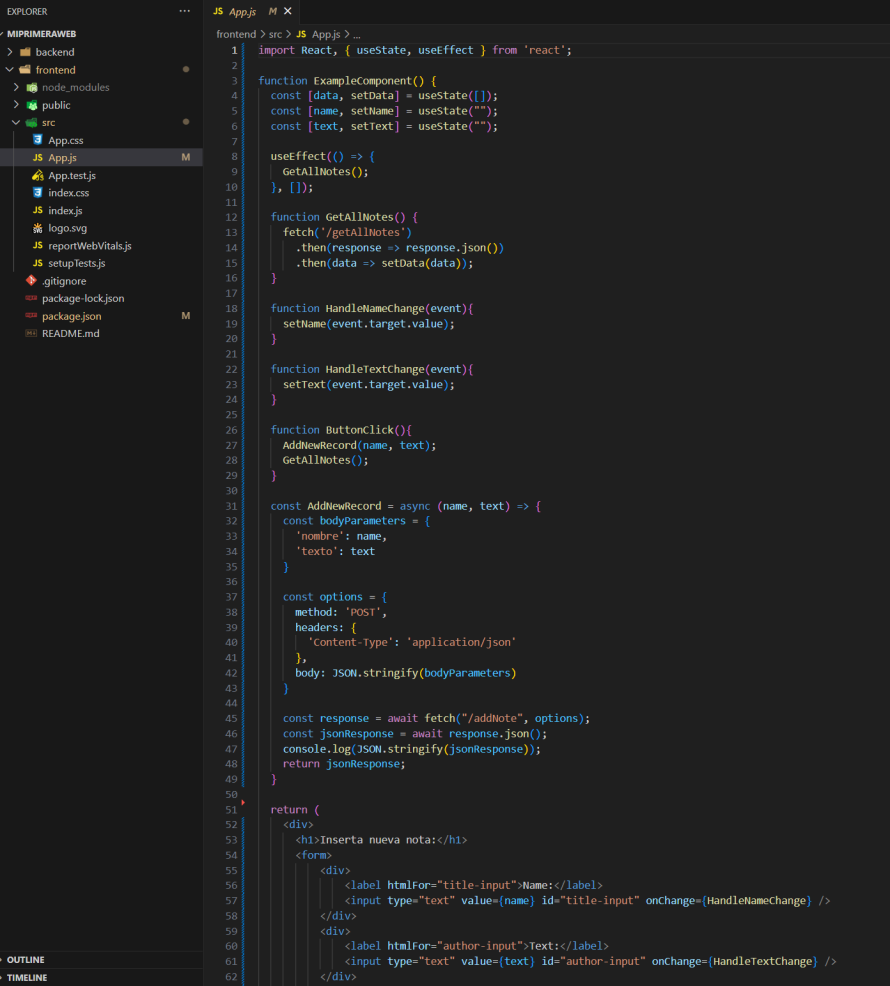
Webscrapping: Programas que leen automáticamente los datos de tu web y se quedan con toda la información para luego ofrecerla a terceros o guardarla.



En la carpeta de src por defecto tenemos los archivos que necesita React para funcionar pero en general solo nos interesa el index.js

Este fichero es el encargado de ejecutar React, no hay que modificar su código bajo ningún concepto, pero que sepamos para que sirve este index.





App.js es el fichero que va a cargar index.js cuando tú le das al play se ejecuta este código que automáticamente ejecuta app.js

Ahora en el terminal de la izquierda en el frontend, vuélvete a ubicar en tu ruta de frontend con el comando cd

Y vamos a usar este comando:  **npm start**

**NOTA:** Si te da error, revisa que en el package.json tiene bien el localhost con <http://localhost:xxxx> o que te encuentres ubicado dentro de la carpeta de frontend en tu proyecto antes de hacer el comando.

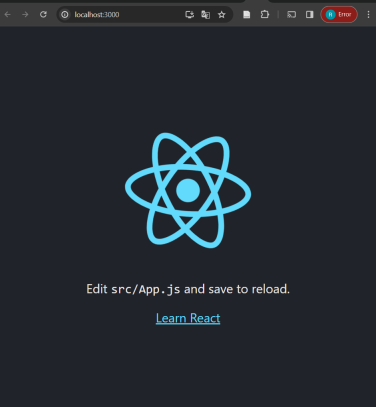
Si el comando react-scripts no se reconoce como un comando interno o externo en tu terminal. Esto sugiere que npm no puede encontrar el paquete react-scripts instalado globalmente o localmente en tu proyecto.

