To include the new Admin Authentication and the management of Admin Users, Branches, Customers, Settings, and Audit Logs into the previous Node.js application structure, we'll also need to integrate JWE (JSON Web Encryption) token handling for secure user authentication and authorization.

**Folder Structure (Updated)**

project/

├── config/

│ ├── db.js # Database connection (MariaDB)

│ ├── logger.js # Logger configuration

│ ├── keys.js # RSA Key management

│ ├── jwe.js # JWE Key management and utilities

├── controllers/

│ ├── adminController.js # Admin-related API logic

│ ├── authController.js # Authentication logic (login, token generation)

│ ├── userController.js # User-related API logic

│ ├── fileController.js # File upload (WAV, Excel) handling

├── routes/

│ ├── adminRoutes.js # Admin-related routes

│ ├── authRoutes.js # Authentication routes

│ ├── userRoutes.js # User-related routes

│ ├── fileRoutes.js # File upload routes

├── services/

│ ├── authService.js # Auth-related logic (token generation, validation)

│ ├── userService.js # User-related logic (user creation, validation)

│ ├── fileService.js # File upload and processing logic

│ ├── dbService.js # DB interaction (MariaDB)

├── utils/

│ ├── tokenUtils.js # JWE generation and validation

│ ├── rsaUtils.js # RSA Key generation, signing, and validation

├── app.js # Main application setup

└── package.json

**1. JWE Key Management and Utility**

config/jwe.js:

const fs = require('fs');

const path = require('path');

const { generateKeyPairSync } = require('crypto');

// Generate JWE RSA keys (private & public) if they don't exist

const keysPath = path.join(\_\_dirname, '../keys/');

if (!fs.existsSync(keysPath)) {

fs.mkdirSync(keysPath);

}

const { publicKey, privateKey } = generateKeyPairSync('rsa', {

modulusLength: 2048,

publicKeyEncoding: {

type: 'pkcs1',

format: 'pem'

},

privateKeyEncoding: {

type: 'pkcs1',

format: 'pem'

}

});

fs.writeFileSync(path.join(keysPath, 'private\_jwe.pem'), privateKey);

fs.writeFileSync(path.join(keysPath, 'public\_jwe.pem'), publicKey);

module.exports = {

publicKey,

privateKey

};

**2. JWE Token Utilities**

utils/tokenUtils.js:

const jwt = require('jsonwebtoken');

const { privateKey, publicKey } = require('../config/jwe');

const generateJweToken = (user) => {

const payload = {

id: user.id,

username: user.username,

role: user.role

};

const token = jwt.sign(payload, privateKey, { algorithm: 'RS256', expiresIn: '1h' });

return token;

};

const verifyJweToken = (token) => {

try {

const decoded = jwt.verify(token, publicKey, { algorithms: ['RS256'] });

return decoded;

} catch (err) {

return null;

}

};

module.exports = {

generateJweToken,

verifyJweToken

};

**3. Admin Authentication Service**

services/authService.js:

const { generateJweToken, verifyJweToken } = require('../utils/tokenUtils');

const pool = require('../config/db');

const logger = require('../config/logger');

// Admin Login

const login = async (username, password) => {

const [rows] = await pool.query('SELECT \* FROM admin\_users WHERE username = ?', [username]);

const admin = rows[0];

if (admin && admin.password === password) {

const token = generateJweToken(admin);

logger.info(`Admin ${username} logged in successfully`);

return { token };

} else {

logger.error(`Invalid login attempt for ${username}`);

throw new Error('Invalid credentials');

}

};

// Token validation

const validateToken = (token) => {

return verifyJweToken(token);

};

module.exports = {

login,

validateToken

};

**4. Admin Controller**

controllers/adminController.js:

const pool = require('../config/db');

const logger = require('../config/logger');

// Create a new admin user

const createUser = async (req, res) => {

const { username, password, role } = req.body;

try {

await pool.query('INSERT INTO admin\_users (username, password, role) VALUES (?, ?, ?)', [username, password, role]);

res.status(201).send('Admin user created');

