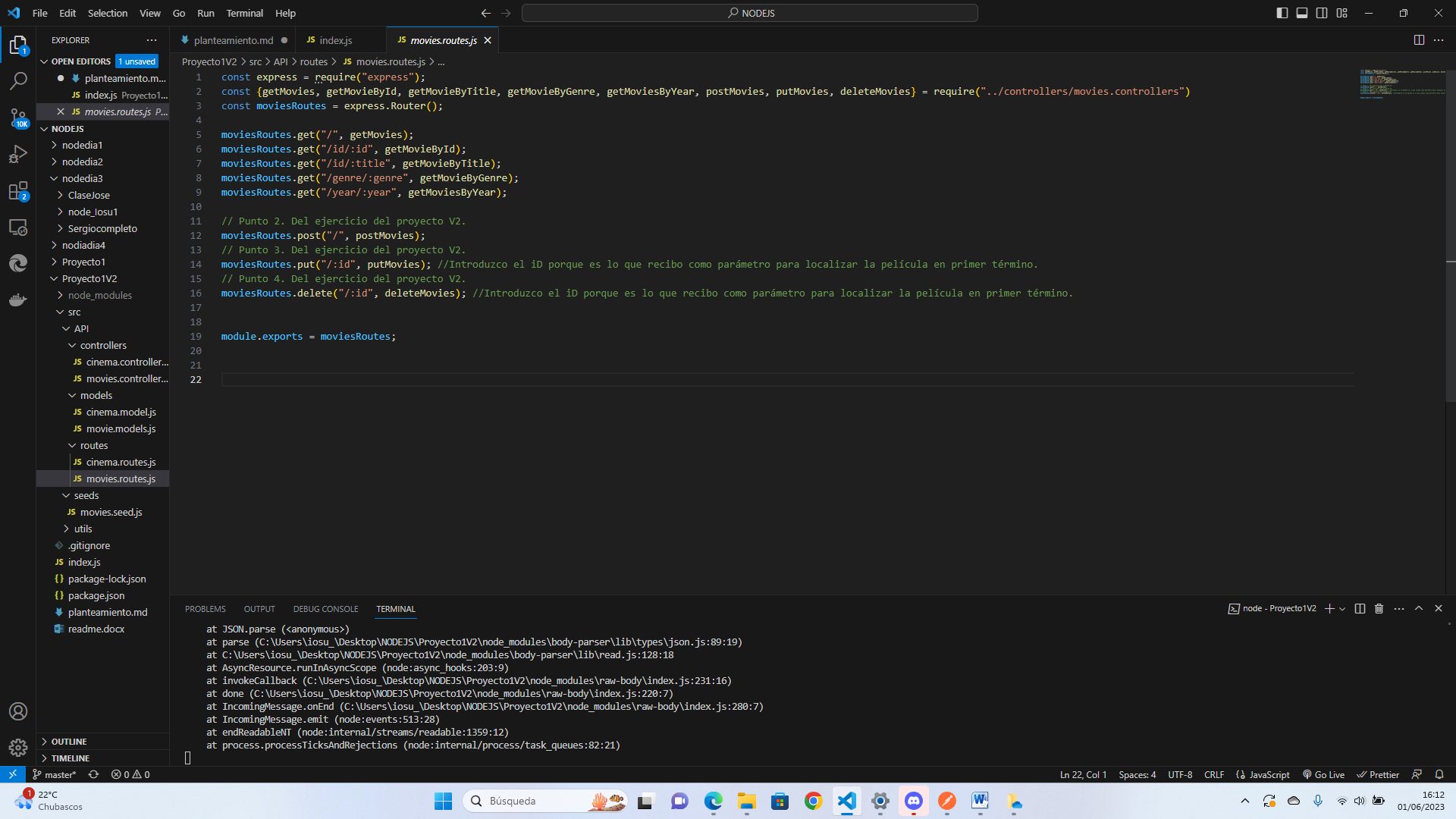
# SERVIDOR MOVIES II

## Punto 1: Crear un directorio de rutas y mover las rutas ya creadas a movie.routes.js.



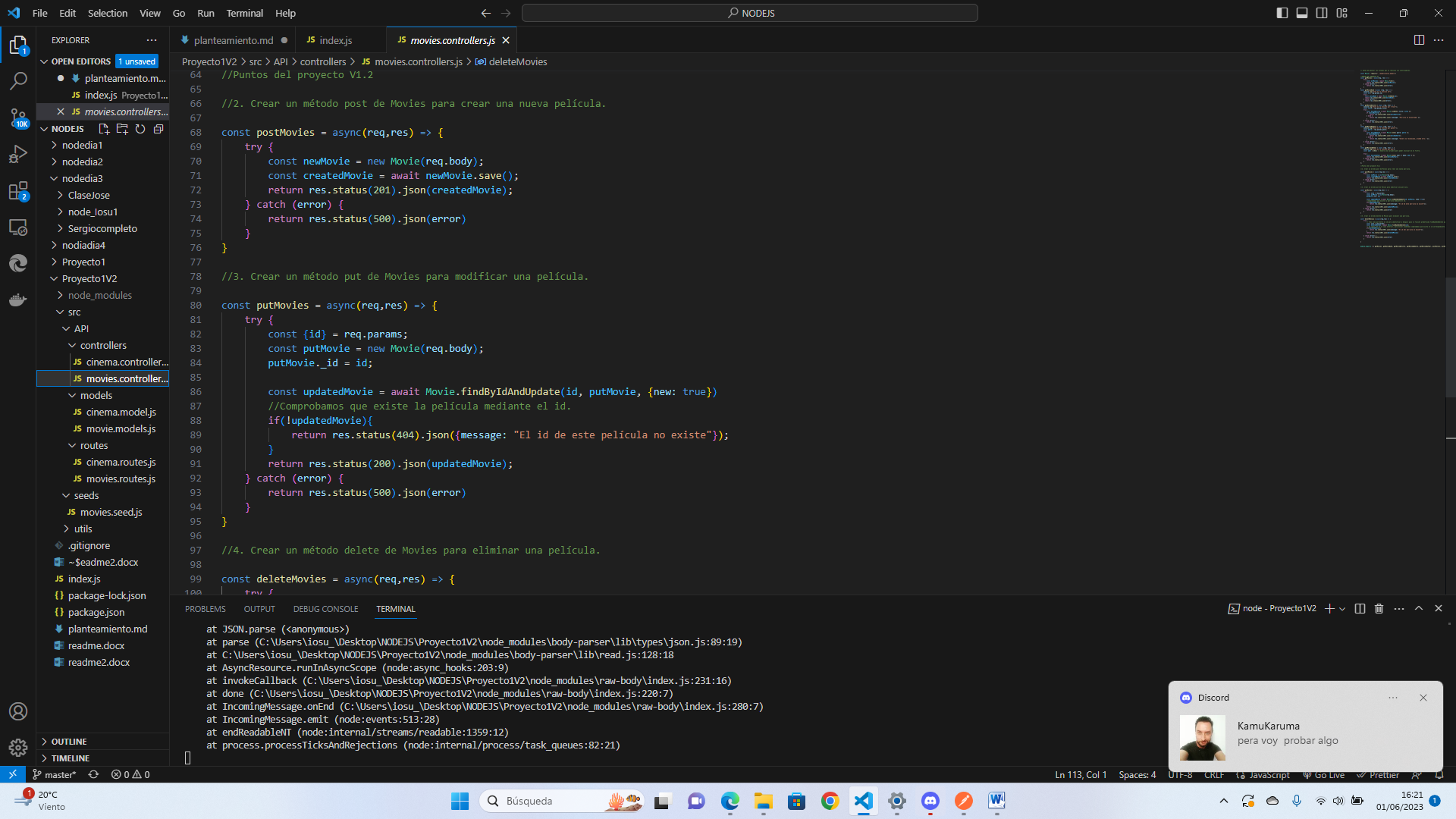
En la imagen sobre estas líneas se ven todas las rutas que se han generado que están relacionadas con las películas. OJO, NO OLVIDAR:

