**Monitoring Tool Project Documentation**

This document provides a comprehensive overview of the Monitoring Tool project, including its architecture, code structure, setup instructions, and explanations of key components.

**Table of Contents**

1. [Project Overview](#1-project-overview)
2. [Architecture](#2-architecture)
3. [Backend Documentation](#3-backend-documentation)
   * [Backend Setup](#31-backend-setup)
   * [Backend Directory Structure](#32-backend-directory-structure)
   * [Backend Key Components & Code Explanations](#33-backend-key-components--code-explana)
     + [server.js](#serverjs)
     + [app.js](#appjs)
     + [db/connection.js](#dbconnectionjs)
     + [jobs/monitoringScheduler.js](#jobsmonitoringschedulerjs)
     + [services/apiMonitorService.js](#servicesapimonitorservicejs)
     + [services/certMonitorService.js](#servicescertmonitorservicejs)
     + [services/notificationService.js](#servicesnotificationservicejs)
     + [controllers/monitoringController.js](#controllersmonitoringcontrollerjs)
     + [middleware/authMiddleware.js](#middlewareauthmiddlewarejs)
     + [utils/validation.js](#utilsvalidationjs)
   * [Backend API Endpoints](#34-backend-api-endpoints)
4. [Frontend Documentation](#4-frontend-documentation)
   * [Frontend Setup](#41-frontend-setup)
   * [Frontend Directory Structure](#42-frontend-directory-structure)
   * [Frontend Key Components & Code Explanations](#43-frontend-key-components--code-explan)
     + [App.js](#appjs-1)
     + [context/AuthContext.js](#contextauthcontextjs)
     + [pages/DashboardPage.js](#pagesdashboardpagejs)
     + [components/dashboard/UrlCard.js](#componentsdashboardurlcardjs)
     + [api/monitoring.js](#apimonitoringjs)
   * [Frontend Styling](#44-frontend-styling)
5. [Database Schema](#5-database-schema)
6. [How to Run the Application](#6-how-to-run-the-application)
7. [Troubleshooting](#7-troubleshooting)
8. [Future Enhancements](#8-future-enhancements)

**1. Project Overview**

The Monitoring Tool is a web application designed to monitor the uptime and health of various URLs, including API endpoints and domains (for SSL/TLS certificate validity). It provides a dashboard for users to view the status of their monitored resources, manage URLs, configure proxy settings, and set up alerts for critical events. The application is built with a Node.js backend (Express.js) and a React.js frontend, utilizing MySQL as its database.

**2. Architecture**

The project follows a client-server architecture:

* **Frontend (React.js):** A single-page application (SPA) that provides the user interface. It consumes RESTful APIs exposed by the backend to fetch and display monitoring data, manage configurations, and handle user authentication.
* **Backend (Node.js with Express.js):** A RESTful API server responsible for:
  + User authentication and authorization (JWT).
  + Managing URL configurations (add, edit, delete).
  + Performing actual URL monitoring (API uptime, domain certificate checks).
  + Storing monitoring logs and status in a MySQL database.
  + Scheduling periodic monitoring jobs using node-cron.
  + Sending notifications (email, SNMP traps) based on alert configurations.
* **Database (MySQL):** Stores user data, URL configurations, proxy settings, alert configurations, and historical monitoring logs.

**Data Flow:**

1. Users interact with the **Frontend**.
2. Frontend makes **API calls** (HTTP requests) to the **Backend**.
3. Backend authenticates requests, processes business logic, and interacts with the **MySQL Database**.
4. Backend's **monitoring scheduler** periodically runs jobs to check URLs.
5. Monitoring jobs update **database records** and trigger **notifications** (email/SNMP) if anomalies are detected.
6. Frontend retrieves updated status from the Backend to display on the **Dashboard**.

**3. Backend Documentation**

The backend is built with Node.js and Express.js. It handles all the core logic, data storage, and monitoring processes.

**3.1. Backend Setup**

**Prerequisites:**

* Node.js (v18.x or later recommended) and npm (comes with Node.js).
* MySQL Server (v8.0 or later recommended).

**Installation Steps:**

1. **Navigate to the backend directory:**
2. cd backend
3. **Install Node.js dependencies:**
4. npm install
5. **Create and configure the .env file:**
   * Create a file named .env in the backend/ directory.
   * Copy the content from backend/.env.example into backend/.env.
   * **Crucially, update the placeholder values** with your actual MySQL credentials, JWT secret, and email/SNMP settings.
   * Example backend/.env:
   * # backend/.env
   * NODE\_ENV=development
   * PORT=3000
   * APP\_TIMEZONE=Asia/Kolkata
   * DB\_HOST=localhost
   * DB\_USER=root
   * DB\_PASSWORD=your\_mysql\_root\_password
   * DB\_DATABASE=monitoring\_tool
   * JWT\_SECRET=a\_very\_long\_and\_complex\_random\_string\_for\_jwt\_security
   * JWT\_EXPIRES\_IN=1d
   * EMAIL\_HOST=your\_smtp\_host.com
   * EMAIL\_PORT=587
   * EMAIL\_SECURE=false
   * EMAIL\_USER=your\_smtp\_username
   * EMAIL\_PASS=your\_smtp\_password
   * EMAIL\_FROM="MonitorPro Alerts <no-reply@monitorpro.com>"
   * ALERTING\_EMAIL\_RECIPIENT=admin@example.com
   * ALERTING\_CERT\_WARNING\_DAYS=30
   * SNMP\_RECEIVER\_HOST=127.0.0.1
   * SNMP\_COMMUNITY=public
   * SNMP\_API\_DOWN\_OID=.1.3.6.1.4.1.9999.1.1
   * SNMP\_CERT\_EXPIRY\_OID=.1.3.6.1.4.1.9999.1.2
6. **Database Initialization:**
   * Open your MySQL client (e.g., MySQL Workbench, command line mysql).
   * If you have an existing monitoring\_tool database from previous attempts, drop it to ensure a clean slate:
   * DROP DATABASE IF EXISTS monitoring\_tool;
   * Navigate back to your backend/ directory in the terminal.
   * Run the database initialization script:
   * node scripts/init\_db.js

This script will create the monitoring\_tool database and all necessary tables with their schemas.

**3.2. Backend Directory Structure**

backend/

├── src/

│ ├── config/ # Application-wide configurations (DB, JWT, Email, SNMP)

│ │ └── index.js

│ ├── controllers/ # Handles incoming requests, calls services, prepares responses

│ │ ├── authController.js

│ │ ├── monitoringController.js

│ │ └── alertController.js

│ ├── services/ # Business logic, external API interactions, database operations

│ │ ├── authService.js

│ │ ├── apiMonitorService.js

│ │ ├── certMonitorService.js

│ │ ├── notificationService.js

│ │ └── proxyService.js # Placeholder for advanced proxy logic

│ ├── middleware/ # Express middleware (e.g., authentication, error handling)

│ │ ├── authMiddleware.js

│ │ └── errorHandler.js

│ ├── routes/ # Defines API endpoints and links them to controllers

│ │ ├── authRoutes.js

│ │ ├── monitorRoutes.js

│ │ └── alertRoutes.js

│ ├── jobs/ # Contains scheduled tasks (cron jobs) for monitoring

│ │ └── monitoringScheduler.js

│ ├── db/

│ │ ├── connection.js # MySQL connection pool management

│ │ └── models/ # Database schema definitions (SQL DDL)

│ │ └── sql\_schema.sql

│ ├── utils/ # Utility functions (logging, validation, custom errors)

│ │ ├── appError.js

│ │ ├── logger.js

│ │ └── validation.js

│ ├── app.js # Express application setup, middleware, and route registration

│ └── server.js # Main entry point: initializes DB, starts scheduler, listens for requests

├── .env # Environment variables (ignored by git)

├── package.json # Project dependencies and scripts

└── scripts/

└── init\_db.js # Script to initialize the MySQL database schema

**3.3. Backend Key Components & Code Explanations**

This section explains the role of critical backend files and provides insights into their code.

**server.js**

This is the main entry point of the backend application. It's responsible for:

* Connecting to the MySQL database.
* Starting the monitoringScheduler to begin periodic health checks.
* Starting the Express.js server to listen for incoming HTTP requests.
* Setting up global error handling for unhandled promise rejections and uncaught exceptions.

