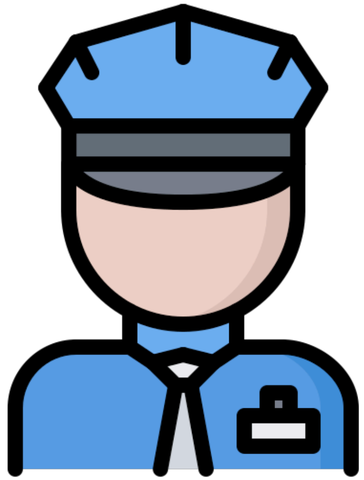
AUTENTICACIÓN JWT NODE JS Y EXPRESS

Ing. David Lozada

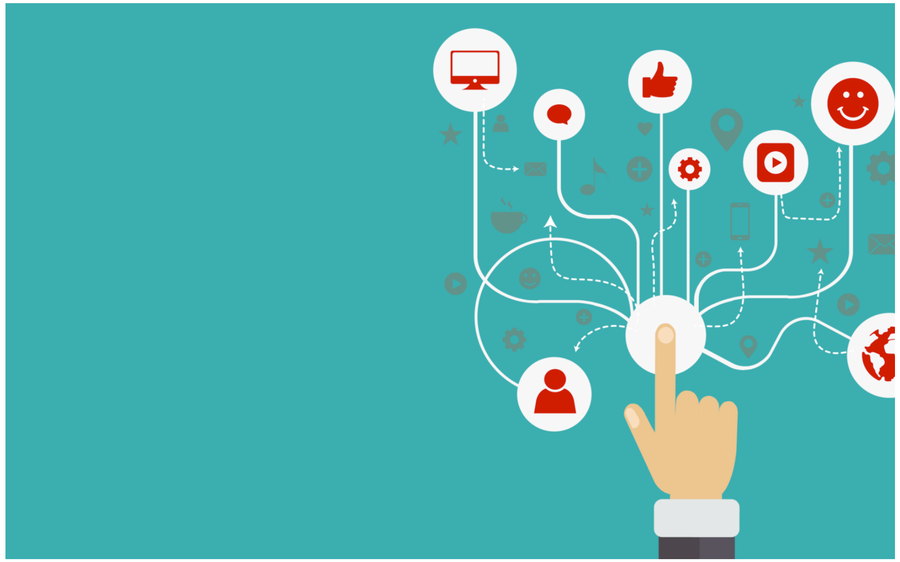
AUT + NODE JS

Autenticacion VS Autorización

- Autenticación: quien eres, y darte acceso a la aplicación.

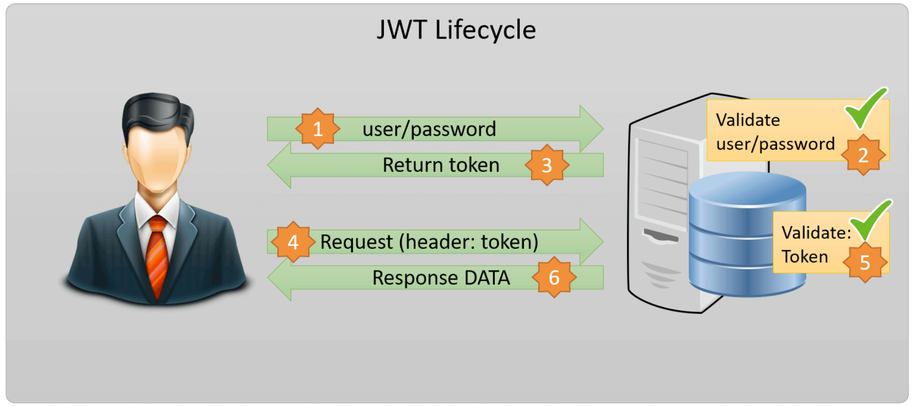


- Autorización: Es la gestion de permisos



JSON WEB TOKEN (JWT)

JWT, o **JSON Web Token**, es un estándar abierto (RFC 7519) que define un formato compacto y seguro para transmitir datos entre partes como un objeto JSON. Generalmente se usa para **autenticación y autorización** en aplicaciones web y móviles, permitiendo la creación de sesiones sin la necesidad de manejar sesiones en el servidor.



Autenticación

JWT es muy útil en la autenticación de usuarios. Cuando un usuario se autentica correctamente (por ejemplo, proporcionando el nombre de usuario y la contraseña correctos), el servidor genera un JWT y se lo envía al cliente. El cliente guarda este token, usualmente en la memoria local del navegador (localStorage) o en cookies.

- A partir de este momento, en cada solicitud que el cliente hace al servidor, puede enviar el JWT en los encabezados de la petición (por ejemplo, en el encabezado `Authorization` con el formato `Bearer <token>`).
- El servidor valida el JWT y, si es válido, puede autorizar el acceso a recursos protegidos.