1. **Importar en el principio del documento los métodos creados en el archivo controllers** **que van a estar asociados a las rutas correspondientes**. En este caso tenemos: getMovies, getMovieById, getMovieByTitle, getMovieByGenre, getMoviesByYear, postMovies, putMovies y deleteMovies.
2. **Generar las rutas de manera coherente**. Ayudar al usuario con rutas coherentes que eviten errores. Ejemplo: movies/id/:id 🡪 Le decimos al usuario que lo que tendrá que poner será un iD.
3. **Al final del documento exportar las rutas para que puedan ser utilizadas desde el index.js**. Veamos las dos líneas clave para usar las rutas en el index.js:

const moviesRoutes = require('./src/API/routes/movies.routes')

app.use("/movies", moviesRoutes);

1. **Revisar el archivo controllers para verificar que tenemos bien organizadas las rutas**.



**Imagen**: Ejemplos de funciones controller que son usadas en el archivo movies.routes.js

## Punto 2: Crear un método put de Movies para modificar una película.

***Elementos de la función***

1. **const putMovies = async(req,res) => { ... }:** Se define una función llamada putMovies que toma dos parámetros: req y res. Esta función manejará la operación de modificación de una película existente.
2. **const {id} = req.params:** Aquí se extrae el valor del parámetro id de la solicitud. Este id representa el identificador único de la película que queremos modificar.
3. **const putMovie = new Movie(req.body):** Creamos una nueva instancia del modelo Movie con los datos proporcionados en el cuerpo de la solicitud (req.body). Suponemos que el cuerpo de la solicitud contiene los datos actualizados de la película.
4. **putMovie.\_id = id:** Asignamos el valor del id extraído de los parámetros de la solicitud al campo \_id de la instancia putMovie. De esta manera, aseguramos que la película modificada tenga el mismo ID que la original.
5. **const updatedMovie = await Movie.findByIdAndUpdate(id, putMovie, {new: true}):** Utilizamos el método findByIdAndUpdate de Mongoose para buscar la película con el ID proporcionado y actualizarla con los datos de putMovie. El tercer argumento {new: true} indica que queremos recibir la versión actualizada de la película como resultado.
6. **if(!updatedMovie){ ... }**: Verificamos si updatedMovie es null o undefined, lo que significa que no se encontró una película con el ID proporcionado. En ese caso, respondemos con un estado 404 (no encontrado) y un mensaje de error.
7. **return res.status(200).json(updatedMovie):** Si la película se actualiza correctamente, respondemos con un estado 200 (éxito) y enviamos el objeto de la película actualizada en formato JSON en la respuesta.