// backend/src/server.js

const app = require('./app');

const monitoringScheduler = require('./jobs/monitoringScheduler');

const { connectDB } = require('./db/connection');

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

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

const PORT = config.app.port;

const startServer = async () => {

try {

await connectDB(); // Establishes database connection

logger.info('MySQL Pool created and connected successfully.');

logger.info('Database connected successfully.');

monitoringScheduler.start(); // Initiates the monitoring cron jobs

logger.info('Periodic monitoring started.');

app.listen(PORT, () => {

logger.info(`Server running on port ${PORT} in ${config.app.env} mode.`);

});

} catch (error) {

logger.error(`Failed to start server: ${error.message}`);

console.error(`Failed to start server:`, error);

process.exit(1); // Exit if server fails to start

}

// Global error handlers to catch unhandled async errors

process.on('unhandledRejection', (reason, promise) => {

logger.error(`Unhandled Rejection: ${reason.message || reason}`);

console.error('Unhandled Rejection at:', promise, 'reason:', reason);

});

process.on('uncaughtException', (err) => {

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

console.error('Uncaught Exception:', err);

process.exit(1);

});

};

startServer();

**app.js**

This file configures the Express.js application, including middleware and route registration.

// backend/src/app.js

const express = require('express');

const cors = require('cors');

const helmet = require('helmet');

const morgan = require('morgan');

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

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

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

const { errorHandler } = require('./middleware/errorHandler'); // Centralized error handler

const AppError = require('./utils/appError'); // Custom error class

const app = express();

// Middleware setup

app.use(express.json()); // Parses incoming JSON requests

app.use(express.urlencoded({ extended: true })); // Parses URL-encoded requests

app.use(cors()); // Enables Cross-Origin Resource Sharing

app.use(helmet()); // Sets various HTTP headers for security

app.use(morgan('dev')); // Logs HTTP requests to console in 'dev' format

// Route registration

app.use('/api/auth', authRoutes); // Authentication routes

app.use('/api/monitor', monitorRoutes); // Monitoring-related routes

app.use('/api/alerts', alertRoutes); // Alert configuration routes

// Health check endpoint

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

res.status(200).send('API is healthy');

});

// Catch-all for 404 Not Found errors (must be before error handler)

app.all('\*', (req, res, next) => {

next(new AppError(`Can't find ${req.originalUrl} on this server!`, 404));

});

// Centralized error handling middleware (must be the last middleware)

app.use(errorHandler);

module.exports = app;

**db/connection.js**

Manages the MySQL database connection pool.

// backend/src/db/connection.js

const mysql = require('mysql2/promise');

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

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

let pool; // Stores the MySQL connection pool

// Establishes and tests the database connection pool

const connectDB = async () => {

try {

pool = mysql.createPool({

host: config.db.host,

user: config.db.user,

password: config.db.password,

database: config.db.database,

waitForConnections: true,

connectionLimit: 10, // Max concurrent connections

queueLimit: 0 // Unlimited queue for waiting connections

});

const connection = await pool.getConnection(); // Get a connection to test

connection.release(); // Release it immediately

logger.info('MySQL Pool created and connected successfully.');

} catch (error) {

logger.error(`Error connecting to MySQL: ${error.message}`);

console.error(`Error connecting to MySQL:`, error);

throw error; // Propagate error to server.js

}

};

// Returns the initialized database connection pool

const getDB = () => {

if (!pool) {

logger.error('Database pool not initialized. Call connectDB first.');

throw new Error('Database pool not initialized. Call connectDB first.');

}

return pool;

};

module.exports = { connectDB, getDB };

**jobs/monitoringScheduler.js**

This file orchestrates the periodic monitoring of URLs using node-cron.

// backend/src/jobs/monitoringScheduler.js

const cron = require('node-cron');

const { getDB } = require('../db/connection');

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

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

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

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

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

const scheduledTasks = new Map(); // Stores cron tasks by URL ID for management

// Starts the main scheduled monitoring tasks

const start = async () => {

const db = getDB();

// Clear any existing tasks on startup to prevent duplicates after a restart

scheduledTasks.forEach(task => task.stop());

scheduledTasks.clear();

logger.info('Cleared existing scheduled tasks upon scheduler start.');

try {

// Load all active URLs from the database at startup

const [urls] = await db.execute(

`SELECT

u.id, u.user\_id, u.name, u.url, u.type, u.monitoring\_interval\_minutes, u.is\_active,

pc.host AS proxy\_host, pc.port AS proxy\_port, pc.protocol AS proxy\_protocol,

pc.username AS proxy\_username, pc.password AS proxy\_password, pc.enabled AS proxy\_enabled

FROM Urls u

LEFT JOIN ProxyConfigs pc ON u.proxy\_config\_id = pc.id

WHERE u.is\_active = TRUE`

);

// Schedule each active URL for monitoring

urls.forEach(urlEntry => {

scheduleMonitor(urlEntry);

});

logger.info(`Scheduled ${urls.length} initial active URLs.`);

} catch (error) {

logger.error(`Error loading initial URLs for scheduling: ${error.message}`);

console.error(`Error loading initial URLs for scheduling:`, error);

}

logger.info('Monitoring scheduler started. Individual URL monitoring will be managed by their respective cron jobs.');

};

// Schedules a monitoring task for a single URL

const scheduleMonitor = (urlEntry) => {

const { id, user\_id, url, name, type, monitoring\_interval\_minutes, is\_active } = urlEntry;

const intervalCron = `\*/${monitoring\_interval\_minutes} \* \* \* \*`; // Cron expression based on interval

stopMonitor(id); // Stop any existing task for this URL before re-scheduling

if (!is\_active) {

logger.info(`Monitoring for URL ID ${id} (${name}) is inactive. Not scheduling.`);

return;

}

const taskKey = `url-${id}`; // Unique key for each URL's task

// The core monitoring job function

const monitorJob = async () => {

logger.info(`Running scheduled ${type} check for: ${name} (URL ID: ${id})`);

try {

if (type === 'API') {

// Prepare proxy config if available

const proxyConfig = urlEntry.proxy\_host ? {

host: urlEntry.proxy\_host,

port: urlEntry.proxy\_port,

protocol: urlEntry.proxy\_protocol,

username: urlEntry.proxy\_username,

password: urlEntry.proxy\_password,

enabled: urlEntry.proxy\_enabled

} : null;

const result = await apiMonitorService.checkApiUptime(id, url, proxyConfig);

if (result.status === 'down') {

await notificationService.sendApiDownAlert(user\_id, url, result.error);

}

} else if (type === 'DOMAIN') {

// Pass full urlEntry to certMonitorService

const result = await certMonitorService.checkCertStatus(urlEntry);

if (result.status === 'expired' || (result.status === 'warning' && result.daysRemaining <= config.alerting.certWarningDays)) {

await notificationService.sendCertificateExpiryAlert(user\_id, url, result.expiryDate, result.daysRemaining);

}

}

} catch (error) {

logger.error(`Error during scheduled monitoring for URL ID ${id} (${name}): ${error.message}`);

console.error(`Error during scheduled monitoring for URL ID ${id} (${name}):`, error);

// Update URL status to 'down' if the monitoring job itself fails

const db = getDB();

try {

await db.execute(

`UPDATE Urls SET last\_status = 'down', last\_checked\_at = NOW(), last\_error = ? WHERE id = ?`,

[`Internal monitoring error: ${error.message}`, id]

);

} catch (dbUpdateError) {

logger.error(`Failed to update URL status after internal monitoring error for ID ${id}: ${dbUpdateError.message}`);

}

}

};

// Schedule the cron task

const task = cron.schedule(intervalCron, monitorJob, {

scheduled: true,

timezone: config.app.timezone

});

scheduledTasks.set(taskKey, task);

logger.info(`Scheduled ${type} monitoring for URL ID ${id} (${name}) every ${monitoring\_interval\_minutes} minutes.`);

// Trigger an immediate check when scheduling a new or updated active URL

monitorJob();

logger.info(`Triggered immediate check for URL: ${name} (ID: ${id}, Type: ${type}).`);

};

// Stops a monitoring task for a single URL

const stopMonitor = (urlId) => {

const taskKey = `url-${urlId}`;

if (scheduledTasks.has(taskKey)) {

const task = scheduledTasks.get(taskKey);

task.stop();

scheduledTasks.delete(taskKey);

logger.info(`Stopped monitoring for URL ID ${urlId}.`);

} else {

logger.debug(`No active task found for URL ID ${urlId} to stop.`);

}

};

// Restarts a monitoring task for a URL

const restartMonitor = (urlEntry) => {

logger.info(`Restarting monitoring for URL ID ${urlEntry.id} (${urlEntry.name}).`);

stopMonitor(urlEntry.id);

if (urlEntry.is\_active) {

scheduleMonitor(urlEntry);

} else {

logger.info(`Monitoring for URL ID ${urlEntry.id} (${urlEntry.name}) is inactive, not restarting.`);

}

};

// Stops all scheduled monitoring tasks

const stopAll = () => {

scheduledTasks.forEach(task => task.stop());

scheduledTasks.clear();

logger.info('All monitoring tasks stopped.');

};

module.exports = { start, stopMonitor, restartMonitor, stopAll };

**services/apiMonitorService.js**

This service handles the actual API uptime checks.

