To build a simple API using Node.js and Express that supports the four HTTP methods (GET, POST, PUT, DELETE) and allows file uploads (in this case, WAV files), then deploy it on a Tomcat server, you can follow these steps.

**Steps Overview:**

1. **Set up Node.js project with Express**: Create a Node.js project and install necessary dependencies.
2. **Implement API routes (GET, POST, PUT, DELETE)**: Implement routes to handle different HTTP methods.
3. **Handle file upload (WAV)**: Implement file upload functionality using the multer middleware.
4. **Deploy the app on Tomcat**: Package the Node.js app and deploy it to Tomcat.

**Step 1: Set Up Node.js Project**

1. **Create a Node.js project**:
2. mkdir wav-api
3. cd wav-api
4. npm init -y
5. **Install necessary dependencies**:
6. npm install express multer

**Step 2: Implement API Routes (GET, POST, PUT, DELETE)**

Create a file called app.js inside your project folder.

const express = require('express');

const multer = require('multer');

const path = require('path');

const app = express();

const port = 3000;

// Set up file upload using multer

const storage = multer.diskStorage({

destination: (req, file, cb) => {

cb(null, 'uploads/');

},

filename: (req, file, cb) => {

cb(null, Date.now() + path.extname(file.originalname));

}

});

const upload = multer({ storage: storage, fileFilter: (req, file, cb) => {

const ext = path.extname(file.originalname);

if (ext !== '.wav') {

return cb(new Error('Only WAV files are allowed'));

}

cb(null, true);

}});

// Create 'uploads' directory if it doesn't exist

const fs = require('fs');

if (!fs.existsSync('uploads')) {

fs.mkdirSync('uploads');

}

// Define routes

// GET - Retrieve list of uploaded WAV files

app.get('/api/files', (req, res) => {

fs.readdir('uploads', (err, files) => {

if (err) {

return res.status(500).send('Failed to retrieve files.');

}

res.json(files);

});

});

// POST - Upload a new WAV file

app.post('/api/upload', upload.single('file'), (req, res) => {

res.status(200).send({

message: 'File uploaded successfully',

filename: req.file.filename

});

});

// PUT - Update an existing file (we'll simulate this by renaming the file)

app.put('/api/files/:filename', upload.single('file'), (req, res) => {

const oldPath = `uploads/${req.params.filename}`;

const newPath = `uploads/${req.file.filename}`;

fs.rename(oldPath, newPath, (err) => {

if (err) {

return res.status(500).send('Failed to update the file.');

}

res.status(200).send({ message: 'File updated successfully', filename: req.file.filename });

});

});

// DELETE - Delete a file

app.delete('/api/files/:filename', (req, res) => {

const filePath = `uploads/${req.params.filename}`;

fs.unlink(filePath, (err) => {

if (err) {

return res.status(500).send('Failed to delete the file.');

}

res.status(200).send({ message: 'File deleted successfully' });

});

});

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

**Explanation:**

1. **GET /api/files**: This route returns a list of all uploaded WAV files stored in the uploads directory.
2. **POST /api/upload**: This route allows the upload of a WAV file. The file will be stored in the uploads directory.
3. **PUT /api/files/:filename**: This simulates updating a file by renaming it to the new filename provided in the request. It expects the new file to be uploaded as well.
4. **DELETE /api/files/:filename**: This route deletes a specified file from the uploads directory.

**Step 3: Handle File Uploads**

The code uses multer, a Node.js middleware, to handle file uploads. The upload.single('file') middleware handles single file uploads from the form field named file. You can adjust the file filter to allow only .wav files, as demonstrated in the fileFilter function.

**Step 4: Deploy Node.js App in Tomcat**

Tomcat is a Java-based web server, but it is possible to run Node.js apps in Tomcat by wrapping them in a web application using a proxy.

Here’s a simple approach:

1. **Install Tomcat**: If you haven't installed Tomcat yet, download it from the [official website](https://tomcat.apache.org/).
2. **Install mod\_proxy for Tomcat**: To run Node.js behind Tomcat, you'll need a reverse proxy setup. Ensure you have mod\_proxy enabled in Tomcat.
3. **Configure Tomcat to Proxy Requests**:
   * Open the conf/server.xml file in Tomcat.
   * Add a <Host> entry to define the reverse proxy setup for your Node.js app:
   * <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">
   * <Context path="/nodeapp" docBase="path/to/your/nodejs/app" />
   * </Host>

This tells Tomcat to forward requests from /nodeapp to your Node.js app.

1. **Run the Node.js app**: Start your Node.js server by running:
2. node app.js
3. **Access the app via Tomcat**: Once everything is configured, you can access the Node.js app via Tomcat. For example:
4. http://localhost:8080/nodeapp/api/upload

**Step 5: Testing the API**

* **Upload a file (POST)**: Use a tool like Postman or curl to upload a .wav file.
* curl -X POST -F "file=@path/to/your/file.wav" http://localhost:3000/api/upload
* **List files (GET)**: To list uploaded files:
* curl http://localhost:3000/api/files
* **Update a file (PUT)**: Update a file by sending a PUT request with the file name and new file:
* curl -X PUT -F "file=@path/to/your/newfile.wav" http://localhost:3000/api/files/filename.wav
* **Delete a file (DELETE)**: Delete a file with the following command:
* curl -X DELETE http://localhost:3000/api/files/filename.wav

**Conclusion**

This guide demonstrates how to build a simple Node.js API using Express to handle file uploads and implement the basic CRUD operations (GET, POST, PUT, DELETE). It also covers deploying the app on Tomcat using a reverse proxy.

If you need any further help with any of the steps, feel free to ask!

To build a Node.js API using Express that handles JSON data (GET, POST, PUT, DELETE) along with a separate API for uploading WAV files (with JSON input as well), and deploy it on Tomcat, we'll break it down into the following steps:

**Steps Overview:**

1. **Set up Node.js project with Express**.
2. **Create APIs for handling JSON data** (GET, POST, PUT, DELETE).
3. **Create an API for WAV file upload with additional JSON data**.
4. **Deploy the Node.js app on Tomcat**.
5. **Test the API**.

**Step 1: Set Up Node.js Project**

1. **Create a Node.js project**:
2. mkdir wav-json-api
3. cd wav-json-api
4. npm init -y
5. **Install necessary dependencies**:
6. npm install express multer body-parser

**Step 2: Create APIs for JSON Data (GET, POST, PUT, DELETE)**

Let's start by building the Express app with the basic CRUD operations for handling JSON data. We'll also create a separate endpoint for handling file uploads.

Create a file called app.js inside your project folder:

const express = require('express');

const multer = require('multer');

const path = require('path');

const fs = require('fs');

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

const app = express();

const port = 3000;

// Middleware to parse JSON bodies

app.use(bodyParser.json());

// In-memory data store (for demonstration purposes)

let dataStore = [

{ id: 1, name: 'Item 1', description: 'This is item 1' },

{ id: 2, name: 'Item 2', description: 'This is item 2' }

];

// File upload setup using multer

const storage = multer.diskStorage({

destination: (req, file, cb) => {

const uploadDir = 'uploads/';

if (!fs.existsSync(uploadDir)) {

fs.mkdirSync(uploadDir);

}

cb(null, uploadDir);

},

filename: (req, file, cb) => {

cb(null, Date.now() + path.extname(file.originalname));

}

});

const upload = multer({

storage: storage,

fileFilter: (req, file, cb) => {

const ext = path.extname(file.originalname);

if (ext !== '.wav') {

return cb(new Error('Only WAV files are allowed'), false);

}

cb(null, true);

}

});

// JSON CRUD APIs

app.get('/api/items', (req, res) => {

res.json(dataStore);

});

app.post('/api/items', (req, res) => {

const { name, description } = req.body;

const newItem = { id: dataStore.length + 1, name, description };

dataStore.push(newItem);

res.status(201).json(newItem);

});

app.put('/api/items/:id', (req, res) => {

const { id } = req.params;

const { name, description } = req.body;

const itemIndex = dataStore.findIndex(item => item.id == id);

if (itemIndex !== -1) {

dataStore[itemIndex] = { id: parseInt(id), name, description };

res.json(dataStore[itemIndex]);

} else {

res.status(404).json({ message: 'Item not found' });

}

});

app.delete('/api/items/:id', (req, res) => {

const { id } = req.params;

const itemIndex = dataStore.findIndex(item => item.id == id);

if (itemIndex !== -1) {

const deletedItem = dataStore.splice(itemIndex, 1);

res.json({ message: 'Item deleted', item: deletedItem });

} else {

res.status(404).json({ message: 'Item not found' });

}

});

// WAV file upload with JSON input

app.post('/api/upload', upload.single('file'), (req, res) => {

const { file } = req;

const { jsonData } = req.body; // Assume JSON data is provided as a string in the form field 'jsonData'

if (!file) {

return res.status(400).json({ message: 'No file uploaded' });

}

let parsedJsonData;

try {

parsedJsonData = JSON.parse(jsonData);

} catch (error) {

return res.status(400).json({ message: 'Invalid JSON data' });

}

res.status(200).json({

message: 'File uploaded successfully',

filename: file.filename,

jsonData: parsedJsonData

});

});

// Start the server

app.listen(port, () => {

console.log(`Server is running at http://localhost:${port}`);

});

**Explanation:**

1. **JSON CRUD APIs**:
   * GET /api/items: Fetches the list of items from the in-memory dataStore.
   * POST /api/items: Creates a new item in the dataStore.
   * PUT /api/items/:id: Updates an item based on the provided id.
   * DELETE /api/items/:id: Deletes an item based on the provided id.
2. **File Upload with JSON Input**:
   * The /api/upload route handles both file uploads and JSON data. The file is uploaded using multer and stored in the uploads/ folder.
   * The route expects the form field file for the WAV file and jsonData for the JSON string.
   * The JSON string in the jsonData field is parsed and returned along with the file information.

**Step 3: Create a uploads Directory**

Make sure to create an uploads/ folder where the uploaded files will be stored. This is done automatically in the code if the directory doesn't exist.