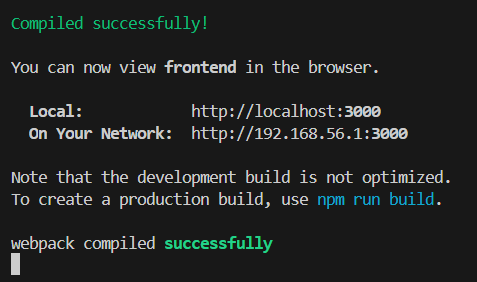
Para resolver este problema, hay algunas cosas que podrías intentar:

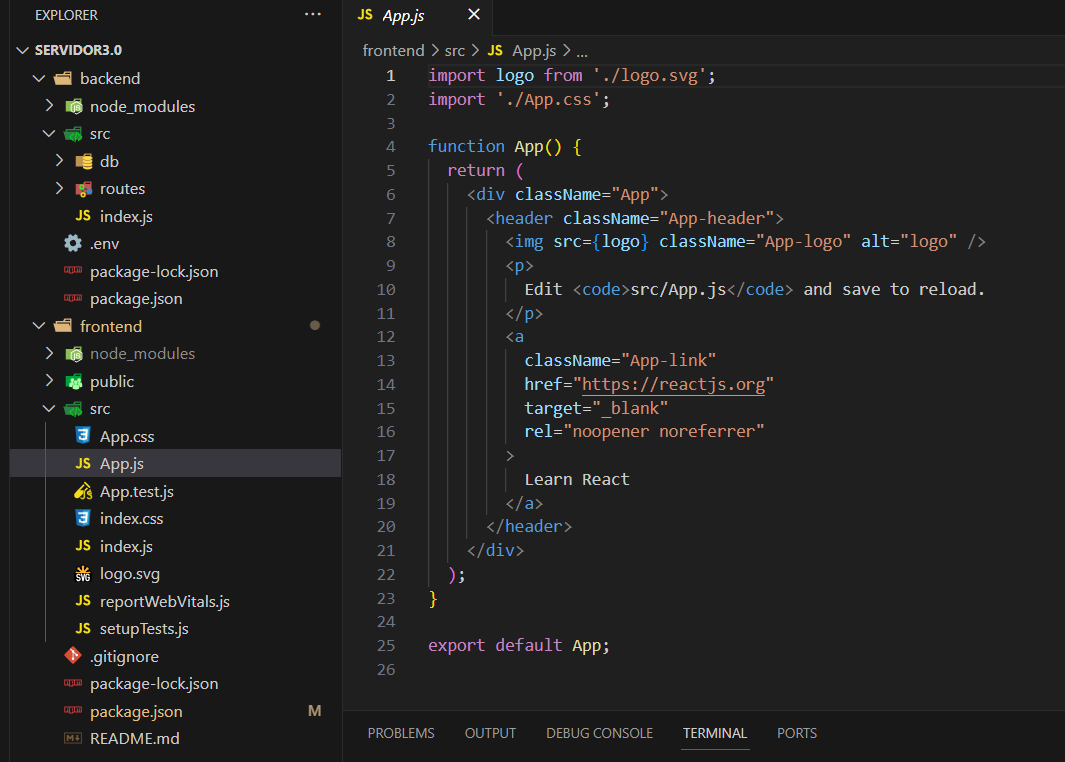
1. Asegúrate de que react-scripts esté instalado en tu proyecto: Ejecuta npm install react-scripts en el directorio de tu proyecto frontend para instalar react-scripts localmente. Esto asegurará que el paquete esté disponible para tu proyecto.
2. Verifica que no haya errores durante la instalación: Si ejecutas npm install react-scripts, asegúrate de revisar la salida para ver si hay algún error durante la instalación. A veces, los problemas de conexión a internet o la configuración del entorno pueden causar problemas durante la instalación de paquetes.
3. Verifica que npm esté configurado correctamente: Asegúrate de que npm esté configurado correctamente en tu sistema y que estés utilizando la versión correcta. Puedes intentar actualizar npm ejecutando npm install -g npm@latest para asegurarte de tener la última versión.
4. Comprueba el archivo package-lock.json: Si tienes un archivo package-lock.json, intenta eliminarlo y luego ejecuta npm install nuevamente para volver a generar el archivo. A veces, los problemas en este archivo pueden causar problemas con la instalación de paquetes.

Una vez que hayas realizado estos pasos, intenta ejecutar npm start nuevamente y verifica si el problema persiste. Si el problema continúa, revisa los mensajes de error para obtener más detalles y sigue investigando en función de eso.