// backend/src/services/apiMonitorService.js

const axios = require('axios');

const { getDB } = require('../db/connection');

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

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

// Checks the uptime and latency for an API endpoint.

const checkApiUptime = async (urlId, url, proxyConfig = null) => {

const db = getDB();

let status = 'down';

let latency = null;

let statusCode = null;

let error = null;

const start = Date.now(); // Start time for latency calculation

try {

const axiosConfig = {

timeout: config.app.apiTimeout, // Use configured timeout (e.g., 10000ms)

validateStatus: (status) => true, // Accept all status codes to process response

};

// Configure proxy if enabled

if (proxyConfig && proxyConfig.enabled) {

axiosConfig.proxy = {

host: proxyConfig.host,

port: proxyConfig.port,

protocol: proxyConfig.protocol || 'http',

...(proxyConfig.username && { auth: { username: proxyConfig.username, password: proxyConfig.password } })

};

logger.debug(`Using proxy for API check: ${proxyConfig.host}:${proxyConfig.port}`);

}

const response = await axios.get(url, axiosConfig);

latency = Date.now() - start;

statusCode = response.status;

// If a response was received (even a 4xx/5xx), the API is considered 'up' in terms of reachability

status = 'up';

logger.info(`API ${url} is UP. Status: ${statusCode}, Latency: ${latency}ms`);

} catch (err) {

latency = Date.now() - start; // Calculate latency even if connection failed

status = 'down'; // API is unreachable or network error

statusCode = err.response ? err.response.status : (err.code === 'ECONNABORTED' ? 408 : null); // 408 for timeout, null for others

error = err.message || 'Unknown error during API check.';

logger.error(`API ${url} is DOWN. Error: ${error}, Status Code: ${statusCode}, Latency: ${latency}ms`);

console.error(`API ${url} check failed:`, err); // Log full error object for debugging

} finally {

// Update the Urls table with the latest status

try {

await db.execute(

`UPDATE Urls SET

last\_status = ?,

last\_latency = ?,

last\_checked\_at = NOW(),

last\_error = ?

WHERE id = ?`,

[status, latency, error, urlId]

);

} catch (dbUpdateError) {

logger.error(`DB error updating Urls table for API ID ${urlId}: ${dbUpdateError.message}`);

}

// Insert into MonitoringLogs table for historical data

try {

await db.execute(

`INSERT INTO MonitoringLogs (url\_id, status, latency, status\_code, error, created\_at) VALUES (?, ?, ?, ?, ?, NOW())`,

[urlId, status, latency, statusCode, error]

);

} catch (dbLogError) {

logger.error(`DB error inserting log for API ID ${urlId}: ${dbLogError.message}`);

}

return { status, latency, statusCode, error };

}

};

module.exports = { checkApiUptime };

**services/certMonitorService.js**

This service performs domain and certificate checks. It's designed to mark the URL as 'up' if a TLS connection is established, and then separately evaluate the certificate's validity.

// backend/src/services/certMonitorService.js

const tls = require('tls');

const { getDB } = require('../db/connection');

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

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

const notificationService = require('./notificationService'); // For sending alerts

// Checks the SSL/TLS certificate status for a given domain URL.

async function checkCertStatus(urlObject) {

console.log('\*\*\* Executing checkCertStatus: Using native Node.js TLS certificate parsing. \*\*\*');

const { id: urlId, url, name, user\_id } = urlObject;

let certificateStatus = null; // 'valid', 'warning', 'expired', 'unavailable', 'error', 'not\_applicable', 'not\_reachable'

let daysRemaining = null;

let expiryDate = null;

let issuer = null;

let subject = null;

let certErrorDetails = null;

let overallStatus = 'down'; // Default for Urls.last\_status, assumes unreachable initially

if (!url.startsWith('https://')) {

certificateStatus = 'not\_applicable';

overallStatus = 'up'; // Non-HTTPS is considered 'up' for overall reachability

logger.info(`URL ID ${urlId} (${name}) is not HTTPS, skipping certificate check. Setting overall status to 'up'.`);

const db = getDB();

try {

await db.execute(

`UPDATE Urls SET

certificate\_status = ?,

days\_remaining = ?,

last\_status = ?,

last\_checked\_at = NOW(),

last\_error = NULL

WHERE id = ?`,

[certificateStatus, daysRemaining, overallStatus, urlId]

);

} catch (dbError) {

logger.error(`DB error updating cert info for non-HTTPS URL ID ${urlId}: ${dbError.message}`);

}

return { status: certificateStatus, expiryDate, daysRemaining, issuer, subject, error: certErrorDetails, overallStatus };

}

const hostname = new URL(url).hostname;

const checkResult = await new Promise((resolve) => {

const socket = tls.connect({

host: hostname,

port: 443,

rejectUnauthorized: false, // Allow inspection of self-signed/invalid certs

timeout: config.app.apiTimeout // Use configured timeout

}, () => {

// If connection established, site is reachable, so overall status is 'up'

overallStatus = 'up';

logger.info(`TLS handshake successful for ${url}. Attempting to get certificate.`);

try {

const cert = socket.getPeerCertificate(); // Get the certificate object

if (cert && typeof cert === 'object' && cert.valid\_to && cert.valid\_from) {

expiryDate = new Date(cert.valid\_to);

const now = new Date();

const diffTime = expiryDate.getTime() - now.getTime();

daysRemaining = Math.ceil(diffTime / (1000 \* 60 \* 60 \* 24));

issuer = cert.issuer.CN || Object.entries(cert.issuer).map(([key, value]) => `${key}=${value}`).join(', ') || 'N/A';

subject = cert.subject.CN || Object.entries(cert.subject).map(([key, value]) => `${key}=${value}`).join(', ') || 'N/A';

if (expiryDate < now) {

certificateStatus = 'expired';

logger.warn(`Cert for ${name} (${url}) is EXPIRED. Days remaining: ${daysRemaining}.`);

} else if (daysRemaining <= config.alerting.certWarningDays) {

certificateStatus = 'warning';

logger.warn(`Cert for ${name} (${url}) is WARNING. Days remaining: ${daysRemaining}.`);

} else {

certificateStatus = 'valid';

logger.info(`Cert for ${name} (${url}) is VALID. Days remaining: ${daysRemaining}.`);

}

} else {

certificateStatus = 'unavailable';

certErrorDetails = 'No complete certificate object found or could not be parsed from peer.';

logger.warn(`Cert for ${name} (${url}) is UNAVAILABLE or incomplete.`);

}

} catch (err) {

certificateStatus = 'error';

certErrorDetails = `Error processing certificate: ${err.message}`;

logger.error(`Error processing certificate for ${url}: ${err.message}`);

console.error(`Error processing certificate for ${url}:`, err);

} finally {

socket.destroy();

resolve({ status: certificateStatus, expiryDate, daysRemaining, issuer, subject, error: certErrorDetails, overallStatus });

}

});

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

certificateStatus = 'not\_reachable';

overallStatus = 'down'; // Network-level failure

certErrorDetails = `TLS connection error: ${err.message}`;

daysRemaining = null;

logger.error(`TLS connection error for ${url}: ${err.message}`);

console.error(`TLS connection error for ${url}:`, err);

socket.destroy();

resolve({ status: certificateStatus, expiryDate, daysRemaining, issuer, subject, error: certErrorDetails, overallStatus });