Autorización

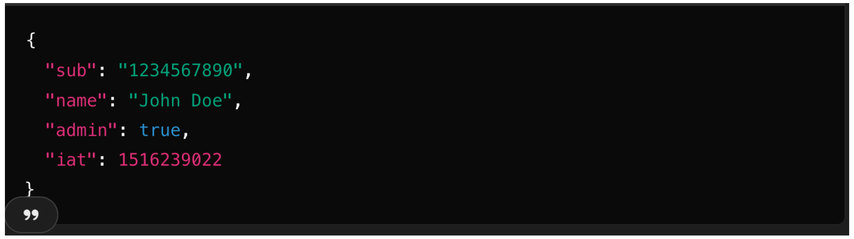
Además de la autenticación, JWT también se usa para la **autorización**. Una vez autenticado, el JWT puede incluir información sobre los permisos o roles del usuario, permitiendo al servidor verificar si el usuario tiene acceso a ciertos recursos o acciones.

Estructura de un JWT

Un JWT se compone de tres partes, separadas por puntos (`.`):

1. **Header** (Encabezado): Define el tipo de token y el algoritmo de cifrado, como HMAC SHA256 o RSA.

2. **Payload** (Cuerpo): Contiene los datos o "claims" que se quieren transmitir. Aquí es donde se almacena la información relevante, como el ID del usuario, el rol o permisos y el tiempo de expiración.



3. **Signature** (Firma): Se genera a partir de los dos primeros componentes (header y payload) y una clave secreta, usando el algoritmo especificado en el encabezado. Esta firma permite verificar la autenticidad del token.

Un JWT completo se vería algo así:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiw...<firma>

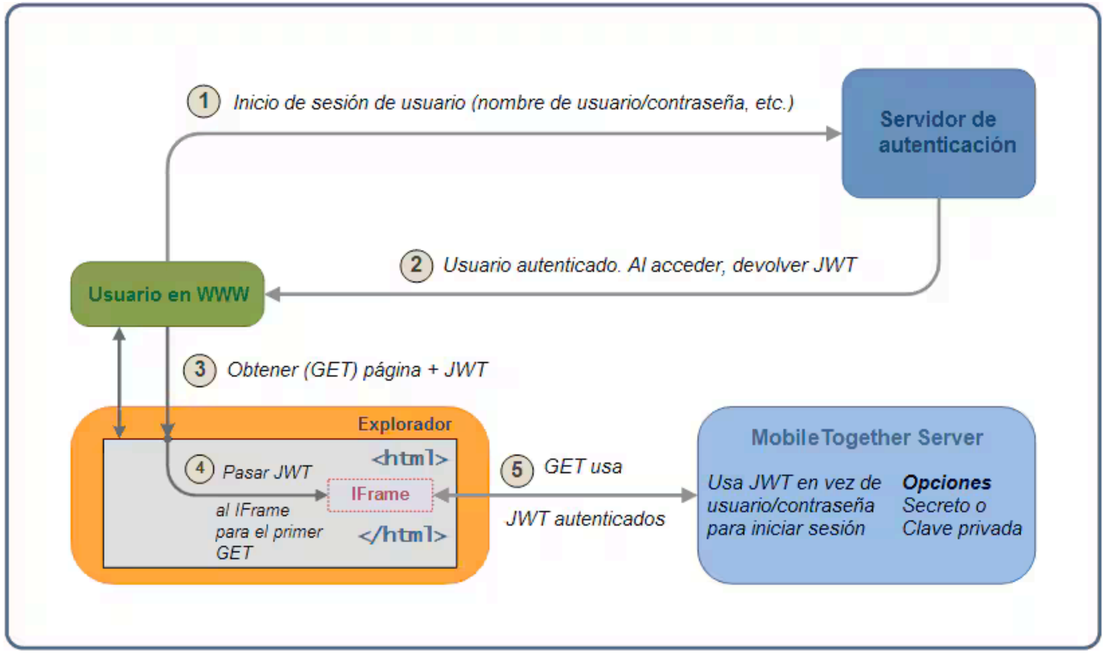
Funcionamiento Básico de JWT

Generación: Al iniciar sesión correctamente, el servidor genera un JWT que contiene la información del usuario y lo envía al cliente.

Almacenamiento: El cliente guarda el JWT localmente (por ejemplo, en localStorage o en una cookie segura).

Solicitud: Para acceder a recursos protegidos, el cliente envía el JWT en cada solicitud al servidor, usualmente en el encabezado `Authorization`.

Validación: El servidor verifica la firma del JWT usando la clave secreta. Si es válido, concede acceso al recurso; si es inválido o ha expirado, el acceso es denegado.