} catch (error) {

logger.error(error);

res.status(500).send('Internal server error');

}

};

// Update an existing admin user

const updateUser = async (req, res) => {

const { id, username, password, role } = req.body;

try {

await pool.query('UPDATE admin\_users SET username = ?, password = ?, role = ? WHERE id = ?', [username, password, role, id]);

res.status(200).send('Admin user updated');

} catch (error) {

logger.error(error);

res.status(500).send('Internal server error');

}

};

// Delete an admin user

const deleteUser = async (req, res) => {

const { id } = req.body;

try {

await pool.query('DELETE FROM admin\_users WHERE id = ?', [id]);

res.status(200).send('Admin user deleted');

} catch (error) {

logger.error(error);

res.status(500).send('Internal server error');

}

};

// Assign role to admin user

const assignRole = async (req, res) => {

const { userId, role } = req.body;

try {

await pool.query('UPDATE admin\_users SET role = ? WHERE id = ?', [role, userId]);

res.status(200).send('Role assigned');

} catch (error) {

logger.error(error);

res.status(500).send('Internal server error');

}

};

module.exports = {

createUser,

updateUser,

deleteUser,

assignRole

};

**5. Admin Routes**

routes/adminRoutes.js:

const express = require('express');

const router = express.Router();

const adminController = require('../controllers/adminController');

const authService = require('../services/authService');

// Admin login

router.post('/login', async (req, res) => {

const { username, password } = req.body;

try {

const { token } = await authService.login(username, password);

res.json({ token });

} catch (err) {

res.status(400).send(err.message);

}

});

// Admin logout (clear session or token)

router.get('/logout', (req, res) => {

// Implement logout logic if session-based, or let the client manage the token

res.send('Admin logged out');

});

// Admin user management

router.post('/create-user', adminController.createUser);

router.put('/update-user', adminController.updateUser);

router.delete('/delete-user', adminController.deleteUser);

router.post('/assign-role', adminController.assignRole);

// Add more routes for branch, customer, settings management as needed

module.exports = router;

**6. Middleware for Token Authentication**

To ensure only authenticated users (admin in this case) can access the APIs, add a middleware for token validation.

middlewares/authMiddleware.js:

const authService = require('../services/authService');

const verifyAdminToken = (req, res, next) => {

const token = req.headers['authorization']?.split(' ')[1];

if (!token) {

return res.status(403).send('No token provided');

}

const decoded = authService.validateToken(token);

if (!decoded) {

return res.status(401).send('Invalid token');

}

req.user = decoded; // Attach user information to request

next();

};

module.exports = {

verifyAdminToken

};

**7. Main Application (app.js)**

app.js:

const express = require('express');

const app = express();

const cors = require('cors');

const bodyParser = require('body-parser');

const adminRoutes = require('./routes/adminRoutes');

const logger = require('./config/logger');

const authMiddleware = require('./middlewares/authMiddleware');

// Middleware

app.use(cors());

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

// Routes with authentication middleware

app.use('/api/admin', authMiddleware.verifyAdminToken, adminRoutes);

// Start the server

const port = process.env.PORT || 3000;

app.listen(port, () => {

logger.info(`Server running on port ${port}`);

});

**8. MariaDB Tables for Admin and User Management**

Here's a summary of the necessary MariaDB tables for this design:

1. **admin\_users**: For storing admin user data.
2. CREATE TABLE admin\_users (
3. id INT AUTO\_INCREMENT PRIMARY KEY,
4. username VARCHAR(255) NOT NULL,
5. password VARCHAR(255) NOT NULL

, role VARCHAR(50) NOT NULL, created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP, updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP );

2. \*\*`branches`\*\*: To store branch information.

```sql

CREATE TABLE branches (

id INT AUTO\_INCREMENT PRIMARY KEY,

customer\_id INT NOT NULL,

name VARCHAR(255) NOT NULL,

location VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

1. **audit\_logs**: To store audit logs for admin actions.
2. CREATE TABLE audit\_logs (
3. id INT AUTO\_INCREMENT PRIMARY KEY,
4. action VARCHAR(255) NOT NULL,
5. admin\_id INT NOT NULL,
6. timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP
7. );

**Conclusion**

In this implementation:

* We have included **admin authentication** using **JWE tokens** for secure authorization.
* We added endpoints for **admin user management**, **branch management**, **customer management**, **settings management**, and **audit logs**.
* **Token-based authentication** ensures secure access to the admin APIs, using JWT encrypted tokens (JWE).
* MariaDB is used to store all the required data with appropriate relationships and foreign keys.

This modular design ensures easy scalability, maintainability, and clear separation of concerns.

To store the **private key** and **public key** in **Redis**, mapped by a **request ID**, we need to modify our application slightly to include Redis integration. We'll use **Redis** as an in-memory cache to store the keys associated with a **request ID**.

**Steps to Integrate Redis for Storing Keys:**

1. **Install Redis and Required Dependencies** First, we need to install the Redis client for Node.js (ioredis), which will allow us to interact with Redis.
2. npm install ioredis
3. **Set Up Redis Connection** We need to set up a connection to Redis. Redis will store the **public key** and **private key** for each request ID. We can use the **in-memory cache** strategy, and Redis will automatically expire the keys after a given TTL (time-to-live).
4. **Modify the Key Management to Use Redis** We'll modify the RSA key generation process to save and retrieve the keys from Redis instead of the local file system.

**1. Redis Setup for Key Storage**

Let's begin by modifying our **key management** file to integrate Redis.

**config/redis.js (New Redis Connection)**

const Redis = require('ioredis');

// Set up Redis connection

const redis = new Redis({

host: 'localhost', // Replace with your Redis server host if not local

port: 6379, // Redis default port

db: 0 // Database index

});

module.exports = redis;

**2. RSA Key Generation and Redis Storage**

We will modify the **RSA key generation** logic to store the generated keys in Redis.

**utils/rsaUtils.js (Store Public & Private Key in Redis)**

const crypto = require('crypto');

const redis = require('../config/redis');

const logger = require('../config/logger');

// Generate RSA keys and store them in Redis (by request ID)

const generateAndStoreKeys = async (requestId) => {

// Generate RSA key pair

const { publicKey, privateKey } = crypto.generateKeyPairSync('rsa', {

modulusLength: 2048,

publicKeyEncoding: {

type: 'pkcs1',

format: 'pem'

},

privateKeyEncoding: {

type: 'pkcs1',

format: 'pem'

}

});

// Store keys in Redis with a TTL of 3600 seconds (1 hour)

await redis.setex(`keys:${requestId}:public`, 3600, publicKey);

await redis.setex(`keys:${requestId}:private`, 3600, privateKey);

logger.info(`Keys stored in Redis for request ID: ${requestId}`);

return { publicKey, privateKey };

};

// Retrieve keys from Redis using the request ID

const getKeysFromRedis = async (requestId) => {

const publicKey = await redis.get(`keys:${requestId}:public`);

const privateKey = await redis.get(`keys:${requestId}:private`);

if (!publicKey || !privateKey) {

logger.error(`Keys not found in Redis for request ID: ${requestId}`);

return null;

}

return { publicKey, privateKey };

};

module.exports = {

generateAndStoreKeys,

getKeysFromRedis

};

In this code:

* **generateAndStoreKeys** generates a new RSA key pair and stores both the **private key** and **public key** in Redis with a TTL of **3600 seconds** (1 hour). You can adjust the TTL as needed.
* **getKeysFromRedis** retrieves the **public key** and **private key** from Redis for a given **request ID**.

**3. Modify Token Validation to Use Redis Keys**

Now that we have the keys stored in Redis, we can modify the authentication part of the application to retrieve these keys from Redis for **device validation**.

**services/authService.js (Using Redis for Key Validation)**

const { generateJweToken, verifyJweToken } = require('../utils/tokenUtils');

const { generateAndStoreKeys, getKeysFromRedis } = require('../utils/rsaUtils');

const pool = require('../config/db');

const logger = require('../config/logger');