});

socket.on('timeout', () => {

certificateStatus = 'not\_reachable';

overallStatus = 'down'; // Connection timed out

certErrorDetails = 'TLS connection timed out.';

daysRemaining = null;

logger.error(`TLS connection to ${url} timed out.`);

console.error(`TLS connection to ${url} timed out.`);

socket.destroy();

resolve({ status: certificateStatus, expiryDate, daysRemaining, issuer, subject, error: certErrorDetails, overallStatus });

});

});

const db = getDB();

try {

let updateQuery = `UPDATE Urls SET

certificate\_status = ?,

days\_remaining = ?,

last\_status = ?,

last\_checked\_at = NOW()`;

let updateValues = [checkResult.status, checkResult.daysRemaining, checkResult.overallStatus];

if (checkResult.error) {

updateQuery += `, last\_error = ?`;

updateValues.push(checkResult.error);

} else {

updateQuery += `, last\_error = NULL`; // Clear error if successful

}

updateQuery += ` WHERE id = ?`;

updateValues.push(urlId);

await db.execute(updateQuery, updateValues);

logger.info(`URL ID ${urlId} (${name}) certificate status updated to: ${checkResult.status || 'N/A'}, Days Remaining: ${checkResult.daysRemaining || 'N/A'}. Overall status: ${checkResult.overallStatus}.`);

} catch (dbError) {

logger.error(`DB error updating certificate info in Urls table for URL ID ${urlId}: ${dbError.message}`);

}

return checkResult;

}

module.exports = {

checkCertStatus

};

**services/notificationService.js**

Handles sending email and SNMP trap notifications.

// backend/src/services/notificationService.js

const nodemailer = require('nodemailer');

const snmp = require('net-snmp');

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

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

const { getDB } = require('../db/connection'); // To fetch user-specific alert configs

// Nodemailer Transporter Setup

const transporter = nodemailer.createTransport({

host: config.email.host,

port: config.email.port,

secure: config.email.secure,

auth: {

user: config.email.user,

pass: config.email.pass,

},

});

// Sends an email notification.

const sendEmail = async (to, subject, text, html) => {

if (!to || !config.email.user || !config.email.pass || !config.email.host) {

logger.warn('Email sending skipped: recipient, sender email, password, or host not configured.');

return;

}

try {

const info = await transporter.sendMail({

from: config.email.from,

to,

subject,

text,

html,

});

logger.info(`Email sent to ${to} with subject: "${subject}". Message ID: ${info.messageId}`);

} catch (error) {

logger.error(`Failed to send email to ${to} for subject "${subject}": ${error.message}`);

if (config.app.env === 'development') {

logger.error(`Nodemailer error details: ${JSON.stringify(error)}`);

}

}

};

// Sends an SNMP trap.

const sendSnmpTrap = (receiverHost, community, oid, value, type = snmp.ObjectType.OctetString) => {

if (!oid || typeof oid !== 'string' || !/^(\.\d+)+$/.test(oid)) {

logger.warn(`SNMP trap sending skipped: Invalid OID string provided: '${oid}'. Must be dot-separated numbers (e.g., .1.3.6.1.4.1.9999.1.1).`);

return;

}

if (!receiverHost || !community) {

logger.warn('SNMP trap sending skipped: receiver host or community not configured.');

return;

}

const varbinds = [{

oid: oid,

type: type,

value: value

}];

const snmpSession = snmp.createSession(

receiverHost,

community,

{ timeout: 5000 }

);

snmpSession.trap(snmp.TrapType.LinkUp, varbinds, (error) => {

if (error) {

logger.error(`Failed to send SNMP trap to ${receiverHost} with OID ${oid}: ${error.message}`);

if (config.app.env === 'development') {

logger.error(`SNMP error details: ${JSON.stringify(error)}`);

}

} else {

logger.info(`SNMP trap sent to ${receiverHost} for OID: ${oid}`);

}

snmpSession.close();

});

};

// Fetches user-specific alert configurations from the database or uses defaults.

const getUserAlertConfig = async (userId) => {

const db = getDB();

try {

const [userConfigs] = await db.execute(

`SELECT email\_recipient, snmp\_receiver\_host, snmp\_community, snmp\_api\_down\_oid, snmp\_cert\_expiry\_oid, cert\_warning\_days

FROM AlertConfigs WHERE user\_id = ?`,

[userId]

);

if (userConfigs.length > 0) {

return userConfigs[0];

}

return {

email\_recipient: config.alerting.emailRecipient,

snmp\_receiver\_host: config.snmp.receiverHost,

snmp\_community: config.snmp.community,

snmp\_api\_down\_oid: config.alerting.snmpApiDownOid,

snmp\_cert\_expiry\_oid: config.alerting.snmpCertExpiryOid,

cert\_warning\_days: config.alerting.certWarningDays

};

} catch (error) {

logger.error(`Failed to fetch alert config for user ${userId}: ${error.message}`);

return {

email\_recipient: config.alerting.emailRecipient,

snmp\_receiver\_host: config.snmp.receiverHost,

snmp\_community: config.snmp.community,

snmp\_api\_down\_oid: config.alerting.snmpApiDownOid,

snmp\_cert\_expiry\_oid: config.alerting.snmpCertExpiryOid,

cert\_warning\_days: config.alerting.certWarningDays

};

}

};

// Sends an alert when an API is detected as down.

const sendApiDownAlert = async (userId, url, errorMessage) => {

const userAlertConfig = await getUserAlertConfig(userId);

const subject = `CRITICAL: API ${url} is DOWN!`;

const text = `The API ${url} is currently unreachable or returning an error.\nError: ${errorMessage}\n\nPlease investigate immediately.`;

const html = `

<p>The API <strong>${url}</strong> is currently unreachable or returning an error.</p>

<p><strong>Error:</strong> ${errorMessage}</p>

<p>Please investigate immediately.</p>

<p>This alert was generated by your Monitoring Tool.</p>

`;

if (userAlertConfig.email\_recipient) {

await sendEmail(userAlertConfig.email\_recipient, subject, text, html);

}

if (userAlertConfig.snmp\_receiver\_host && userAlertConfig.snmp\_api\_down\_oid) {

sendSnmpTrap(

userAlertConfig.snmp\_receiver\_host,

userAlertConfig.snmp\_community,

userAlertConfig.snmp\_api\_down\_oid,

`API Down: ${url} - ${errorMessage}`

);

}

};

// Sends an alert when a domain certificate is expiring soon or has expired.

const sendCertificateExpiryAlert = async (userId, url, expiryDate, daysRemaining) => {

const userAlertConfig = await getUserAlertConfig(userId);

let subject, text, html;

if (daysRemaining <= 0) {

subject = `CRITICAL: Certificate for ${url} has EXPIRED!`;

text = `The SSL certificate for ${url} expired on ${expiryDate.toDateString()}.\n\nPlease renew the certificate immediately to avoid service disruption.`;

html = `

<p>The SSL certificate for <strong>${url}</strong> has <strong>EXPIRED</strong> on <strong>${expiryDate.toDateString()}</strong>.</p>

<p>Please renew the certificate immediately to avoid service disruption.</p>

<p>This alert was generated by your Monitoring Tool.</p>

`;

} else {

subject = `WARNING: Certificate for ${url} expires in ${daysRemaining} days!`;

text = `The SSL certificate for ${url} expires on ${expiryDate.toDateString()} (${daysRemaining} days remaining).\n\nPlease renew the certificate.`;

html = `

<p>The SSL certificate for <strong>${url}</strong> expires on <strong>${expiryDate.toDateString()}</strong> (<strong>${daysRemaining} days remaining</strong>).</p>

<p>Please renew the certificate to avoid service disruption.</p>

<p>This alert was generated by your Monitoring Tool.</p>

`;

}

if (userAlertConfig.email\_recipient) {

await sendEmail(userAlertConfig.email\_recipient, subject, text, html);

}

if (userAlertConfig.snmp\_receiver\_host && userAlertConfig.snmp\_cert\_expiry\_oid) {

sendSnmpTrap(

userAlertConfig.snmp\_receiver\_host,

userAlertConfig.snmp\_community,

userAlertConfig.snmp\_cert\_expiry\_oid,

`Cert Expiry: ${url} expires in ${daysRemaining} days`

);

}

};

module.exports = {

sendEmail,

sendSnmpTrap,

sendApiDownAlert,

sendCertificateExpiryAlert,

getUserAlertConfig

};

**controllers/monitoringController.js**

Handles API requests related to URL and proxy management.