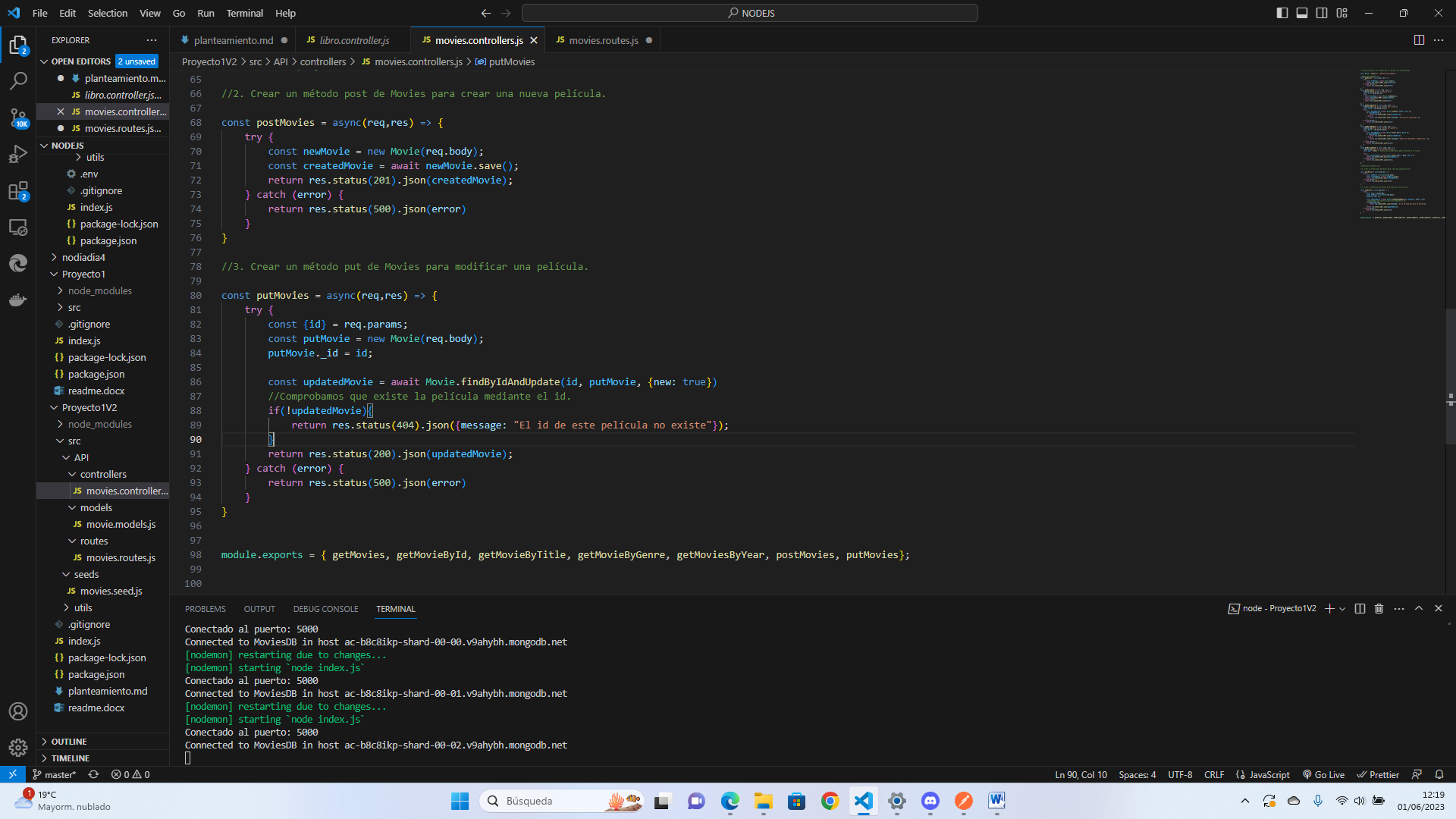
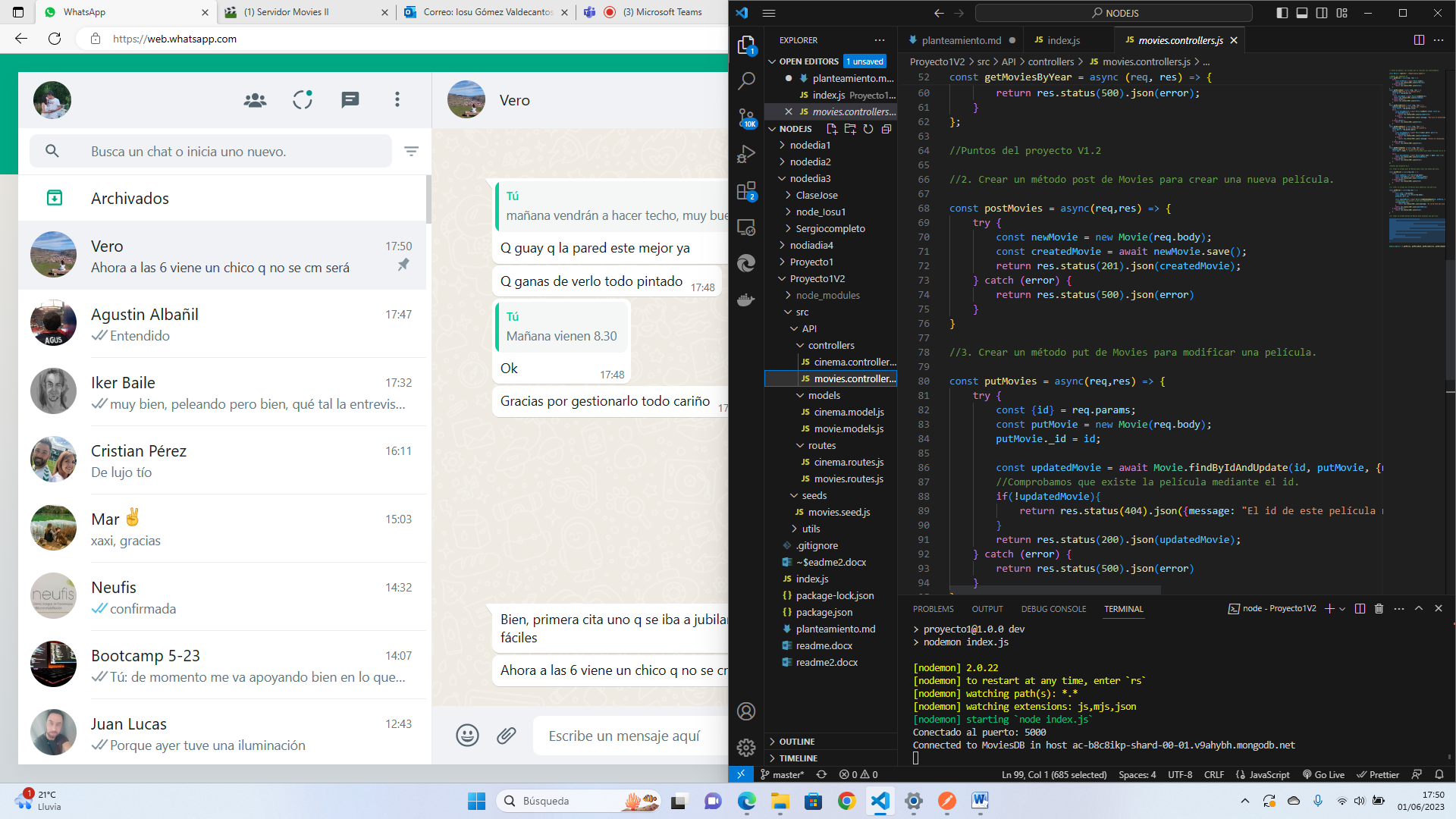


Imagen: Función putMovies extraída del archivo de texto.