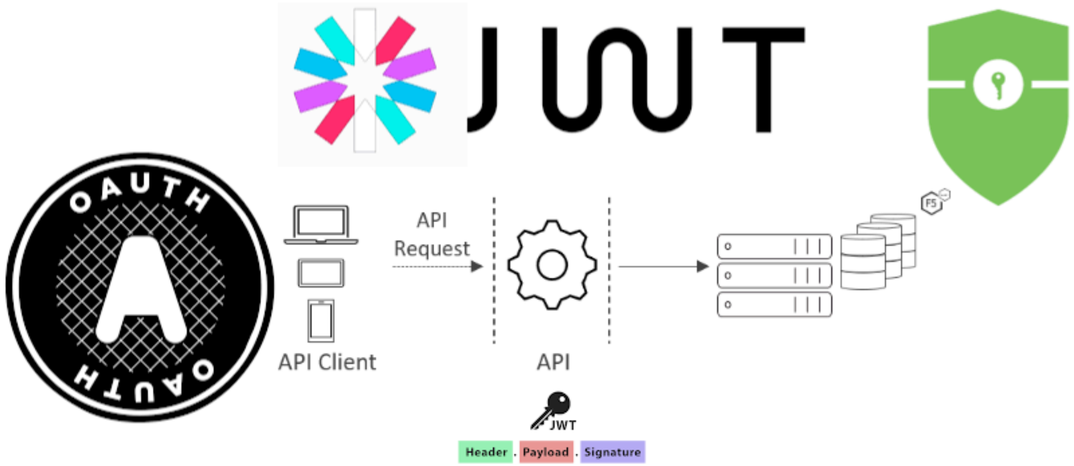


Ventajas de JWT

Sin estado en el servidor: JWT permite mantener sesiones sin que el servidor almacene información de sesión, lo que reduce la carga en el servidor.

Escalable: Funciona bien en aplicaciones distribuidas y microservicios.

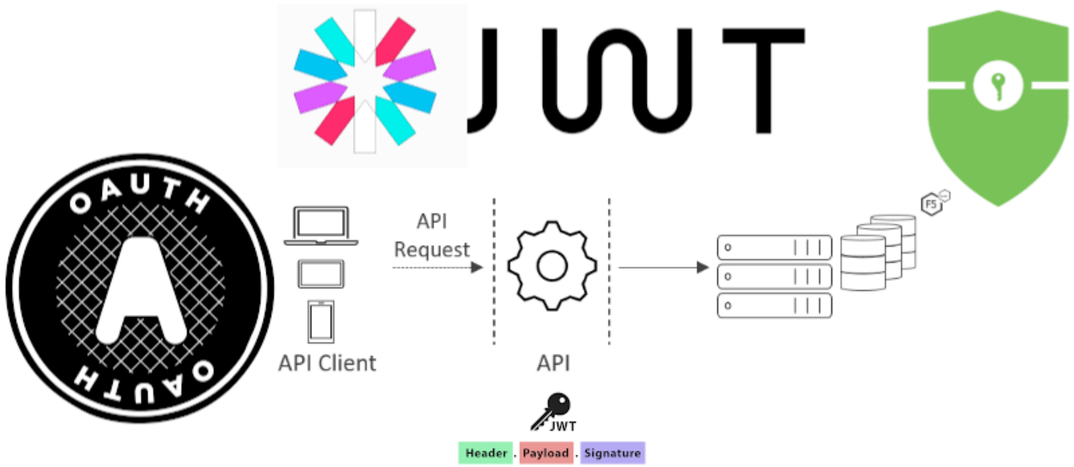
Compacto y seguro: Al ser una cadena de texto, es fácil de transmitir y seguro, siempre que se cifre y maneje de manera adecuada.



Desventajas de JWT

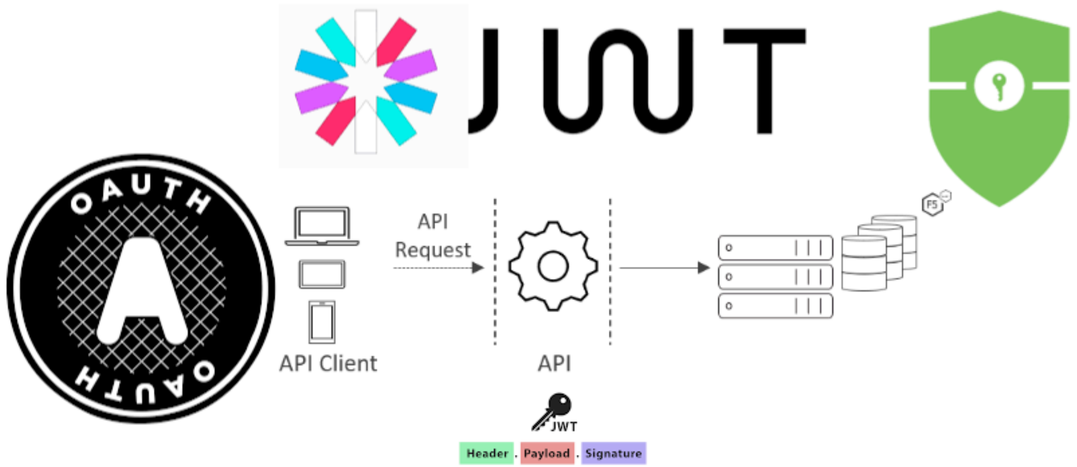
No es revocable: Una vez emitido, es difícil invalidar un JWT antes de que expire (a menos que se guarde una lista de tokens revocados en el servidor).