// backend/src/controllers/monitoringController.js

const { getDB } = require('../db/connection');

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

const { validateUrl, validateProxyConfig } = require('../utils/validation');

const monitoringScheduler = require('../jobs/monitoringScheduler');

const AppError = require('../utils/appError');

// Adds a new URL for monitoring.

exports.addUrl = async (req, res, next) => {

const db = getDB();

const userId = req.user.id;

try {

const { error } = validateUrl(req.body);

if (error) {

logger.warn(`Validation error adding URL for user ${userId}: ${error.details[0].message}`);

return next(new AppError(error.details[0].message, 400));

}

const { name, url, type, monitoring\_interval\_minutes, proxy\_config\_id, is\_active } = req.body;

const [result] = await db.execute(

`INSERT INTO Urls (user\_id, name, url, type, monitoring\_interval\_minutes, proxy\_config\_id, is\_active) VALUES (?, ?, ?, ?, ?, ?, ?)`,

[userId, name, url, type, monitoring\_interval\_minutes, proxy\_config\_id || null, is\_active]

);

const newUrlId = result.insertId;

// Fetch the newly added URL with complete details for scheduling

const [newUrlEntry] = await db.execute(

`SELECT u.id, u.user\_id, u.name, u.url, u.type, u.monitoring\_interval\_minutes, u.is\_active,

pc.host AS proxy\_host, pc.port AS proxy\_port, pc.protocol AS proxy\_protocol,

pc.username AS proxy\_username, pc.password AS proxy\_password, pc.enabled AS proxy\_enabled

FROM Urls u

LEFT JOIN ProxyConfigs pc ON u.proxy\_config\_id = pc.id

WHERE u.id = ?`,

[newUrlId]

);

if (newUrlEntry.length > 0 && newUrlEntry[0].is\_active) {

monitoringScheduler.scheduleMonitor(newUrlEntry[0]);

}

res.status(201).json({

status: 'success',

message: 'URL added successfully',

data: { url: newUrlEntry[0] }

});

logger.info(`URL added by user ${userId}: ${name} (${url})`);

} catch (error) {

logger.error(`Error adding URL for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

if (error.code === 'ER\_DUP\_ENTRY') {

return next(new AppError('A URL with this name or URL already exists for this user.', 409));

}

next(new AppError('Failed to add URL. Please check input or try again.', 500));

}

};

// Retrieves all URLs for the authenticated user.

exports.getUrls = async (req, res, next) => {

const db = getDB();

const userId = req.user.id;

try {

const [urls] = await db.execute(

`SELECT

u.id, u.name, u.url, u.type, u.monitoring\_interval\_minutes, u.is\_active,

u.last\_status, u.last\_latency, u.last\_checked\_at,

u.certificate\_status, u.days\_remaining,

pc.name AS proxy\_name, pc.host AS proxy\_host, pc.port AS proxy\_port, pc.protocol AS proxy\_protocol,

u.user\_id

FROM Urls u

LEFT JOIN ProxyConfigs pc ON u.proxy\_config\_id = pc.id

WHERE u.user\_id = ?

ORDER BY u.name ASC`,

[userId]

);

res.status(200).json({

status: 'success',

results: urls.length,

data: urls,

});

logger.info(`User ${userId} fetched ${urls.length} URLs.`);

} catch (error) {

logger.error(`Error fetching URLs for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to retrieve URLs.', 500));

}

};

// Updates an existing URL.

exports.updateUrl = async (req, res, next) => {

const db = getDB();

const { urlId } = req.params;

const userId = req.user.id;

try {

const { name, url, type, monitoring\_interval\_minutes, proxy\_config\_id, is\_active } = req.body;

const { error } = validateUrl(req.body, true);

if (error) {

logger.warn(`Validation error updating URL ${urlId} for user ${userId}: ${error.details[0].message}`);

return next(new AppError(error.details[0].message, 400));

}

const [result] = await db.execute(

`UPDATE Urls SET name = ?, url = ?, type = ?, monitoring\_interval\_minutes = ?, proxy\_config\_id = ?, is\_active = ?, updated\_at = NOW() WHERE id = ? AND user\_id = ?`,

[name, url, type, monitoring\_interval\_minutes, proxy\_config\_id || null, is\_active, urlId, userId]

);

if (result.affectedRows === 0) {

return next(new AppError('URL not found or not authorized to update.', 404));

}

const [updatedUrlEntry] = await db.execute(

`SELECT u.id, u.user\_id, u.name, u.url, u.type, u.monitoring\_interval\_minutes, u.is\_active,

pc.host AS proxy\_host, pc.port AS proxy\_port, pc.protocol AS proxy\_protocol,

pc.username AS proxy\_username, pc.password AS proxy\_password, pc.enabled AS proxy\_enabled

FROM Urls u

LEFT JOIN ProxyConfigs pc ON u.proxy\_config\_id = pc.id

WHERE u.id = ?`,

[urlId]

);

if (updatedUrlEntry.length > 0) {

monitoringScheduler.restartMonitor(updatedUrlEntry[0]);

}

res.status(200).json({

status: 'success',

message: 'URL updated successfully',

data: { url: updatedUrlEntry[0] }

});

logger.info(`URL ${urlId} updated by user ${userId}.`);

} catch (error) {

logger.error(`Error updating URL ID ${urlId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

if (error.code === 'ER\_DUP\_ENTRY') {

return next(new AppError('A URL with this name or URL already exists for this user.', 409));

}

next(new AppError('Failed to update URL.', 500));

}

};

// Deletes a URL.

exports.deleteUrl = async (req, res, next) => {

const db = getDB();

const { urlId } = req.params;

const userId = req.user.id;

try {

const [result] = await db.execute(

`DELETE FROM Urls WHERE id = ? AND user\_id = ?`,

[urlId, userId]

);

if (result.affectedRows === 0) {

return next(new AppError('URL not found or not authorized to delete.', 404));

}

monitoringScheduler.stopMonitor(urlId);

res.status(204).json({

status: 'success',

data: null,

message: 'URL deleted successfully'

});

logger.info(`URL ${urlId} deleted by user ${userId}.`);

} catch (error) {

logger.error(`Error deleting URL ID ${urlId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to delete URL.', 500));

}

};

// Gets monitoring logs for a specific URL.

exports.getMonitoringLogs = async (req, res, next) => {

const db = getDB();

const { urlId } = req.params;

const userId = req.user.id;

try {

const limit = parseInt(req.query.limit || '50', 10);

const [urlCheck] = await db.execute('SELECT id FROM Urls WHERE id = ? AND user\_id = ?', [urlId, userId]);

if (urlCheck.length === 0) {

return next(new AppError('URL not found or not authorized.', 404));

}

const [logs] = await db.execute(

`SELECT status, latency, status\_code, error, created\_at

FROM MonitoringLogs

WHERE url\_id = ?

ORDER BY created\_at DESC

LIMIT ?`,

[urlId, limit]

);

res.status(200).json({

status: 'success',

results: logs.length,

data: { logs }

});

logger.info(`User ${userId} fetched ${logs.length} logs for URL ${urlId}.`);

} catch (error) {

logger.error(`Error fetching logs for URL ID ${urlId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to retrieve monitoring logs.', 500));

}

};

// Gets certificate information for a specific URL.

exports.getCertificateInfo = async (req, res, next) => {

const db = getDB();

const { urlId } = req.params;

const userId = req.user.id;

try {

const [urlEntry] = await db.execute('SELECT id, type, certificate\_status, days\_remaining, last\_checked\_at FROM Urls WHERE id = ? AND user\_id = ? AND type = "DOMAIN"', [urlId, userId]);

if (urlEntry.length === 0) {

return next(new AppError('Domain URL not found or not authorized, or not a DOMAIN type URL.', 404));

}

const certInfo = {

status: urlEntry[0].certificate\_status,

days\_remaining: urlEntry[0].days\_remaining,

last\_checked\_at: urlEntry[0].last\_checked\_at,

};

res.status(200).json({

status: 'success',

data: { certificate: certInfo }

});

logger.info(`User ${userId} fetched certificate info for URL ${urlId}.`);

} catch (error) {

logger.error(`Error fetching certificate info for URL ID ${urlId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to retrieve certificate information.', 500));

}

};