**Step 4: Deploy the Node.js App on Tomcat**

To deploy a Node.js application on Tomcat, we need to set up a reverse proxy because Tomcat is a Java web server. Here’s how you can set it up:

1. **Install and Configure Tomcat**:
   * Download and install Tomcat from the [official website](https://tomcat.apache.org/).
   * Ensure that Tomcat is running on your machine.
2. **Configure Apache HTTP or Nginx as Reverse Proxy (Optional)**: While Tomcat doesn’t directly support Node.js, you can use a reverse proxy (Apache HTTP or Nginx) to route HTTP requests to your Node.js app. This step is optional but provides more flexibility.
3. **Run the Node.js App**: You can run the Node.js app using pm2 or forever to ensure it runs continuously.
4. pm2 start app.js
5. **Proxy Requests from Tomcat to Node.js**: In your Tomcat conf/server.xml, add a <Proxy> setup that will route requests to your Node.js app. For example:
6. <Host name="localhost" appBase="webapps" unpackWARs="true" autoDeploy="true">
7. <Context path="/nodeapp" docBase="path/to/your/nodejs/app" />
8. </Host>

You can then access your Node.js API via:

http://localhost:8080/nodeapp/api/items

**Step 5: Test the API**

1. **CRUD operations** (using curl, Postman, etc.):
   * **GET**:
   * curl http://localhost:3000/api/items
   * **POST** (Create item):
   * curl -X POST http://localhost:3000/api/items -H "Content-Type: application/json" -d '{"name": "Item 3", "description": "This is item 3"}'
   * **PUT** (Update item):
   * curl -X PUT http://localhost:3000/api/items/1 -H "Content-Type: application/json" -d '{"name": "Updated Item", "description": "Updated description"}'
   * **DELETE** (Delete item):
   * curl -X DELETE http://localhost:3000/api/items/1
2. **WAV file upload with JSON data**: You can upload a file using Postman or curl by sending the file and jsonData as form fields:
3. curl -X POST http://localhost:3000/api/upload -F "file=@path/to/your/file.wav" -F "jsonData={\"key\":\"value\"}"

**Conclusion**

You now have a Node.js API that supports CRUD operations for JSON data, as well as an endpoint for uploading WAV files along with JSON input. The application can be deployed to Tomcat with the reverse proxy configuration, allowing it to run alongside other Java applications on Tomcat.

If you need further assistance or clarification on any step, feel free to ask!

To run a Node.js application as a service on a Unix-based system (such as Linux or macOS), you can use a process manager like systemd (on modern Linux systems) or pm2 (a popular Node.js process manager). Both methods ensure that your application runs continuously, even after reboots.

Here are two methods to achieve this:

**Method 1: Using systemd to Run the Node.js App as a Service**

systemd is the default service manager on many modern Linux distributions (such as Ubuntu, CentOS, and Debian). You can use systemd to create a service that will start your Node.js app on boot and ensure it keeps running.

**1. Create a systemd Service File**

1. First, navigate to the systemd service configuration directory:
2. cd /etc/systemd/system/
3. Create a new service file, e.g., nodejs-app.service:
4. sudo nano nodejs-app.service
5. In the service file, add the following configuration (replace paths with your actual file paths):
6. [Unit]
7. Description=Node.js App
8. After=network.target
9. [Service]
10. ExecStart=/usr/bin/node /path/to/your/app.js
11. WorkingDirectory=/path/to/your
12. StandardOutput=inherit
13. StandardError=inherit
14. Restart=always
15. User=yourusername
16. Group=yourgroup
17. Environment=PATH=/usr/bin:/usr/local/bin
18. Environment=NODE\_ENV=production
19. RestartSec=10
20. [Install]
21. WantedBy=multi-user.target
    * **ExecStart**: The command to run your Node.js app (replace /path/to/your/app.js with the full path to your application).
    * **WorkingDirectory**: The directory where your app is located.
    * **User** and **Group**: The user and group under which the service will run. This should be a non-root user for security reasons.
    * **Restart=always**: This ensures that the service will restart if it crashes.
    * **RestartSec=10**: Wait 10 seconds before restarting the service.

**2. Reload systemd and Start the Service**

Once you've created the service file, reload the systemd configuration and start your service.

sudo systemctl daemon-reload # Reload systemd to recognize the new service

sudo systemctl start nodejs-app.service # Start the service

sudo systemctl enable nodejs-app.service # Enable the service to start on boot

**3. Check the Status of the Service**

You can check the status of your service using the following command:

sudo systemctl status nodejs-app.service

This will show you if the service is running or if there are any issues.

**4. Logs**

To view logs for your Node.js app running as a service, you can use the following command:

journalctl -u nodejs-app.service

**Method 2: Using pm2 to Manage the Node.js App**

pm2 is a popular process manager for Node.js applications. It allows you to run, monitor, and manage your Node.js applications as background services.

**1. Install pm2**

First, install pm2 globally if you haven't already:

npm install -g pm2

**2. Start Your Node.js Application with pm2**

Navigate to the directory where your app.js file is located and start the app using pm2:

pm2 start app.js --name "nodejs-app"

This will start your Node.js app in the background and assign it a name (nodejs-app).

**3. Set up pm2 to Start on Boot**

To make sure your Node.js application starts automatically when the server reboots, you need to configure pm2 to start on boot. You can do this with the following command:

pm2 startup

This will generate a command that you can run to configure your system to start pm2 on boot. It should look something like this:

sudo env PATH=$PATH:/usr/bin pm2 startup systemd -u yourusername --hp /home/yourusername

Run the command it outputs to set up the startup process.

**4. Save the pm2 Process List**

Now, save the process list so that pm2 knows which applications to restart when the system reboots:

pm2 save

**5. Check the Status of Your Application**

You can check the status of your app with the following command:

pm2 status

This will show you a list of all applications managed by pm2 and their status.

**6. Logs**

To view logs of your application managed by pm2, you can use:

pm2 logs

This will show both the output and error logs for your app.

**Conclusion**

You have two main options to run a Node.js app as a service on Unix-based systems:

1. **Using systemd**: This is ideal for most modern Linux distributions. It gives you fine-grained control over the app as a service, and you can manage it using standard Linux tools like systemctl.
2. **Using pm2**: This is a simpler and more specialized tool for managing Node.js applications. It provides features like monitoring, log management, and automatic restart of your app if it crashes.

Choose the method that best fits your needs. If you're managing a single app or just want an easy way to keep your app running, pm2 is a great choice. If you're looking for more advanced management or integration with the system's service manager, systemd is the better option.

To add logging for observability in a Node.js application, you can use a logging library like **Winston** or **Pino**. These libraries provide flexible logging mechanisms and are widely used in production environments to capture logs, monitor the health of an application, and troubleshoot issues.

In this guide, I'll demonstrate how to integrate **Winston** into your Node.js app for logging and provide insights into different log levels, log formatting, and log file handling.

**Step 1: Install Winston**

First, you need to install the Winston package:

npm install winston

**Step 2: Set Up Winston in Your Node.js App**

You will need to set up a logger in your application, usually at the beginning of your app.js or main application file. Here's how to integrate **Winston** for logging:

**Example: Setting Up Winston Logger**

const express = require('express');

const winston = require('winston');

const path = require('path');

const fs = require('fs');

// Create a log folder if it doesn't exist

const logDir = 'logs';

if (!fs.existsSync(logDir)) {

fs.mkdirSync(logDir);

}

// Create a Winston logger instance

const logger = winston.createLogger({

level: 'info', // Minimum log level

format: winston.format.combine(

winston.format.timestamp({ format: 'YYYY-MM-DD HH:mm:ss' }), // Add timestamp

winston.format.printf(({ timestamp, level, message }) => {

return `${timestamp} [${level}]: ${message}`;

})

),

transports: [

// Log to a file

new winston.transports.File({ filename: path.join(logDir, 'app.log') }),

// Log to the console (for development)

new winston.transports.Console({ format: winston.format.simple() })

]

});

// Optional: Log to different files for different log levels

logger.add(new winston.transports.File({

filename: path.join(logDir, 'error.log'),

level: 'error' // Only log error messages in this file

}));

const app = express();

const port = 3000;

// Simple middleware to log every request

app.use((req, res, next) => {

logger.info(`Request received: ${req.method} ${req.originalUrl}`);

next();

});

// Example routes

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

logger.info('GET / route accessed');

res.send('Hello, world!');

});

app.post('/api/data', (req, res) => {

logger.info('POST /api/data route accessed');

// Simulate a warning log

logger.warn('Potential issue with the data submission');

res.status(201).json({ message: 'Data received' });

});

app.use((err, req, res, next) => {

// Error logging

logger.error(`Error occurred: ${err.message}`);

res.status(500).send('Something went wrong!');

});

// Start the server

app.listen(port, () => {

logger.info(`Server running at http://localhost:${port}`);

});

**Explanation:**

1. **Winston Logger Setup**:
   * **Level**: The level field in the logger determines the minimum level of logs that will be recorded. In this case, it's set to info, meaning it will record all logs of level info and above (info, warn, error).
   * **Format**: The logs are formatted with a timestamp and a simple output that shows the log level and the log message.
   * **Transports**: Winston allows logs to be directed to various places (called transports). We’re logging to two destinations:
     + **File**: Logs are saved to a file called app.log.
     + **Console**: Logs are also output to the console in a simple format for easy debugging.
   * **Error File**: Logs for errors (level: 'error') are stored separately in error.log.
2. **Logging in Routes**:
   * We use logger.info(), logger.warn(), and logger.error() in different parts of the application to log messages at different levels:
     + info for regular log messages (e.g., route access).
     + warn for warnings or potential issues.
     + error for errors that occur within the app.
3. **Middleware for Request Logging**:
   * The middleware logs every incoming request to the app with logger.info(), making it easier to monitor all traffic in and out of the server.
4. **Error Handling**:
   * In case of errors, the error message is logged with logger.error(), and the error is sent as a response.

**Step 3: Use Different Log Levels**

Winston supports several log levels, including:

* **info**: General information messages (e.g., server started, route accessed).
* **warn**: Warnings, which indicate something unexpected but not necessarily an error.
* **error**: Errors, indicating problems that may need attention.

You can adjust the log level depending on your use case, like debug, verbose, info, warn, error, etc.

**Step 4: Test the Logging**

1. **Start the server**:
2. node app.js
3. **Access the application**:

Open a browser or use curl to send requests:

curl http://localhost:3000/

curl -X POST http://localhost:3000/api/data

1. **Check the Logs**:
   * The logs will be saved in the logs/ folder.
   * Check the app.log file for general logs:
   * cat logs/app.log
   * Check the error.log file for any errors:
   * cat logs/error.log

**Step 5: Optional Enhancements**

1. **Log Rotation**: Over time, log files can grow large. You can use the winston-daily-rotate-file transport to rotate logs daily or based on file size.
   * Install the package:
   * npm install winston-daily-rotate-file
   * Example configuration with daily rotation:
   * const DailyRotateFile = require('winston-daily-rotate-file');
   * const rotateTransport = new DailyRotateFile({
   * filename: path.join(logDir, 'application-%DATE%.log'),
   * datePattern: 'YYYY-MM-DD',
   * zippedArchive: true,
   * maxSize: '20m',
   * maxFiles: '14d' // Retain logs for 14 days
   * });
   * logger.add(rotateTransport);
2. **Logging User Activities**: You can also add detailed logs for user activities, database queries, or any external API requests to improve observability.
3. **Error Tracking Tools**: Integrate your logging with error tracking tools like **Sentry** or **Loggly** for advanced observability and real-time monitoring.

