**¿Qué es Express?**

Se usa para levantar un servidor web usando Express.js (un framework para Node.js). Este servidor está diseñado para ser la "espalda" (backend) de una aplicación, es decir, la parte que se encarga de recibir peticiones (como cuando un usuario se registra, inicia sesión, o pide información de artistas), procesarlas, y enviar respuestas.

app.use(express.json());//usa express.json() para que el servidor pueda recibir datos en formato JSON del frontend, es decir, que el servidor pueda recibir datos en formato JSON y los convierta a un objeto de JavaScript para manipularlos en los controladores del backend

// Rutas

app.use('/api/auth', authRoutes); // Rutas. Establece /api/auth como la ruta base o prefijo para todas las rutas que están definidas dentro del archivo authRoutes.js. Por ejemplo, si tienes una ruta de registro en authRoutes.js que es /register, la ruta completa para acceder a ella sería /api/auth/register.

app.listen(port, () => {// inicia tu servidor Express, haciendo que

empiece a escuchar peticiones en el puerto que le indicaste. Cuando procesa una petición, envía una respuesta de vuelta al cliente

  console.log(`Servidor corriendo en http://localhost:${port}`);

});

**¿Qué es Dotenv?**

**permite usar "variables de entorno". Las variables de entorno son como cajitas donde guardamos información secreta o que puede cambiar (como la contraseña de la base de datos o el puerto del servidor). No se sube a github o sitios publicos, queda guardado en nuestro proyecto personal.**

dotenv.config();

**Busca un archivo llamado .env en la carpeta principal de tu backend. Dentro de ese archivo .env es donde guardas información sensible o de configuración, como la contraseña de la base de datos, una clave secreta para los tokens JWT, o el puerto en el que quieres que corra el servidor. dotenv.config() lee ese archivo y hace que esas variables estén disponibles dentro de tu código**

**¿Qué es Cors?**

CORS es como un "permiso de comunicación". Cuando tu frontend (que corre, por ejemplo, en localhost:5500) quiere hablar con tu backend (que corre en localhost:3000), el navegador, por seguridad, necesita que el backend le dé permiso explícito para escuchar peticiones desde cualquier origen (direccion).

const cors = require('cors');//importamos cors para permitir el acceso a

la API desde el frontend, le dice al servidor que acepte peticiones de cualquier origen

En resumen para tus notas:

* server.js es el corazón de tu backend.
* Usa express para manejar las peticiones.
* Carga configuraciones con dotenv.
* Usa cors para permitir que tu frontend hable con él.
* Entiende JSON con express.json().
* Sirve tus archivos HTML/CSS/JS del frontend con express.static.
* Dirige las peticiones de API (como login, registro) a tus archivos de rutas (authRoutes, userRoutes).
* Finalmente, se pone a "escuchar" en un puerto con app.listen.

Para que es JWT:

El JWT (JSON Web Token) que generas en tu backend después de un login exitoso es como una credencial digital o un "Pase VIP" que le das al usuario. Este pase le permite demostrar quién es y qué permisos tiene sin necesidad de que el servidor tenga que buscarlo en la base de datos cada vez que hace algo.

* Creación (Login): Se empaqueta información del usuario (id, role) en un "pase" (token), se firma con una clave secreta y se le da una fecha de caducidad. Se envía al navegador.
* Utilización (Peticiones Protegidas): El navegador envía el "pase" de vuelta con cada petición. El servidor usa la clave secreta para verificar que el pase sea auténtico y no esté caducado. Si es válido, confía en la información del pase para saber quién es el usuario y qué puede hacer.

Para que es LocalStorage:

Utiliza para guardar el token JWT (y el rol del usuario) después de un login exitoso. Esto permite que el token esté disponible para ser enviado en futuras peticiones a rutas protegidas y ayuda a mantener la sesión del usuario activa entre visitas (mientras el token sea válido).