Riesgo si es robado: Si un atacante obtiene el JWT, puede usarlo hasta que expire.



Uso Común en Desarrollo

JWT es ampliamente usado con frameworks y librerías como Express.js, Laravel, Django, entre otros, para implementar autenticación sin estado en APIs REST.



Proyecto

Front

Vamos a crear un proyecto fron en react y back con node y express, el fornt debe cumplir con todo el estandar de inicio de session.

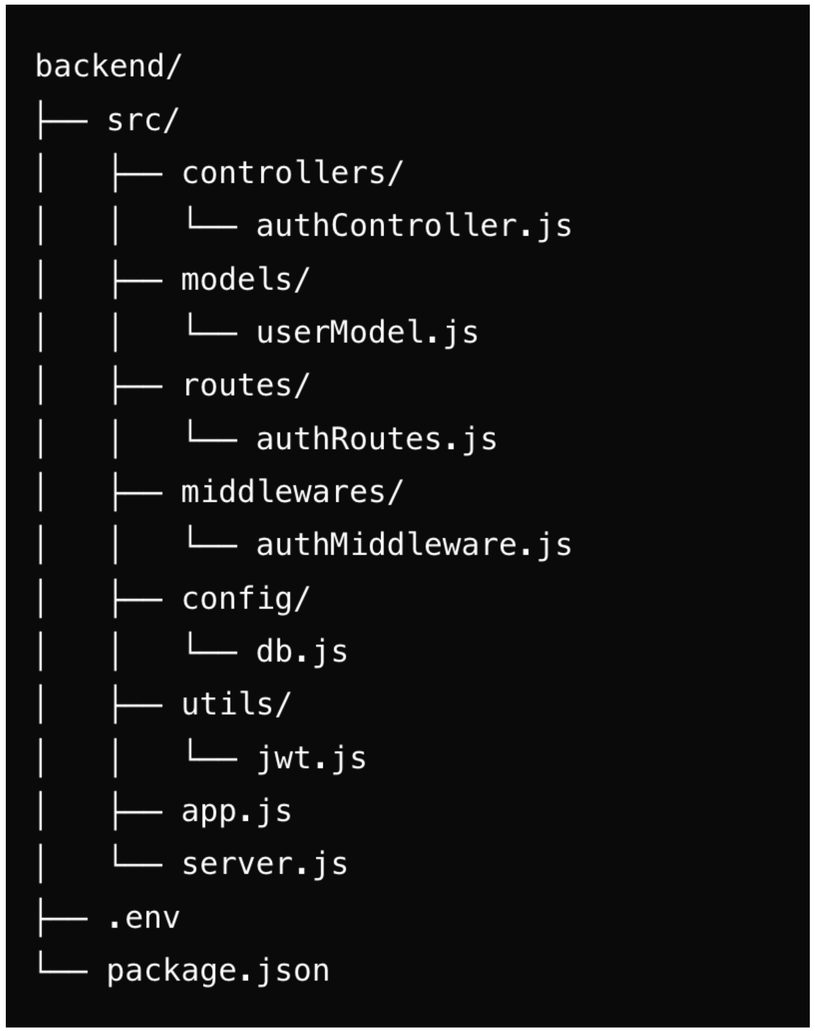
Para realizar este proyecto debes tener conocimientos en React, npm, note js, Express, JWT, bases de datos relacional / no relaciona

Para crear un sistema completo de inicio de sesión con un frontend en React y un backend en Node.js con Express, utilizaremos JWT para manejar la autenticación sin estado. Este enfoque es estándar en aplicaciones web modernas y cumple con buenas prácticas de seguridad.

AQUI VA LA PRESENTACION DEL FRONT

BACKEND CON NODE JS Y EXPRESS

Primero, configuremos el servidor backend que manejará el registro, inicio de sesión y verificación de usuarios autenticados.