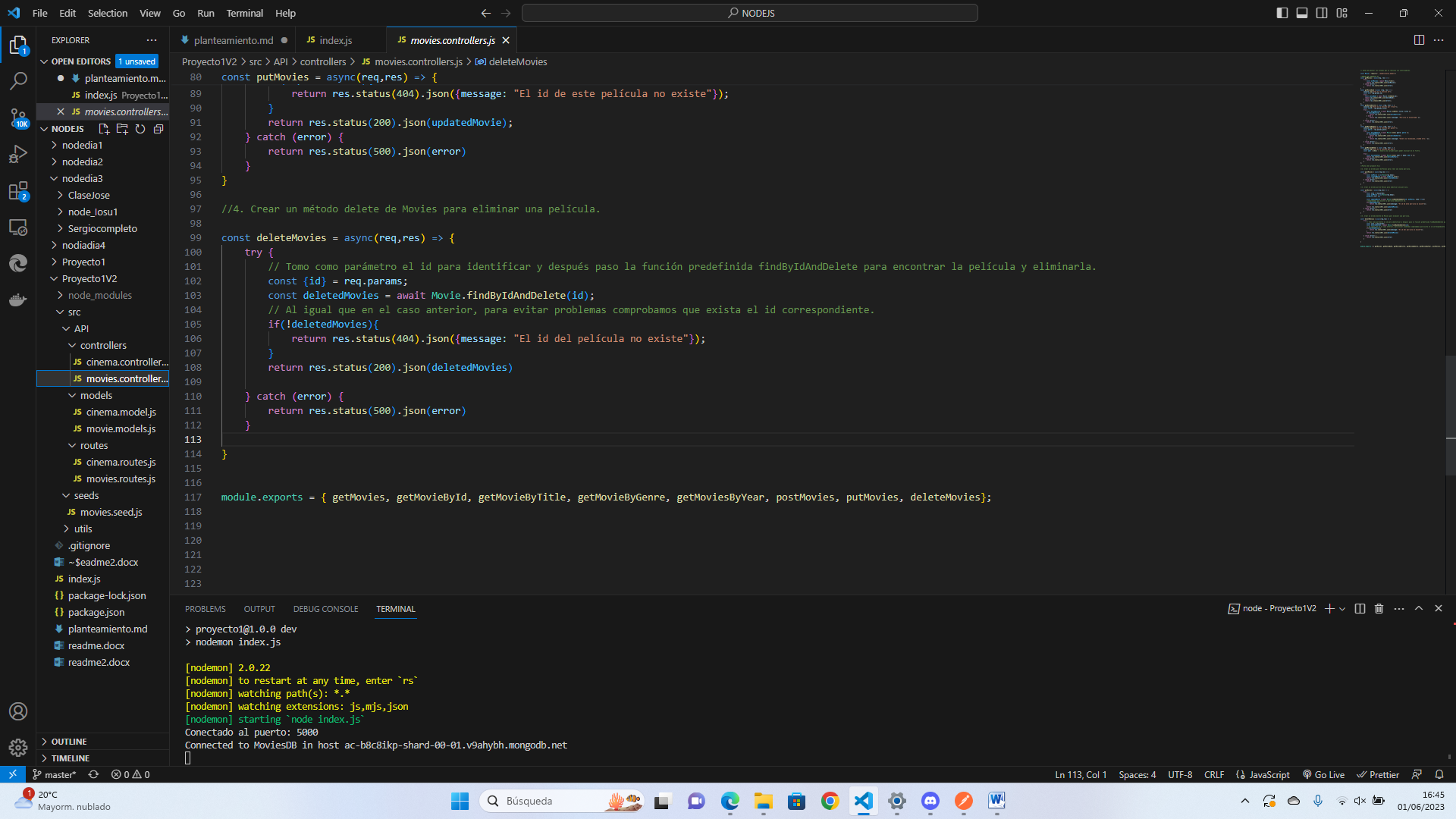
## Punto 3: Crear un método Post



Aquí está una explicación paso a paso del código:

1. La función `postMovies` toma dos parámetros, `req` (la solicitud HTTP) y `res` (la respuesta HTTP).
2. Dentro del bloque `try`, se crea una nueva instancia del modelo `Movie` utilizando los datos proporcionados en el cuerpo de la solicitud (`req.body`).
3. Luego, se utiliza el método `save()` para guardar la nueva película en la base de datos. Esto devuelve una promesa, por lo que se utiliza la palabra clave `await` para esperar a que se resuelva la promesa y se asigna el resultado a la variable `createdMovie`.
4. Si la película se crea correctamente, se envía una respuesta con un código de estado 201 (creado) y se envía el objeto de película creado en formato JSON utilizando el método `json()` de la respuesta.
5. En caso de que ocurra algún error durante el proceso, se captura en el bloque `catch` y se envía una respuesta con un código de estado 500 (error interno del servidor) y se envía el error en formato JSON utilizando el método `json()` de la respuesta.

## Punto 4: Crear un método delete de Movies para eliminar una película.

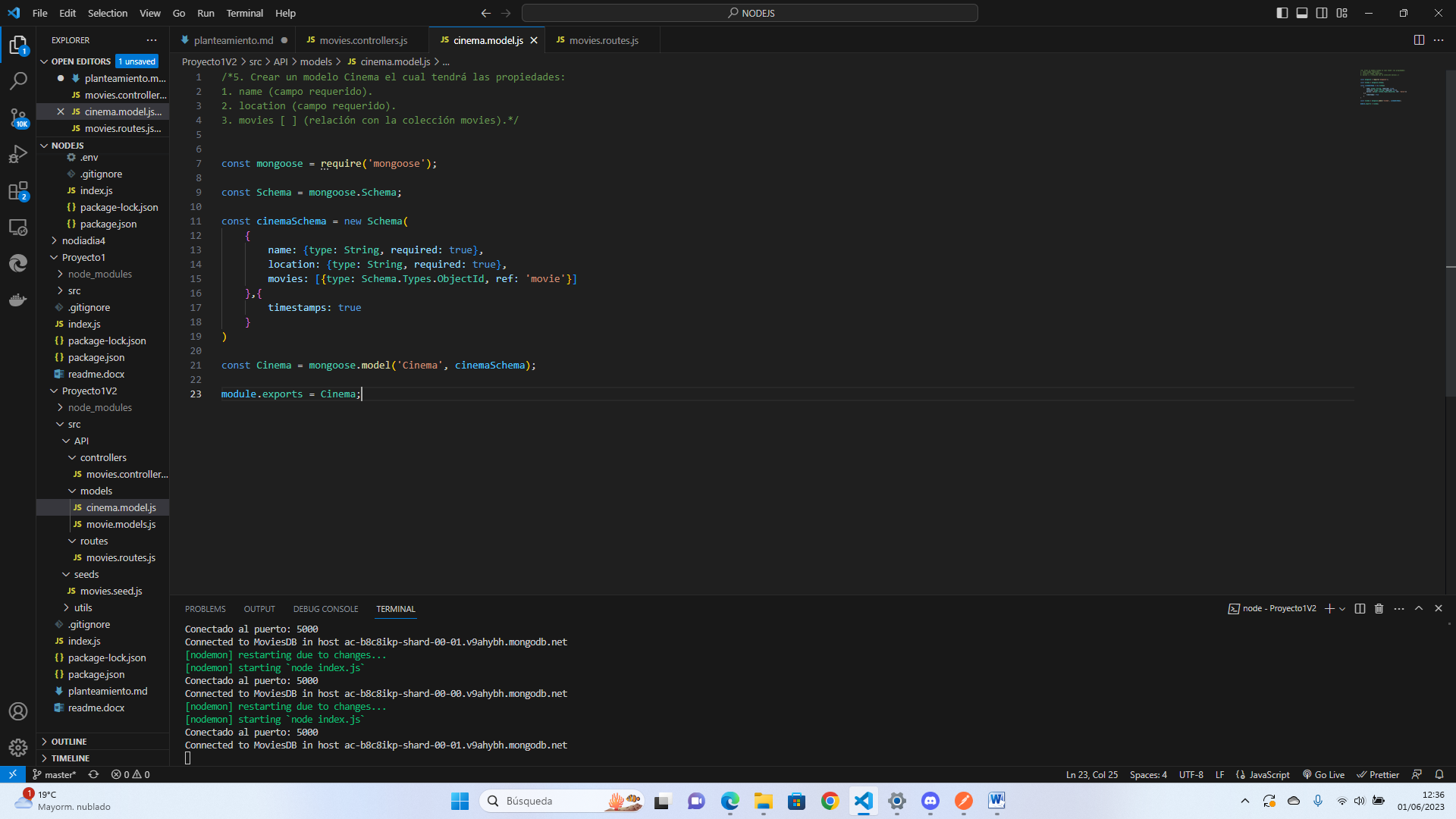


Como la función es similar a la anterior, la comentaré de manera resumida. La función deleteMovies es un controlador que se utiliza para eliminar una película de la base de datos.