**Conclusion**

Adding logging to your Node.js application enhances observability, allowing you to track application behavior, errors, and user interactions. By integrating **Winston**, you can manage different log levels, log to multiple destinations (files and console), and implement error tracking for easier debugging.

With this setup, your application will be better equipped to handle production environments and provide the necessary information for maintenance and troubleshooting.

To integrate **MariaDB** with a Node.js application, you can use the popular mysql2 package, which is compatible with MariaDB. This allows you to interact with a MariaDB database for storing and retrieving data.

Here’s a step-by-step guide to set up **MariaDB** with Node.js using the mysql2 package:

**Step 1: Install MariaDB Server**

1. **Install MariaDB on your system** (if you haven't already). For example, on a **Debian-based** Linux system (such as Ubuntu), you can install MariaDB by running:
2. sudo apt update
3. sudo apt install mariadb-server
4. **Start MariaDB** and enable it to run on startup:
5. sudo systemctl start mariadb
6. sudo systemctl enable mariadb
7. **Secure the MariaDB installation** (optional but recommended):
8. sudo mysql\_secure\_installation

Follow the prompts to set up a root password and secure your installation.

1. **Create a database** for your Node.js application (e.g., nodejs\_app):
2. sudo mysql -u root -p
3. CREATE DATABASE nodejs\_app;
4. CREATE USER 'nodeuser'@'localhost' IDENTIFIED BY 'your\_password';
5. GRANT ALL PRIVILEGES ON nodejs\_app.\* TO 'nodeuser'@'localhost';
6. FLUSH PRIVILEGES;

**Step 2: Install the mysql2 Package**

In your Node.js application, you’ll use the mysql2 package to connect to MariaDB.

1. **Install the mysql2 package**:
2. npm install mysql2

**Step 3: Connect to MariaDB from Node.js**

You can now create a database connection and perform queries using the mysql2 package.

**Example: Integrating MariaDB with Node.js**

1. **Set up a new file** (e.g., db.js) for the database connection.

const mysql = require('mysql2');

// Create a connection to the MariaDB database

const db = mysql.createConnection({

host: 'localhost', // MariaDB host

user: 'nodeuser', // The database user

password: 'your\_password',// The user's password

database: 'nodejs\_app' // The database to connect to

});

// Connect to the database

db.connect((err) => {

if (err) {

console.error('Error connecting to the database:', err.stack);

return;

}

console.log('Connected to MariaDB as ID ' + db.threadId);

});

module.exports = db;

This file will export the database connection, so you can import it into your application’s main files.

1. **Create a simple Express app that performs CRUD operations on the database**.

const express = require('express');

const db = require('./db'); // Import the database connection

const app = express();

const port = 3000;

// Middleware to parse JSON bodies

app.use(express.json());

// Create a new item in the database

app.post('/api/items', (req, res) => {

const { name, description } = req.body;

const query = 'INSERT INTO items (name, description) VALUES (?, ?)';

db.query(query, [name, description], (err, result) => {

if (err) {

return res.status(500).json({ message: 'Error inserting item into database' });

}

res.status(201).json({ message: 'Item created', itemId: result.insertId });

});

});

// Get all items from the database

app.get('/api/items', (req, res) => {

const query = 'SELECT \* FROM items';

db.query(query, (err, rows) => {

if (err) {

return res.status(500).json({ message: 'Error fetching items from database' });

}

res.json(rows);

});

});

// Get a specific item by ID

app.get('/api/items/:id', (req, res) => {

const { id } = req.params;

const query = 'SELECT \* FROM items WHERE id = ?';

db.query(query, [id], (err, rows) => {

if (err) {

return res.status(500).json({ message: 'Error fetching item from database' });

}

if (rows.length === 0) {

return res.status(404).json({ message: 'Item not found' });

}

res.json(rows[0]);

});

});

// Update an existing item

app.put('/api/items/:id', (req, res) => {

const { id } = req.params;

const { name, description } = req.body;

const query = 'UPDATE items SET name = ?, description = ? WHERE id = ?';

db.query(query, [name, description, id], (err, result) => {

if (err) {

return res.status(500).json({ message: 'Error updating item in database' });

}

if (result.affectedRows === 0) {

return res.status(404).json({ message: 'Item not found' });

}

res.json({ message: 'Item updated' });

});

});

// Delete an item

app.delete('/api/items/:id', (req, res) => {

const { id } = req.params;

const query = 'DELETE FROM items WHERE id = ?';

db.query(query, [id], (err, result) => {

if (err) {

return res.status(500).json({ message: 'Error deleting item from database' });

}

if (result.affectedRows === 0) {

return res.status(404).json({ message: 'Item not found' });

}

res.json({ message: 'Item deleted' });

});

});

// Start the server

app.listen(port, () => {

console.log(`Server running at http://localhost:${port}`);

});

**Step 4: Create the Database Table**

Before running the Node.js app, you need to create a table in the MariaDB database to store the data. You can execute the following SQL command inside your MariaDB client:

USE nodejs\_app;

CREATE TABLE items (

id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

description TEXT NOT NULL

);

**Step 5: Run Your Node.js Application**

1. Start the Node.js application by running:
2. node app.js
3. The app will be running on http://localhost:3000, and you can interact with the API using Postman or curl:
   * **GET** all items:
   * curl http://localhost:3000/api/items
   * **POST** a new item:
   * curl -X POST http://localhost:3000/api/items -H "Content-Type: application/json" -d '{"name": "Item 1", "description": "This is item 1"}'
   * **GET** a specific item by ID:
   * curl http://localhost:3000/api/items/1
   * **PUT** to update an item:
   * curl -X PUT http://localhost:3000/api/items/1 -H "Content-Type: application/json" -d '{"name": "Updated Item", "description": "Updated description"}'
   * **DELETE** an item:
   * curl -X DELETE http://localhost:3000/api/items/1

**Conclusion**

You’ve now successfully integrated MariaDB with your Node.js application using the mysql2 package. This setup allows you to interact with your MariaDB database, storing and retrieving data for your app.

You can now extend this app by adding more routes, performing complex queries, or incorporating advanced features like pagination or joins depending on your use case.

If you need further enhancements or have specific questions, feel free to ask!

To call another API from your Node.js Express app, you can use the popular axios library, which makes it easy to perform HTTP requests (GET, POST, etc.) to external APIs.

Here’s how to handle different API call scenarios using axios from your Node.js Express server:

**Step 1: Install axios**

First, you need to install the axios library:

npm install axios

**Step 2: Make API Calls Using axios**

Below are examples of how you can call external APIs for each scenario you mentioned.

**1. POST - JSON Input**

To send a JSON object via a POST request to another API, you can use axios.post with the appropriate data.

const express = require('express');

const axios = require('axios');

const app = express();

app.use(express.json());

app.post('/call-api/json', async (req, res) => {

const inputData = req.body; // JSON input from the client

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, inputData, {

headers: {

'Content-Type': 'application/json',

},

});

res.status(200).json(response.data); // Return the response from the external API

} catch (error) {

console.error('Error calling the external API:', error);

res.status(500).json({ message: 'Error calling external API' });

}

});

**Explanation**:

* You use axios.post to send a JSON body to the external API.
* The response from the external API is then sent back to the client.

**2. GET Request**

To make a simple GET request to another API and retrieve data, you can use axios.get.

app.get('/call-api/get', async (req, res) => {

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.get(apiUrl);

res.status(200).json(response.data); // Send back the data received from the external API

} catch (error) {

console.error('Error calling the external API:', error);

res.status(500).json({ message: 'Error calling external API' });

}

});

**Explanation**:

* axios.get is used to send a GET request to the external API.
* The response is sent back to the client.

**3. POST - Form Input**

To send form data via POST, you can use the FormData object. You will also need to set the correct content type (multipart/form-data).

const FormData = require('form-data');

app.post('/call-api/form', async (req, res) => {

const form = new FormData();

form.append('name', req.body.name);

form.append('email', req.body.email);

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: {

...form.getHeaders(), // Automatically set the right content-type

},

});

res.status(200).json(response.data); // Return the response from the external API

} catch (error) {

console.error('Error calling the external API:', error);

res.status(500).json({ message: 'Error calling external API' });

}

});

**Explanation**:

* A FormData object is created and populated with the form data (e.g., name, email).
* The form.getHeaders() function automatically generates the correct Content-Type for multipart/form-data.

**4. POST - File and JSON Input**

To send both a file and JSON data, you can use FormData as well. The file will be sent as multipart/form-data, and the JSON data can be included in the same form.

const fs = require('fs');

const path = require('path');

app.post('/call-api/file-json', async (req, res) => {

const form = new FormData();

form.append('file', fs.createReadStream(path.join(\_\_dirname, 'path\_to\_your\_file.jpg'))); // Add the file

form.append('json', JSON.stringify({ key: 'value', anotherKey: 'anotherValue' })); // Add JSON data

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: {

...form.getHeaders(), // Automatically set the right content-type

},

});

res.status(200).json(response.data); // Return the response from the external API

} catch (error) {

console.error('Error calling the external API:', error);

res.status(500).json({ message: 'Error calling external API' });

}

});

**Explanation**:

* The file is appended using fs.createReadStream() to read the file from the file system.
* JSON data is serialized using JSON.stringify() and appended to the form data.

**5. POST - File and Form Input**

To send both a file and form data (i.e., non-JSON values like plain text fields), use FormData and append both the file and form fields.

app.post('/call-api/file-form', async (req, res) => {

const form = new FormData();

form.append('file', fs.createReadStream(path.join(\_\_dirname, 'path\_to\_your\_file.jpg'))); // Add the file

form.append('name', req.body.name); // Add form data (e.g., name)

form.append('description', req.body.description); // Add form data (e.g., description)

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: {

...form.getHeaders(), // Automatically set the right content-type

},

});

res.status(200).json(response.data); // Return the response from the external API

} catch (error) {

console.error('Error calling the external API:', error);

res.status(500).json({ message: 'Error calling external API' });

}

});

**Explanation**:

* This approach allows you to send a file along with form data (non-JSON data like name, description).
* The FormData object is used to append the file and form data.