Si compila correctamente veras este mensaje y en el navegador en el localhost:3000 verás la animación predefinida por React.







Lo que vemos que se ejecuta en nuestro navegador, es el código predefinido de App.js

Vale, pues ahora vamos a sustituir este código predefinido por este otro, lo borramos todo y ponemos esto encuenta:

import React, { useState, useEffect } from 'react';  
  
function ExampleComponent() {  
const [data, setData] = useState([]);  
  
useEffect(() => {  
fetch('/getAllNotes')  
.then(response => response.json())  
.then(data => setData(data));  
}, []);  
  
return (  
<div>  
{data.map(item => (  
<div key={item.id}>  
<h2>{item.nombre}</h2>  
<p>{item.texto}</p>  
</div>  
))}  
</div>  
);  
}

**¿COMO FUNCIONA REACT? ENTENDIENDO LOS COMPONENTES**

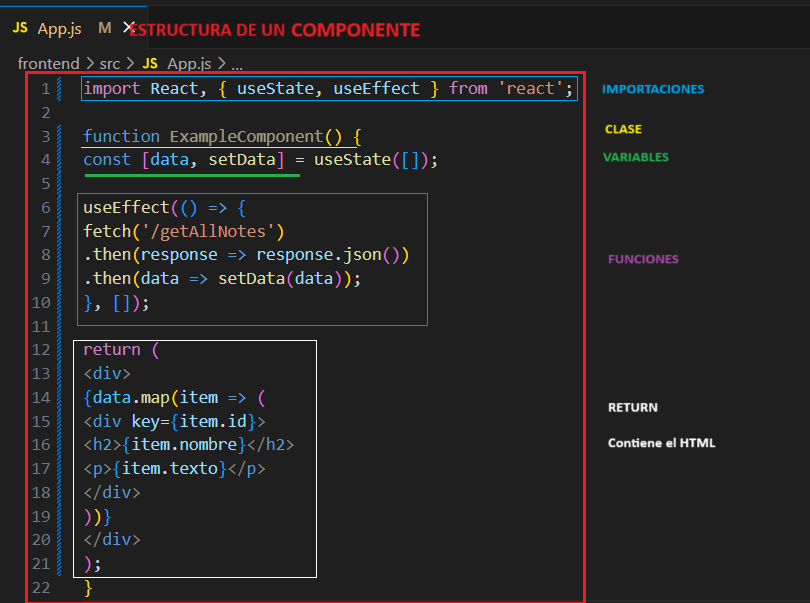
Tienes que entender que, React no se programa por pantallas como en HTML, se denominan componentes y una pantalla en React puede estar formada por tantos componentes como se quiera.

Por ejemplo podemos decir que tenemos un header, un body y un footer, bien, pues en React cada uno de estos elementos es un componente y podemos pedirle que los cargue donde queramos indicándole el orden.

Esto quiere decir que, si se registra un cambio, en el body por ejemplo, solo se refresca el body, el header y el footer no se refrescan porque no tienen cambios y no ocurre nada con ellos.

Lo que permite es modularizar partes de una página, cosa que en HTML no pasa, resfrescas todo el contenido de una página.

Esta es la estructura de un componente desglosado, es SIEMPRE IGUAL por lo que copiamos y pegamos el código y modificamos lo que nos interese cambiar.



Aquí usamos las **variables de estado**, se crean de la siguiente forma:

-Ponemos: const [ ]

-Ponemos el nombre de la variable: data

-Creamos un setter como en Java: setData

-Después le damos un valor inicial dentro de useState( );

La variable data va a contener en este caso una tabla useState ( [ ] ) con valores vacíos que es como se inicia.

Entonces tu crear una variable de estado que va a estar vacia y le indicas de que tipo va a ser, para luego decirle que equivale a el valor de otra variable del mismo tipo, en el caso del ejemplo, de un Array.