Configuración Básica

npm init -y
npm install express mongoose bcryptjs jsonwebtoken dotenv cors

Conexión a la Base de Datos (Mongo)

En `src/config/db.js`:

```
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGO_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });
    console.log('Base de datos conectada');
  } catch (error) {
    console.error('Error al conectar a la base de datos:', error);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Este código está configurado para conectar una aplicación de Node.js con una base de datos MongoDB utilizando la biblioteca `mongoose`, que facilita la interacción con MongoDB desde JavaScript. Veamos cada parte del código:

1. **`const mongoose = require('mongoose');`**: Importa el módulo `mongoose`, que se utiliza para trabajar con MongoDB en Node.js de manera más intuitiva y organizada.
2. **`const connectDB = async () => { ... };`**: Define una función asíncrona llamada `connectDB` para conectar la aplicación a la base de datos. La función usa `async` porque la conexión con la base de datos es una operación asincrónica.
3. **Intento de conexión con `try`**:
   - **`await mongoose.connect(process.env.MONGO_URI, { ... });`**: Utiliza el método `connect` de `mongoose` para establecer una conexión con MongoDB. Aquí, `process.env.MONGO_URI` es una variable de entorno donde se almacena la URI de conexión (especifica la ubicación de la base de datos).
   - Los parámetros:
     - **`useNewUrlParser: true`**: Permite que `mongoose` use el nuevo analizador de URL para conectarse a MongoDB.
     - **`useUnifiedTopology: true`**: Habilita el motor de administración de conexiones de `mongoose`, lo que proporciona mejor compatibilidad y estabilidad.
4. Si la conexión es exitosa, se muestra el mensaje: **`console.log('Base de datos conectada');`**.
5. **Captura de errores con `catch`**:
   - Si ocurre algún error durante la conexión, el bloque `catch` captura la excepción.
   - **`console.error('Error al conectar a la base de datos:', error);`**: Muestra el mensaje de error en la consola.
   - **`process.exit(1);`**: Finaliza el proceso de Node.js con un código de error `1`, indicando que ocurrió un problema.
6. **`module.exports = connectDB;`**: Exporta la función `connectDB` para que se pueda importar y usar en otras partes de la aplicación.

Modelo de Usuario

En `src/models/userModel.js`:

```
const mongoose = require('mongoose');
const bcrypt = require('bcryptjs');

const userSchema = new mongoose.Schema({
  username: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});

userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) return next();
  this.password = await bcrypt.hash(this.password, 10);
  next();
});

userSchema.methods.comparePassword = async function (password) {
  return await bcrypt.compare(password, this.password);
};

module.exports = mongoose.model('User', userSchema);
```

Este código define un modelo de usuario en MongoDB utilizando `mongoose` y aplica lógica para encriptar contraseñas y comparar contraseñas de manera segura. Desglosemos cada parte:

1. **`const mongoose = require('mongoose');`**: Importa `mongoose`, que es una biblioteca de modelado de datos para MongoDB.
2. **`const bcrypt = require('bcryptjs');`**: Importa `bcryptjs`, que es una biblioteca para encriptar y comparar contraseñas de manera segura.
3. **Definición del esquema de usuario `userSchema`**:

Aquí se define el esquema de la colección `User` en MongoDB con dos campos:

- **`username`**: tipo `String`, obligatorio (`required: true`) y único (`unique: true`), lo que significa que no pueden existir dos usuarios con el mismo nombre de usuario.
- **`password`**: tipo `String`, obligatorio (`required: true`), para almacenar la contraseña del usuario.

4. Middleware `pre` para encriptar la contraseña antes de guardar el usuario:
Aquí se usa el método `pre` de `mongoose` para ejecutar una función antes de guardar el documento en la base de datos. La función toma `next` como parámetro para continuar el flujo de ejecución.

- **`if (!this.isModified('password')) return next();`**: Esta línea verifica si el campo `password` ha sido modificado. Si no lo ha sido, llama a `next()` para pasar al siguiente paso y evita volver a encriptar la contraseña.
- **`this.password = await bcrypt.hash(this.password, 10);`**: Encripta la contraseña antes de guardarla. El `10` es el factor de "salting" que hace el cifrado más seguro, al añadir un pequeño valor adicional en cada contraseña.
- **`next();`**: Continua con el proceso de guardado.

5. Método `comparePassword` para verificar contraseñas:
Aquí se agrega un método personalizado llamado `comparePassword` al esquema `userSchema`. Este método compara una contraseña sin encriptar (por ejemplo, ingresada por el usuario en el login) con la contraseña encriptada almacenada en la base de datos.

- **`bcrypt.compare(password, this.password);`**: Usa `bcrypt` para comparar la contraseña ingresada (`password`) con `this.password` (la contraseña guardada en el documento), devolviendo `true` si coinciden o `false` en caso contrario.

6. **`module.exports = mongoose.model('User', userSchema);`**: Exporta el modelo `User` creado a partir del esquema `userSchema`. Esto permite que el modelo sea utilizado en otras partes de la aplicación para crear, leer, actualizar y eliminar usuarios.

Controlador de Autenticación

En `src/controllers/authController.js`:

```
const User = require('../models/userModel');
const jwt = require('jsonwebtoken');

const generateToken = (user) => {
  return jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });
};

exports.register = async (req, res) => {
  try {
    const { username, password } = req.body;
    const user = new User({ username, password });
    await user.save();
    res.status(201).json({ message: 'Usuario registrado con éxito' });
  } catch (error) {
    res.status(400).json({ error: error.message });
  }
};

exports.login = async (req, res) => {
  try {
    const { username, password } = req.body;
    const user = await User.findOne({ username });
    if (!user || !(await user.comparePassword(password))) {
      return res.status(401).json({ error: 'Credenciales inválidas' });
    }
    const token = generateToken(user);
    res.json({ token });
  } catch (error) {
    res.status(500).json({ error: error.message });
  }
};
```

Este código define las funciones para registrar y autenticar usuarios en una aplicación Node.js, utilizando JWT para la generación de tokens de acceso. A continuación, se explica cada parte:

1. **Importar Dependencias**:
   - **`const User = require('../models/userModel');`**: Importa el modelo `User` desde la carpeta `models`, donde se define la estructura del usuario.
   - **`const jwt = require('jsonwebtoken');`**: Importa `jsonwebtoken`, que se usa para crear y verificar tokens JWT (JSON Web Tokens), los cuales se emplean para gestionar la autenticación de usuarios.
2. **Función `generateToken`**:

Esta función genera un token JWT para un usuario:

- **`jwt.sign({ id: user._id }, process.env.JWT_SECRET, { expiresIn: '1h' });`**: Usa `jsonwebtoken` para crear un token basado en el `id` del usuario (`user._id`).
- **`process.env.JWT_SECRET`**: Es una clave secreta almacenada en las variables de entorno, que se utiliza para firmar el token de forma segura.
- **`{ expiresIn: '1h' }`**: Define que el token expirará en 1 hora, limitando su tiempo de validez.

3. **Función `register` para registrar un nuevo usuario**:

Esta función crea un nuevo usuario en la base de datos:

- **`const { username, password } = req.body;`**: Extrae el nombre de usuario y la contraseña del cuerpo de la solicitud (datos enviados por el cliente).
- **`const user = new User({ username, password });`**: Crea una instancia del modelo `User` con los datos recibidos.
- **`await user.save();`**: Guarda el usuario en la base de datos. El middleware en `userModel` se encargará de encriptar la contraseña.
- **`res.status(201).json({ message: 'Usuario registrado con éxito' });`**: Envía una respuesta de éxito al cliente si el registro es exitoso.
- **Manejo de Errores**: Si ocurre un error, se captura y envía una respuesta con código `400` y el mensaje del error.

4. **Función `login` para autenticar al usuario**:

Esta función autentica a un usuario y le genera un token JWT:

   - **`const { username, password } = req.body;`**: Extrae el nombre de usuario y la contraseña del cuerpo de la solicitud.
   - **`const user = await User.findOne({ username });`**: Busca un usuario en la base de datos cuyo nombre de usuario coincida con el ingresado.
   - **Verificación de Credenciales**:
     - **`if (!user || !(await user.comparePassword(password)))`**: Si el usuario no existe o la contraseña no coincide (verificada con el método `comparePassword`), envía un mensaje de error con código `401` (No Autorizado).
   - **Generación de Token**:
     - **`const token = generateToken(user);`**: Genera un token usando la función `generateToken`.
     - **`res.json({ token });`**: Envía el token generado al cliente, que podrá usarlo para autenticarse en futuras solicitudes.
   - **Manejo de Errores**: Si ocurre un error, se captura y se envía una respuesta con código `500` (Error Interno del Servidor) y el mensaje del error.

Este código permite registrar nuevos usuarios y autenticar usuarios existentes, devolviendo un token JWT si las credenciales son correctas.

Rutas de Autenticación

En `src/routes/authRoutes.js`:

```
const express = require('express');
const authController = require('../controllers/authController');
const router = express.Router();

router.post('/register', authController.register);
router.post('/login', authController.login);

module.exports = router;
```

Este código configura rutas en Express para manejar el registro y el inicio de sesión de los usuarios, y luego exporta el router para su uso en otras partes de la aplicación. Aquí está la explicación detallada:

1. **Importar Dependencias**:
   - **`const express = require('express');`**: Importa el módulo `express`, una biblioteca de Node.js para crear aplicaciones web y APIs de forma sencilla.
   - **`const authController = require('../controllers/authController');`**: Importa el controlador de autenticación desde el archivo `authController.js`, que contiene las funciones `register` y `login` para registrar y autenticar usuarios.
2. **Crear el Router**:
   - **`const router = express.Router();`**: Crea una instancia del enrutador de Express. Este router ayuda a organizar las rutas relacionadas con la autenticación en un solo lugar y permite modularizar la aplicación.
3. **Definir las Rutas**:
   - **`router.post('/register', authController.register);`**:
     - Define una ruta de tipo `POST` en `/register`. Cuando un cliente realiza una solicitud POST a `/register`, se llama a la función `register` del `authController`, que se encarga de registrar un nuevo usuario.
   - **`router.post('/login', authController.login);`**:
     - Define una ruta de tipo `POST` en `/login`. Cuando un cliente realiza una solicitud POST a `/login`, se llama a la función `login` del `authController`, que se encarga de autenticar al usuario y devolver un token de acceso si las credenciales son correctas.
4. **Exportar el Router**:
   - **`module.exports = router;`**: Exporta el router, permitiendo que este conjunto de rutas pueda ser importado y utilizado en la aplicación principal de Express.

Este archivo facilita el manejo de las rutas de autenticación (`/register` y `/login`) y las conecta con las funciones correspondientes en el `authController`.

Middleware de Autenticación

En `src/middlewares/authMiddleware.js`:

```
const jwt = require('jsonwebtoken');

exports.protect = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) return res.status(401).json({ error: 'No estás autorizado' });

  jwt.verify(token, process.env.JWT_SECRET, (err, decoded) => {
    if (err) return res.status(403).json({ error: 'Token no válido' });
    req.user = decoded;
    next();
  });
};
```

Este código define un middleware llamado `protect` que se usa para proteger rutas en una aplicación. Verifica si el usuario está autenticado mediante un token JWT (JSON Web Token) antes de permitir el acceso a la ruta protegida. A continuación se explica cada parte:

1. **Importar JWT**:

- Importa la biblioteca `jsonwebtoken` que permite crear, firmar y verificar tokens JWT.

2. **Función `protect`:**

- Se exporta la función `protect` como un middleware, que se puede aplicar a las rutas que necesitan autenticación.
- Recibe tres parámetros estándar en Express: `req` (solicitud), `res` (respuesta) y `next` (función que continúa con el siguiente middleware o la ruta final si se pasa la verificación).

3. **Obtener el Token del Encabezado de Autorización:**

- **`req.headers.authorization`**: Intenta obtener el encabezado de `authorization` de la solicitud, donde normalmente se envía el token.
- **`split(' ')[1]`**: Separa el valor del encabezado en palabras (usando el espacio como delimitador) y toma el segundo elemento. Esto es porque los tokens a menudo se envían en el formato `Bearer <token>`, donde `Bearer` es el primer elemento y el token es el segundo.
- **Si no se encuentra el token**: Devuelve un estado `401` (No Autorizado) y un mensaje indicando que el usuario no está autorizado, deteniendo el proceso sin pasar al siguiente middleware.

4. **Verificar el Token:**
    - **`jwt.verify(token, process.env.JWT_SECRET, ...)`**: Usa `jsonwebtoken` para verificar el token.
        - **`token`**: El token extraído del encabezado.
        - **`process.env.JWT_SECRET`**: La clave secreta usada para verificar la autenticidad del token.
        - **Callback `(err, decoded)`**: Llamado tras la verificación, recibe dos posibles resultados:
            - **`err`**: Indica que el token es inválido o expiró, en cuyo caso se devuelve un estado `403` (Prohibido) y un mensaje de error.
            - **`decoded`**: Si el token es válido, `decoded` contiene los datos del token (por ejemplo, el `id` del usuario). Esto se asigna a `req.user` para que el usuario autenticado esté disponible en las siguientes funciones.

2. **Pasar al Siguiente Middleware o Ruta**:

    - **`next();`**: Llama a `next()` si el token es válido, permitiendo el acceso a la ruta o middleware que sigue.

Este middleware `protect` se utiliza para garantizar que solo los usuarios autenticados puedan acceder a ciertas rutas, añadiendo así una capa de seguridad a la aplicación.

Configuración de la Aplicación

En `src/app.js`:

```
const express = require('express');
const cors = require('cors');
const connectDB = require('./config/db');
const authRoutes = require('./routes/authRoutes');

const app = express();
connectDB();

app.use(cors());
app.use(express.json());
app.use('/api/auth', authRoutes);

module.exports = app;
```

Este código configura una aplicación básica de Express que permite manejar autenticación con rutas específicas, conexión a la base de datos y soporte para solicitudes de diferentes orígenes mediante CORS. Vamos a analizarlo en detalle:

1. **Importar Dependencias**:

- **`const express = require('express');`**: Importa Express, que facilita la creación y gestión de aplicaciones web y APIs.
- **`const cors = require('cors');`**: Importa `cors` para habilitar CORS (Cross-Origin Resource Sharing), permitiendo que la API reciba solicitudes de otros dominios.
- **`const connectDB = require('./config/db');`**: Importa la función `connectDB` desde un archivo de configuración (`db.js`), que se encarga de conectar la aplicación a una base de datos.
- **`const authRoutes = require('./routes/authRoutes');`**: Importa las rutas de autenticación desde `authRoutes.js`, donde están definidas las rutas de registro e inicio de sesión.

2. Inicializar la Aplicación:
   - **`const app = express();`**: Crea una instancia de la aplicación Express, que se usará para configurar middleware, rutas y opciones de la aplicación.
3. Conectar a la Base de Datos:
   - **`connectDB();`**: Llama a la función `connectDB` para establecer una conexión a la base de datos al iniciar la aplicación.
4. Configurar Middleware:

   - **`app.use(cors());`**: Activa CORS, permitiendo que la aplicación reciba solicitudes de diferentes orígenes (por ejemplo, desde un frontend en otro dominio).
   - **`app.use(express.json());`**: Habilita la interpretación de JSON en el cuerpo de las solicitudes entrantes. Esto permite que los datos enviados en formato JSON se conviertan en objetos JavaScript accesibles en `req.body`.
5. Definir Rutas:
   - **`app.use('/api/auth', authRoutes);`**: Monta las rutas de autenticación en el endpoint `/api/auth`. Esto significa que cualquier ruta definida en `authRoutes` estará disponible bajo el prefijo `/api/auth` (por ejemplo, `/api/auth/register` y `/api/auth/login`).
6. Exportar la Aplicación:
   - **`module.exports = app;`**: Exporta la instancia de la aplicación para que pueda ser utilizada en otros archivos, como el archivo principal que inicia el servidor (`server.js` o similar).

Resumen

Este archivo configura una aplicación Express con:

- Conexión a la base de datos.
- Habilitación de CORS.
- Interpretación de JSON en las solicitudes entrantes.
- Rutas para manejar la autenticación.

Este archivo es el corazón de la configuración de la API, donde se inicializan las conexiones y las rutas necesarias para ejecutar la aplicación.

Inicio del Servidor

En `src/server.js`:

```
require('dotenv').config();
const app = require('./app');

const PORT = process.env.PORT || 5000;
app.listen(PORT, () => {
  console.log(`Servidor corriendo en el puerto ${PORT}`);
});
```

Configuración de .env en el Frontend

En `frontend/.env`:

```
REACT_APP_API_URL=http://localhost:5000
```

Este código configura y arranca el servidor de la aplicación Express. Vamos a desglosarlo:

1. **Cargar Variables de Entorno**:

- **`require('dotenv').config();`**: Importa y ejecuta el paquete `dotenv`, que permite cargar variables de entorno desde un archivo `.env` en el proyecto. Estas variables son accesibles mediante `process.env`.
- Esto es útil para configurar detalles confidenciales o configurables, como la conexión a la base de datos (`MONGO_URI`) o el puerto en el que el servidor escuchará (`PORT`), sin que estén hardcodeadas en el código.

2.Importar la Aplicación:

- **`const app = require('./app');`**: Importa la instancia de la aplicación Express que se configuró en el archivo `app.js` (donde se definen las rutas, la conexión a la base de datos y los middleware necesarios).
- Al importar la aplicación desde `app.js`, se encapsulan todas las configuraciones en un solo archivo que el servidor puede iniciar.

3.Definir el Puerto del Servidor:

- **`const PORT = process.env.PORT || 5000;`**: Asigna el puerto en el que se ejecutará el servidor.
  - **`process.env.PORT`**: Si existe una variable de entorno `PORT` (definida en el archivo `.env` o en el entorno de producción), se usará este valor.
  - **`5000`**: Si `PORT` no está definida, se usará `5000` como puerto por defecto.

4. **Iniciar el Servidor**:

  - **`app.listen(PORT, () => { ... })`**: Inicia el servidor Express en el puerto especificado por `PORT`.
  - **Callback `( ) => { ... }`**: Este callback se ejecuta una vez que el servidor está en funcionamiento y registra en la consola el mensaje `"Servidor corriendo en el puerto <PORT>"` para confirmar que la aplicación está en marcha y en qué puerto.

Resumen

Este código carga las configuraciones de entorno, establece el puerto, e inicia el servidor Express en dicho puerto. Es el punto de entrada del servidor, que controla cuándo se arranca la aplicación.