* ARCHIVO DB.JS
* Trae las herramientas: Importa las librerías necesarias (mysql2 para MySQL y dotenv para secretos).
* Lee los secretos: Carga la información de conexión (usuario, contraseña, etc.) desde tu archivo .env.
* Prepara la conexión: Crea un objeto (db) que sabe cómo conectarse a tu base de datos MySQL usando esos secretos**. const db = mysql.createConnection({...}) crea la instancia db con toda la configuración necesaria (host, usuario, etc.), pero sin conectarse todavía.**
* **db.connect(...)**: Es la acción de **intentar** establecer la comunicación con tu base de datos MySQL. db.connect(...) inicia el intento de conexión.
* err => { ... } (El Callback): Es la instrucción que le das para después de que termine de marcar. Le dices: "Cuando sepas si te contestaron o no (si la conexión funcionó o falló), haz lo siguiente:".

if (err): "Si te dio error (err tiene algo), muéstrame el error por pantalla (console.error)".

else: "Si no hubo error (err está vacío), dime que todo fue bien (console.log)".

* module.exports = db; hace que tu objeto de conexión db esté disponible para ser

importado y usado por otros archivos de tu proyecto

db.connect(err => {

    if (err) {

        // Si hay un error, lo muestra en la consola

        console.error('Error de conexión a la base de datos:', err);

    } else {

        // Si todo va bien, avisa que se conectó

        console.log('Conectado a la base de datos');

    }

});

module.exports = db;

**Entender flujo de autenticación**

Fase 1: El Usuario Quiere Entrar (Frontend)

* El Usuario Interactúa (en index.html para Login o register.html para Registro):

El artista va a tu página de login o de registro.

Llena los campos del formulario (nombre de usuario, contraseña).

Cuando el artista hace clic en "Login" o "Registrarse", el JavaScript de esa página se activa.

Previene el envío normal: event.preventDefault(); evita que la página se recargue.

Recolecta los datos: Toma el nombre de usuario y la contraseña que el artista escribió.

Validaciones básicas: Verifica si los campos están vacíos o si las contraseñas coinciden (en el caso del registro).

Prepara el "Paquete" para el Servidor: Si todo está bien, usa fetch para enviar una petición POST a tu backend.

Destino: /api/auth/login o /api/auth/register.

Contenido del Paquete (body): Los datos del usuario (nombre de usuario y contraseña) se convierten a formato JSON con JSON.stringify().

Aviso de Contenido (headers): Se incluye 'Content-Type': 'application/json' para decirle al servidor que el paquete viene en JSON.

Fase 2: El Servidor Recibe y Procesa la Solicitud (Backend)

El Servidor Escucha en server.js:

Tu servidor Express está siempre atento a nuevas peticiones.

Middleware express.json(): Antes de que la petición llegue a tu lógica específica, este "ayudante" toma el paquete JSON que envió el frontend y lo convierte en un objeto JavaScript fácil de usar, que estará disponible en req.body.

Enrutamiento: app.use('/api/auth', authRoutes); dirige todas las peticiones que empiezan con /api/auth hacia tu archivo de rutas de autenticación (que a su vez las pasará al authController).

El Controlador de Autenticación Decide authController.js:

Aquí es donde ocurre la "magia" de la autenticación.

Si es un REGISTRO (exports.register):

Extrae datos: Toma username y password de req.body.

Rol fijo: Asigna role = "artist".

Verifica si existe: Llama a User.findByUsername (del modelo User.js) para ver si el nombre de usuario ya está en la base de datos.

Si ya existe, devuelve un error (HTTP 400).

Si no existe, llama a User.create.

Dentro de User.create (en User.js): La contraseña se "esconde" (hashea) usando bcrypt.hash().

Se guarda el nuevo usuario (con la contraseña hasheada y el rol) en la base de datos.

Devuelve un mensaje de éxito (HTTP 201).

Si es un LOGIN (exports.login):

Extrae datos: Toma username y password de req.body.

Busca al usuario: Llama a User.findByUsername para encontrar al usuario en la base de datos.

Si no lo encuentra, devuelve un error (HTTP 400 - "Credenciales inválidas").

