REACT FrontEnd, Backend MYSQL y Nodejs para conexión

Primero para crear un apartado en react ocupar

* Npx create-react-app client (este es el frontend)

Tambien se debe crear la carpeta de backend(mysql) creando en la raíz una carpeta llamada server

Posteriormente se crea mediante npm -y init el package.json para las dependencias principales:  
A screenshot of a computer program

Description automatically generated

Ya teniendo esto se comienza la instalación de paquetes,

* Express para el backend y la creación de servidor: npm i express
* Mysql2 es el modulo de conexión mysql con promesas, lo que hace es agregar características más modernas a mysql normal: npm i mysql2
* Morgan para ver los mensajes en consola del backend: npm i Morgan
* Nodemon para no reiniciar el servicio en cada actualización y que se actualice solo al guardar: npm i nodemon -D
* O npm i express mysql morgan y luego npm i nodemon -D

Ya teniendo todo esto se crea el archivo raíz del backend, en este caso dentro de la carpeta server, se crea index.js

Además en el package.json se crea un script especial para la ejecución que ejecuta por uno mismo nodemon así solo se hace npm run dev y el cambio se hace solo, por ende con solo agregar npm run dev se ejecuta el backend que esta en server/index.js : “dev”: “nodemon server/index.js”

A screenshot of a computer program

Description automatically generated

Ahora para poder importar módulos node se debe asignar en el package que se pueda asignar módulos, sino esto no funcionaria:  
A screenshot of a computer

Description automatically generated

* Para ello nos vamos a package.json y se declara el “type”: “module” de esta forma:
* A screenshot of a computer program

  Description automatically generated

Ahora se guarda la información traida de express en app y se agrega que el servidor va a estar ejecutando y escuchando en el puerto 3000

A screenshot of a computer

Description automatically generated

Ahora por tema de buena practica se puede guardar configuraciones en un archivo llamado config.js que va a estar con index.js, dentro de config.js va por ejemplo el numero de puerto, … Entonces para tomar esa información de config.js se importa y se agrega

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**NOTA: COMO EL FRONTEND DE REACT SE CONECTA NORMALMENTE AL 3000, POR ELLO EL BACKEND SE ENVIO AL 4000**

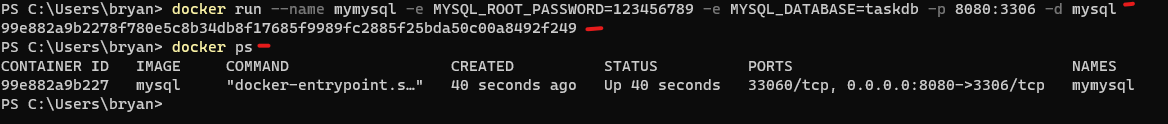
**CONEXIÓN DEL BACKEND AL SQL**

Ahora para la conexión de sql con el backend se crea un archivo dentro de server llamado db.js, donde se hace la importación y llamado a mysql

* Entonces primero se importa mysql haciendo un llamado a createPool, que permite la utilización de consultas a mysql2/promise: import {createPool} from ‘mysql2/promise’
* Para ello se guarda pool en una variable y así guardar sus funciones para las consultas, como métodos queries, etc.. Esto se hace en otro archivo.
* Aquí dentro lo que se debe asignar son las configuraciones de la base de datos como el host, usuario, password, nombre de base de datos,etc..
* Entonces quedaría así:
* Export const pool = createPool({
  + Host: ‘localhost’,
  + })

En caso de no tener instalado mysql lo que se pueda hacer es uso de Docker

* Se ingresa a Docker desktop
* Y se descarga la imagen de mysql de Docker, buscando en google: Docker mysql: <https://hub.docker.com/_/mysql>
* Entonces según la documentación para instalar Docker se puede hacer uso de la consola y colocar Docker pull mysql, de esta forma se descarga o instala mysql para Docker.
* Con Docker images se puede ver las imágenes que se tiene y en este caso el de mysql
* A screen shot of a computer

  Description automatically generated
* **docker run --name mymysql -e MYSQL\_ROOT\_PASSWORD=123456789 -e MYSQL\_DATABASE=taskdb -p 3306:3306 -d mysql** (aquí se ejecuta el contenedor de mysql, en este caso tiene algunos atributos agregamos como se va a llamar mymysql(con –name), luego una variable de entorno para la contraseña del usuario MYSQL\_ROOT\_PASS(con -e)=123456789 para tener una contraseña inicial en la base de datos, en este caso 123456789. Además se puede crear una base de datos inicial altiro haciendo uso de MYSQL\_DATABASE=taskdb. Por ultimo se agrega un puerto para la conexión que por defecto es el 3306 el mysql, haciendo que el primero sea el puerto interno de mi computador para poder ingresar y el externo de mysql que hace uso, por ello los mismos valores. En caso de no funcionar el 3306 por estar en uso, se puede ocupar el 8080:3306
* 
* De esta forma ya estaría listo, ahora si se desea ejecutar esto seria así:
* docker exec -it mymysql bash (ejecuta docker de forma interactivo al contenedor mymysql con bash que viene)
* Ahora si se ejecuta mysql -u root --password y se coloca la password como 123456789, se puede ver que se entro a mysql mediante Docker.
* Otra forma es ya tener mysql en la computadora y hacer uso de mysql -u root -p12345678 y se hace lo mismo
* A screenshot of a computer program

  Description automatically generated
* Y se puede ver la base de datos creada:

A screenshot of a computer

Description automatically generated

* Ahora ya con esto se puede comenzar a conectarse mediante el código, en el db.js se configura todo, el host, el puerto, el nombre de la base de datos, todo lo que se hizo aquí:
  + A screenshot of a computer program

    Description automatically generated

LLAMADOS A BASE DE DATOS

* Ahora ya teniendo esto configurado se crea la carpeta routes dentro de server
* Aquí se van a crear las rutas para GET y POST, para ello se crea un archivo llamado index.routes.js
* Dentro de este archivo se va hacer uso de una prueba llamada ping, donde para ello se hace un llamado a express
* Ahora con router se hace un get a ping siendo asíncrono al ser un llamado a la base de datos, este llamado se hace mediante await y con la llamada a la base de datos(pool) haciendo un query que crea una base de datos, haciendo uso de 1+1
* Luego lo muestra en un console.log y que lo muestre como un json
* A screenshot of a computer program

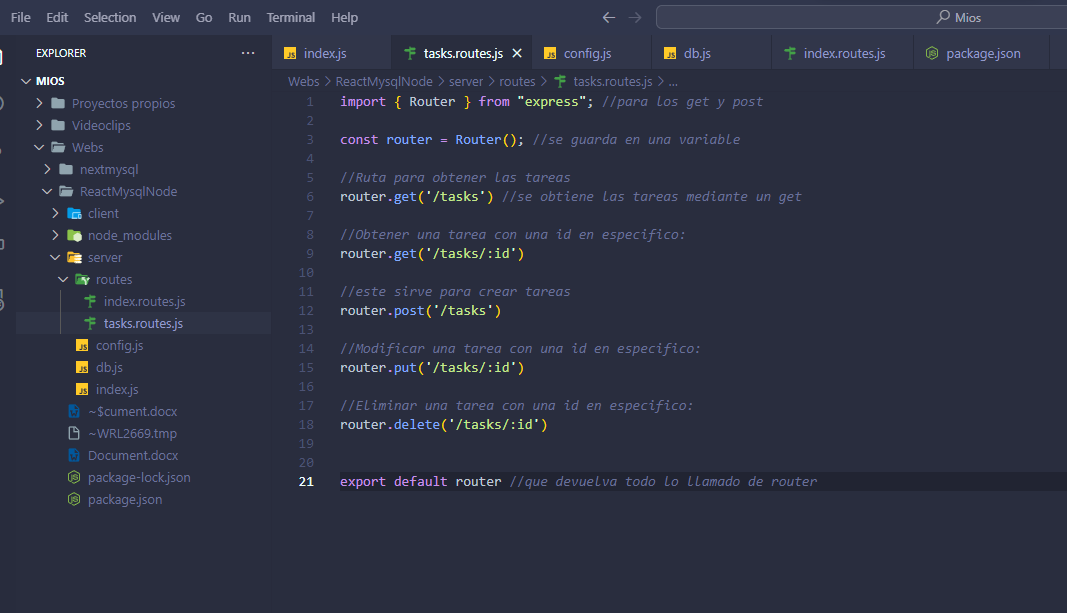
  Description automatically generated
* Ahora para que se ejecute este route se debe llamar también en el index, en la parte principal del backend llamar a esa ruta así:
  + Donde primero se importa en index.js se importa las rutas
  + Posteriormente se dice que la aplicación(app) de index haga uso de las rutas del index
  + A screenshot of a computer program

    Description automatically generated
  + Y listo ahora debería funcionar
* A screenshot of a computer

  Description automatically generated
* Se puede ver que la llamada resulto
* A screen shot of a computer code

  Description automatically generated
* Además se ve que en consola se hizo la ejecución del backend haciendo que en la primera columna este el result 2

**CREACIÓN DE RUTAS DEL BACKEND:**

* Rutas de las tareas
  + Se crea otro archivo rutas en routes, llamado tasks.routes.js
  + Entonces nuevamente se importa Router de express para los get y post
  + Posteriormente se guarda en una variable y se comienza la creación de rutas
  + Aquí se agrega los CRUD: Eliminar, crear, modificar
  + En la ruta /tasks se va a obtener las tareas haciendo uso de get
  + Luego otra ruta es /tasks pero mediante un POST donde se van a crear tareas(POST Ya que se envia información mediante un formulario, mayor seguridad)
  + Modificar haciendo uso de PUT, con una id en especifico, es decir ‘/tasks/:id’ (el :id significa que no se cual id va a llegar pero sirve para un parámetro genérico de cualquier id
  + Luego obtener una única tarea según una id, haciendo uso de GET /tasks/:id
  + Luego DELETE para eliminar una tarea en especifico /task/:id
  + 

Que ocurre que en la anterior prueba se agrego las funcionalidades dentro de routes, como en index.routes pero era solo un ejemplo. Una buena practica es hacer uso de un archivo llamado controllers en server, donde ahí dentro se hacen las operaciones de las rutas

A screenshot of a computer program

Description automatically generated

De esta forma se comienza a crear las operaciones ahí

OPERACIONES DE TASK EN CONTROLLERS

* Se crean varias funciones para obtener, crear, actualizar y eliminar.
* Retornando una respuesta mediante el response (res)
* A screenshot of a computer program

  Description automatically generated
* De esta forma son creadas las funciones, se debe asignar el export debido a que las funciones van a ser exportadas por las rutas para ser usadas
* Por ende ahora nos vamos a routes para importas las funciones de task.controllers, una por una.
* A screenshot of a computer program

  Description automatically generated
* Posteriormente son agregadas en cada ruta. Por ende, dice que en la ruta /tasks va a ejecutar la función de getTasks, luego en la ruta tasks/:id hace la llamada de una tarea en especifico mediatne la función createTask y así sucesivamente
* Ahora ya teniendo esto se agrega las rutas al index principal del backend para que importe las rutas creadas mediante el import tasksRoutes from ‘../server/controllers/task.controllers.js’ y luego que la aplicación utilice esas rutas para que sean funcionales
* A screenshot of a computer program

  Description automatically generated
* Esto se puede ver funcionando en la web para los gets:
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* Para los post, put, delete se hace uso de Postman:
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated
* A screenshot of a computer

  Description automatically generated

Ahora ya teniendo esto, se deben empezar a comenzar a realizar las funciones para la conexión de MYSQL

**CONECCIÓN DE FUNCIONES CON FUNCIONES CRUD DE MYSQL en tasks.controllers**

* Primero que todo el post hace uso del request, debido a que la función de agrega el request y response(res y req), por ende al momento de agregar por ejemplo un nombre, eso se enviar mediante un request que llega a la función. Por lo que si se hace uso un console.log(req.body) se mostraria el contenido de la petición
* A computer screen with text

  Description automatically generated
* Con esto tranforma lo ingresado a función json para que lo lea:
* A black background with white text

  Description automatically generated
* A screenshot of a computer

  Description automatically generated

Entonces siguiendo ese ejemplo, para guadar la información traida lo que se hace es guardar las variables en este caso con un const {title, description} = req.body (se guarda la información traida en un title y description para luego enviarlo a la base de datos.