**Complete Example of the Express App**

const express = require('express');

const axios = require('axios');

const FormData = require('form-data');

const fs = require('fs');

const path = require('path');

const app = express();

app.use(express.json());

// POST - JSON input

app.post('/call-api/json', async (req, res) => {

const inputData = req.body;

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, inputData, {

headers: { 'Content-Type': 'application/json' },

});

res.status(200).json(response.data);

} catch (error) {

res.status(500).json({ message: 'Error calling external API' });

}

});

// GET Request

app.get('/call-api/get', async (req, res) => {

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.get(apiUrl);

res.status(200).json(response.data);

} catch (error) {

res.status(500).json({ message: 'Error calling external API' });

}

});

// POST - Form Input

app.post('/call-api/form', async (req, res) => {

const form = new FormData();

form.append('name', req.body.name);

form.append('email', req.body.email);

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: { ...form.getHeaders() },

});

res.status(200).json(response.data);

} catch (error) {

res.status(500).json({ message: 'Error calling external API' });

}

});

// POST - File and JSON Input

app.post('/call-api/file-json', async (req, res) => {

const form = new FormData();

form.append('file', fs.createReadStream(path.join(\_\_dirname, 'path\_to\_file.jpg')));

form.append('json', JSON.stringify({ key: 'value' }));

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: { ...form.getHeaders() },

});

res.status(200).json(response.data);

} catch (error) {

res.status(500).json({ message: 'Error calling external API' });

}

});

// POST - File and Form Input

app.post('/call-api/file-form', async (req, res) => {

const form = new FormData();

form.append('file', fs.createReadStream(path.join(\_\_dirname, 'path\_to\_file.jpg')));

form.append('name', req.body.name);

form.append('description', req.body.description);

const apiUrl = 'https://external-api.com/endpoint';

try {

const response = await axios.post(apiUrl, form, {

headers: { ...form.getHeaders() },

}); res.status(200).json(response.data); } catch (error) { res.status(500).json({ message: 'Error calling external API' }); } });

const port = 3000; app.listen(port, () => { console.log(Server running on http://localhost:${port}); });

### Conclusion

This example shows how to interact with different types of APIs (POST with JSON, GET, POST with form data, file upload with JSON, and file upload with form data) from a Node.js Express server using `axios`. You can expand on this by adding more sophisticated error handling, validation, and logging as needed.

To create an API endpoint in Node.js (with Express) that accepts both a **file** and **JSON input** (where the JSON data requires some manipulation), you can use **multer** for handling the file upload and process the JSON data accordingly.

Here's how you can set this up:

**Step 1: Install Dependencies**

You'll need to install multer for handling file uploads and axios for sending the request (if you're forwarding the file somewhere else).

npm install express multer axios

* **multer** is used to handle the file upload, allowing the file to be sent in a multipart/form-data format along with JSON data.
* **axios** (optional) is used if you want to forward the file and manipulated JSON to another API.

**Step 2: Create the Express Application**

1. **Set up the necessary dependencies and file upload handling.**
2. **Extract the file and JSON input.**
3. **Manipulate the JSON data as required.**
4. **Forward the manipulated JSON and the file (if needed) to another service (using axios).**

Here's an example code to achieve this:

const express = require('express');

const multer = require('multer');

const axios = require('axios');

const fs = require('fs');

const path = require('path');

const app = express();

const port = 3000;

// Setup multer for file upload

const storage = multer.diskStorage({

destination: (req, file, cb) => {

cb(null, './uploads'); // Uploads folder to store files

},

filename: (req, file, cb) => {

cb(null, Date.now() + path.extname(file.originalname)); // Unique file name

},

});

const upload = multer({ storage: storage });

// Ensure the 'uploads' directory exists

if (!fs.existsSync('./uploads')) {

fs.mkdirSync('./uploads');

}

// API endpoint that accepts a file and JSON input

app.post('/api/file-json', upload.single('file'), async (req, res) => {

try {

// Extract the file and JSON from the request

const file = req.file; // File uploaded via form-data

const jsonData = req.body.jsonData; // JSON input sent in the form

if (!file || !jsonData) {

return res.status(400).json({ message: 'File and JSON data are required.' });

}

// Manipulate the JSON data

let manipulatedData;

try {

manipulatedData = JSON.parse(jsonData); // Parse JSON string

// Example manipulation: add a field to the JSON object

manipulatedData.timestamp = new Date().toISOString();

manipulatedData.fileName = file.filename; // Attach file name to JSON

} catch (err) {

return res.status(400).json({ message: 'Invalid JSON data.', error: err.message });

}

// Log the manipulated data

console.log('Manipulated Data:', manipulatedData);

// Optional: Forward the file and manipulated JSON to another API

const forwardApiUrl = 'https://external-api.com/endpoint';

const form = new FormData();

form.append('file', fs.createReadStream(file.path));

form.append('json', JSON.stringify(manipulatedData)); // Send manipulated JSON as string

// Send to the external API

try {

const response = await axios.post(forwardApiUrl, form, {

headers: { ...form.getHeaders() },

});

res.status(200).json(response.data);

} catch (err) {

console.error('Error forwarding to external API:', err);

res.status(500).json({ message: 'Error forwarding to external API' });

}

} catch (err) {

console.error('Error processing file or JSON:', err);

res.status(500).json({ message: 'Error processing file and JSON data', error: err.message });

}

});

// Start the server

app.listen(port, () => {

console.log(`Server running on http://localhost:${port}`);

});

**Explanation of Code:**

1. **Multer Setup**:
   * The multer middleware is set up to handle file uploads. The storage configuration specifies where to store the uploaded files (./uploads folder), and each file is given a unique name based on the current timestamp.
2. **API Endpoint (/api/file-json)**:
   * The endpoint accepts both the file and the JSON data. The file is uploaded using the upload.single('file') middleware (which expects the file to be included in the request under the field name file).
   * The JSON data is retrieved from req.body.jsonData. This assumes that the client is sending the JSON as part of the form-data request body.
   * The JSON data is manipulated by parsing it and adding additional fields (timestamp and fileName).
3. **Forwarding the File and JSON to Another API**:
   * After manipulating the JSON data, the file and JSON are sent to another API (if necessary) using the axios library.
   * We use FormData to construct the request with both the file and JSON data. The file is read using fs.createReadStream() to send it as part of the request.
4. **Error Handling**:
   * Error handling ensures that invalid inputs (missing file or JSON, malformed JSON) are appropriately caught and sent back with a helpful message.

**Step 3: Testing the API**

To test the above API, you need to send a POST request with multipart/form-data that contains:

* A **file** (use the file field).
* A **JSON object** (use the jsonData field).

**Example curl Command**:

curl -X POST http://localhost:3000/api/file-json \

-F "file=@path\_to\_your\_file.jpg" \

-F "jsonData={\"name\":\"test\",\"description\":\"This is a test\"}"

**Explanation**:

* -F "file=@path\_to\_your\_file.jpg" uploads the file from your local machine.
* -F "jsonData={\"name\":\"test\",\"description\":\"This is a test\"}" sends a JSON object in the jsonData field.

**Conclusion**

This setup allows you to:

* Accept both file uploads and JSON input in a single request.
* Manipulate the JSON data (such as adding timestamps or attaching the filename).
* Optionally forward the file and the manipulated JSON to another external API.

You can extend the code to handle other scenarios or add validation, logging, and other features based on your needs.

To create a **JWE (JSON Web Encryption) token** for user authentication in Node.js, we will use the following steps:

1. **Generate RSA Private and Public Keys** for encryption and decryption.
2. **Use the jose library** to generate and verify JWE tokens.
3. **Store RSA keys in memory or Redis**.
4. **Verify the JWE token in subsequent API calls** by checking the Authorization header.

**Prerequisites:**

1. **Install Required Dependencies**:
   * jose: Library for creating and verifying JWE tokens.
   * crypto: Built-in Node.js module for RSA key generation.
   * redis (optional, if you want to store keys in Redis).

Run the following command to install the necessary libraries:

npm install jose redis dotenv

**Step 1: Generate RSA Private and Public Keys**

RSA keys can be generated using Node.js' **crypto** module. We'll generate a private and public key pair and store them either in memory or Redis.

**Step 2: Create JWE Token for User Authentication**

The **JWE** token will be created with the RSA public key for encryption, and it will be decrypted using the corresponding private key.

**Step 3: Verify JWE Token Using Authorization Header**

The **JWE token** is passed in the Authorization header, and it will be verified using the RSA private key (from either memory or Redis).

**Full Code Example**

**Step 1: Set Up Environment**

Create a .env file to store configuration details for Redis and any other sensitive information:

REDIS\_HOST=localhost

REDIS\_PORT=6379

**Step 2: Generate RSA Keys**

The following code demonstrates how to generate RSA keys and store them in memory or Redis. It also includes functions for generating and verifying the JWE token.

const { generateKeyPairSync, privateEncrypt, publicDecrypt } = require('crypto');

const { JWE, JWK } = require('jose');

const redis = require('redis');

const dotenv = require('dotenv');

dotenv.config();

// Generate RSA Key Pair

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

modulusLength: 2048, // Key size in bits

publicKeyEncoding: {

type: 'spki',

format: 'pem',

},

privateKeyEncoding: {

type: 'pkcs8',

format: 'pem',

},

});

// Store the RSA keys (Option 1 - In-memory, Option 2 - Redis)

let keys = {

privateKey: privateKey,

publicKey: publicKey,

};

const redisClient = redis.createClient({

host: process.env.REDIS\_HOST,

port: process.env.REDIS\_PORT,

});

redisClient.on('error', (err) => {

console.log('Redis error:', err);

});

// Optionally store the keys in Redis

redisClient.set('privateKey', privateKey);

redisClient.set('publicKey', publicKey);

// Function to generate JWE token

const generateJWE = (payload) => {

const jwe = JWE.encrypt(JSON.stringify(payload), JWK.asKey(publicKey), {

alg: 'RSA-OAEP',

enc: 'A256GCM',

});

return jwe;

};

// Function to verify JWE token

const verifyJWE = (token) => {

// Option 1 - In-memory keys

const decodedToken = JWE.decrypt(token, JWK.asKey(keys.privateKey));

return JSON.parse(decodedToken);

};

// Option 2 - Using Redis to retrieve the private key and verify JWE

const verifyJWEWithRedis = (token) => {

return new Promise((resolve, reject) => {

redisClient.get('privateKey', (err, privateKey) => {

if (err || !privateKey) {

reject('Private key not found in Redis');

}

const decodedToken = JWE.decrypt(token, JWK.asKey(privateKey));

resolve(JSON.parse(decodedToken));

});

});

};

// Express setup

const express = require('express');

const app = express();

const port = 3000;

app.use(express.json());

// Endpoint to generate a JWE token for authentication

app.post('/generate-token', (req, res) => {

const { username, email } = req.body;

const payload = {

username,

email,

exp: Math.floor(Date.now() / 1000) + 60 \* 60, // 1 hour expiry

};

const token = generateJWE(payload);

res.json({ token });

});

// Endpoint to verify the JWE token

app.post('/verify-token', async (req, res) => {

const { token } = req.body;

try {

// Option 1 - In-memory keys

const userData = verifyJWE(token);

res.json({ message: 'Token verified successfully!', data: userData });

// Option 2 - Redis keys

// const userData = await verifyJWEWithRedis(token);

// res.json({ message: 'Token verified successfully!', data: userData });