Compara contraseñas: Si encuentra al usuario, usa bcrypt.compare() para comparar la contraseña que envió el artista con la contraseña hasheada que está guardada en la base de datos.

Si las contraseñas no coinciden, devuelve un error (HTTP 400 - "Credenciales inválidas").

¡Éxito! Genera el "Pase VIP" (Token JWT): Si las contraseñas coinciden:

Usa jwt.sign() para crear un token.

Contenido del Token (Payload): Incluye id, username y role del usuario.

Firma Secreta: Se firma con tu process.env.JWT\_SECRET.

Expiración: Se establece un tiempo de vida (ej. "1h").

Envía Respuesta al Navegador: Devuelve un JSON con el token y el role del usuario.

Fase 3: El Navegador Recibe la Respuesta y Actúa (Frontend)

JavaScript Maneja la Respuesta de nuevo en register.js

El fetch recibe la respuesta del servidor.

const data = await response.json(); convierte la respuesta JSON del servidor en un objeto JavaScript.

Si fue un REGISTRO exitoso:

Muestra un mensaje de éxito.

Usa setTimeout para esperar un par de segundos y luego redirige al usuario a index.html (la página de login).

Si fue un LOGIN exitoso:

Muestra un mensaje de éxito.

Guarda el "Pase VIP":

localStorage.setItem('token', data.token); guarda el token en el "bolsillo" del navegador.

localStorage.setItem('userRole', data.role); guarda el rol.

Usa setTimeout para esperar un poco y luego redirige (actualmente a index.html, pero podría ser a un panel de artista).

Si hubo un error (en registro o login): Muestra el mensaje de error que envió el servidor.

Fase 4: Usando el "Pase VIP" (Token JWT) para Acciones Futuras

Accediendo a Rutas Protegidas:

Cuando el usuario (ya logueado) intenta acceder a una parte de tu aplicación que requiere autenticación (por ejemplo, su perfil, o si un admin quiere ver todos los usuarios):

El JavaScript del frontend tomará el token guardado en localStorage.

Enviará este token al servidor en la cabecera Authorization de la petición HTTP (normalmente como Authorization: Bearer <token\_aqui>).

El Servidor Verifica el "Pase VIP" (Backend - c:\Users\Usuario\Documents\ISFT 151\4to ANIO cursada\Practicas Profesionalizantes 2\proyecto-1 - copia\MUSICAA\backend\middlewares\authMiddleware.js):

El middleware protect se ejecuta antes que el controlador de la ruta protegida.

Extrae el token: Saca el token de la cabecera Authorization.

Verifica el token: Usa jwt.verify() y tu JWT\_SECRET para:

Asegurarse de que la firma sea válida (que el token sea auténtico y no haya sido modificado).

Asegurarse de que el token no haya expirado.

Si el token es válido, protect extrae la información del usuario (el payload: id, username, role) y la añade a req.user.

Opcional: Middleware authorize: Si la ruta también usa authorize(['admin']), este middleware revisará req.user.role para ver si el usuario tiene el permiso necesario.

Si todo está bien, la petición continúa hacia el controlador de la ruta (por ejemplo, una función en userController.js.

Si el token es inválido o el usuario no tiene el rol adecuado, se devuelve un error (HTTP 401 o 403).

Conceptos Clave del JavaScript del Navegador (Frontend):

fetch (El Cartero Moderno):

Qué es: Una herramienta de JavaScript para enviar "cartas" (peticiones) a tu servidor (backend) y recibir "respuestas".

Para qué lo usas: Para mandar los datos del formulario de registro o login a tu servidor para que los procese.

Importante: Es asíncrono. Cuando fetch envía la carta, no se queda esperando la respuesta; sigue con otras cosas y te avisa cuando la respuesta llega.

async function (Función Preparada para Esperar):

Qué es: Una forma especial de escribir una función en JavaScript (como async (event) => { ... }).

Para qué sirve: Le dice a JavaScript: "Dentro de esta función, es posible que tengamos que esperar a que terminen cosas que tardan (como fetch). Prepárate para pausar esta función sin detener todo lo demás".

