Seguridad Basada En Tokens

Middleware de Autenticación (auth.js)

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

const authMiddleware = (req, res, next) => {
    const authHeader = req.headers['authorization'];
    if (!authHeader) return res.status(401).json({ error: 'No token provided' });

    const token = authHeader.split(' ')[1];
    if (!token) return res.status(401).json({ error: 'Malformed token' });

    try {
        const decoded = jwt.verify(token, '22644');
        req.user = decoded;
        next();
    } catch (err) {
        return res.status(401).json({ error: 'Token inválido' });
    }
};

module.exports = authMiddleware;
```

- Este código es fundamental para la seguridad del sistema

- jwt.require('jsonwebtoken'): Importa la biblioteca para manejar tokens JWT

- La función authMiddleware verifica cada petición:

1. Comprueba si existe el encabezado de autorización
2. Extrae el token del formato "Bearer [token]"
3. Verifica el token usando la clave secreta '22644'
4. Si todo es correcto, permite continuar con la petición
5. Si hay algún error, devuelve una respuesta 401 (No autorizado)
6.

Gestión de Clientes (clientes.js)

```
const express = require('express');
const router = express.Router();
const Clientes = require('../modelos/clientes');
const authMiddleware = require('../middlewares/auth');

router.use(authMiddleware);

// Obtener todos los clientes
router.get('/', async (req, res) => {
    try {
        const clientes = await Clientes.findAll();
        res.json(clientes);
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener clientes' });
    }
});

// Obtener cliente por ID
router.get('/:id', async (req, res) => {
    try {
        const cliente = await Clientes.findByPk(req.params.id);
        if (cliente) {
            res.json(cliente);
        } else {
            res.status(404).json({ error: 'Cliente no encontrado' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener el cliente' });
    }
});

// Crear nuevo cliente
router.post('/', async (req, res) => {
    try {
        const nuevoCliente = await Clientes.create(req.body);
        res.status(201).json(nuevoCliente);
    } catch (error) {
        res.status(400).json({ error: 'Error al crear el cliente' });
    }
});

// Actualizar cliente
router.put('/:id', async (req, res) => {
    try {
        const cliente = await Clientes.findByPk(req.params.id);
        if (cliente) {
            await cliente.update(req.body);
            res.json(cliente);
        } else {
            res.status(404).json({ error: 'Cliente no encontrado' });
        }
    } catch (error) {
```

```
        res.status(400).json({ error: 'Error al actualizar el cliente' });
    }
});

// Eliminar cliente
router.delete('/:id', async (req, res) => {
    try {
        const eliminado = await Clientes.destroy({ where: { id: req.params.id } });
        if (eliminado) {
            res.json({ mensaje: 'Cliente eliminado correctamente' });
        } else {
            res.status(404).json({ error: 'Cliente no encontrado' });
        }
    } catch (error) {
        res.status(500).json({ error: 'Error al eliminar el cliente' });
    }
});

module.exports = router;
```

- Este módulo implementa todas las operaciones CRUD para clientes
- Utiliza el patrón Router de Express para organizar las rutas
- Cada ruta está protegida por el middleware de autenticación

Operaciones implementadas:

1. GET / - Lista todos los clientes
   - Usa findAll() de Sequelize
   - Maneja errores con try/catch
2. GET /:id - Obtiene un cliente específico
   - Usa findByPk() para buscar por ID
   - Devuelve 404 si no encuentra el cliente
3. POST / - Crea un nuevo cliente
   - Usa create() con los datos del body
   - Devuelve 201 si se crea correctamente
4. PUT /:id - Actualiza un cliente
   - Primero busca el cliente
   - Si existe, lo actualiza con los nuevos datos
5. DELETE /:id - Elimina un cliente
   - Usa `destroy()` para eliminar por ID
   - Confirma si se eliminó correctamente

Gestión de Artículos (articulos.js)

```
const express = require('express');
const router = express.Router();
const Articulos = require('../modelos/articulos');

// Obtener todos los artículos
router.get('/', async (req, res) => {
    try {
        const articulos = await Articulos.findAll();
        res.json(articulos);
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener artículos' });
    }
});

// Obtener un artículo por ID
router.get('/:id', async (req, res) => {
    try {
        const articulo = await Articulos.findByPk(req.params.id);
        articulo ? res.json(articulo) : res.status(404).json({ error: 'Artículo no
encontrado' });
    } catch (error) {
        res.status(500).json({ error: 'Error al obtener el artículo' });
    }
});

// Crear un nuevo artículo
router.post('/', async (req, res) => {
    try {
        const nuevo = await Articulos.create(req.body);
        res.status(201).json(nuevo);
    } catch (error) {
        res.status(400).json({ error: 'Error al crear artículo' });
    }
});

// Actualizar artículo
router.put('/:id', async (req, res) => {
    try {
        const articulo = await Articulos.findByPk(req.params.id);
        if (articulo) {
            await articulo.update(req.body);
            res.json(articulo);
        } else {
            res.status(404).json({ error: 'Artículo no encontrado' });
        }
    } catch (error) {
        res.status(400).json({ error: 'Error al actualizar artículo' });
    }
});
```

```
// Eliminar artículo
router.delete('/:id', async (req, res) => {
    try {
        const eliminado = await Articulos.destroy({ where: { id: req.params.id } });
        eliminado ? res.json({ mensaje: 'Artículo eliminado' }) : res.status(404).json({
error: 'No encontrado' });
    } catch (error) {
        res.status(500).json({ error: 'Error al eliminar artículo' });
    }
});

module.exports = router;
```

- Similar al módulo de clientes, pero para gestionar artículos
- Implementa las mismas operaciones CRUD pero con sintaxis más concisa
- Características específicas:

1. Manejo de Errores
    - Usa operadores ternarios para respuestas más concisas
    - Mantiene los mismos códigos de estado HTTP

2. Operaciones de Base de Datos

- Utiliza el modelo Articulos para todas las operaciones
- Implementa las mismas funciones de Sequelize

3. Respuestas
- Formato JSON consistente
- Mensajes de error específicos para artículos
- Códigos de estado HTTP apropiados

Estos tres archivos forman el núcleo del sistema:

- auth.js proporciona la seguridad
- clientes.js maneja la lógica de negocio para clientes
- articulos.js maneja la lógica de negocio para artículos