} catch (error) {

res.status(400).json({ message: 'Invalid or expired token', error });

}

});

// Middleware to check authorization header for JWE token

app.use((req, res, next) => {

const authHeader = req.headers['authorization'];

if (!authHeader) {

return res.status(403).json({ message: 'No authorization header provided' });

}

const token = authHeader.split(' ')[1]; // Bearer <token>

if (!token) {

return res.status(403).json({ message: 'Invalid authorization header format' });

}

try {

// Option 1 - In-memory keys

const userData = verifyJWE(token);

req.user = userData; // Add user data to the request object

next();

// Option 2 - Redis keys

// await verifyJWEWithRedis(token);

// next();

} catch (error) {

res.status(401).json({ message: 'Invalid or expired token', error });

}

});

// Sample protected route

app.get('/protected', (req, res) => {

res.json({ message: 'This is a protected route', user: req.user });

});

// Start the server

app.listen(port, () => {

console.log(`Server running at http://localhost:${port}`);

});

**Explanation:**

1. **RSA Key Generation**:
   * generateKeyPairSync('rsa', { modulusLength: 2048 }) is used to generate a 2048-bit RSA key pair (private and public keys).
   * The public and private keys are encoded in PEM format, which is commonly used for encryption and decryption.
2. **In-memory Storage (Option 1)**:
   * The RSA keys are stored in memory in the keys object, which allows for fast access.
   * The generateJWE function encrypts a payload using the **public key** and creates a JWE token.
3. **Redis Storage (Option 2)**:
   * Alternatively, the keys are stored in **Redis** using redis.set() to persist them for later use.
   * The verifyJWEWithRedis function retrieves the **private key** from Redis and decrypts the token.
4. **Generating and Verifying the JWE Token**:
   * The generateJWE function encrypts the payload using the **RSA public key** and returns the JWE token.
   * The verifyJWE function decrypts the JWE token using the **RSA private key** (either in-memory or from Redis).
5. **Authorization Middleware**:
   * The authorization middleware checks for the Authorization header in incoming requests.
   * The JWE token is extracted and validated before accessing any protected routes.
6. **Protected Route**:
   * /protected is a sample protected route that requires a valid JWE token to access.

**Step 3: Testing the Application**

1. **Generate the JWE Token**:

You can send a POST request to /generate-token with a JSON payload like:

{

"username": "john\_doe",

"email": "john.doe@example.com"

}

This will return a JWE token.

1. **Verify the JWE Token**:

Send a POST request to /verify-token with the token you received:

{

"token": "your\_jwe\_token"

}

1. **Access Protected Route**:

Send a GET request to /protected with the Authorization header:

Authorization: Bearer your\_jwe\_token

**Conclusion**

This solution demonstrates how to generate and verify a **JWE token** using RSA keys in **Node.js**, with the ability to store the keys in either **memory** or **Redis**. This setup ensures secure user authentication with encrypted tokens, and the server can verify these tokens before granting access to protected routes.

To implement **in-memory caching** with **TTL (Time-to-Live)** for storing RSA keys in Node.js, we can use a simple in-memory cache that stores keys and automatically expires after a specified duration.

Here's how to implement the solution:

1. **In-Memory Cache with TTL**: Use a cache object in memory to store the RSA keys and set a TTL for expiration.
2. **Expire Keys Automatically**: After the TTL is reached, the keys will be deleted from the cache.

You can use the built-in setTimeout to handle TTL expiration in the cache.

**Step-by-Step Implementation**

**Step 1: Define the In-Memory Cache with TTL**

We'll store RSA keys in memory with a TTL, and use setTimeout to expire the cache.

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

const { JWE, JWK } = require('jose');

const express = require('express');

const dotenv = require('dotenv');

dotenv.config();

const app = express();

app.use(express.json());

// In-Memory Cache with TTL

const cache = {

keys: null,

expiration: null,

};

const CACHE\_TTL = 60000; // TTL in milliseconds (1 minute for this example)

// Generate RSA Key Pair

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

modulusLength: 2048, // Key size in bits

publicKeyEncoding: {

type: 'spki',

format: 'pem',

},

privateKeyEncoding: {

type: 'pkcs8',

format: 'pem',

},

});

// Store keys in cache with TTL

const storeKeysInCache = () => {

cache.keys = { privateKey, publicKey };

cache.expiration = Date.now() + CACHE\_TTL;

// Set a timeout to clear the cache after the TTL expires

setTimeout(() => {

cache.keys = null;

cache.expiration = null;

console.log('Keys expired and cleared from cache');

}, CACHE\_TTL);

};

// Function to generate JWE token

const generateJWE = (payload) => {

if (!cache.keys || Date.now() > cache.expiration) {

console.log('Cache expired, regenerating RSA keys');

storeKeysInCache(); // Re-store keys if expired

}

const jwe = JWE.encrypt(JSON.stringify(payload), JWK.asKey(cache.keys.publicKey), {

alg: 'RSA-OAEP',

enc: 'A256GCM',

});

return jwe;

};

// Function to verify JWE token

const verifyJWE = (token) => {

if (!cache.keys || Date.now() > cache.expiration) {

console.log('Cache expired, regenerating RSA keys');

storeKeysInCache(); // Re-store keys if expired

}

const decodedToken = JWE.decrypt(token, JWK.asKey(cache.keys.privateKey));

return JSON.parse(decodedToken);

};

// Endpoint to generate a JWE token for authentication

app.post('/generate-token', (req, res) => {

const { username, email } = req.body;

const payload = {

username,

email,

exp: Math.floor(Date.now() / 1000) + 60 \* 60, // 1 hour expiry

};

const token = generateJWE(payload);

res.json({ token });

});

// Endpoint to verify the JWE token

app.post('/verify-token', (req, res) => {

const { token } = req.body;

try {

const userData = verifyJWE(token);

res.json({ message: 'Token verified successfully!', data: userData });

} catch (error) {

res.status(400).json({ message: 'Invalid or expired token', error });

}

});

// Middleware to check authorization header for JWE token

app.use((req, res, next) => {

const authHeader = req.headers['authorization'];

if (!authHeader) {

return res.status(403).json({ message: 'No authorization header provided' });

}

const token = authHeader.split(' ')[1]; // Bearer <token>

if (!token) {

return res.status(403).json({ message: 'Invalid authorization header format' });

}

try {

const userData = verifyJWE(token);

req.user = userData; // Add user data to the request object

next();

} catch (error) {

res.status(401).json({ message: 'Invalid or expired token', error });

}

});

// Sample protected route

app.get('/protected', (req, res) => {

res.json({ message: 'This is a protected route', user: req.user });

});

// Start the server

const port = 3000;

app.listen(port, () => {

console.log(`Server running at http://localhost:${port}`);

storeKeysInCache(); // Initialize cache with keys when the server starts

});

**Explanation:**

1. **In-Memory Cache with TTL**:
   * We create an in-memory cache (cache object) that stores the keys and expiration time (cache.expiration).
   * The keys are stored using the storeKeysInCache function, which sets a TTL for the keys in the cache.
   * The setTimeout function is used to automatically clear the cache after the TTL expires (in this case, 60 seconds).
2. **JWE Token Generation**:
   * The generateJWE function generates a JWE token using the public key stored in the cache.
   * If the cache has expired (i.e., the current time is greater than cache.expiration), the keys are regenerated and stored again in the cache.
3. **JWE Token Verification**:
   * The verifyJWE function verifies the token using the private key stored in the cache.
   * If the cache has expired, the keys are regenerated and stored again.
4. **Express Server**:
   * The /generate-token endpoint generates and returns a JWE token.
   * The /verify-token endpoint verifies a JWE token.
   * The Authorization header is checked in the middleware, and the token is verified before accessing the protected route /protected.
5. **TTL**:
   * In this example, the cache TTL is set to 60 seconds (1 minute) for simplicity. You can adjust the CACHE\_TTL constant based on your requirements.

**Step 2: Testing the Application**

1. **Generate the JWE Token**:

You can send a POST request to /generate-token with a JSON payload like:

{

"username": "john\_doe",

"email": "john.doe@example.com"

}

This will return a JWE token.

1. **Verify the JWE Token**:

Send a POST request to /verify-token with the token you received:

{

"token": "your\_jwe\_token"

}

1. **Access Protected Route**:

Send a GET request to /protected with the Authorization header:

Authorization: Bearer your\_jwe\_token

**Conclusion**

In this setup, the RSA keys are stored in an in-memory cache with TTL. The keys are automatically regenerated and replaced in the cache when they expire. This ensures that the JWE token can be validated securely without the overhead of continuously generating keys. This solution is ideal for short-lived keys or situations where the RSA keys don’t need to persist for long periods of time.

To implement an in-memory **cache as a Map** where the **key** is a **request ID** and the **value** is a pair of **private** and **public keys**, we can modify the existing structure and logic slightly. This approach will allow us to store and manage multiple sets of RSA keys for different requests, keyed by their unique **request ID**.

Here's how we can adjust the code:

1. **In-Memory Cache (Map)**: We will use a Map object to store keys for each request. The request ID will be the **key**, and the value will be an object containing the privateKey and publicKey.
2. **TTL for Each Request**: Each entry in the map will have a **TTL** for expiration. When the TTL is reached, the key pair for that specific request will be removed.

**Step-by-Step Implementation**

**1. Modify the Cache to Use Map for Storing RSA Keys**

We'll store the RSA key pair in a Map where the **request ID** is the key, and the **value** is an object containing the privateKey and publicKey. We'll also handle the **TTL** for each request's key pair.