Para ello se debe importar la conexión del backend arriba de tasks.controllers, mediante un import, haciendo el import del objeto pool que es la conexión. Import {pool} from ‘../db.js’

* A screenshot of a computer program

  Description automatically generated
* De esta forma se tiene la conexión a la base de datos y las variables guardadas que se van a ingresar.
* Por ende ahora se debe enviar la información a tablas de la base de datos, pero como actualmente no tenemos tablas, estas se deben crear mediante comandos por ejemplo. En este caso comandos mysql
* Entonces nos contenemos en el contenedor en caso de tenerlo cerrado(esto en caso de ocupar Docker) con Docker exec -it mymysql bash
  + A black screen with white text

    Description automatically generated
  + Y nos conectamos a la base de datos mediante el root y las password
  + A close up of a black background

    Description automatically generated
  + Por ende, ahora vemos los databases e ingresamos a la base de datos taskdb que no tiene tablas actualmente
  + A screen shot of a computer program

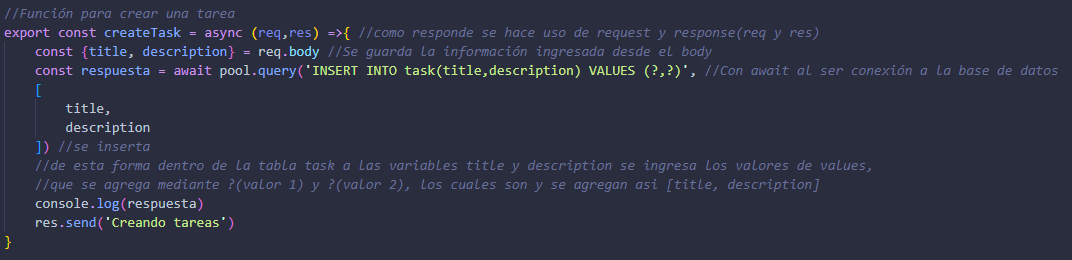
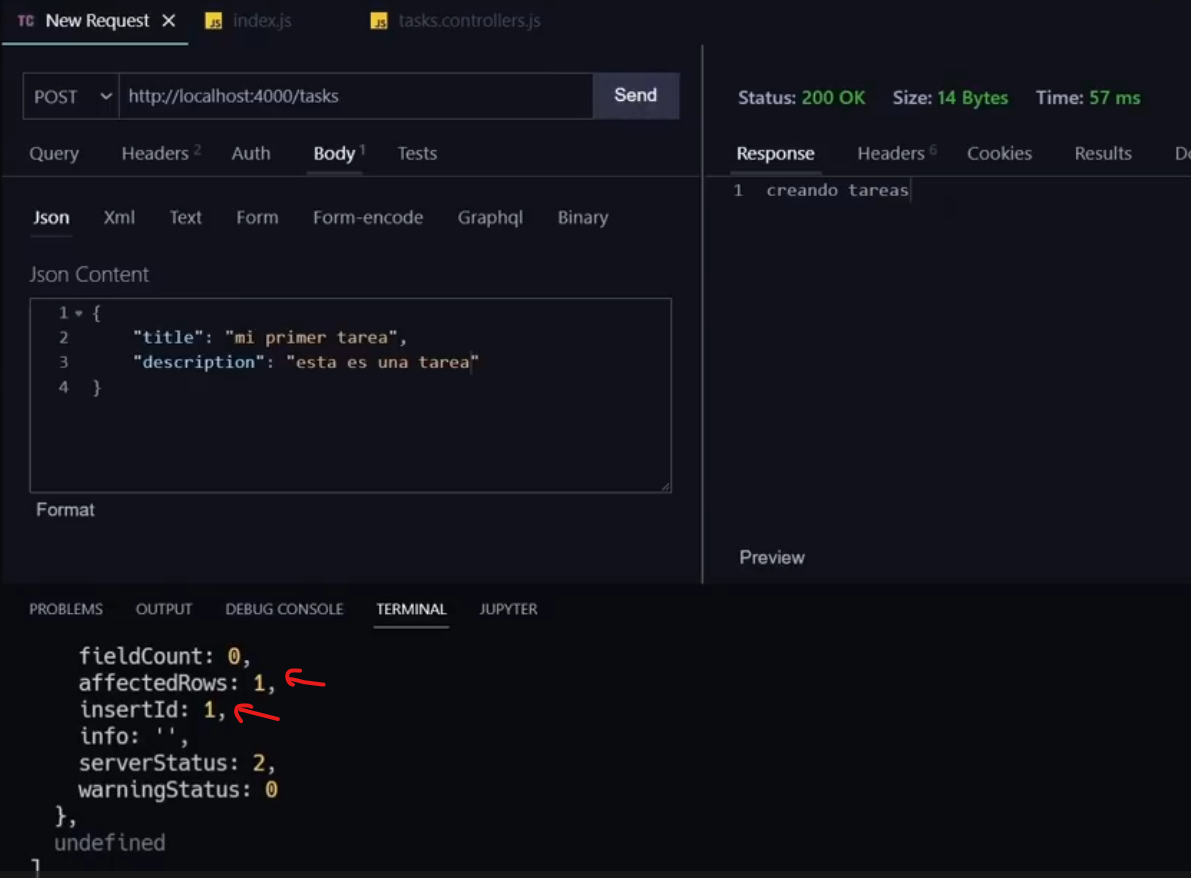
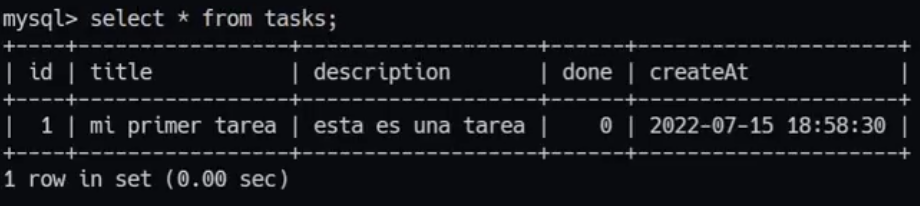
    Description automatically generated
  + Ahora aquí se puede crear las tablas haciendo comandos directamente aquí como CREATE TABLE task(..), el problema que puede ser un poco incomodo, por ende lo que se puede hacer es crear una carpeta en algún lado como en la raiz llamada database y un archivo llamado db.sql donde se hacen los comandos y luego se ejecuta y listo
  + A screenshot of a computer

    Description automatically generated
  + Aquí se puede visualizar que se crea la tabla task que va a tener los atributos id, title, description, done(sirve para guardar si la tarea esta hecha o no) y un createAt(sirve para agregar cuando fue creada la tarea).
  + Posteriormente se definen las variables como INTEGER, VARCHAR, etc..
  + El id es INTEGER y es PRIMARY KEY y AUTO\_INCREMENT(permite que cada nuevo valor se vaya incrementando el valor de la id)
  + El title va a ser tipo varchar(string) con 200 caracteres y es obligatorio, no puede ser null
  + La descripción es mas larga, varchar y es opcional, es decir puede ser null
  + El done al ser si la tarea esta hecha o no es un boolean(true o false) que no puede ser nullo y por defecto al crearse recién empieza como en 0, es decir en false(no hecho)
  + Y finalmente el createAt es un tipo tiempo que no es nulo ya que se crea al crear la tarea y que haga uso de la hora actual de la creación de la tarea
  + A screenshot of a computer program

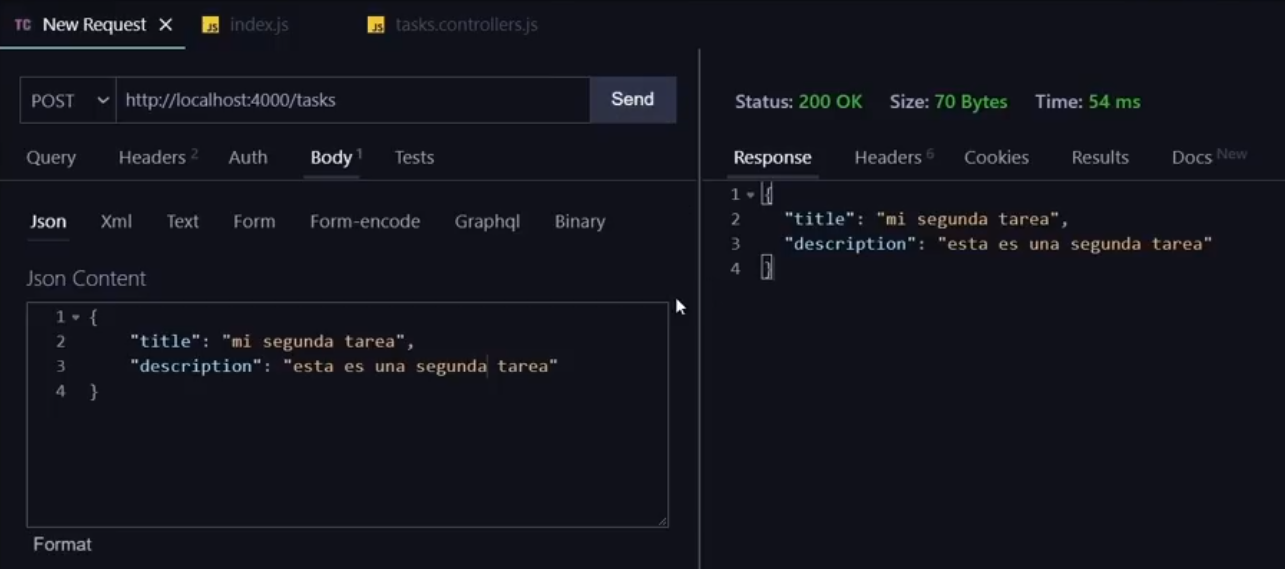
    Description automatically generated
* Ahora ya teniendo esto para la ejecución, lo bueno que se puede hacer es copiar el código y pegarlo en la consola
* A screenshot of a computer program