// Adds a new proxy configuration.

exports.addProxyConfig = async (req, res, next) => {

const db = getDB();

const userId = req.user.id;

try {

const { error } = validateProxyConfig(req.body);

if (error) {

logger.warn(`Validation error adding proxy config for user ${userId}: ${error.details[0].message}`);

return next(new AppError(error.details[0].message, 400));

}

const { name, host, port, protocol, username, password, enabled } = req.body;

const [result] = await db.execute(

`INSERT INTO ProxyConfigs (user\_id, name, host, port, protocol, username, password, enabled) VALUES (?, ?, ?, ?, ?, ?, ?, ?)`,

[userId, name, host, port, protocol, username || null, password || null, enabled]

);

const newProxyId = result.insertId;

const [newProxyEntry] = await db.execute(`SELECT \* FROM ProxyConfigs WHERE id = ?`, [newProxyId]);

res.status(201).json({

status: 'success',

message: 'Proxy configuration added successfully',

data: { proxyConfig: newProxyEntry[0] }

});

logger.info(`Proxy config added by user ${userId}: ${name}`);

} catch (error) {

logger.error(`Error adding proxy config for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

if (error.code === 'ER\_DUP\_ENTRY') {

return next(new AppError('A proxy configuration with this name or host/port combination already exists for this user.', 409));

}

next(new AppError('Failed to add proxy configuration.', 500));

}

};

// Retrieves all proxy configurations for the authenticated user.

exports.getProxyConfigs = async (req, res, next) => {

const db = getDB();

const userId = req.user.id;

try {

const [proxyConfigs] = await db.execute(

`SELECT id, name, host, port, protocol, username, enabled

FROM ProxyConfigs WHERE user\_id = ? ORDER BY created\_at DESC`,

[userId]

);

res.status(200).json({

status: 'success',

results: proxyConfigs.length,

data: { proxyConfigs }

});

logger.info(`User ${userId} fetched ${proxyConfigs.length} proxy configs.`);

} catch (error) {

logger.error(`Error fetching proxy configs for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to retrieve proxy configurations.', 500));

}

};

// Updates an existing proxy configuration.

exports.updateProxyConfig = async (req, res, next) => {

const db = getDB();

const { proxyId } = req.params;

const userId = req.user.id;

try {

const { name, host, port, protocol, username, password, enabled } = req.body;

const { error } = validateProxyConfig(req.body, true);

if (error) {

logger.warn(`Validation error updating proxy config ${proxyId} for user ${userId}: ${error.details[0].message}`);

return next(new AppError(error.details[0].message, 400));

}

const [result] = await db.execute(

`UPDATE ProxyConfigs SET name = ?, host = ?, port = ?, protocol = ?, username = ?, password = ?, enabled = ?, updated\_at = NOW() WHERE id = ? AND user\_id = ?`,

[name, host, port, protocol, username || null, password || null, enabled, proxyId, userId]

);

if (result.affectedRows === 0) {

return next(new AppError('Proxy configuration not found or not authorized to update.', 404));

}

const [updatedProxyEntry] = await db.execute(`SELECT \* FROM ProxyConfigs WHERE id = ?`, [proxyId]);

res.status(200).json({

status: 'success',

message: 'Proxy configuration updated successfully',

data: { proxyConfig: updatedProxyEntry[0] }

});

logger.info(`Proxy config ${proxyId} updated by user ${userId}.`);

} catch (error) {

logger.error(`Error updating proxy config ID ${proxyId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

if (error.code === 'ER\_DUP\_ENTRY') {

return next(new AppError('A proxy configuration with this name or host/port combination already exists for this user.', 409));

}

next(new AppError('Failed to update proxy configuration.', 500));

}

};

// Deletes a proxy configuration.

exports.deleteProxyConfig = async (req, res, next) => {

const db = getDB();

const { proxyId } = req.params;

const userId = req.user.id;

try {

const [urlsUsingProxy] = await db.execute(

`SELECT id FROM Urls WHERE proxy\_config\_id = ? AND user\_id = ?`,

[proxyId, userId]

);

if (urlsUsingProxy.length > 0) {

return next(new AppError(`Cannot delete proxy config. It is currently used by ${urlsUsingProxy.length} monitored URL(s).`, 409));

}

const [result] = await db.execute(

`DELETE FROM ProxyConfigs WHERE id = ? AND user\_id = ?`,

[proxyId, userId]

);

if (result.affectedRows === 0) {

return next(new AppError('Proxy configuration not found or not authorized to delete.', 404));

}

res.status(204).json({

status: 'success',

data: null,

message: 'Proxy configuration deleted successfully'

});

logger.info(`Proxy config ${proxyId} deleted by user ${userId}.`);

} catch (error) {

logger.error(`Error deleting proxy config ID ${proxyId} for user ${userId}: ${error.message}. SQL Error: ${error.sqlMessage || 'N/A'}. Error Code: ${error.code || 'N/A'}`);

next(new AppError('Failed to delete proxy configuration.', 500));

}

};

**middleware/authMiddleware.js**

Middleware for JWT authentication.

// backend/src/middleware/authMiddleware.js

const jwt = require('jsonwebtoken');

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

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

const AppError = require('../utils/appError');

// Middleware to protect routes by verifying JWTs.

module.exports = (req, res, next) => {

const authHeader = req.header('Authorization');

if (!authHeader || !authHeader.startsWith('Bearer ')) {

logger.warn('Access denied: No token or invalid token format.');

return next(new AppError('No token, authorization denied', 401));

}

const token = authHeader.replace('Bearer ', '');

try {

const decoded = jwt.verify(token, config.jwt.secret);

req.user = decoded;

logger.debug(`Token verified for user ID: ${req.user.id}`);

next();

} catch (error) {

if (error.name === 'TokenExpiredError') {

logger.warn('Access denied: Token expired.');

return next(new AppError('Token expired', 401));

}

if (error.name === 'JsonWebTokenError') {

logger.warn('Access denied: Invalid token.');

return next(new AppError('Token is not valid', 401));

}

logger.error(`Unexpected error verifying token: ${error.message}`);

next(new AppError('Authentication failed', 500));

}

};

**utils/validation.js**

Contains Joi schemas for validating incoming request data.

// backend/src/utils/validation.js

const Joi = require('joi');

// User Validation Schemas

const registerSchema = Joi.object({

username: Joi.string().min(3).max(30).required(),

email: Joi.string().email().required(),

password: Joi.string().min(6).required(),

});

const loginSchema = Joi.object({

username: Joi.string().required(),

password: Joi.string().required(),

});

// URL Monitoring Validation Schema

const urlSchema = Joi.object({

name: Joi.string().min(3).max(255).required(),

url: Joi.string().uri().max(2048).required(),

type: Joi.string().valid('API', 'DOMAIN').required(),

monitoring\_interval\_minutes: Joi.number().integer().min(1).max(1440).required(),

proxy\_config\_id: Joi.number().integer().allow(null).optional(),

is\_active: Joi.boolean().truthy(1).falsy(0).required(), // Handles 1/0 for boolean

});

// Proxy Configuration Validation Schema

const proxyConfigSchema = Joi.object({

name: Joi.string().min(3).max(255).required(),

host: Joi.string().required(),

port: Joi.number().integer().min(1).max(65535).required(),

protocol: Joi.string().valid('http', 'https', 'socks4', 'socks5').required(),

username: Joi.string().allow('').optional(),

password: Joi.string().allow('').optional(),

enabled: Joi.boolean().required(),

});

// Alert Configuration Validation Schema

const alertConfigSchema = Joi.object({

email\_recipient: Joi.string().email().allow('').optional(),

snmp\_receiver\_host: Joi.string().allow('').optional(),

snmp\_community: Joi.string().allow('').optional(),

snmp\_api\_down\_oid: Joi.string().pattern(/^(\.\d+)+$/).allow('').optional(),

snmp\_cert\_expiry\_oid: Joi.string().pattern(/^(\.\d+)+$/).allow('').optional(),

cert\_warning\_days: Joi.number().integer().min(1).max(365).required(),

});

// Validation Functions

exports.validateRegister = (data) => registerSchema.validate(data);

exports.validateLogin = (data) => loginSchema.validate(data);