**La forma de funcionar en React es distinta a Java, cuando insertas algo no vas a verlo porque React no funciona así, solamente vas a poner visualizar algo CUANDO se produzca un cambio en alguna de las VARIABLES DE ESTADO a través de useEffect ( )**

**Se da el caso de que, alguien puede introducir un nuevo registro en una tabla de SQL, y no vas a poder verlo porque no se modifica nada y por lo tanto el useEffect no se activa**

**Vale, esto que acabo de explicar se puede cambiar indicando a useEffect el comportamiento que tiene que tener**

**Información oficial 🡪** [**https://es.react.dev/reference/react/hooks**](https://es.react.dev/reference/react/hooks)

Esta estructura SIEMPRE es igual:

**useEffect( () => {**

**fetch (AQUÍ PONEMOS LA PETICION)**

**.then(response => response.json () )  
.then (NOMBRE DE LA VARIABLE => setNOMBRE DE LA VARIABLE (NOMBRE DE LA VARIABLE) ) ;**

**}, [ ] );**

fetch ( ) 🡪 Se encarga de mandar una petición al endPoint que escribas. El controlador del servidor en el backend recogerá la petición, la procesara y enviara devuelta al cliente (en un formato especifico) y un código de estado (1XX,2XX,3XX,4XX,5XX)

.then(response => response.json () ) 🡪 En esta línea conviertes la respuesta recibida en JSON

.then (NOMBRE DE LA VARIABLE => setNOMBRE DE LA VARIABLE (NOMBRE DE LA VARIABLE) ) ; 🡪 La metes en el nombre de la variable

Por último, tenemos el formato HTML en el que se va a mostrar esta información.

Dentro de un div, encontramos un bucle for (si aunque no lo creas)

En el ejemplo, data es una tabla, por lo tanto por cada ítem (que es cada valor de la celda) dentro de la tabla, crea un div que contiene el contenido de la tabla, que en este caso es un nombre y un texto

Este código es mas complejo, incluye funciones de escritura en la tabla de datos y mostrar.

A la derecha es lo que muestra React mostrando el código HTML.

El proceso seria el siguiente:

-Poner a nuestro servidor a la escucha en el backend. Node /src/index.js

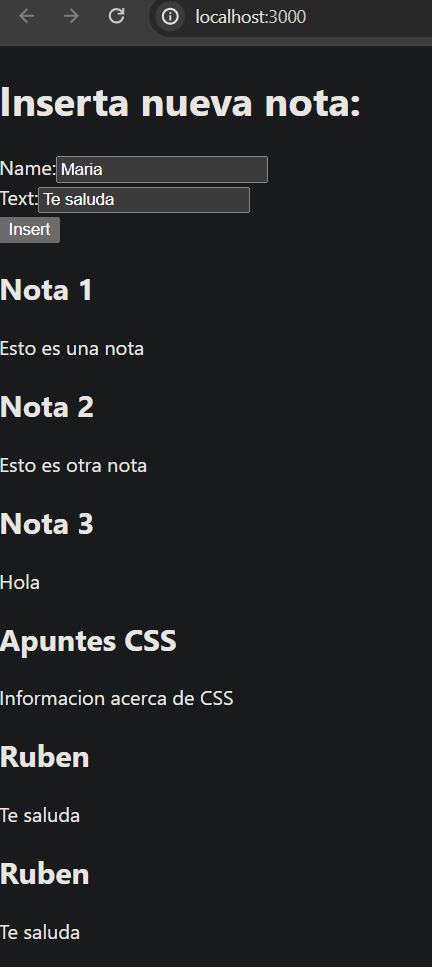
-Iniciar React en el frontend npm Start

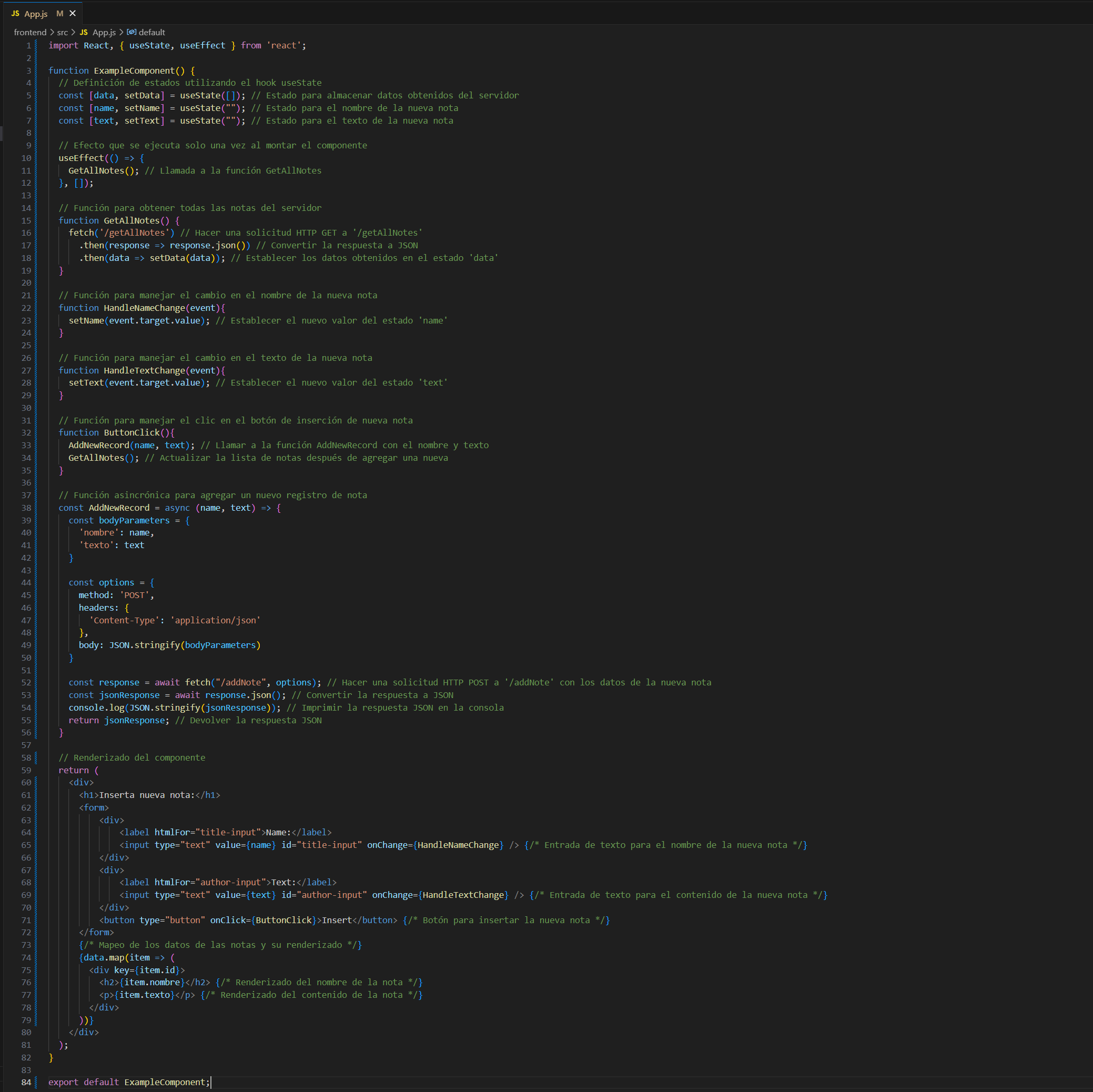
Y nos mostrara en localhost:3000 en el navegador lo que hayamos programado en React.

Con esto:

-En el puerto 3001 tendriamos el servidor (BACKEND)

-En el puerto 3000 tendriamos la API con React (FRONEND)





Notas: Ojo con la etiqueta <input> porque es con n, si pones <imput> no funcionará y además no te advierte, el programa traga, por lo que si no ves un cajetín para introducir texto se debe a esto.

En la carpeta Routes tienes las rutas de la tabla que tienes en la BBDD. Ejemplo: nombredelatablaRoute.js

Si quieres acceder al código de cada ruta, vas a la carpeta services y buscas nombredelatablaService.js

Es decir para el proyeto, tendremos en la carpeta routes con una ruta por cada tabla:

-studentRoute

-companyRoute

-contactsRoute

-adressRoute

…

En cada ruta programaremos los endPoints que vayan a trabajar sobre esa tabla en concreto

/addStudent /deleteStudent /getStudent /updateStudent …

Asi con cada tabla que tenga sentido tener funcionalidad.

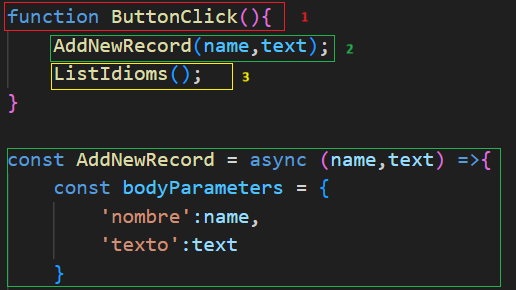
En el index.js de routes (el principal) tendremos que tener la direccion de cada ruta anterior que es especifica de una tabla. El tema es que tenemos uno principal en el que llamamos a la ruta de la tabla que se necesite en cada petición cuando lo solicite el usuario y luego un arhcivo .js por cada tabla, asi es mas fácil trabajar sin tener todo en un único .js.