  Description automatically generated
* Listo, de esta forma ya estaría creada la tabla y se puede empezar a ingresar información y mostrarla

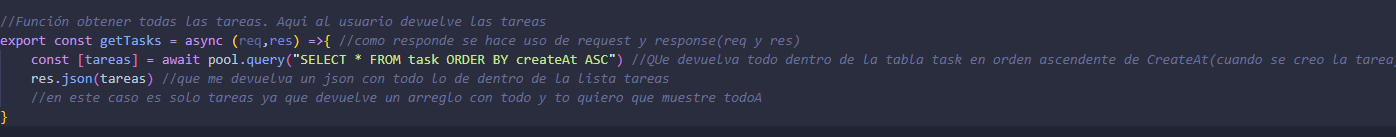
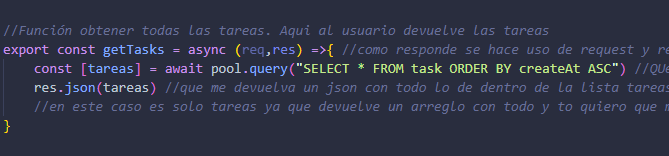
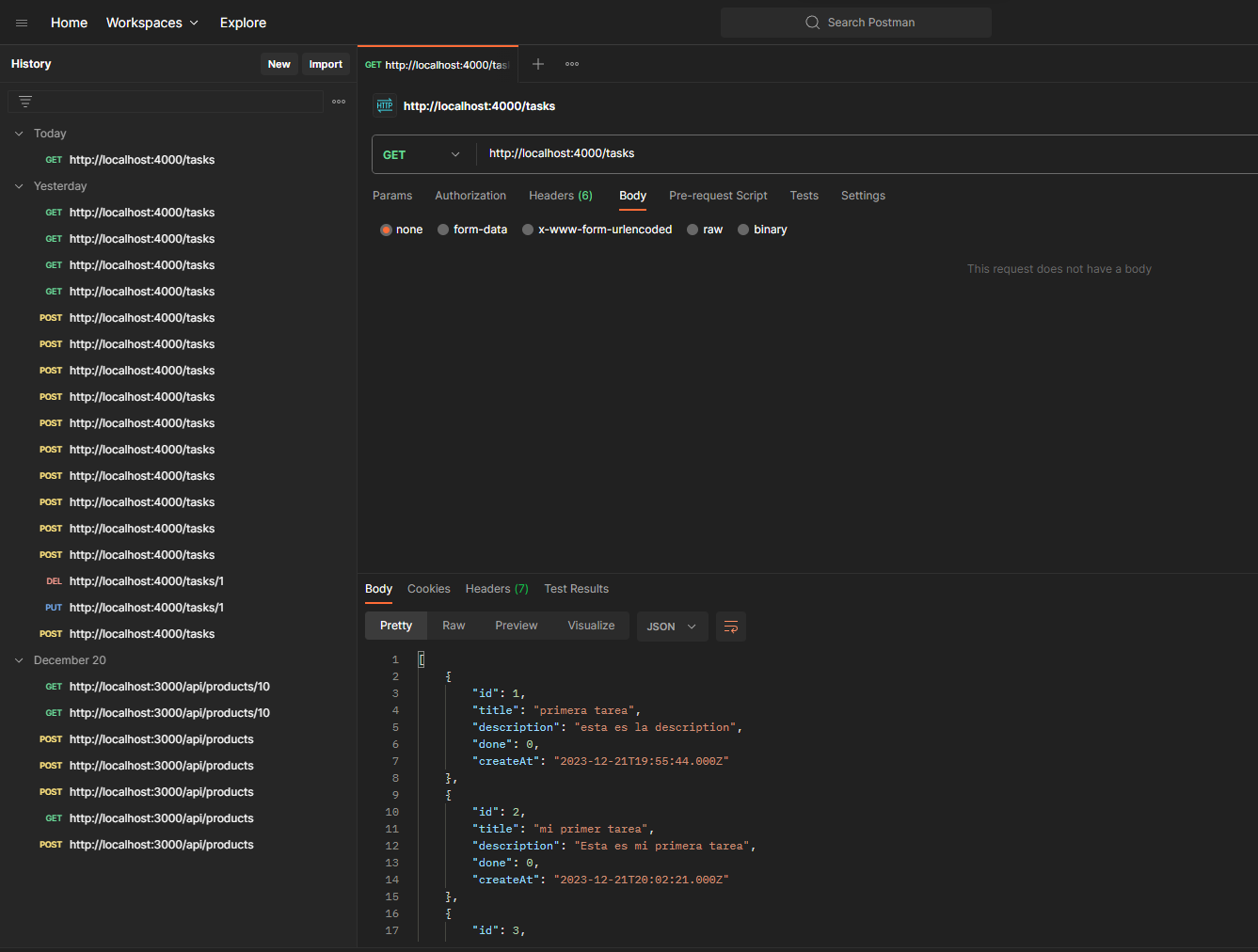
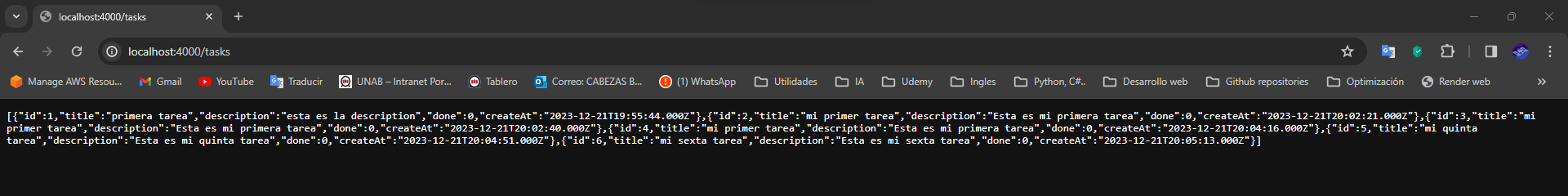
Agregación de variables a la tabla task

* Ahora ya teniendo la tabla creada, no podemos ir al código y agregar el código mysql para agregar las variables guardadas que son enviadas por el usuario en el body y crear una tarea en la tablas tarea.
* 
* DE esta forma, lo que se hace es guardar las variables enviadas mediante el const = req.body.
* Además la función es asíncrona debido que se conecta con la base de datos por ello el async y el await
* Posteriormente se conecta a la base de datos pool que se importo y con el query se agrega el comando de mysql
* Que dice que se inserta algo dentro de la tabla task que modifica los valores title y description, siendo estos valores los siguientes (¿(valor 1) , ¿(valor 2)) que son los siguientes , [title, description]
* Luego nos podemos ir al postman y hacer el post con el body de title y description
* 
* Donde se puede visualizar que se inserto 1 ID y fue afectada una tabla.
* Por ende si en la base de datos se coloca SELECT \* from task (selecciona todo de tareas y muestra)
* se puede ver la id, titulo, description, done y createAt
* Finalmente lo que se puede hacer es que también devuelva lo que se agrego
* A screenshot of a computer

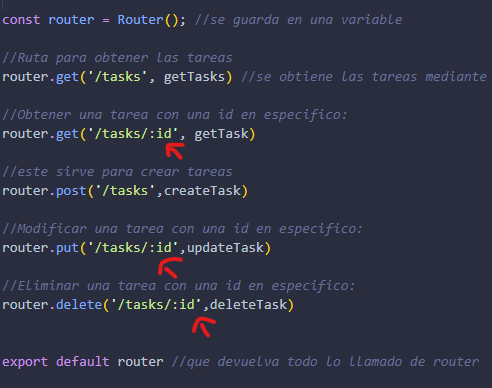
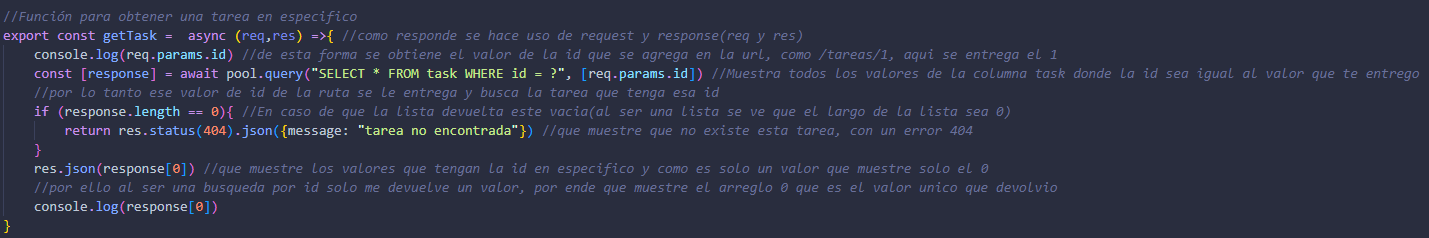
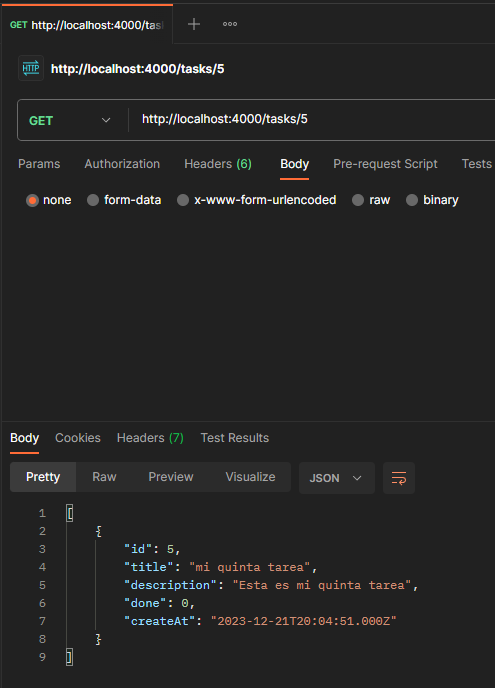
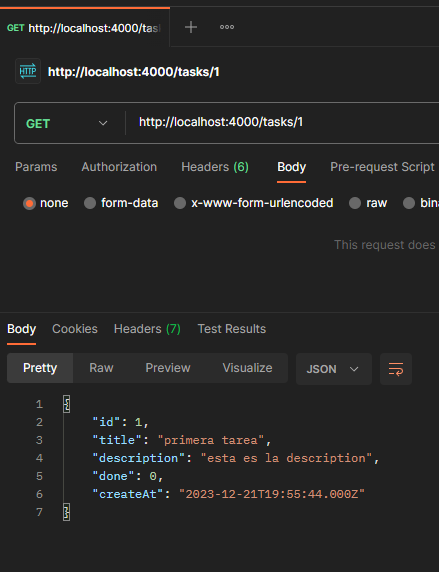
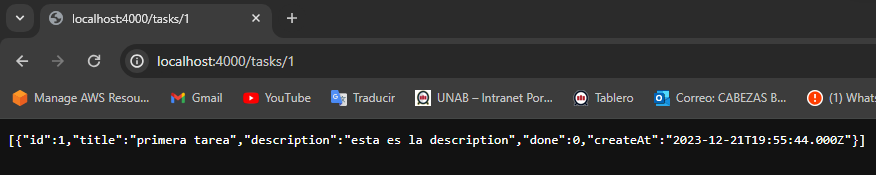
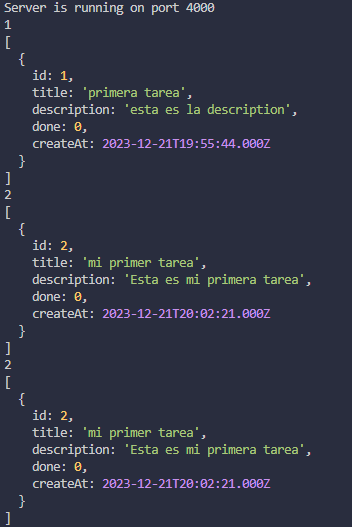
  Description automatically generated
* Por ende lo que se devuelve(res) devuelve una respuesta tipo json que tenga la id que es obtenida de result.insertId, el titulo y description. El insertId(es lo de la respuesta anterior, ahí se agrega el insertId)
* Además se deja la respuesta como tipo arreglo debido a que como se puede ver el resultado es un arreglo, por ello la id se ve como una respuesta.insertId y simplemente el title y description como es traído se deja como title y description
* A computer screen shot of a code