// Validates URL data, with optional fields for updates

exports.validateUrl = (data, isUpdate = false) => {

const schemaToUse = isUpdate ? urlSchema.keys({

name: Joi.string().min(3).max(255).optional(),

url: Joi.string().uri().max(2048).optional(),

type: Joi.string().valid('API', 'DOMAIN').optional(),

monitoring\_interval\_minutes: Joi.number().integer().min(1).max(1440).optional(),

proxy\_config\_id: Joi.number().integer().allow(null).optional(),

is\_active: Joi.boolean().truthy(1).falsy(0).optional(),

}).min(1) : urlSchema;

return schemaToUse.validate(data);

};

// Validates proxy config data, with optional fields for updates

exports.validateProxyConfig = (data, isUpdate = false) => {

const schemaToUse = isUpdate ? proxyConfigSchema.keys({

name: Joi.string().min(3).max(255).optional(),

host: Joi.string().optional(),

port: Joi.number().integer().min(1).max(65535).optional(),

protocol: Joi.string().valid('http', 'https', 'socks4', 'socks5').optional(),

username: Joi.string().allow('').optional(),

password: Joi.string().allow('').optional(),

enabled: Joi.boolean().optional(),

}).min(1) : proxyConfigSchema;

return schemaToUse.validate(data);

};

exports.validateAlertConfig = (data) => alertConfigSchema.validate(data);

**3.4. Backend API Endpoints**

All API endpoints are prefixed with /api. Authentication middleware (authMiddleware) protects most routes, requiring a valid JWT in the Authorization: Bearer <token> header.

**Authentication (/api/auth)**

* POST /api/auth/register: Register a new user.
* POST /api/auth/login: Authenticate user and get JWT.
* GET /api/auth/profile: Get authenticated user's profile.

**Monitoring (/api/monitor)**

* POST /api/monitor/urls: Add a new URL for monitoring.
* GET /api/monitor/urls: Get all URLs monitored by the user.
* PUT /api/monitor/urls/:urlId: Update an existing URL.
* DELETE /api/monitor/urls/:urlId: Delete a URL.
* GET /api/monitor/urls/:urlId/logs: Get monitoring logs for a specific URL.
* GET /api/monitor/urls/:urlId/certificate: Get certificate info for a domain URL.
* POST /api/monitor/proxy-configs: Add a new proxy configuration.
* GET /api/monitor/proxy-configs: Get all proxy configurations for the user.
* PUT /api/monitor/proxy-configs/:proxyId: Update an existing proxy configuration.
* DELETE /api/monitor/proxy-configs/:proxyId: Delete a proxy configuration.

**Alerts (/api/alerts)**

* GET /api/alerts/config: Get user's alert configuration.
* PUT /api/alerts/config: Save (create/update) user's alert configuration.

**Health Check (/api/health)**

* GET /api/health: Basic health check to confirm API is running.

**4. Frontend Documentation**

The frontend is a React.js application that provides the user interface for the monitoring tool.

**4.1. Frontend Setup**

**Prerequisites:**

* Node.js and npm (same as backend).

**Installation Steps:**

1. **Navigate to the frontend directory:**
2. cd frontend
3. **Install Node.js dependencies:**
4. npm install
5. **Create and configure the .env file:**
   * Create a file named .env in the frontend/ directory.
   * Copy the content from frontend/.env.example into frontend/.env.
   * **Crucially, update the REACT\_APP\_API\_BASE\_URL** to point to your backend server's address.
   * Example frontend/.env:
   * # frontend/.env
   * REACT\_APP\_API\_BASE\_URL=http://localhost:3000/api

**4.2. Frontend Directory Structure**

frontend/

├── public/ # Public assets (index.html, favicon)

│ └── index.html

├── src/

│ ├── api/ # Functions for making API calls to the backend

│ │ ├── auth.js

│ │ └── monitoring.js

│ ├── components/

│ │ ├── common/ # General reusable UI components (e.g., buttons, inputs)

│ │ ├── dashboard/ # Components specific to the dashboard (e.g., UrlCard, UptimeChart)

│ │ │ ├── UrlCard.js

│ │ │ ├── UptimeChart.js # Placeholder/Future

│ │ │ └── CertificateDisplay.js # Placeholder/Future

│ │ ├── forms/ # Reusable form components

│ │ │ ├── AddUrlForm.js # Not explicitly made, but conceptual

│ │ │ └── ProxyConfigForm.js # Not explicitly made, but conceptual

│ │ └── Navbar.js # Navigation bar

│ ├── context/ # React Context for global state management (e.g., Auth)

│ │ └── AuthContext.js

│ ├── pages/ # Top-level components representing application pages

│ │ ├── LoginPage.js

│ │ ├── DashboardPage.js

│ │ ├── ManageUrlsPage.js

│ │ └── SettingsPage.js

│ ├── App.js # Main React component, handles routing

│ ├── index.js # React app entry point (ReactDOM rendering)

│ ├── index.css # Global CSS, Tailwind imports, custom styles

│ └── assets/ # Images, icons, fonts (if not from CDN)

├── tailwind.config.js # Tailwind CSS configuration

├── postcss.config.js # PostCSS configuration for Tailwind

├── .env # Environment variables for frontend

└── package.json

**4.3. Frontend Key Components & Code Explanations**

This section explains the role of critical frontend files and provides insights into their code.

**App.js**

The root component of the React application. It sets up client-side routing using react-router-dom and integrates the AuthContext to provide authentication state globally.

import React from 'react';

import { BrowserRouter as Router, Routes, Route, Navigate } from 'react-router-dom';

import LoginPage from './pages/LoginPage.js';

import DashboardPage from './pages/DashboardPage.js';

import ManageUrlsPage from './pages/ManageUrlsPage.js';

import SettingsPage from './pages/SettingsPage.js';

import Navbar from './components/Navbar.js';

import { AuthProvider, useAuth } from './context/AuthContext.js'; // AuthContext for JWT

// PrivateRoute component: Protects routes, redirects to login if not authenticated

const PrivateRoute = ({ children }) => {

const { isAuthenticated, isLoading } = useAuth();

if (isLoading) {

return (

<div className="flex items-center justify-center min-h-screen text-xl text-gray-700">

<div className="animate-spin rounded-full h-12 w-12 border-b-2 border-blue-500"></div>

<span className="ml-4">Loading application...</span>

</div>

);

}

return isAuthenticated ? children : <Navigate to="/login" />;

};

// AppContent handles routing and conditional Navbar rendering

function AppContent() {

const { isAuthenticated } = useAuth();

return (

<>

{isAuthenticated && <Navbar />} {/\* Navbar shown only if authenticated \*/}

<div className="p-4 md:p-6 lg:p-8 flex-grow">

<Routes>

<Route path="/login" element={<LoginPage />} />

<Route path="/dashboard" element={<PrivateRoute><DashboardPage /></PrivateRoute>} />

<Route path="/manage-urls" element={<PrivateRoute><ManageUrlsPage /></PrivateRoute>} />

<Route path="/settings" element={<PrivateRoute><SettingsPage /></PrivateRoute>} />

<Route path="\*" element={isAuthenticated ? <Navigate to="/dashboard" /> : <Navigate to="/login" />} />

</Routes>

</div>

</>

);

}

// Root App component providing AuthProvider

function App() {

return (

<Router>

<AuthProvider>

<AppContent />

</AuthProvider>

</Router>

);

}

export default App;

**context/AuthContext.js**

Manages the global authentication state (user login status, user data, JWT token) and provides functions for logging in and out. It uses localStorage to persist the JWT.

import React, { createContext, useContext, useState, useEffect } from 'react';

const AuthContext = createContext(null);

export const AuthProvider = ({ children }) => {

const [isAuthenticated, setIsAuthenticated] = useState(false);

const [user, setUser] = useState(null); // Stores user data including role

const [isLoading, setIsLoading] = useState(true); // Tracks initial auth check

useEffect(() => {

const loadAuthStatus = async () => {

const token = localStorage.getItem('jwtToken');

if (token) {

try {

// Decode token (basic, client-side decoding) to get user info

const base64Url = token.split('.')[1];

const base64 = base64Url.replace(/-/g, '+').replace(/\_/g, '/');

const decodedPayload = JSON.parse(window.atob(base64));

// Check token expiry

if (decodedPayload.exp \* 1000 < Date.now()) {

console.warn('AuthContext: JWT token found but is expired locally.');

logout();