Regla: Solo puedes usar await dentro de una función que hayas marcado como async.

await (El "Espera Aquí"):

Qué es: Una palabra que usas dentro de una async function, justo antes de algo que tarda (como await fetch(...)).

Para qué sirve: Le dice a la función async: "Haz esto (fetch), y espera aquí mismo hasta que termine y te dé un resultado. Cuando lo tengas, sigue con la siguiente línea dentro de esta función".

Importante: Mientras await pausa la función async, el resto de tu página y otros scripts pueden seguir funcionando. No congela todo.

setTimeout (El Temporizador):

Qué es: Una herramienta de JavaScript que actúa como un reloj con alarma.

Para qué sirve: Le dices: "Oye, Temporizador, cuando pasen X segundos (o milisegundos), por favor, haz esta otra cosa (ejecuta esta función)".

En tu código: Lo usas para mostrar un mensaje de éxito ("¡Login correcto!") y esperar un par de segundos antes de cambiar de página. Así, el usuario tiene tiempo de leer el mensaje.

Importante: También es asíncrono. Pones la alarma y tu código sigue adelante; la tarea se ejecutará cuando el tiempo se cumpla.

**localStorage vs. sessionStorage**

Ambos son mecanismos de almacenamiento web del lado del cliente, pero tienen diferencias fundamentales:

|  |  |  |
| --- | --- | --- |
| Característica | localStorage | sessionStorage |
| Persistencia | Los datos persisten indefinidamente hasta que el usuario los borra manualmente o tu código los elimina. Sobreviven al cierre del navegador y reinicios del sistema. | Los datos se eliminan automáticamente cuando la "sesión" de navegación termina (es decir, cuando se cierra la pestaña o ventana del navegador). Si abres la misma URL en una nueva pestaña, será una nueva sesión. |
| Alcance | Compartido entre todas las pestañas y ventanas del mismo origen (dominio). | Limitado a la pestaña/ventana que lo creó. No se comparte entre pestañas, incluso si son del mismo origen. |
| Capacidad de Almacenamiento | Generalmente 5MB por origen. | Generalmente 5MB por origen. |
| Accesibilidad | Ambos son accesibles vía JavaScript (window.localStorage y window.sessionStorage). | Ambos son accesibles vía JavaScript (window.localStorage y window.sessionStorage). |

¿Qué son las LOGICAS DE NEGOCIOS?

La lógica de negocio son todas las reglas, validaciones y procesos que hacen que tu aplicación funcione como un negocio real. No es solo técnico (como conectar a la base de datos), sino que define cómo se deben hacer las cosas.

¿Qué patron arquitectónico de N capas estoy aplicando?

Esoy aplicando el **patrón arquitectónico de N capas**, específicamente el clásico **Modelo-Vista-Controlador (MVC)**, que es una variante de N capas.  
Te justifico por qué:

**1. Separación en Capas**

* **Modelo (Model):**  
  Carpeta models/ en backend (User.js, [Artist.js](vscode-file://vscode-app/c:/Users/Usuario/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), Event.js, db.js).  
  Aquí defines la lógica de acceso a datos y las funciones para interactuar con la base de datos.
* **Controlador (Controller):**  
  Carpeta controllers/ (userController.js, [artistController.js](vscode-file://vscode-app/c:/Users/Usuario/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html), etc.).  
  Aquí procesas las peticiones, validas datos, aplicas reglas de negocio y llamas a los modelos.
* **Rutas (Router):**  
  Carpeta routes/ (userRoutes.js, artistRoutes.js, etc.).  
  Aquí defines los endpoints de la API y conectas cada ruta con su controlador.
* **Vista (View):**  
  Carpeta [frontend](vscode-file://vscode-app/c:/Users/Usuario/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html" \o ") con archivos HTML y CSS, y la carpeta js/ para la lógica de interacción con la API.  
  El frontend consume la API y muestra los datos al usuario, sin mezclar lógica de backend.