  Description automatically generated
* Ahí retorna:  
  

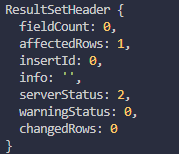
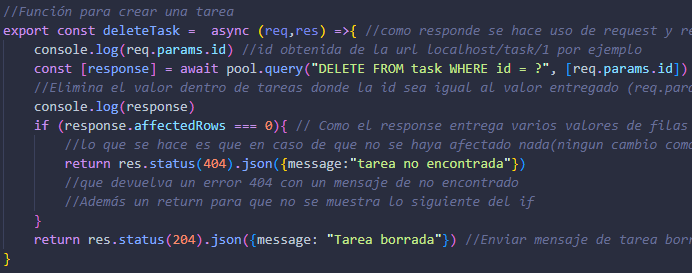
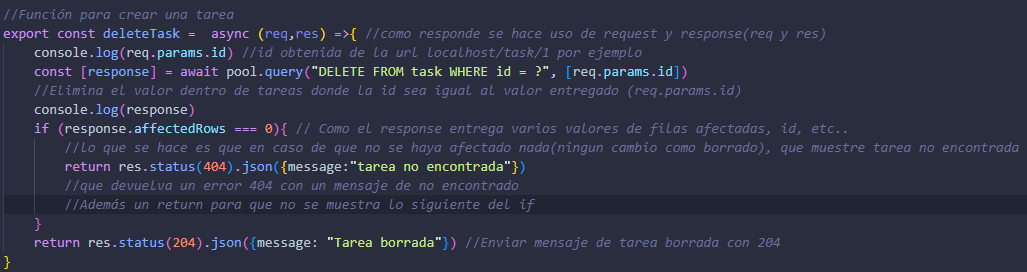
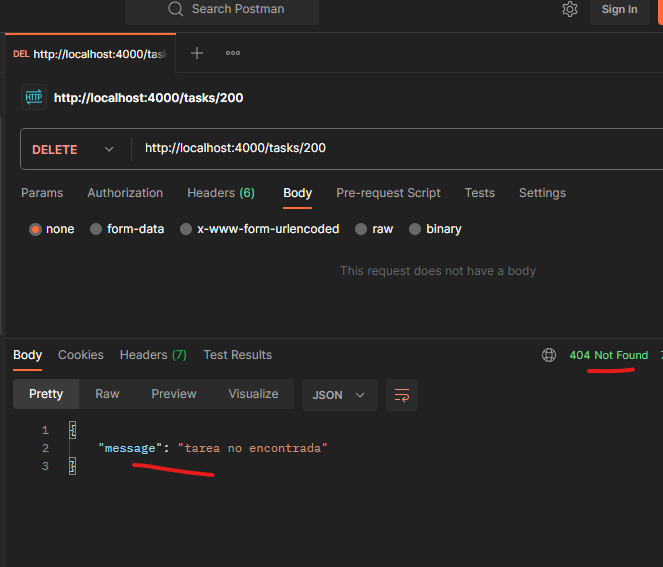
**Obtener todas las tareas GetTasks**

* Ahora para getTasks(mostrar todas las tareas)
* Lo que se hace es nuevamente un llamado a la base ed datos mediante el pool(conexión al backend) y con un query se va hacer. En este caso el query a la base de datos es: (SELECT \* FROM task ORDER BY createAt ASC) lo que significa que muestre todo de la tabla task de forma ordenada por el createAt(que es la fecha de creación de la tarea) de forma ascendente
* Es await y asyncronico por la conexión al backend
* Y se entrega directamente todo tareas ya que como se puede ver al hacer un console.log(tareas) esto devuelve como una arreglo, y como quiero todo lo dejo tal cual.
* 
* 
* Se puede visualizar en postman que devuelve el arreglo de tareas
* 
* 

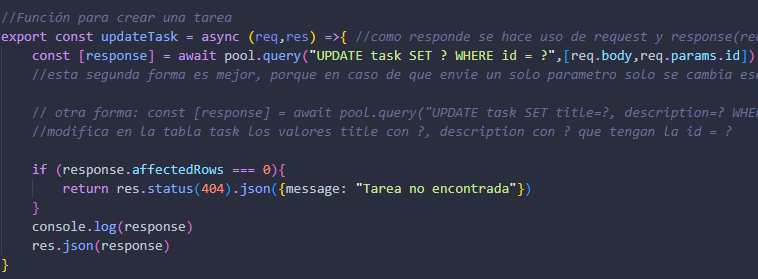
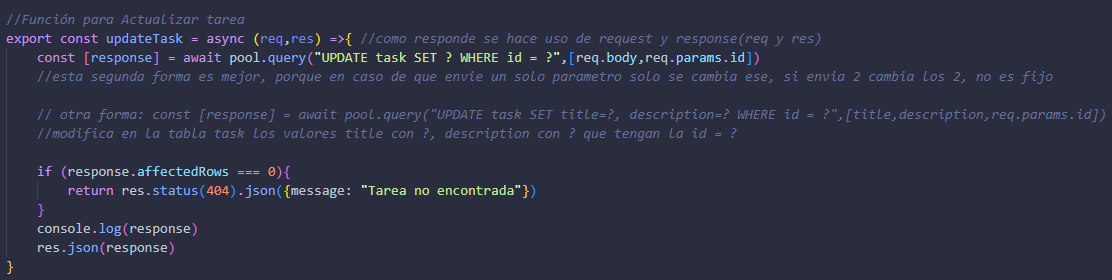
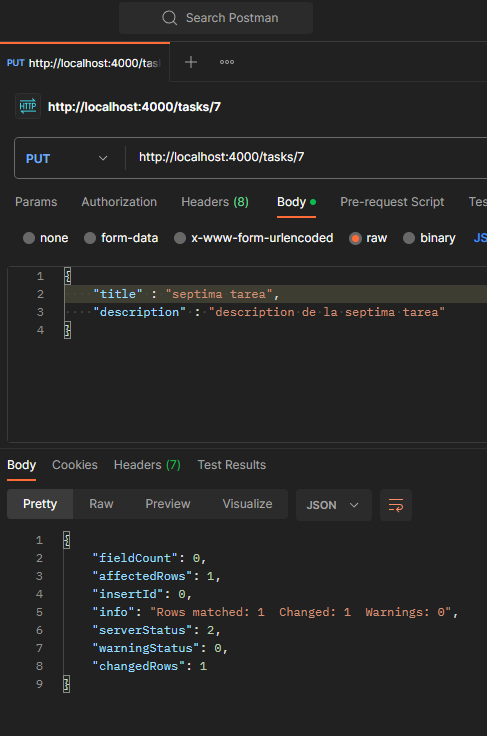
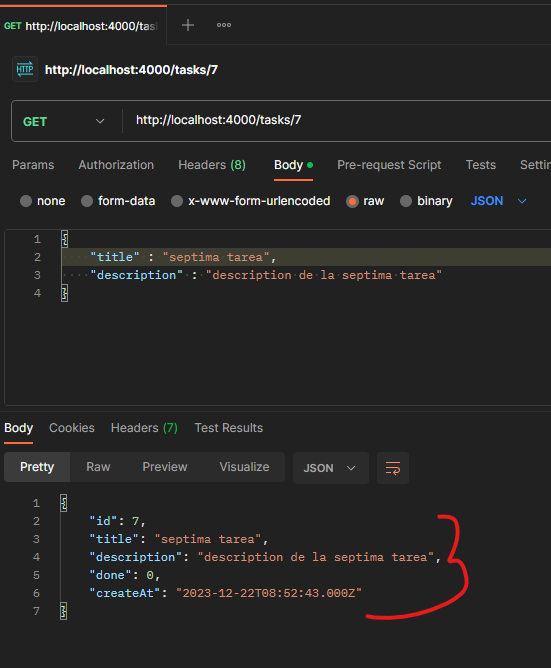
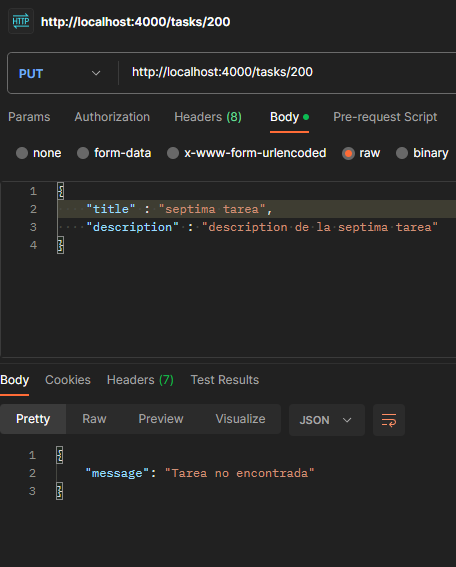
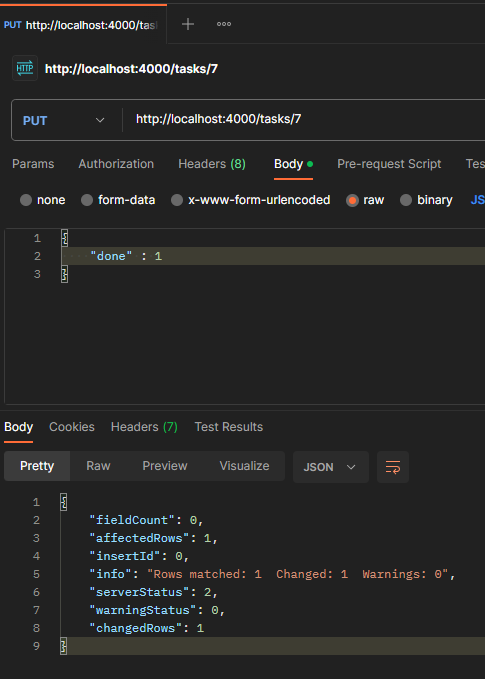
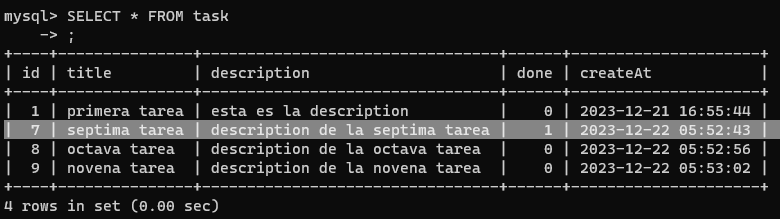
**Obtener una tarea getTask(por id):**