**Full Code Example**

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

const { JWE, JWK } = require('jose');

const express = require('express');

const dotenv = require('dotenv');

dotenv.config();

const app = express();

app.use(express.json());

// In-memory Cache (Map) with TTL per Request ID

const cache = new Map(); // Map to store key pairs by request ID

const CACHE\_TTL = 60000; // TTL in milliseconds (1 minute for this example)

// Function to generate RSA Key Pair and store it in the cache

const generateKeysForRequest = (requestId) => {

// Generate RSA key pair

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

modulusLength: 2048, // Key size in bits

publicKeyEncoding: {

type: 'spki',

format: 'pem',

},

privateKeyEncoding: {

type: 'pkcs8',

format: 'pem',

},

});

// Create cache entry for the request ID

const keyPair = {

privateKey,

publicKey,

expiration: Date.now() + CACHE\_TTL, // Set expiration for TTL

};

// Store in the cache

cache.set(requestId, keyPair);

// Set a timeout to remove the key pair after TTL expires

setTimeout(() => {

cache.delete(requestId);

console.log(`Cache entry for request ID ${requestId} expired and removed`);

}, CACHE\_TTL);

};

// Function to generate JWE token

const generateJWE = (requestId, payload) => {

if (!cache.has(requestId) || Date.now() > cache.get(requestId).expiration) {

console.log('Cache expired or missing for request ID, regenerating keys');

generateKeysForRequest(requestId); // Regenerate keys if expired or missing

}

const publicKey = cache.get(requestId).publicKey;

// Encrypt the payload with the public key from the cache

const jwe = JWE.encrypt(JSON.stringify(payload), JWK.asKey(publicKey), {

alg: 'RSA-OAEP',

enc: 'A256GCM',

});

return jwe;

};

// Function to verify JWE token

const verifyJWE = (requestId, token) => {

if (!cache.has(requestId) || Date.now() > cache.get(requestId).expiration) {

console.log('Cache expired or missing for request ID, regenerating keys');

generateKeysForRequest(requestId); // Regenerate keys if expired or missing

}

const privateKey = cache.get(requestId).privateKey;

// Decrypt the token with the private key from the cache

const decodedToken = JWE.decrypt(token, JWK.asKey(privateKey));

return JSON.parse(decodedToken);

};

// Endpoint to generate a JWE token for a request ID

app.post('/generate-token', (req, res) => {

const { requestId, username, email } = req.body;

const payload = {

username,

email,

exp: Math.floor(Date.now() / 1000) + 60 \* 60, // 1 hour expiry

};

generateKeysForRequest(requestId); // Ensure keys are generated and cached for the request ID

const token = generateJWE(requestId, payload);

res.json({ token });

});

// Endpoint to verify the JWE token

app.post('/verify-token', (req, res) => {

const { requestId, token } = req.body;

try {

const userData = verifyJWE(requestId, token);

res.json({ message: 'Token verified successfully!', data: userData });

} catch (error) {

res.status(400).json({ message: 'Invalid or expired token', error });

}

});

// Middleware to check authorization header for JWE token

app.use((req, res, next) => {

const authHeader = req.headers['authorization'];

if (!authHeader) {

return res.status(403).json({ message: 'No authorization header provided' });

}

const token = authHeader.split(' ')[1]; // Bearer <token>

if (!token) {

return res.status(403).json({ message: 'Invalid authorization header format' });

}

const requestId = req.headers['request-id']; // Custom header for request ID

if (!requestId) {

return res.status(400).json({ message: 'Request ID is missing in headers' });

}

try {

const userData = verifyJWE(requestId, token);

req.user = userData; // Add user data to the request object

next();

} catch (error) {

res.status(401).json({ message: 'Invalid or expired token', error });

}

});

// Sample protected route

app.get('/protected', (req, res) => {

res.json({ message: 'This is a protected route', user: req.user });

});

// Start the server

const port = 3000;

app.listen(port, () => {

console.log(`Server running at http://localhost:${port}`);

});

**Explanation:**

1. **In-Memory Cache as a Map**:
   * A Map object (cache) is used to store the RSA key pairs, where the **key** is the requestId and the **value** is an object containing the privateKey, publicKey, and expiration timestamp.
   * This allows us to store key pairs for each unique request ID separately.
2. **TTL Management**:
   * Each entry in the cache has a TTL (CACHE\_TTL) and an **expiration time**.
   * We use setTimeout to remove the key pair from the cache once the TTL expires, ensuring that the cache doesn't grow indefinitely.
3. **Key Generation**:
   * The generateKeysForRequest function generates a new RSA key pair for a given **request ID** and stores it in the cache along with the expiration time.
4. **JWE Token Generation**:
   * The generateJWE function uses the **public key** from the cache (associated with the provided requestId) to encrypt the payload and generate a JWE token.
5. **JWE Token Verification**:
   * The verifyJWE function uses the **private key** from the cache (associated with the provided requestId) to decrypt the JWE token and verify its authenticity.
6. **Request ID**:
   * The requestId is passed as part of the **body** for the /generate-token and /verify-token endpoints.
   * In the **authorization middleware**, the requestId is expected to be sent as a **custom header** (e.g., Request-ID) for token verification.

**Step 2: Testing the Application**

1. **Generate the JWE Token**:

Send a POST request to /generate-token with a JSON payload like:

{

"requestId": "req\_12345",

"username": "john\_doe",

"email": "john.doe@example.com"

}

This will return a JWE token.

1. **Verify the JWE Token**:

Send a POST request to /verify-token with the following JSON body:

{

"requestId": "req\_12345",

"token": "your\_jwe\_token"

}

1. **Access Protected Route**:

Send a GET request to /protected with the Authorization header:

Authorization: Bearer your\_jwe\_token

Request-ID: req\_12345

**Conclusion**

With this setup, the **RSA key pairs** are stored in an **in-memory cache (Map)**, where each entry is keyed by the **request ID**. This allows multiple requests to have their own RSA key pairs, with TTL to automatically expire keys when they are no longer needed. The **cache** ensures that keys are only regenerated when they are either missing or expired.

**API Design for Application**

Below is a detailed API design based on your requirements. It covers the **Super Admin**, **Admin**, **User**, and **Report Modules**.

**1. Super Admin Module API**

**1.1 Create Customer**

* **Endpoint**: POST /admin/customers
* **Description**: Create a new customer.
* **Request Body**:
* {
* "name": "Customer Name"
* }
* **Response**:
* {
* "message": "Customer created successfully",
* "customerId": 1
* }

**1.2 Create Admin User for Customer**

* **Endpoint**: POST /admin/customers/:customerId/admin
* **Description**: Create an admin user for a customer.
* **Request Body**:
* {
* "username": "admin\_username",
* "password": "admin\_password",
* "email": "admin@example.com"
* }
* **Response**:
* {
* "message": "Admin user created successfully",
* "adminId": 1
* }

**2. Admin Module (Customer-Specific) API**

**2.1 Create User for Customer**

* **Endpoint**: POST /admin/users
* **Description**: Create a new user for a specific customer.
* **Request Body**:
* {
* "customerId": 1,
* "username": "new\_user",
* "password": "password123",
* "email": "user@example.com",
* "phone": "1234567890"
* }
* **Response**:
* {
* "message": "User created successfully",
* "userId": 1
* }

**2.2 Import Excel for Users**

* **Endpoint**: POST /admin/users/import
* **Description**: Admin imports users from an Excel file.
* **Request Body**: Form-data with file (Excel)
* **Response**:
* {
* "message": "Users imported successfully",
* "importedCount": 100,
* "failedCount": 5
* }

**2.3 Update / Inactivate / Activate User**

* **Endpoint**: PATCH /admin/users/:userId
* **Description**: Update the status of a user (Activate/Inactivate).
* **Request Body**:
* {
* "status": "ACTIVE" // or "INACTIVE"
* }
* **Response**:
* {
* "message": "User status updated successfully"
* }

**2.4 Approval List for User Registration**

* **Endpoint**: GET /admin/user-approvals
* **Description**: Get a list of pending user registrations that require approval.
* **Response**:
* {
* "approvals": [
* {
* "userId": 1,
* "username": "new\_user",
* "status": "PENDING",
* "createdAt": "2024-12-01T12:00:00Z"
* }
* ]
* }

**2.5 Approve / Reject User Registration**

* **Endpoint**: POST /admin/user-approvals/:approvalId
* **Description**: Admin approves or rejects a user registration request.
* **Request Body**:
* {
* "status": "APPROVED", // or "REJECTED"
* "branchId": 2
* }
* **Response**:
* {
* "message": "User registration approved successfully"
* }

**2.6 Create Branch for Customer**

* **Endpoint**: POST /admin/branches
* **Description**: Admin creates a new branch for the customer.
* **Request Body**:
* {
* "customerId": 1,
* "name": "New Branch",
* "location": "City Location",
* "didNumber": "DID1234",
* "numberOfRetry": 3,
* "failureActionType": "RETRY"
* }
* **Response**:
* {
* "message": "Branch created successfully",
* "branchId": 1
* }

**2.7 Create App Settings for Customer**

* **Endpoint**: POST /admin/settings
* **Description**: Admin sets global settings (Retry Count, Failure Action Type, etc.).
* **Request Body**:
* {
* "customerId": 1,
* "globalRetryCount": 3,
* "globalFailureActionType": "RETRY"
* }
* **Response**:
* {
* "message": "App settings updated successfully"
* }

**2.8 Create Code Settings for Branches**

* **Endpoint**: POST /admin/codes
* **Description**: Admin creates code settings for a branch.
* **Request Body**:
* {
* "branchId": 1,
* "codeName": "CODE123",
* "didNumber": "DID1234",
* "numberOfRetry": 3,
* "failureActionType": "RETRY"
* }
* **Response**:
* {
* "message": "Code settings created successfully",
* "codeId": 1
* }

**3. User Module API**

**3.1 Register for the Customer**

* **Endpoint**: POST /user/register
* **Description**: User registers for a specific customer using customer ID.
* **Request Body**:
* {
* "customerId": 1,
* "username": "new\_user",
* "email": "user@example.com",
* "phone": "1234567890"
* }
* **Response**:
* {
* "message": "Registration successful. Awaiting approval."
* }

**3.2 Approve and Change Branch**

* **Endpoint**: POST /user/branch-change
* **Description**: User requests to change their branch after approval.
* **Request Body**:
* {
* "branchId": 2
* }
* **Response**:
* {
* "message": "Branch change request submitted successfully"
* }

**3.3 Login**

* **Endpoint**: POST /user/login
* **Description**: User logs in using username and password.
* **Request Body**:
* {
* "username": "new\_user",
* "password": "password123"
* }
* **Response**:
* {
* "message": "Login successful",
* "userId": 1,
* "token": "JWT\_TOKEN"
* }

**3.4 Register Device for User**

* **Endpoint**: POST /user/devices
* **Description**: Register a device for a logged-in user.
* **Request Body**:
* {
* "deviceId": "device\_123",
* "deviceType": "MOBILE"
* }
* **Response**:
* {
* "message": "Device registered successfully"
* }

**3.5 Load Code Dashboard**

* **Endpoint**: GET /user/codes
* **Description**: Fetch a list of codes available for the user.
* **Response**:
* {
* "codes": [
* {
* "codeId": 1,
* "codeName": "CODE123",
* "branchId": 1,
* "didNumber": "DID1234"
* }
* ]
* }

**3.6 Register Voice for Code**

* **Endpoint**: POST /user/register-voice
* **Description**: User registers their voice for a specific code.
* **Request Body**:
* {
* "codeId": 1,
* "voiceData": "voice\_file\_content"
* }
* **Response**:
* {
* "message": "Voice registered successfully for the code"
* }

**4. Reporting API**

**4.1 Fetch User Activity Reports**

* **Endpoint**: GET /admin/reports
* **Description**: Fetch activity logs for users based on action type or user.
* **Request Params**: userId, actionType (optional)
* **Response**:
* {
* "reports": [
* {
* "reportId": 1,
* "userId": 1,
* "actionType": "REGISTER\_VOICE",
* "actionDetails": "Voice data submitted",
* "createdAt": "2024-12-01T12:00:00Z"
* }
* ]
* }

**Conclusion**

This API design provides the structure for building the necessary endpoints for your application, allowing:

* **Super Admin** to manage customers and admin users.
* **Admin** to manage users, settings, branches, and approval processes.
* **User** to register, login, and interact with codes.
* **Reporting** to track user activity.