***Veamos su funcionalidad:***

1. Toma el parámetro id de la solicitud, que se utiliza para identificar la película que se desea eliminar**. Para ello usamos la estructura const {id} = req.params**.
2. Utiliza la función predefinida findByIdAndDelete proporcionada por Mongoose para buscar y eliminar la película correspondiente en la base de datos. Al ser una función de búsqueda es una función de naturaleza asíncrona.
3. Si la película no existe (es decir, no se encuentra ningún documento con el ID especificado), devuelve una respuesta con un código de estado 404 y un mensaje indicando que el ID de la película no existe. ***Esto catchea el error y evita que la aplicación caiga ante la falta de un id***.
4. Si la película se elimina correctamente, devuelve una respuesta con un código de estado 200 y el objeto de la película eliminada.
5. Si ocurre algún error durante el proceso, devuelve una respuesta con un código de estado 500 y el error generado.

## Punto 5: Crear un modelo Cinema



Este código utiliza una biblioteca llamada "mongoose" para trabajar con una base de datos MongoDB. MongoDB es un tipo de base de datos no relacional que almacena la información de una manera similar a los documentos.

Aquí tienes una explicación más detallada de lo que hace cada parte del código:

1. **const mongoose = require('mongoose'):** Esta línea importa la biblioteca "mongoose" para poder utilizarla en el código. Es como abrir la caja de herramientas de mongoose para usar sus funciones.

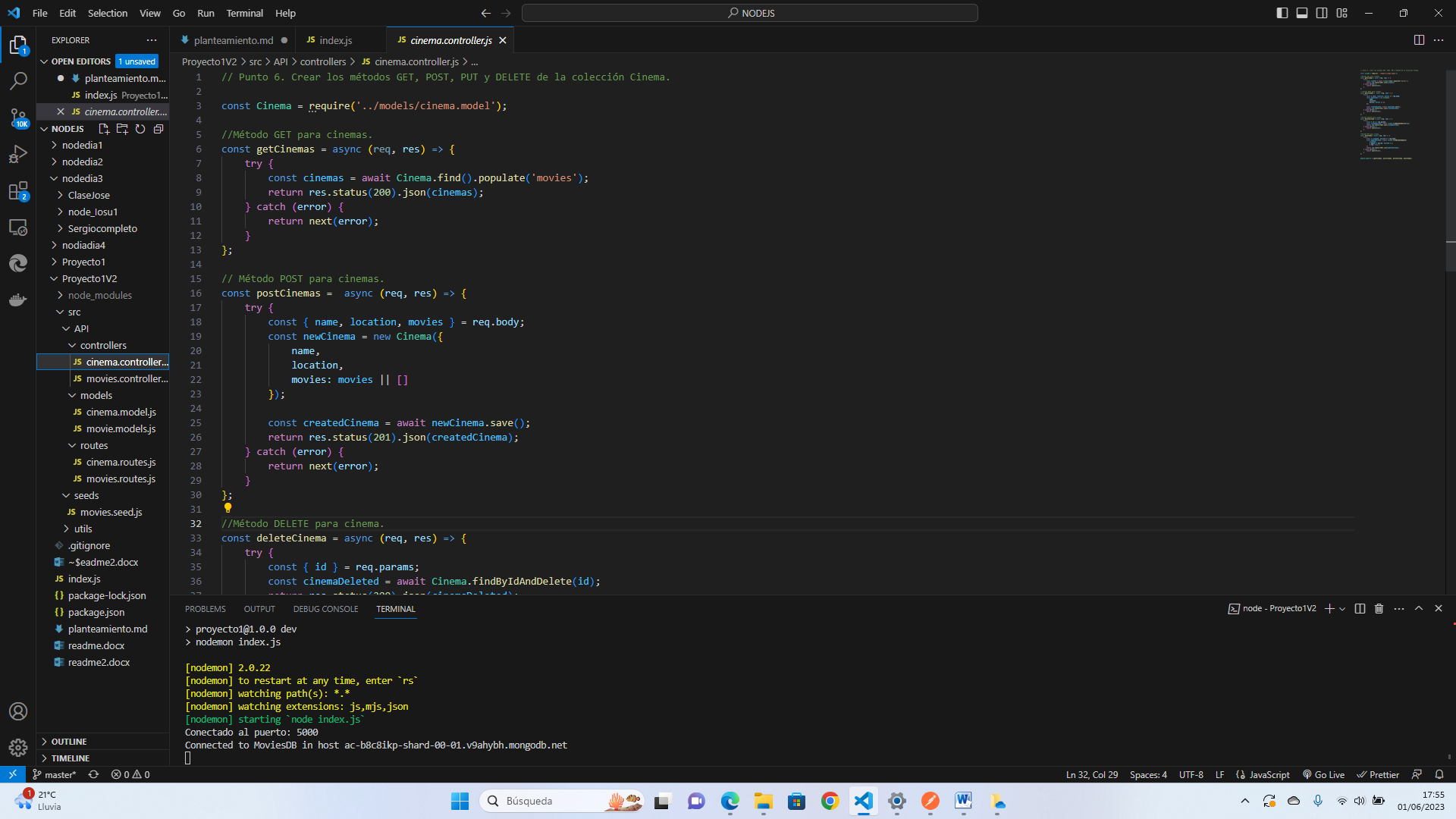
2. **const Schema = mongoose.Schema**: Aquí se crea una variable llamada "Schema" que contiene una referencia a la función "Schema" de mongoose. Esto permite crear esquemas de datos para estructurar la información que se almacenará en la base de datos.

**3. const cinemaSchema = new Schema({ ... }):** Se define un nuevo esquema llamado "cinemaSchema" utilizando la función "Schema" de mongoose. El esquema define la estructura de los datos que se almacenarán para representar un cine. En este caso, el esquema tiene tres campos: "name" (nombre), "location" (ubicación) y "movies" (películas). Cada campo tiene un tipo de dato y otras propiedades.

**4. const Cinema = mongoose.model('Cinema', cinemaSchema):** Aquí se crea un modelo llamado "Cinema" utilizando el esquema "cinemaSchema" definido anteriormente. El modelo actúa como una interfaz para interactuar con la colección de datos de la base de datos que representa los cines. Permite realizar operaciones como crear, leer, actualizar y eliminar datos relacionados con los cines.

**5. module.exports = Cinema:** Finalmente, se exporta el modelo "Cinema" para que pueda ser utilizado en otros archivos del proyecto. Esto permite a otros módulos acceder y utilizar el modelo para interactuar con los datos de los cines.

## Punto 6: Crear los métodos GET, POST, PUT y DELETE de la colección Cinema.



En la imagen superior mostramos los métodos GET y POST para Cinema.

***A comentar del método GET***: Funcionalidad similar a las funciones anteriormente explicadas. Resaltar la siguiente línea de código: const cinemas = await Cinema.find().populate(‘movies’).

Este código se utiliza para obtener todos los documentos de la colección cinemas de la base de datos y, al mismo tiempo, cargar los datos relacionados de la colección movies en cada uno de los documentos de cinemas.