* Ahora lo que se va a hacer es obtener la tarea en especifica por id
* Para ello lo que se hace es hacer uso del req.params.id (esto lo que significa es obtener el valor de la id que tiene la url, es decir, cuando se busca una tarea en especifico la id se vería así en la url localhost/task/1 . el 1 en este caso seria la id, por lo tanto para obtener esa id lo que se hace es uso del req.params.id.
* Como la id se obtiene mediante la url por ello en las routes(rutas) se definen de esta forma:
* 
* Donde la id esta en la ruta, donde indica que es un valor también dinámico que cambia, es un parámetro. Para obtener el parámetro se hace mediante el nombre, es decir en este caso si no fuera id, sino taskid, se obtendría mediante req.params.taskid y así sucesivamente
* Posteriormente se hace el uso de la base de datos, donde nuevamente se hace un llamado al pool(conexión al backend) con el query a realizar que es: “SELECT \* FROM task WHERE id = ?” , [req.params.id] (esto significa que muestre todos los valores de tareas donde la id sea igual a un valor ?, donde el valor ? es: req.parms.id (que es la id de la url))
* El resultado recordar que se devuelve como un arreglo, por ello el [response] de esta forma.
* Luego se muestra todo como un json(los atributos de la tarea traida), en este caso haciendo uso del [0] debido a que solo devuelve un valor por el momento, por ello solo quiero mostrar ese valor, así entro al valor y no me muestra un arreglo si al final es un valor.
* Además en caso de que el arreglo response tenga un largo de 0(al ser arreglo tiene largos) que devuelva tarea no encontrada
* 
* 
* Así se vería sin el [0] en el arreglo
* 
* Con el [0]
* 
* 
* Y así se ve al momento de ver el console.log(req.params.id) se puede ver que se trae el valor 1, 2, 2 según el valor que se coloque en la url, con su respectiva tarea encontrada
* 
* En caso de no encontrar el valor:
* 

**Borrar tareas:**

* Ahora para la eliminación es muy parecida a la de ver por única id
* Lo que se hace es nuevamente hacer uso del req.params.id, es decir según la id de la url como localhost/task/1 (pero hacindo un llamado tipo delete)
* Aquí lo que es “DELETE FROM task WHERE ID = ?”, [req.params.id] que dice que borre el valor ed tabla donde la id sea igual a un valor, que en este caso el valor es el req.params.id
* Posteriormente se hace un console.log(response) para ver lo que responde, como se puede ver responde como la columna afectada es 1, por lo tanto hizo un cambio que en este caso es el delete
* 
* Por ello se hace que en caso de que la columna afectada sea 0, es decir no se afecto ninguna que muestre error 404 y entregue un json de tarea no encontrada
* De lo contrario, encaso de encontrar que envie un status de 204 de elemento borrado con un mensaje de tarea borrada
* 
* 
* 
* S
* 

**ACTUALIZAR TAREA**

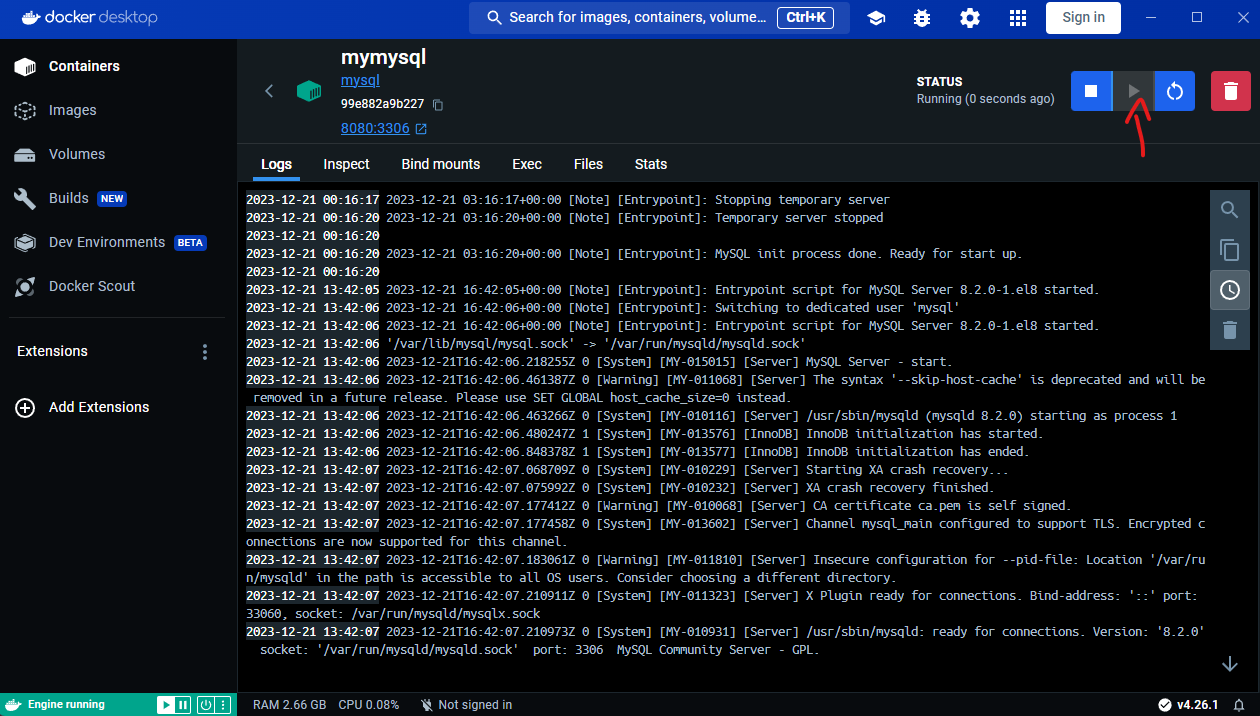
* Ahora para la actualización es parecido al anterior haciendo uso del req.params.id pero con el put.
* Además se hace uso del req.body que es la información que se agrega para actualizar los valores actuales.
* Por ende se hace un const del response como lista que retorna y un pool.query para la conexión al backend.
* El query es “UPDATE task SET ? WHERE id = ?”, [req.body,req.params.id] , de forma que dice que actualizar la tabla tarea con los valores entregado donde la id sea un valor.
* Donde el primer valor es el req.body. Lo bueno de esta forma que según el valor se actualiza, es decir, puede ser uno, dos o tres valores y se cambian solo sin asignar un valor en especifico como title= ? , sino que puede ser description solo o description, title o Date o cosas así, puede variar, no es fijo. Lo bueno que se le puede entregar ese json directamente.
* Además el req.params.id es la id de la url par el cambio.
* Posteriormente se hace una validación, que en caso de que la respuesta entregue que las filas afectadas son 0 que retorne que no existe la tarea
* Y que retorna finalmente
* 
* 
* Así se vería, al momento de hacer un PUT(update) a la tarea 7 con los nuevos valores de title y description
* 
* Se hace un get y se muestra que se actualizo
* 
* En caso de escoger una tarea que no existe se valida
* 
* Como se puede es dinámico, se puede elegir otro valor para actualizar y lo hace, esto gracias a la entrega del json completo de req.body
* 
* D
* 

Documentación de mysql2 para códigos: <https://www.npmjs.com/package/mysql2> , de igual forma como mysql2 se guía de mysql se puede buscar la guía de mysql e igual sirve:

* <https://www.npmjs.com/package/mysql>

DOCKER:

* En caso de ya tener una imagen y reiniciar la pc y tener que correr nuevamente esa imagen hacer uso de:
  + Docker image ls (se obtienen todas las imágenes y la id de la imagen)
  + A black background with white text

    Description automatically generated
  + Luego se debe corer la imagen en Docker desktop
  + 
  + E ingresamos mediante Docker exec -it mymysql bash
  + Luego Logearse: mysql -u root -p123456789
  + A screenshot of a computer screen

    Description automatically generated

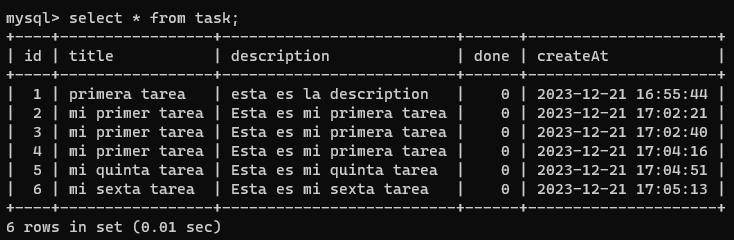
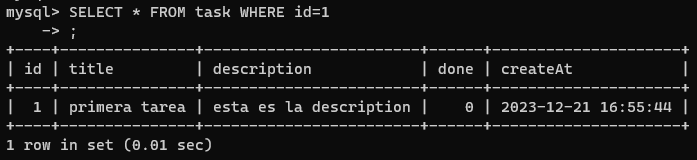
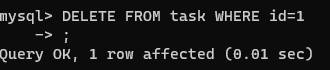
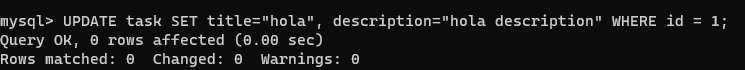
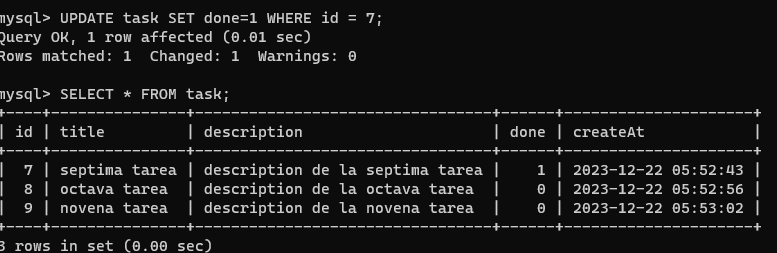
COMANDO MYSQL:

* Ver databases: show databases;
* Ingresar o usar una base de datos: use taskdb(nombre de la base de datos)
* A screen shot of a computer