This design also covers the handling of **user registration**, **branch management**, \*\*code interactions

\*\*, **user approvals**, and **report generation**.

Certainly! Given your updated requirements, we will modify the database design to include:

1. **Separate tables for Customers and Users**, with a **mapping table** that tracks which user is associated with which customer. This mapping table will also include the **active/inactive flag** to manage user status per customer.
2. **Code as a common configuration** with additional **branch-specific configurations**. We'll also use **foreign keys** and **composite keys** for relationships between tables.

**Updated DB Schema:**

**1. Users Table (Main Users Table)**

This table stores the basic user information that is common across all customers.

CREATE TABLE users (

id INT PRIMARY KEY AUTO\_INCREMENT,

username VARCHAR(255) NOT NULL UNIQUE,

password VARCHAR(255) NOT NULL,

email VARCHAR(255),

phone VARCHAR(50),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

**2. Customers Table (Stores customer information)**

CREATE TABLE customers (

id INT PRIMARY KEY AUTO\_INCREMENT,

name VARCHAR(255) NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

**3. User-Customer Mapping Table (Maps users to customers, with active/inactive flag)**

This mapping table will allow a user to be linked to multiple customers, with an active or inactive status per customer.

CREATE TABLE user\_customer\_mapping (

user\_id INT NOT NULL,

customer\_id INT NOT NULL,

status ENUM('ACTIVE', 'INACTIVE') DEFAULT 'ACTIVE',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

PRIMARY KEY (user\_id, customer\_id),

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (customer\_id) REFERENCES customers(id)

);

* **Composite Key**: (user\_id, customer\_id) ensures that a user can only be linked to the same customer once.
* **Status Field**: The status field tracks whether the user's account for that specific customer is active or inactive.

**4. Branches Table (Stores customer branches and configurations)**

CREATE TABLE branches (

id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT NOT NULL,

name VARCHAR(255) NOT NULL,

location VARCHAR(255),

did\_number VARCHAR(50) NOT NULL,

number\_of\_retry INT DEFAULT 3,

failure\_action\_type ENUM('RETRY', 'ALERT', 'IGNORE') DEFAULT 'RETRY',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

FOREIGN KEY (customer\_id) REFERENCES customers(id)

);

**5. Codes Table (Common code configuration)**

CREATE TABLE codes (

id INT PRIMARY KEY AUTO\_INCREMENT,

code\_name VARCHAR(100) NOT NULL,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP

);

**6. Branch Code Settings Table (Links codes to specific branches)**

CREATE TABLE branch\_code\_settings (

branch\_id INT NOT NULL,

code\_id INT NOT NULL,

number\_of\_retry INT DEFAULT 3,

failure\_action\_type ENUM('RETRY', 'ALERT', 'IGNORE') DEFAULT 'RETRY',

PRIMARY KEY (branch\_id, code\_id),

FOREIGN KEY (branch\_id) REFERENCES branches(id),

FOREIGN KEY (code\_id) REFERENCES codes(id)

);

* **Composite Key**: (branch\_id, code\_id) allows multiple codes to be linked to a branch with specific settings like retry count and failure action type.

**7. Settings Table (Global settings per customer)**

CREATE TABLE settings (

id INT PRIMARY KEY AUTO\_INCREMENT,

customer\_id INT NOT NULL,

global\_retry\_count INT DEFAULT 3,

global\_failure\_action\_type ENUM('RETRY', 'ALERT', 'IGNORE') DEFAULT 'RETRY',

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

FOREIGN KEY (customer\_id) REFERENCES customers(id)

);

**8. User Devices Table (Tracks the devices registered by users)**

CREATE TABLE user\_devices (

id INT PRIMARY KEY AUTO\_INCREMENT,

user\_id INT NOT NULL,

device\_id VARCHAR(255) NOT NULL,

device\_type ENUM('MOBILE', 'TABLET', 'WEB'),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

updated\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP ON UPDATE CURRENT\_TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users(id)

);

**9. User Approval Table (Tracks user approvals and branch changes)**

CREATE TABLE user\_approval (

id INT PRIMARY KEY AUTO\_INCREMENT,

user\_id INT NOT NULL,

branch\_id INT NOT NULL,

status ENUM('PENDING', 'APPROVED', 'REJECTED') DEFAULT 'PENDING',

approval\_date TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users(id),

FOREIGN KEY (branch\_id) REFERENCES branches(id)

);

**10. Reports Table (Stores user activity logs)**

CREATE TABLE reports (

id INT PRIMARY KEY AUTO\_INCREMENT,

user\_id INT NOT NULL,

action\_type ENUM('REGISTER\_VOICE', 'SUBMIT\_CODE', 'LOGIN', 'BRANCH\_CHANGE') NOT NULL,

action\_details TEXT,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (user\_id) REFERENCES users(id)

);

**Summary of Changes:**

1. **User-Customer Mapping**:
   * The new user\_customer\_mapping table allows a user to be linked to multiple customers. The status field lets you mark the user as **ACTIVE** or **INACTIVE** for each customer.
2. **Code as a Common Configuration**:
   * The codes table stores common code configurations, and the branch\_code\_settings table links codes to branches and allows branch-specific settings such as **retry counts** and **failure actions**.
3. **Foreign Keys and Composite Keys**:
   * Foreign keys are used to enforce relationships between tables (e.g., between users, customers, branches, etc.).
   * Composite keys (e.g., (user\_id, customer\_id)) ensure unique and non-redundant mappings between users and customers, as well as branch-code associations.

**Example Use Cases and SQL Queries:**

1. **Assigning a User to a Customer**:
   * To assign an existing user to a customer, you'd insert a record into the user\_customer\_mapping table:
2. INSERT INTO user\_customer\_mapping (user\_id, customer\_id, status)
3. VALUES (1, 2, 'ACTIVE');
4. **Fetching All Active Users for a Customer**:
   * To fetch all active users for a specific customer:
5. SELECT u.id, u.username, u.email
6. FROM users u
7. JOIN user\_customer\_mapping ucm ON u.id = ucm.user\_id
8. WHERE ucm.customer\_id = 2 AND ucm.status = 'ACTIVE';
9. **Fetching Codes for a Branch**:
   * To fetch all codes linked to a specific branch, along with retry settings:
10. SELECT c.code\_name, bcs.number\_of\_retry, bcs.failure\_action\_type
11. FROM branch\_code\_settings bcs
12. JOIN codes c ON bcs.code\_id = c.id
13. WHERE bcs.branch\_id = 1;
14. **Updating User Status for a Specific Customer**:
    * To update a user's status for a specific customer (e.g., inactivating a user):
15. UPDATE user\_customer\_mapping
16. SET status = 'INACTIVE'
17. WHERE user\_id = 1 AND customer\_id = 2;
18. **Adding a Code to a Branch**:
    * To assign a code to a branch with specific settings:
19. INSERT INTO branch\_code\_settings (branch\_id, code\_id, number\_of\_retry, failure\_action\_type)
20. VALUES (1, 1, 3, 'RETRY');

This updated design ensures flexibility, scalability, and integrity by properly organizing customer-user relationships, managing branch-specific configurations for codes, and leveraging composite keys and foreign keys to maintain strong relationships between entities.

To modularize a **Node.js application** similar to **Java packaging** and organize it into different modules such as **Super Admin**, **Admin**, **User**, and **Report**, we can structure the project in a way that separates each functionality into its own directory. This will allow us to maintain cleaner code, improve scalability, and make it easier to navigate the codebase. The modules will communicate via API routes, models, services, and utilities.

Here’s a suggested structure for your Node.js project with all the functionalities outlined:

**1. Project Structure**

my-node-app/

│

├── config/ # Configuration files (e.g., database connection, app settings)

│ ├── db.js # MariaDB connection and pool configuration

│ └── config.js # Global settings (e.g., retries, failure action types)

│

├── controllers/ # API Controllers for each module

│ ├── superAdminController.js

│ ├── adminController.js

│ ├── userController.js

│ └── reportController.js

│

├── models/ # Database models for each module

│ ├── customerModel.js

│ ├── userModel.js

│ ├── branchModel.js

│ ├── deviceModel.js

│ ├── settingModel.js

│ └── codeModel.js

│

├── routes/ # API routes for each module

│ ├── superAdminRoutes.js

│ ├── adminRoutes.js

│ ├── userRoutes.js

│ └── reportRoutes.js

│

├── services/ # Business logic and services for each module

│ ├── superAdminService.js

│ ├── adminService.js

│ ├── userService.js

│ └── reportService.js

│

├── utils/ # Utility functions (e.g., file upload, JWT validation, RSA signing)

│ ├── fileUpload.js

│ ├── jwtUtils.js

│ └── rsaUtils.js

│

├── app.js # Main app file to initialize routes and middlewares

├── package.json # Project dependencies and scripts

└── .env # Environment variables (e.g., DB credentials, JWT secret)

**2. Explanation of the Structure**

**1. config/ - Configuration files**

* db.js: Contains the MariaDB database connection pool setup.
* config.js: Holds global settings like retry attempts, failure action types, and other configurable parameters.

**2. controllers/ - Handles incoming requests**

* superAdminController.js: Manages the routes and logic for **Super Admin** tasks (e.g., create customers, create admin users).
* adminController.js: Handles **Admin**-specific logic (e.g., create users for a customer, manage branches, etc.).
* userController.js: Manages **User**-related routes and logic (e.g., registration, login, device registration).
* reportController.js: Handles report-related functionality.

**3. models/ - Database models (tables and queries)**

* customerModel.js: Defines functions for managing customer-related data (CRUD operations).
* userModel.js: Defines functions for managing user-related data (CRUD operations).
* branchModel.js: Functions related to branch management (CRUD).
* deviceModel.js: Functions to manage devices (e.g., store public keys, register devices).
* settingModel.js: Functions to manage app and code settings.
* codeModel.js: Functions for managing codes, DID numbers, retries, failure actions.

**4. routes/ - API routes for each module**

* superAdminRoutes.js: Defines the routes for **Super Admin** operations.
* adminRoutes.js: Defines the routes for **Admin** operations.
* userRoutes.js: Defines the routes for **User**-related functionality.
* reportRoutes.js: Defines the routes for report generation and retrieval.

**5. services/ - Business logic layer**

* superAdminService.js: Contains business logic for super admin actions.
* adminService.js: Handles logic for admin-specific actions like user and branch management.
* userService.js: Contains logic for user registration, login, and other related actions.
* reportService.js: Handles the report-related logic.