Utilizando el modelo Cinema, se llama al método find() sin ningún filtro para obtener todos los documentos de la colección cinemas. ***Se utiliza el método populate('movies') para indicar a Mongoose que cargue los datos relacionados de la colección movies en el campo movies de cada documento de cinemas.*** Esto se logra a través del uso de referencias mediante el campo movies que contiene identificadores de documentos en la colección movies.

Al utilizar await antes de Cinema.find().populate('movies'), se espera a que la consulta se complete y los datos se carguen correctamente antes de continuar con la siguiente línea de código.

***El resultado de la consulta y la carga de datos se asigna a la variable cinemas.***

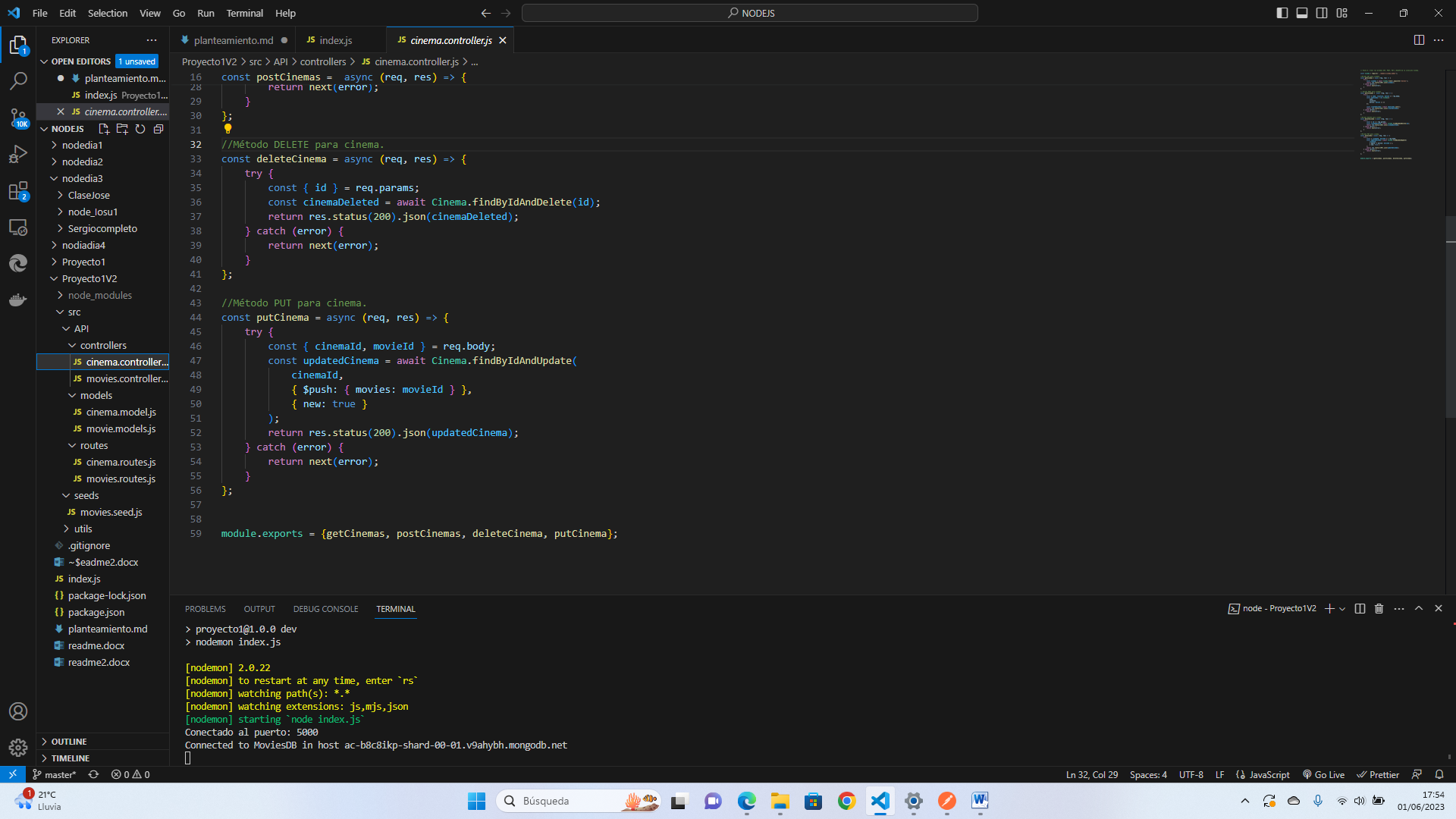
***A COMENTAR DEL POST: Método que se utiliza para solicitar la creación de un nuevo cine en la base de datos. Veamos el código explicado paso a paso.***

Se define una función asíncrona llamada postCinemas que toma los parámetros req y res. Estos parámetros representan como siempre el objeto de solicitud y el objeto de respuesta respectivamente.

Dentro del bloque try, se desestructuran las propiedades name, location y movies del cuerpo de la solicitud (req.body). Se crea una nueva instancia del modelo Cinema usando los valores desestructurados. El campo movies se establece en movies si existe, de lo contrario se establece en un arreglo vacío []. Este campo se rellanará relacionando las películas y los cines (queda explicado en el siguiente punto).

Se llama al método save() en la instancia de newCinema para guardar el nuevo cine en la base de datos. Si el guardado es exitoso, se devuelve una respuesta con el código de estado 201 (creado) y se envía el objeto createdCinema como una respuesta en formato JSON.

Si se produce un error durante el proceso, se captura en el bloque catch y se pasa al siguiente middleware utilizando next(error) para manejar el error de manera adecuada.



***El método DELETE para cinema tiene la siguiente explicación:***

1. Recibe una solicitud para eliminar un cine mediante su identificador (`id`) a través del objeto de solicitud (`req.params`).

2. Utiliza el método `findByIdAndDelete` del modelo `Cinema` para buscar y eliminar el cine correspondiente en la base de datos.

3. Si el cine es encontrado y eliminado exitosamente, devuelve una respuesta con el código de estado `200` (éxito) y envía el cine eliminado como respuesta en formato JSON.

4. En caso de que ocurra un error durante el proceso, el controlador lo captura en el bloque `catch` y lo pasa al siguiente middleware para su manejo adecuado.

***El método PUT para cinema tiene la siguiente explicación:***

1. Recibe una solicitud para actualizar un cine mediante los parámetros `cinemaId` y `movieId` a través del objeto de solicitud (`req.body`).

2. Utiliza el método `findByIdAndUpdate` del modelo `Cinema` para buscar el cine correspondiente por su identificador (`cinemaId`) y actualizar el campo `movies` agregando (`$push`) el identificador de la película (`movieId`).

3. El parámetro `{ new: true }` indica que se debe devolver la versión actualizada del cine después de la actualización.

4. Si la actualización es exitosa, devuelve una respuesta con el código de estado `200` (éxito) y envía el cine actualizado como respuesta en formato JSON.