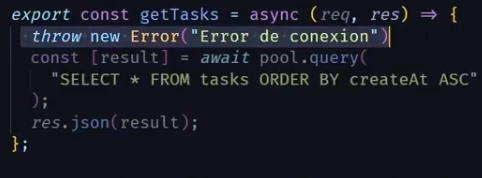
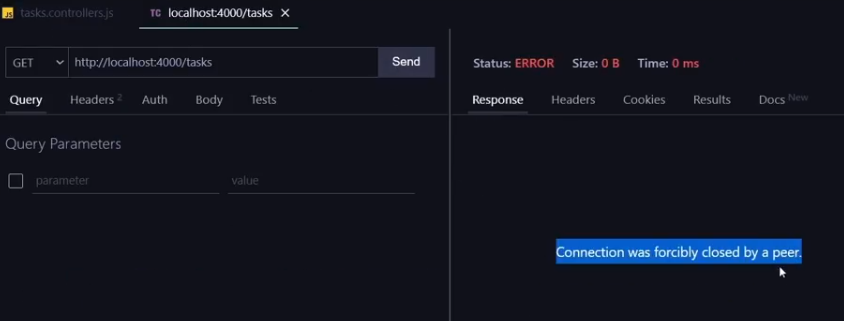
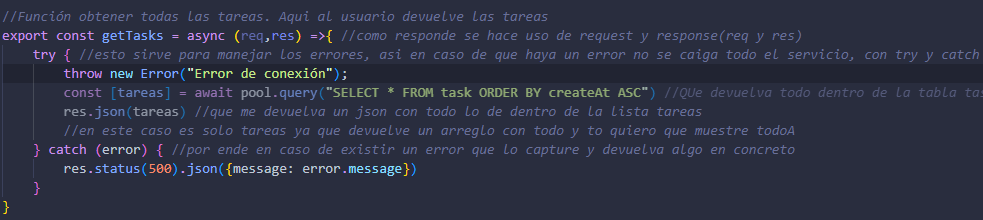
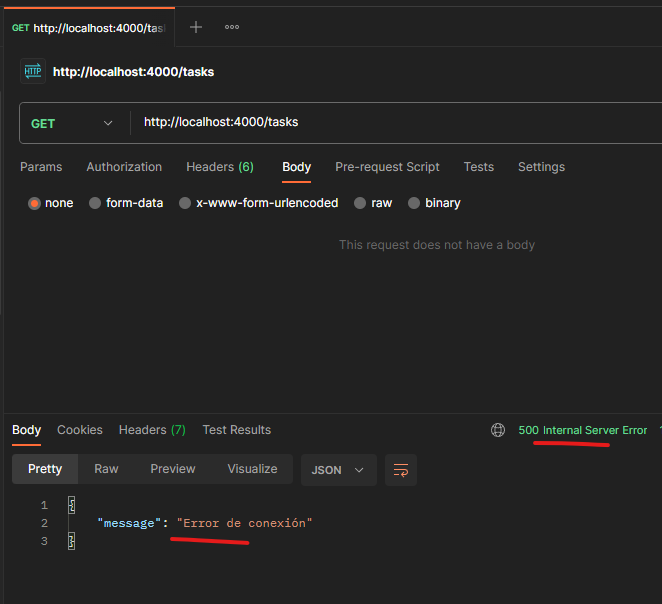
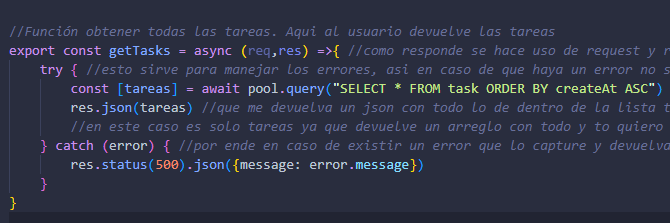
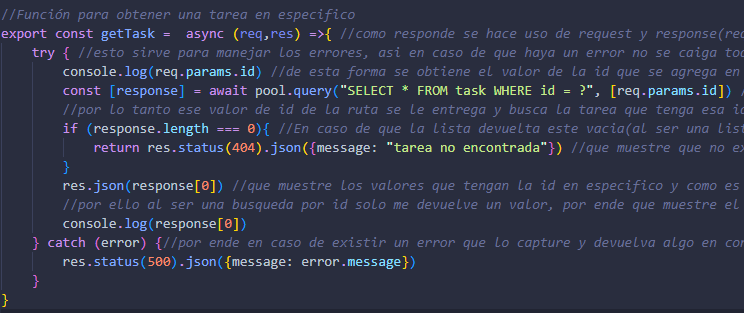
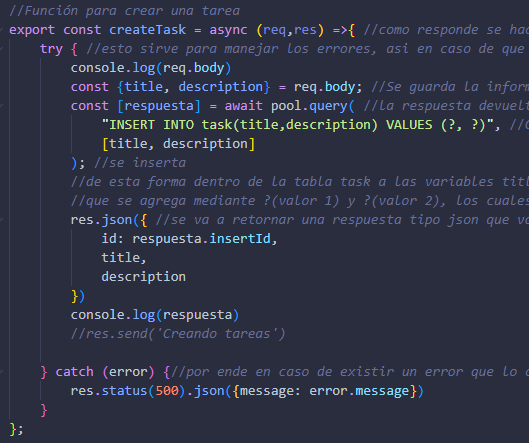
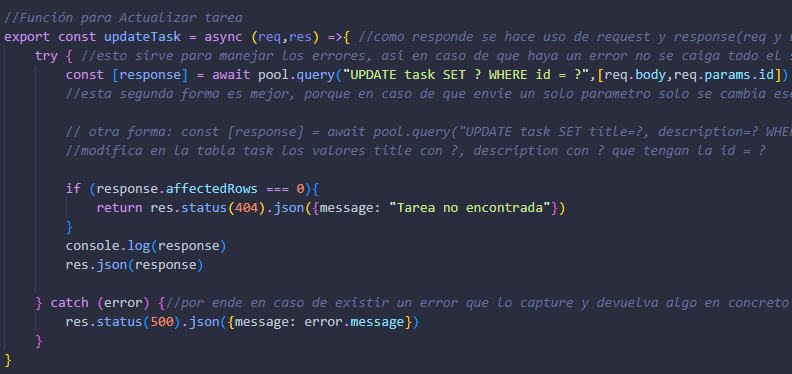
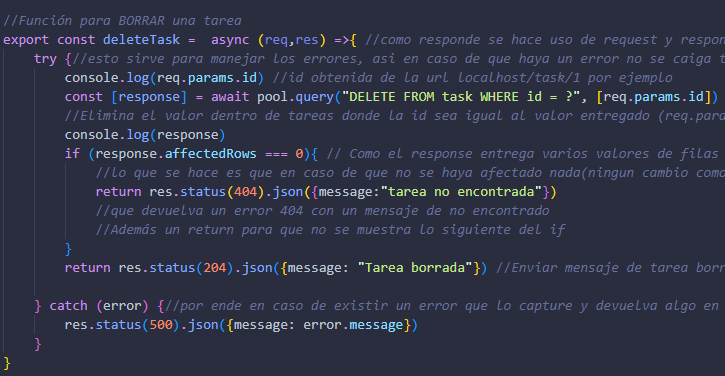
  Description automatically generated
* Mostrar tablas de la base de datos: show tables;
* A black screen with white text

  Description automatically generated
* Mostar columnas de una tabla y su tipo,llave, etc.. : describe task;
* A screen shot of a computer