La función onChange permite accionarse cuando ocurre cualquier cambio en el inputtext, a diferencia de Summit que requiere de la acción enter para accionarse. Por lo tanto, esta cosntantemente buscarndo cambios en un bucle infinito.

Por lo tanto, la variable name va a tener un valor en tiempo real de lo que tu escribas.

Lo mismo va a pasar con la variable text, va a actualizar la variable en tiempo real.

Despues tendremos la función button que lo que hara será accionar el endPoint /addNewRecord en la tabla notas con las variables name y text y después volverá a listar las notas para que se muestren por pantalla.



1 – Creas la función

2 – Creas un parámetro (const) que reciba los datos en formato JSON.

Invocas el parámetro en la función creada.

3 – Para este ejemplo, después de insertar un nuevo registro queremos ver el resultado, por lo que llamo a la función listar de, en este caso, la tabla idiomas.

En el código HTML, creamos un button y le asociamos la función.

**¿QUÉ ES UN HOOK?**

Es una función especial que te permite usar características de React sin tener que escribir una clase.

Los hooks son funciones especiales que comienzan con el prefijo "use" (por ejemplo, useState, useEffect, useContext, etc.)

Símil: Es como el objeto Scanner en Java. Un objeto que lo utilizas como herramienta.

Estos son algunos de más básicos y más comunes:

1. **useState**:
   * Permite agregar estado local a componentes funcionales.
   * Devuelve una variable de estado y una función para actualizarla.
   * Se utiliza para manejar valores que pueden cambiar durante la vida útil del componente.
2. **useEffect**:
   * Permite realizar efectos secundarios en componentes funcionales.
   * Se ejecuta después de que el componente se renderiza por primera vez y después de cada actualización.
   * Puede utilizarse para realizar tareas como suscripciones a eventos, llamadas a API, actualización del DOM, etc.
   * Puede especificar dependencias para controlar cuándo se vuelve a ejecutar el efecto.
3. **useContext**:
   * Permite acceder al contexto de React en componentes funcionales.
   * Proporciona una forma de pasar datos a través del árbol de componentes sin tener que pasar props manualmente en cada nivel.
   * Es útil para acceder a datos globales o compartir funcionalidades comunes entre componentes.
4. **useReducer**:
   * Proporciona una alternativa más escalable a **useState** para manejar estados más complejos.
   * Se utiliza cuando el estado del componente depende de acciones y transiciones entre diferentes estados.
   * Similar al concepto de reducers en Redux, permite definir cómo el estado del componente debe cambiar en respuesta a acciones.
5. **useCallback**:
   * Devuelve una versión memoizada de una función callback.
   * Es útil cuando se pasa una función a un componente hijo y se quiere evitar que el componente hijo se vuelva a renderizar innecesariamente si la función no cambia.
   * Se utiliza para optimizar el rendimiento de los componentes.
6. **useMemo**:
   * Devuelve un valor memoizado calculado.
   * Se utiliza para memoizar valores computados costosos, de manera que solo se recalculen cuando alguna de sus dependencias cambie.
   * Ayuda a optimizar el rendimiento evitando cálculos innecesarios en cada renderizado.
7. **useRef**:
   * Devuelve un objeto mutable cuyo atributo **current** se puede utilizar para almacenar cualquier valor mutable.
   * Se utiliza para mantener una referencia mutable a un elemento del DOM, un valor mutable o cualquier otro dato que necesite persistir entre renderizaciones.
   * Es útil para acceder a elementos del DOM de manera imperativa o para mantener valores que no provocan un nuevo renderizado.

En la carpeta cliente tendremos crearemos una carpeta componente para guardar componentes como App.js

Aquí iremos creando componentes como la barra de navegador etc, todos estos componentes iran en app.js

Ahora tendremos dos tipos de peticiones distintas: localhost 3000 (servidor) y localhost 3001 (cliente)

Con el componente BrowserRouter nos permite cambiar entre htmls (inicio, estudiante, empresas etc…)

Lo que antes hacimos con href ahora lo haremos con el componente BrowserRoutes indicándole el html que tiene que cargar

Pasos para instalar

nmp install tanto en backend como en frontend

server 🡪 npm run dev

cliente 🡪 npm start

Tendremos que crear una BBDD llamada PlenaInclusion y el programa instalara la BBDD automáticamente.

Aparecera el navegador en blanco porque no hay un componente / que muestre algo

-------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Es necesario instalar npm install react-router-dom para poder ejecutar los componentes y sus rutas dentro de app.js en la parte de frontend. Debes instalarlo en la parte de frontentd.