// Admin Login (JWT Generation)

const login = async (username, password) => {

const [rows] = await pool.query('SELECT \* FROM admin\_users WHERE username = ?', [username]);

const admin = rows[0];

if (admin && admin.password === password) {

const token = generateJweToken(admin);

logger.info(`Admin ${username} logged in successfully`);

return { token };

} else {

logger.error(`Invalid login attempt for ${username}`);

throw new Error('Invalid credentials');

}

};

// Device Authentication (using public/private keys from Redis)

const authenticateDevice = async (requestId, signedPayload) => {

// Retrieve keys from Redis

const keys = await getKeysFromRedis(requestId);

if (!keys) {

throw new Error('Keys not found for the request ID');

}

const { publicKey, privateKey } = keys;

// Validate signed payload using the private key

const isValid = verifySignedPayload(signedPayload, privateKey);

if (!isValid) {

throw new Error('Invalid signature');

}

// Do other checks like device ID and user authentication here...

return { message: 'Device authenticated successfully' };

};

// Function to verify the signed payload

const verifySignedPayload = (signedPayload, privateKey) => {

// You can implement your signature verification logic here.

// For example, using crypto module to verify the signed payload using the private key.

// For demonstration purposes, we're assuming it's valid:

return true;

};

module.exports = {

login,

authenticateDevice

};

* **authenticateDevice** is the function where we retrieve the **public key** and **private key** from Redis based on the **request ID**.
* After retrieving the keys, we would verify the **signed payload** using the private key (this can be your device signature).

**4. Integrate Redis for Key Storage in Controller**

We need to integrate the **generate and store keys** functionality in the **admin authentication** or **device authentication** flow.

**controllers/adminController.js (Generate RSA Keys for Admin Authentication)**

const { generateAndStoreKeys, getKeysFromRedis } = require('../utils/rsaUtils');

const logger = require('../config/logger');

// Generate RSA keys and store in Redis for each request

const generateKeysForRequest = async (req, res) => {

const { requestId } = req.body; // Assuming request ID is passed in request body

try {

const { publicKey, privateKey } = await generateAndStoreKeys(requestId);

res.json({ publicKey, privateKey });

} catch (error) {

logger.error(error);

res.status(500).send('Error generating keys');

}

};

// Device Authentication using Redis keys

const deviceAuthentication = async (req, res) => {

const { requestId, signedPayload } = req.body;

try {

const result = await authenticateDevice(requestId, signedPayload);

res.json(result);

} catch (error) {

logger.error(error);

res.status(400).send(error.message);

}

};

module.exports = {

generateKeysForRequest,

deviceAuthentication

};

In this example:

* **generateKeysForRequest** generates RSA keys for a given **request ID** and stores them in Redis.
* **deviceAuthentication** validates the device using the keys stored in Redis.

**5. Add Routes for RSA Key Generation and Device Authentication**

**routes/adminRoutes.js (Add Routes for Key Generation and Device Authentication)**

const express = require('express');

const router = express.Router();

const adminController = require('../controllers/adminController');

const authService = require('../services/authService');

const authMiddleware = require('../middlewares/authMiddleware');

// Route to generate RSA keys for a request ID

router.post('/generate-keys', adminController.generateKeysForRequest);

// Route for device authentication (validate signed payload)

router.post('/authenticate-device', adminController.deviceAuthentication);

module.exports = router;

**6. Test and Monitor**

Now, with Redis storing the RSA **public key** and **private key**, you can:

* **Generate keys** on the fly and store them in Redis with a **request ID**.
* **Authenticate devices** based on their signed payload and the stored private/public keys from Redis.

**Redis Expiry**: Keys will be automatically deleted from Redis after the TTL (1 hour in this case).

**Conclusion**

This approach integrates Redis as an in-memory cache to store **private keys** and **public keys** for each **request ID**. It ensures that:

* **Key management** is centralized and efficient.
* **RSA keys** are securely stored and accessed from Redis.
* Device validation is handled using these keys, leveraging the **request ID** for each session.

This modular approach allows scalability and improved performance for key management and authentication.