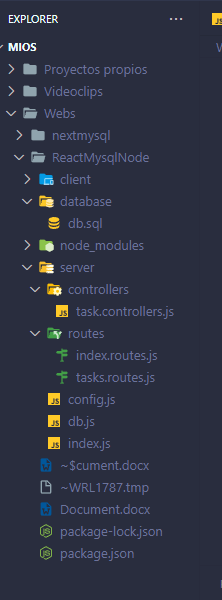
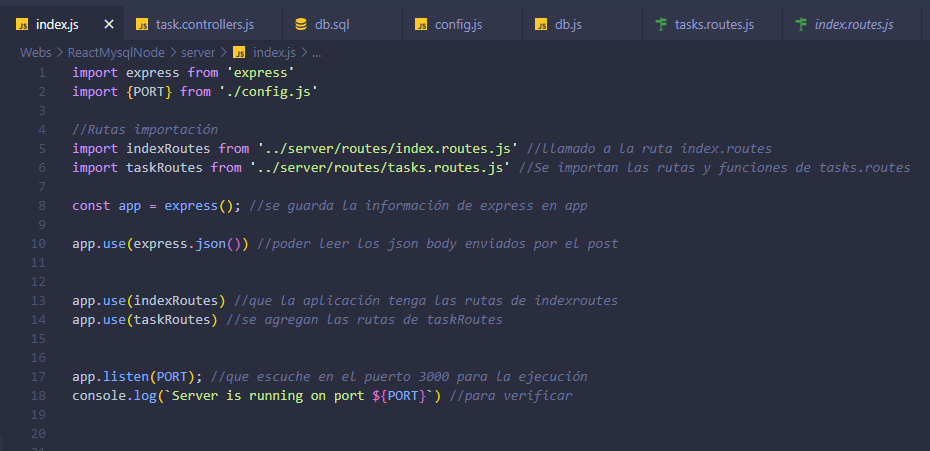
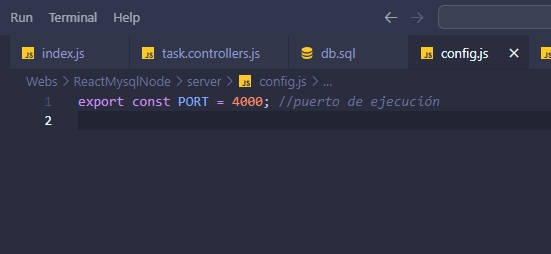
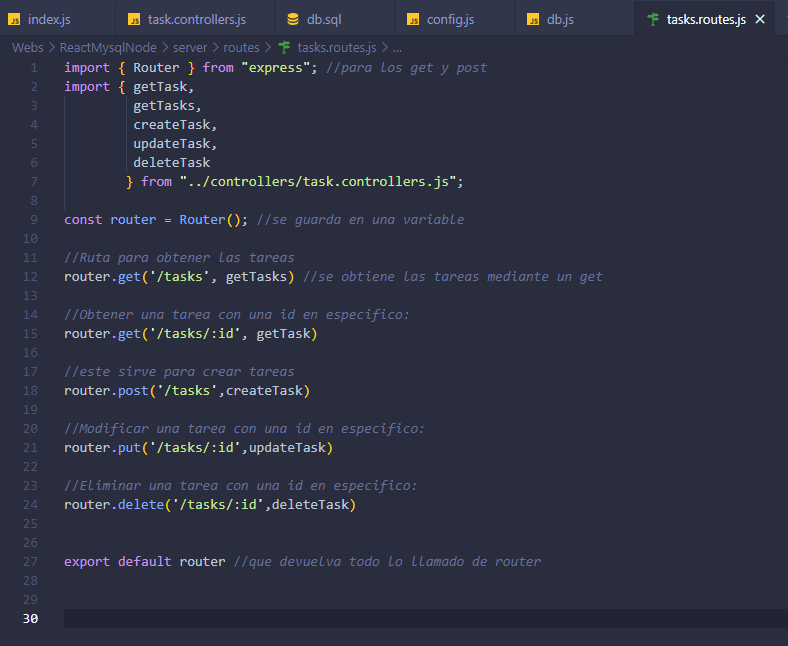
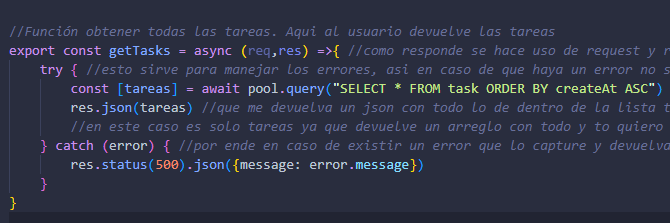
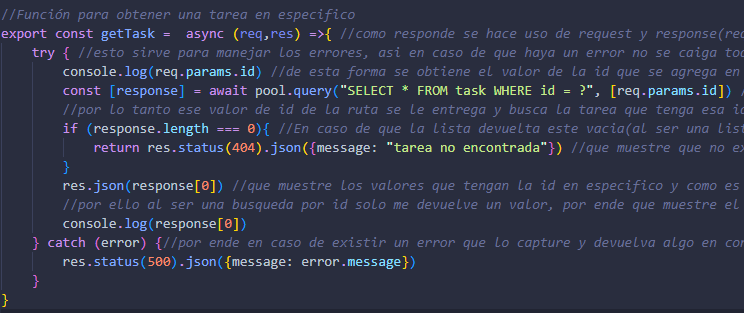
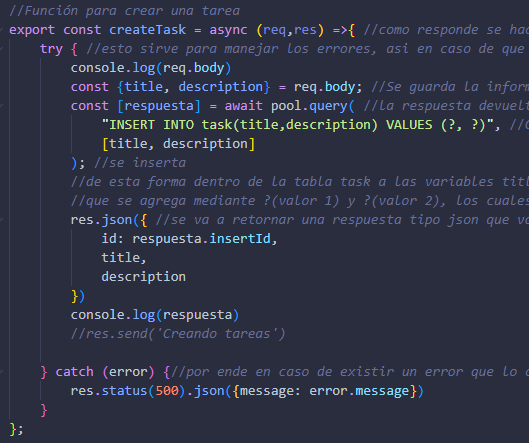
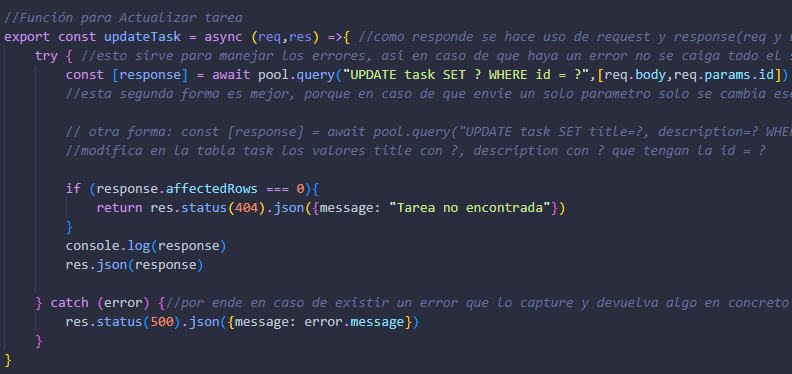
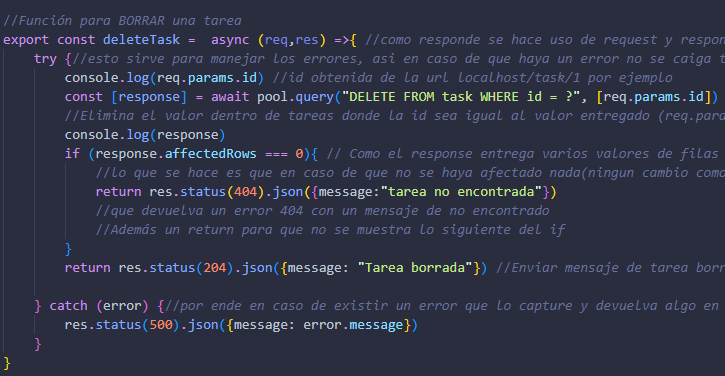
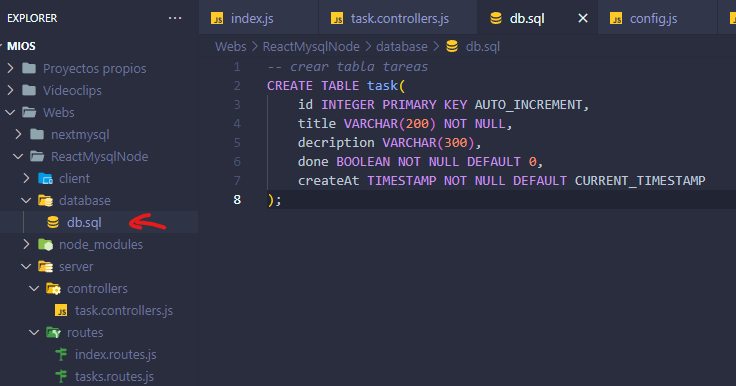
  Description automatically generated
* Modificar un nombre de una columna de la tabla : ALTER TABLE task RENAME COLUMN decription TO description; (se modifica la columna decription por description de la tabla task)
* A screenshot of a computer program

  Description automatically generated
* **AGREGAR INFORMÁCIÓN** A LAS COLUMNAS TITLE Y DESCRIPTION LOS VALORES DE MI PRIMERA TAREA Y ESTA ES LA DESCRIPCIÓN:
  + 
* Mostar todo lo que esta dentro de una tabla:
  + SELECT \* from task;
  + 
* Mostar solo las tareas que tenga un parámetro de ID igual al buscado
  + SELECT \* FROM task WHERE id= 1
  + 
* Borrar una tarea en especifico con su id
  + DELETE FROM task WHERE id=1
  + 
* Actualizar los valores de title y description de la tabla task que tengan una cierta id
  + UPDATE task SET title= “hola”, description=”hola descripcion” WHERE id = 1
  + 
  + Otra forma:
  + 

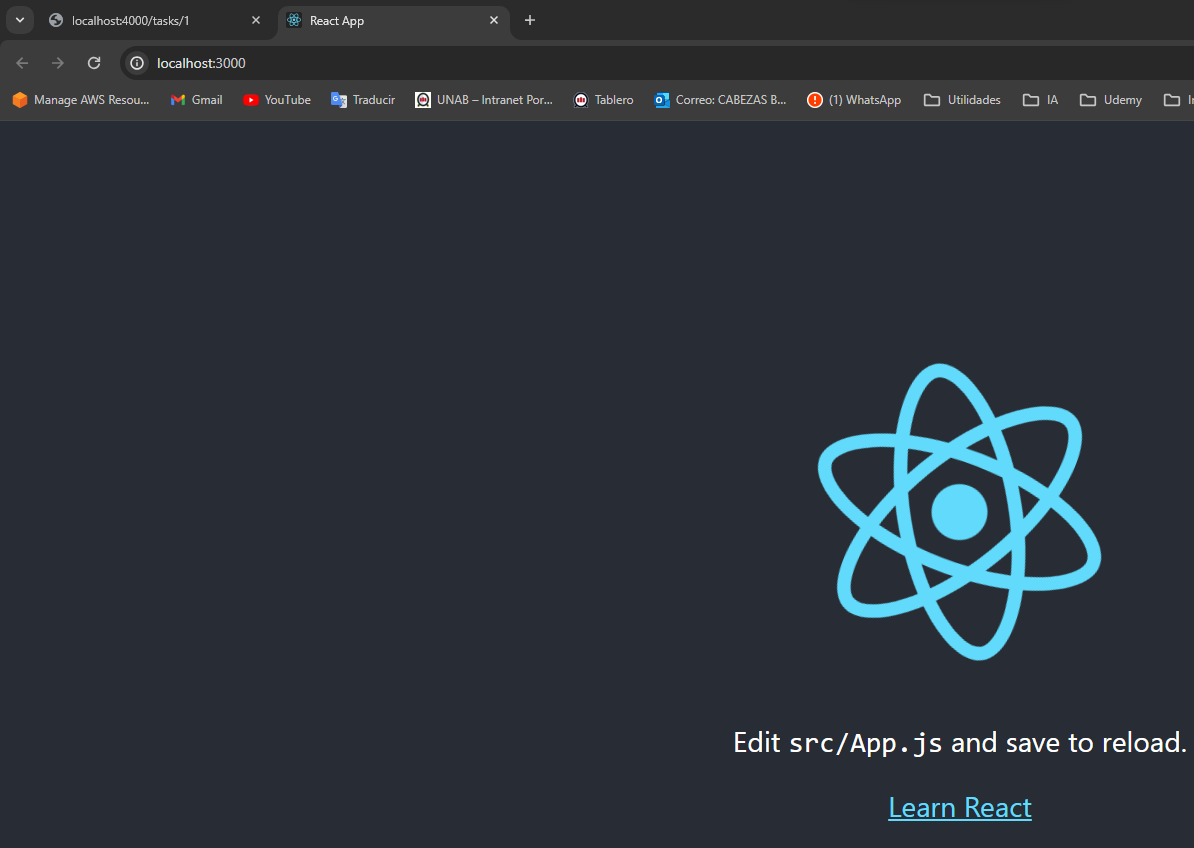
**AHORA existe una forma para validar errores,** para que en caso de que una consulta llegue a fallar como en la realidad, el usuario no vea ese error. Esto es para manejar errores. Con el try, catch se hace eso.