**6. utils/ - Utility functions**

* fileUpload.js: Logic for handling file uploads (e.g., for importing Excel files).
* jwtUtils.js: Functions for JWT generation and validation.
* rsaUtils.js: Functions for handling RSA encryption/decryption and signature verification.

**3. Example Code Implementation**

**1. Database Configuration (config/db.js)**

const mysql = require('mysql2');

// Database connection pool

const pool = mysql.createPool({

host: process.env.DB\_HOST,

user: process.env.DB\_USER,

password: process.env.DB\_PASSWORD,

database: process.env.DB\_NAME,

waitForConnections: true,

connectionLimit: 10,

queueLimit: 0

});

// Helper function for queries

const query = (sql, params) => {

return new Promise((resolve, reject) => {

pool.execute(sql, params, (err, results) => {

if (err) {

return reject(err);

}

resolve(results);

});

});

};

module.exports = { query };

**2. Super Admin Controller (controllers/superAdminController.js)**

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

// Create Customer

exports.createCustomer = async (req, res) => {

try {

const { customerName, customerDetails } = req.body;

const result = await superAdminService.createCustomer(customerName, customerDetails);

res.status(201).json({ status: 'success', data: result });

} catch (error) {

res.status(500).json({ status: 'error', message: error.message });

}

};

// Create Admin User for Customer

exports.createAdminUser = async (req, res) => {

try {

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

const result = await superAdminService.createAdminUser(customerId, username, password);

res.status(201).json({ status: 'success', data: result });

} catch (error) {

res.status(500).json({ status: 'error', message: error.message });

}

};

**3. Super Admin Service (services/superAdminService.js)**

const customerModel = require('../models/customerModel');

exports.createCustomer = async (customerName, customerDetails) => {

try {

const query = 'INSERT INTO customers (name, details) VALUES (?, ?)';

const result = await customerModel.createCustomer(query, [customerName, customerDetails]);

return result;

} catch (error) {

throw new Error('Error creating customer: ' + error.message);

}

};

exports.createAdminUser = async (customerId, username, password) => {

try {

const query = 'INSERT INTO users (customer\_id, username, password) VALUES (?, ?, ?)';

const result = await customerModel.createAdminUser(query, [customerId, username, password]);

return result;

} catch (error) {

throw new Error('Error creating admin user: ' + error.message);

}

};

**4. Routes (routes/superAdminRoutes.js)**

const express = require('express');

const router = express.Router();

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

// Super Admin Routes

router.post('/create-customer', superAdminController.createCustomer);

router.post('/create-admin-user', superAdminController.createAdminUser);

module.exports = router;

**5. Main Application File (app.js)**

const express = require('express');

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

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

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

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

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

const app = express();

app.use(bodyParser.json());

// Routes

app.use('/api/superadmin', superAdminRoutes);

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

app.use('/api/user', userRoutes);

app.use('/api/reports', reportRoutes);

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

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

**4. Benefits of Modularization**

* **Separation of Concerns**: Each module (Super Admin, Admin, User, Report) is encapsulated in its own set of controllers, services, and models.
* **Easier Maintenance**: Any updates to a specific feature or module can be handled in isolation without affecting other parts of the system.
* **Scalability**: As the project grows, you can easily add more functionality or modify existing modules.
* **Reusability**: Services and models are reusable across different parts of the application (e.g., user-related logic can be reused in both the Admin and User modules).

**5. Next Steps**

* Implement **middleware** for common logic (e.g., authentication, validation).
* Add **error handling** to ensure clean error messages.
* Set up **unit tests** for each module to ensure reliability.
* Enhance security (e.g., for JWT authentication, file validation).
* Add **logging** for observability using libraries like `winston

Sure! Let's go through how to set up the **Routes**, **Controllers**, and **Services** in Node.js to handle the following scenarios:

1. **Send WAV file, Excel file, and JSON input from Front End**.
2. **Send WAV file, Excel file, and form input from Front End**.

For handling files like WAV and Excel, we'll use the **multer** package for handling file uploads in Node.js.

**Step 1: Install Required Packages**

First, install the necessary packages:

* **multer** for file uploads.
* **xlsx** for parsing Excel files.
* **express** for routing.

npm install express multer xlsx body-parser

**Step 2: Project Structure**

Here is a suggested folder structure:

my-node-app/

│

├── controllers/

│ ├── fileController.js

│

├── routes/

│ ├── fileRoutes.js

│

├── services/

│ ├── fileService.js

│

├── app.js

├── package.json

└── .env

**Step 3: Implementing the Routes, Controllers, and Services**

**1. Routes (routes/fileRoutes.js)**

This will define the endpoints that handle file uploads.

const express = require('express');

const router = express.Router();

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

const multer = require('multer');

// Setup Multer for handling file uploads

const storage = multer.diskStorage({

destination: (req, file, cb) => {

cb(null, 'uploads/'); // Set the destination folder for file uploads

},

filename: (req, file, cb) => {

cb(null, `${Date.now()}-${file.originalname}`); // Set the filename to be unique

}

});

const upload = multer({ storage: storage });

// Routes for uploading files and handling JSON data

router.post('/upload-wav-json', upload.single('wavFile'), fileController.uploadWavJson); // Upload WAV file + JSON

router.post('/upload-wav-excel', upload.single('wavFile'), upload.single('excelFile'), fileController.uploadWavExcel); // Upload WAV file + Excel file

module.exports = router;

**2. Controller (controllers/fileController.js)**

The controller handles the request logic and calls the service methods.

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

exports.uploadWavJson = async (req, res) => {

try {

const { wavFile } = req.file;

const { jsonInput } = req.body;

// Call the service to handle the logic

const result = await fileService.handleWavJsonUpload(wavFile, JSON.parse(jsonInput));

res.status(200).json({

status: 'success',

message: 'WAV file and JSON processed successfully',

data: result

});

} catch (error) {

console.error('Error uploading WAV and JSON:', error);

res.status(500).json({ status: 'error', message: 'Failed to upload files and JSON.' });

}

};

exports.uploadWavExcel = async (req, res) => {

try {

const { wavFile, excelFile } = req.files;

// Call the service to handle WAV and Excel file processing

const result = await fileService.handleWavExcelUpload(wavFile, excelFile);

res.status(200).json({

status: 'success',

message: 'WAV file and Excel processed successfully',

data: result

});

} catch (error) {

console.error('Error uploading WAV and Excel:', error);

res.status(500).json({ status: 'error', message: 'Failed to upload WAV and Excel.' });

}

};

**3. Service (services/fileService.js)**

This service handles the actual business logic after receiving the files and data.

const fs = require('fs');

const xlsx = require('xlsx');

exports.handleWavJsonUpload = async (wavFile, jsonInput) => {

try {

// Process WAV file (example: save it, analyze it, etc.)

console.log(`Processing WAV file: ${wavFile.path}`);

// Process JSON input (example: save to database, validate, etc.)

console.log('Processing JSON Input:', jsonInput);

// For example, you could write the WAV file path and JSON input to the database

const result = {

wavFilePath: wavFile.path,

jsonData: jsonInput

};

// Return the result for further handling

return result;

} catch (error) {

throw new Error('Error processing WAV and JSON upload: ' + error.message);

}

};

exports.handleWavExcelUpload = async (wavFile, excelFile) => {

try {

// Process WAV file (example: save it, analyze it, etc.)

console.log(`Processing WAV file: ${wavFile.path}`);

// Process Excel file (example: parse it and extract data)

console.log(`Processing Excel file: ${excelFile.path}`);

const workbook = xlsx.readFile(excelFile.path);

const sheetNames = workbook.SheetNames;

const sheet = workbook.Sheets[sheetNames[0]];

// Convert the first sheet to JSON

const excelData = xlsx.utils.sheet\_to\_json(sheet);

// Return the result with Excel data and WAV file path

const result = {

wavFilePath: wavFile.path,

excelData: excelData

};

// Return the result for further handling

return result;

} catch (error) {

throw new Error('Error processing WAV and Excel upload: ' + error.message);

}

};

**Step 4: Setting Up the Main Application (app.js)**

Finally, integrate everything into your main application.

const express = require('express');

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

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

const app = express();

// Middleware for parsing JSON

app.use(bodyParser.json());

// Route handling for file uploads

app.use('/api/files', fileRoutes);

// Set the folder for file uploads

const path = require('path');

app.use('/uploads', express.static(path.join(\_\_dirname, 'uploads')));

// Start the server

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

app.listen(PORT, () => {

console.log(`Server running on port ${PORT}`);

});

**Step 5: Frontend Example for File Upload**

Here’s an example of how you might send the data from the **frontend** (HTML form with files and JSON):

**HTML Form for Sending WAV file, JSON, and Excel file**

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>File Upload</title>

</head>

<body>

<h2>Upload WAV file and JSON</h2>

<form id="uploadForm" enctype="multipart/form-data">

<input type="file" id="wavFile" name="wavFile" accept=".wav" required><br><br>

<textarea id="jsonInput" name="jsonInput" rows="4" cols="50" placeholder="Enter JSON data here" required></textarea><br><br>

<button type="submit">Upload WAV + JSON</button>

</form>

<h2>Upload WAV file and Excel file</h2>

<form id="excelForm" enctype="multipart/form-data">

<input type="file" id="wavFileExcel" name="wavFile" accept=".wav" required><br><br>

<input type="file" id="excelFile" name="excelFile" accept=".xlsx, .xls" required><br><br>

<button type="submit">Upload WAV + Excel</button>

</form>

<script>

document.getElementById('uploadForm').addEventListener('submit', function(event) {

event.preventDefault();

const wavFile = document.getElementById('wavFile').files[0];

const jsonInput = document.getElementById('jsonInput').value;

const formData = new FormData();

formData.append('wavFile', wavFile);

formData.append('jsonInput', jsonInput);

fetch('/api/files/upload-wav-json', {

method: 'POST',

body: formData

}).then(response => response.json())

.then(data => alert('Files uploaded successfully!'))

.catch(error => alert('Error uploading files: ' + error));

});

document.getElementById('excelForm').addEventListener('submit', function(event) {

event.preventDefault();

const wavFile = document.getElementById('wavFileExcel').files[0];

const excelFile = document.getElementById('excelFile').files[0];

const formData = new FormData();

formData.append('wavFile', wavFile);

formData.append('excelFile', excelFile);

fetch('/api/files/upload-wav-excel', {

method: 'POST',

body: formData

}).then(response => response.json())

.then(data => alert('Files uploaded successfully!'))

.catch(error => alert('Error uploading files: ' + error));

});

</script>

</body>

</html>

**Explanation of the Code:**

1. **Frontend**:
   * The frontend consists of two forms: one for uploading the WAV file and JSON