5. En caso de que ocurra un error durante el proceso, el controlador lo captura en el bloque `catch` y lo pasa al siguiente middleware para su manejo adecuado.

## Punto 7 y 8: Insertar dos documentos en la colección cinema que tengan al menos una relación con una película.

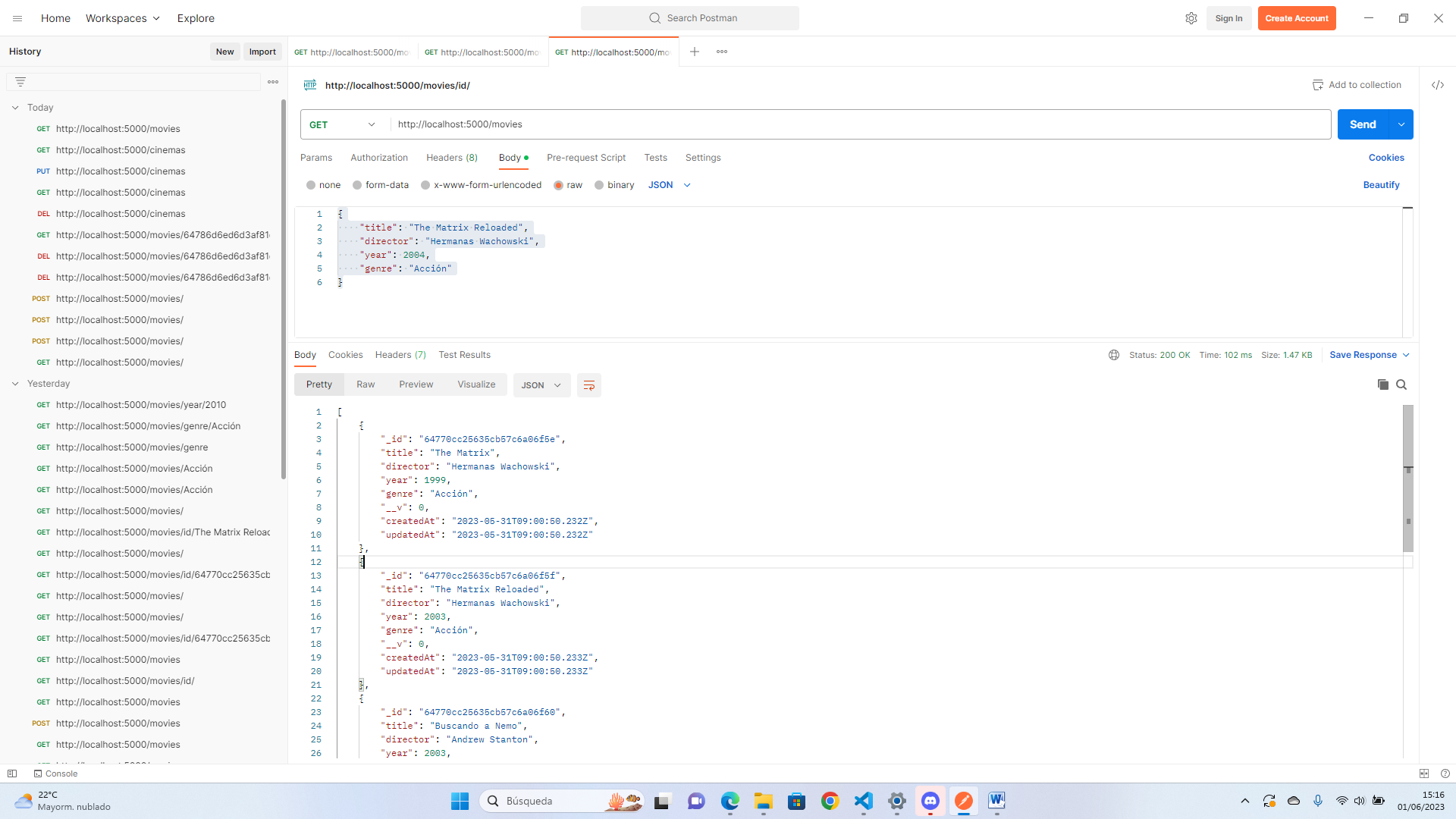
**METODOLOGÍA**

1. Abre Postman y asegúrate de tener tu servidor en ejecución y escuchando en el puerto correspondiente (en este caso, el puerto 5000).

2. Primero, obtendremos los ID de las películas que queremos asociar a los cines. Para ello, haremos una solicitud GET a la ruta `/movies` para obtener todas las películas. Debemos seleccionar el método GET en Postman y proporcionar la URL completa: `http://localhost:5000/movies`.

3. Haz clic en el botón "Enviar" y recibirás la respuesta con todas las películas. Asegúrate de revisar la respuesta para obtener los ID de las películas que deseas asociar a los cines.

Para este ejemplo usamos estas dos:

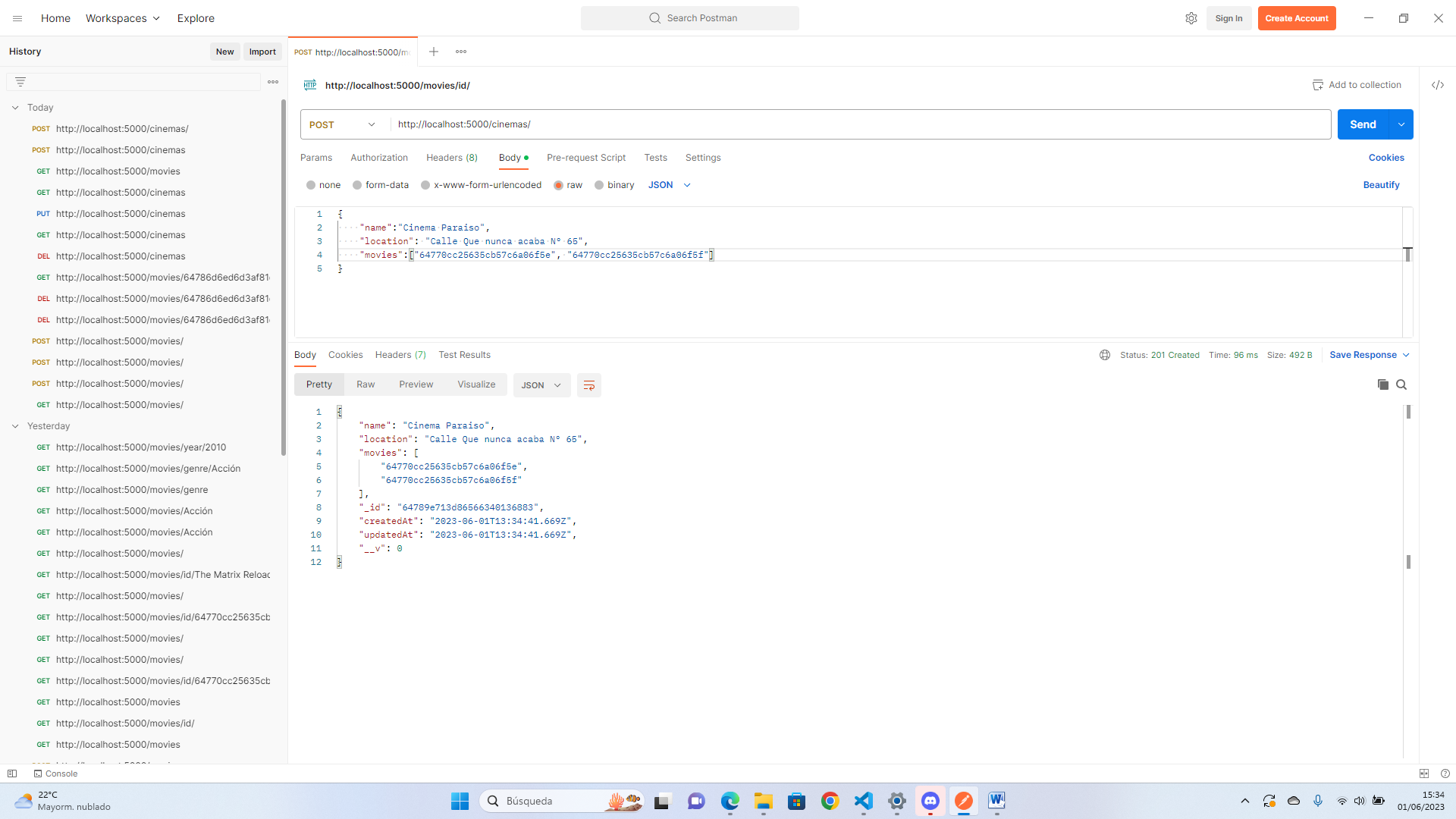


**Id**: 64770cc25635cb57c6a06f5e.

**Id**: 64770cc25635cb57c6a06f5f.

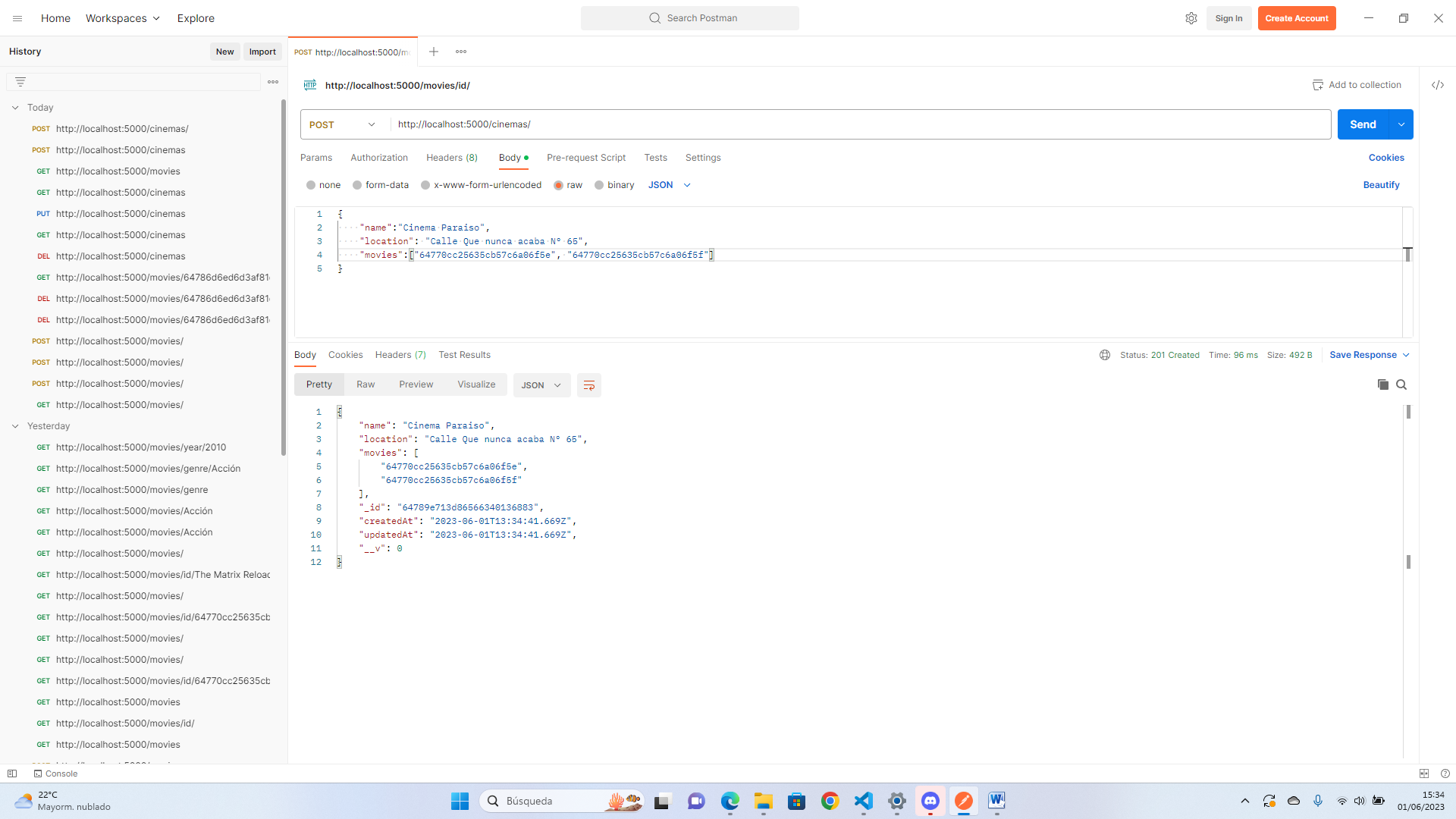
4. Ahora, podemos hacer las solicitudes POST para crear los cines. En Postman, selecciona el método POST y proporciona la URL completa para crear un cine: `http://localhost:5000/cinemas`.

5. En la pestaña "Cuerpo", selecciona el formato "JSON" y proporciona los datos del cine que deseas crear. Aquí tienes un ejemplo de cómo debería ser la estructura de datos para crear un cine con dos las películas asociadas:



Así nos quedaría previo a dar al botón “Send”.

6. Hacemos clic en el botón “Send" y recibimos la respuesta con el cine creado.



7. Repetimos los pasos y nos inventamos otro cine.

Con estos pasos, realizamos dos solicitudes POST a la API para crear dos cines, cada uno con dos películas asociadas. OJO: Tenemos que asegurarnos de proporcionar los datos correctos mediante consultas get apropiadas que nos proporcionen los ID de interés.

El catcheo de los errores se realizado evitando la aparición de casos extraños que hicieran crashear a la aplicación.