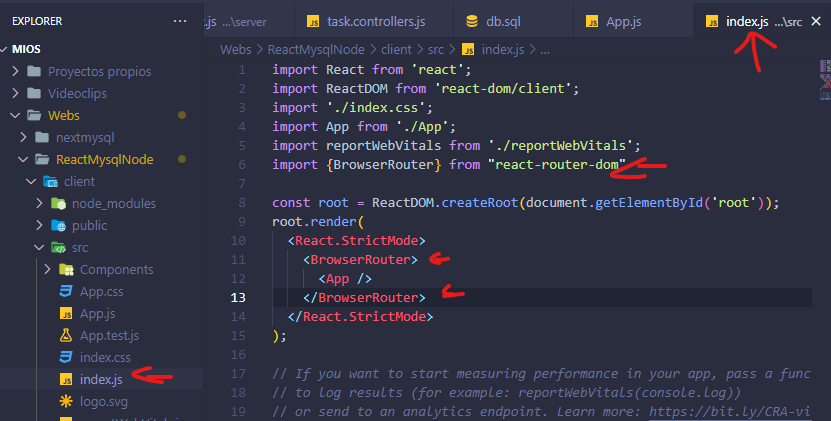
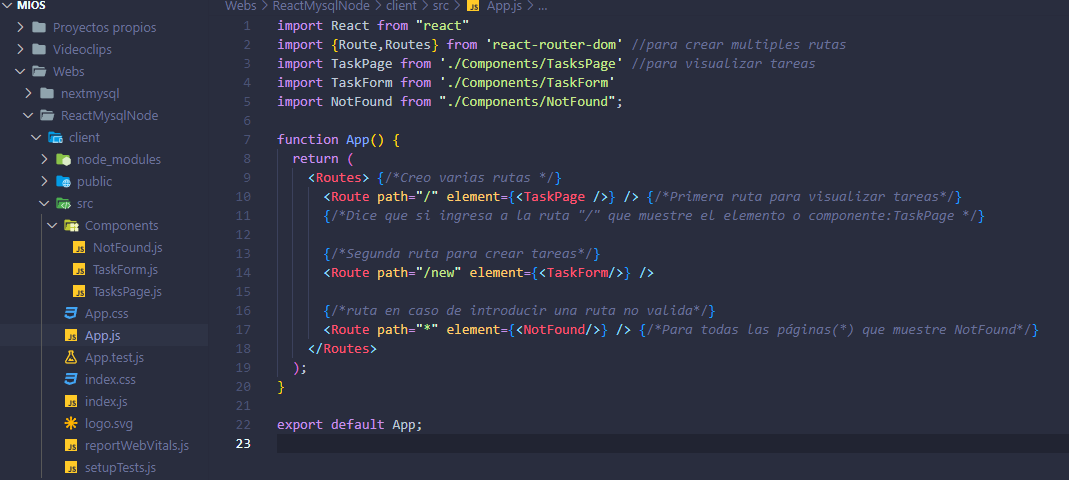
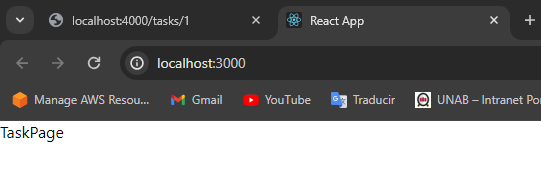
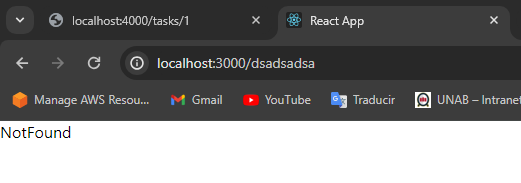
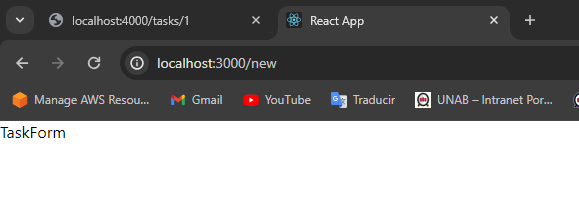
* Por ejemplo se puede simular un error para mostrar como se vería ese error y como cae el servidor:
  + Se agrega el throw new Error(“error de conexión”)
  + 
  + Daría este error, lo cual se cae el servidor, y esto es un error grave, por ende que afecta todas las peticiones.
  + 
* Para solucionar esto se hace uso de try, catch
  + Por ende esto funciona que en el try se coloca todo el código que quiero que haga correctamente, pero en caso de que en alguna parte falle, es automáticamente capturado por el catch, donde aun da una respuesta, pero no se cae el servidor y seguirá procesando peticiones
  + Por ende en caso de que ahora lance un error intencionado para probar al principio
  + 
  + Ahora ya no se cae el servidor, sino da un mensaje directamente
  + 
  + De esta forma las conexiones cliente pueden seguir funcionando en caso de error y no se cae el servidor
  + Existe también una librería que permite el manejo de error llamado http-errors que puede servir igual
* Finalmente así quedaría los CRUD
  + **GetTasks**
  + 
  + **getTask**
  + 
  + **createTask**
  + 
  + **updateTask**
  + 
  + **deleteTask**
  + 

Ahora como también se va a comenzar con el frontend entonces el server(backend) queda así:

* Orden:
  + 
* Base del backend : Index.js
* 
* Configuración del backend mysql : db.js
* 
* Configuración adicional en diferente archivo : conf.js (esto después puede servir como enviroment)
* 
* Routes (esto sirven para hacer las rutas de cada ejecución, es decir, en caso de hacer uso de get, post, put, delete que ruta deben tomar para hacer esas operaciones y funciones)
  + 
* Finalmente en controllers están las funciones que son llamadas por las rutas
  + **GetTasks**
  + 
  + **getTask**
  + 
  + **createTask**
  + 
  + **updateTask**
  + 
  + **deleteTask**
  + 
* Además se tiene una carpeta llamada database que sirve para crear las tablas a ocupar de la base de datos, que luego puede servir en producción
  + 

**CREACIÓN DEL FRONTEND**

* Como se va hacer uso de react se puede hacer uso de los siguientes comandos en la raiz para crear la carpeta client
  + Npx create-react-app client
  + O npm create vit (Que es lo mismo que el create react app pero para proyectos mas pequeños) aquí luego se escoge el nombre client y en react. En caso de utilizar esta forma luego hay que entrar a la carpeta client y darle a npm i para instalar los módulos
* Para ejecutar se hace uso de npm start
  + 
  + 
* Ahora como se van a tener varias rutas en el frontend como ruta /tareas/ , inicio, logeo, etc.. esto se hace uso mediante el modulo **react router dom** que se hace mediante **npm install react-router-dom@6**
  + A screen shot of a computer

    Description automatically generated
  + Esto obviamente en la ruta de client
  + Entonces también para que funcione, se solicita que el app del index.js este dentro de un BrowserRouter
    - 
  + Lo que se hace es la configuración de las rutas por ejemplo en app, donde se hace uso de varias rutas con Routes y luego se especifica cada una con Route:
  + 
  + Donde se puede ver que la ruta principal “/” hace uso del componente TaskPage. En caso de crear una tarea hace uso de la ruta “/new” y en caso de escribir cualquier url en el navegador que no esta asignada que lo envie a NotFound y diga que la ruta no fue encontrada “NotFound”
  + 
  + 
  + 
* Ya sabiendo como hacer las rutas ahora vendría el tema de las peticiones al backend

**ENVIAR PETICIONES BACKEND(CONEXIÓN FULLSTACK)**

* Enviar tareas mediante un formulario y que sea enviado al backend para que se cree en la base de datos
* Ahora lo que se hizo fue un navbar para que envie a la página “/new” que es la url de TaskForm donde se van a crear las tareas
* Crear tareas(TaskForm)
  + Se desarrolla un formulario de ingreso de datos por parte del usuario para crear la tarea, pero para ello se va a hacer una librería llamada Formik(<https://formik.org/>) que tiene funcionalidades para validar los datos ingresados como “ingreso correcto, mayúscula, validar estados, manejar estados, etc..” Para ello se hace uso de npm install formik en el Frontend.
  + Para funcionar se importa los componentes de formik, con import {Formik,Form} from ‘formik’. Siendo que el componente form permite la creación del formulario y Formik manejar los estados, es decir, que los datos ingresados en los inputs se guarden automáticamente en sus respectivas variables para luego enviarlas, es decir si se agrega un input con name title, al principio se crea una variable title y al agregar algo en title automáticamente se guarda, en los valoresiniciales.
  + Además es importante agregar el nombre de cada input o textarea para identificar al valor que se refiere y así se guarde en el Formik los valores iniciales.
  + De esta forma no es necesario crear un useState, un UseRef, handleSubmit, ni nada de ello, sino que viene directamente implementado en la librería por defecto

NOTA, los archivos a excepción de app.js y index.js fueron cambiados a jsx para hacer uso de funciones que puedan obtener y hacer ejecuciones de otras funciones dentro del return haciendo uso de {() => ( )}

CONEXIÓN AL BACKEND Y AL FRONTEND FULLSTACK:

* Para ello en el frontend fuera de componentes, en la raíz, se crea un carpeta llamado api
* Esta carpeta api es la que se encarga de tener funciones que se comunican con el backend, para ello se hace uso de una biblioteca como **axios**.
* Por ello se instala mediante npm install axios. La cual es una biblioteca que permite la conexión o enviar peticiones http sencillas entre el backend y frontend
* Ya instalado se crea un archivo dentro de api con las rutas, llamado por ejemplo task.api.js
  + Aquí se crea la función de crear tareas muy parecido a lo del backend, pero ahora en el server.
  + Donde aquí se importa el axios que es la librería para enviar y recibir peticiones del backend
  + Luego se crea la función que hace un post a la url que crea las tareas, que fue designada al crear el backend, que es un POST al http//localhost:400/tasks como lo veíamos aquí con postman:
  + A screenshot of a computer

    Description automatically generated
  + Y como el backend recibe json se envia la tarea como json:
  + A screen shot of a computer

    Description automatically generated
  + A computer screen shot of a program

    Description automatically generated
* Llamado frontend al axios(taskForm.jsx)
  + Ahora para el llamado se realiza en este caso al momento de hacer el submit, ya que se envia en ese momento las variables, de ello ahí esta el return de los valores guardados por el formulario de abajo
  + Por ende se crea un await que devuelve una respuesta a la cual llama a la función del api y envia la tarea que son los values.
  + A screen shot of a computer program

    Description automatically generated
  + A screenshot of a computer code

    Description automatically generated
  + A screen shot of a computer program

    Description automatically generated
  + El problema es que al momento de enviar la creación de una tarea da error, debido que al tener cors, se tiene el frontend en el 3000 y el backend en el 4000, por lo que se tiene 2 servidores y se tiene prácticamente 2 backend, el que esta en 4000 y la llamada del 3000 al backend. Por lo que se debe mencionar que permita la conexión sin restricción a nuestro puerto por temas de seguridad.
* Permitir conexión habilitada del frontend y backend quitando la restricción mediante cors
  + Para ello en el backend en la consola en server, se debe instalar cors mediante npm i cors
  + A screen shot of a computer

    Description automatically generated
  + Ahora en el package.json se ve cors:
  + A screen shot of a computer program

    Description automatically generated
  + Para utilizarlo en el index.js se importa primero cors con import cors from ‘cors’
  + Y luego se hace uso en el servidor con **app.use(cors())**
  + Posteriormente se puede ajustar los parámetros del cors ya que actualmente dice que cualquiera se puede conectar, para ello se hace uso del origin, de forma que solo se conecte el puerto 3000 que es el frontend en este caso:
  + A computer screen shot of a program

    Description automatically generated
* Creación de tarea
  + Ahora si nos vamos al frontend se crea la tarea
  + A screen shot of a computer

    Description automatically generated
  + A screen shot of a computer

    Description automatically generated
  + Ahora como extra para limpiar el formulario al crear la tarea se puede hacer de esta forma, haciendo uso de actions
  + A screen shot of a computer

    Description automatically generated
  + A screen shot of a computer program

    Description automatically generated
  + A computer code with many colorful lines

    Description automatically generated with medium confidence
* GETTAREAS
  + Ahora el gettareas se va a llevar a cabo en la página principal TasksPage.
  + Donde inicialmente primero hay que crear la función de api, ahora una axios con get a la url de tareas que seria de esta forma:
  + A screen shot of a computer

    Description automatically generated

1:36:12