} else {

setIsAuthenticated(true);

setUser({

id: decodedPayload.id,

username: decodedPayload.username,

email: decodedPayload.email,

role: decodedPayload.role // Store the role

});

console.info(`AuthContext: User ${decodedPayload.username} (${decodedPayload.role}) authenticated from localStorage.`);

}

} catch (error) {

console.error('AuthContext: Error decoding or validating token from localStorage:', error);

logout();

}

}

setIsLoading(false); // Auth check complete

};

loadAuthStatus();

}, []);

const login = (token, userData) => {

localStorage.setItem('jwtToken', token);

setIsAuthenticated(true);

setUser(userData); // Store full user data from login response

console.info(`User ${userData.username} (${userData.role}) logged in.`);

};

const logout = () => {

localStorage.removeItem('jwtToken');

setIsAuthenticated(false);

setUser(null);

console.info('User logged out.');

};

return (

<AuthContext.Provider value={{ isAuthenticated, user, isLoading, login, logout }}>

{children}

</AuthContext.Provider>

);

};

export const useAuth = () => {

const context = useContext(AuthContext);

if (context === undefined) {

throw new Error('useAuth must be used within an AuthProvider');

}

return context;

};

**pages/DashboardPage.js**

Displays the main monitoring dashboard, fetching and presenting a summary of all monitored URLs.

import React, { useEffect, useState } from 'react';

import UrlCard from '../components/dashboard/UrlCard.js';

import { fetchUrls } from '../api/monitoring.js';

import { useAuth } from '../context/AuthContext.js';

function DashboardPage() {

const { user } = useAuth();

const [urls, setUrls] = useState([]);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

const loadData = async () => {

setLoading(true);

setError(null);

try {

const fetchedUrls = await fetchUrls();

setUrls(fetchedUrls.data); // Access the 'data' property from the backend response

} catch (err) {

console.error('Failed to fetch URLs:', err);

setError(err.message || 'Failed to load monitoring data. Please try again.');

} finally {

setLoading(false);

}

};

useEffect(() => {

loadData();

const interval = setInterval(loadData, 60000); // Poll every minute for updates

return () => clearInterval(interval); // Cleanup interval on component unmount

}, [user]);

if (loading && !error) {

return (

<div className="flex items-center justify-center min-h-[calc(100vh-80px)] text-xl text-gray-700">

<div className="animate-spin rounded-full h-12 w-12 border-b-2 border-blue-500"></div>

<span className="ml-4">Loading dashboard data...</span>

</div>

);

}

if (error) {

return (

<div className="bg-red-100 border border-red-400 text-red-700 px-4 py-3 rounded-md mx-auto max-w-lg mt-8" role="alert">

<strong className="font-bold">Error!</strong>

<span className="block sm:inline ml-2">{error}</span>

</div>

);

}

return (

<div className="container mx-auto px-4 py-8">

<h1 className="text-3xl font-bold text-gray-800 mb-8 text-center md:text-left">

Monitoring Dashboard

</h1>

<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-6">

{urls.length > 0 ? (

urls.map((url) => (

<UrlCard key={url.id} url={url} />

))

) : (

<p className="col-span-full text-center text-gray-600 p-8 bg-white rounded-lg shadow-md">

No URLs added yet. Go to "Manage URLs" to start monitoring!

</p>

)}

</div>

<h2 className="text-2xl font-semibold text-gray-800 mt-12 mb-6 text-center md:text-left">

Uptime Trends (Coming Soon!)

</h2>

<div className="bg-white p-6 rounded-lg shadow-md text-gray-600 text-center">

Detailed uptime charts will appear here for selected URLs.

</div>

<h2 className="text-2xl font-semibold text-gray-800 mt-12 mb-6 text-center md:text-left">

Certificate Expiry Details (Coming Soon!)

</h2>

<div className="bg-white p-6 rounded-lg shadow-md text-gray-600 text-center">

Comprehensive certificate information for your domains will be displayed here.

</div>

</div>

);

}

export default DashboardPage;

**components/dashboard/UrlCard.js**

A reusable component to display the status and key information of a single monitored URL on the dashboard.

import React from 'react';

import { Clock, CheckCircle, XCircle, AlertTriangle } from 'lucide-react'; // Icons

// Renders a card displaying monitoring information for a single URL.

function UrlCard({ url }) {

const lastStatus = url.last\_status || 'N/A';

const statusColor = lastStatus === 'up' ? 'text-green-600' : 'text-red-600';

const statusBgColor = lastStatus === 'up' ? 'bg-green-100' : 'bg-red-100';

const statusIcon = lastStatus === 'up' ? <CheckCircle size={20} /> : <XCircle size={20} />;

const typeBadgeColor = url.type === 'API' ? 'bg-blue-600' : 'bg-purple-600';

const lastCheckTime = url.last\_checked\_at

? new Date(url.last\_checked\_at).toLocaleString()

: 'Never';

const certExpiryStatus = url.certificate\_status || 'N/A';

let certExpiryColor = 'text-gray-600';

let certIcon = null;

if (url.type === 'DOMAIN') {

if (certExpiryStatus === 'valid') {

certExpiryColor = 'text-green-600';

certIcon = <CheckCircle size={18} className="inline-block mr-1" />;

} else if (certExpiryStatus === 'warning') {

certExpiryColor = 'text-yellow-600';

certIcon = <AlertTriangle size={18} className="inline-block mr-1" />;

} else if (certExpiryStatus === 'expired' || certExpiryStatus === 'error' || certExpiryStatus === 'unavailable' || certExpiryStatus === 'not\_reachable') {

certExpiryColor = 'text-red-600';

certIcon = <XCircle size={18} className="inline-block mr-1" />;

}

}

const certDaysRemaining = url.days\_remaining !== undefined && url.days\_remaining !== null

? `${url.days\_remaining} day${url.days\_remaining !== 1 ? 's' : ''}`

: 'N/A';

return (

<div className="bg-white rounded-lg shadow-lg p-6 flex flex-col justify-between border-t-4 border-blue-600 hover:shadow-xl transition-shadow duration-300">

<div>

<div className="flex justify-between items-start mb-4">

<h3 className="text-xl font-semibold text-gray-800 break-words pr-2">{url.name}</h3>

<span className={`${typeBadgeColor} text-white text-xs font-semibold px-2.5 py-0.5 rounded-full`}>

{url.type}

</span>

</div>

<p className="text-gray-600 text-sm mb-4 truncate">{url.url}</p>

<div className={`flex items-center text-sm font-medium p-2 rounded-md ${statusBgColor} ${statusColor} mb-2`}>

{statusIcon}

<span className="ml-2">Uptime Status: <span className="capitalize">{lastStatus}</span></span>

</div>

{url.type === 'API' && url.last\_latency !== null && (

<div className="text-sm text-gray-700 mb-2 flex items-center">

<Clock size={16} className="inline-block mr-1 text-gray-500" />

<span className="font-medium">Latency:</span> {url.last\_latency}ms

</div>

)}

{url.type === 'DOMAIN' && (

<div className="text-sm text-gray-700 mb-2">

<p className={`font-medium ${certExpiryColor} flex items-center`}>

{certIcon} Cert Status: <span className="capitalize ml-1">{certExpiryStatus.replace(/\_/g, ' ')}</span>

</p>

{url.days\_remaining !== undefined && url.days\_remaining !== null && (

<p className="font-medium text-gray-700 mt-1">Expires In: {certDaysRemaining}</p>

)}

</div>

)}

{url.proxy\_name && (

<div className="text-sm text-gray-700 mb-2">

<span className="font-medium">Proxy:</span> {url.proxy\_name} ({url.proxy\_protocol}://{url.proxy\_host}:{url.proxy\_port})

</div>

)}

</div>

<div className="mt-4 pt-4 border-t border-gray-200 text-xs text-gray-500 flex justify-between items-center">

<span>Last Checked: {lastCheckTime}</span>

<button className="text-blue-600 hover:text-blue-800 font-medium transition duration-200">

View Details

</button>

</div>

</div>

);

}

export default UrlCard;

**api/monitoring.js**

Contains all the API call functions for interacting with the backend's monitoring, proxy, and alerts endpoints.

// frontend/src/api/monitoring.js

const API\_BASE\_URL = process.env.REACT\_APP\_API\_BASE\_URL || 'http://localhost:3000/api';

const getAuthHeaders = () => {

const token = localStorage.getItem('jwtToken');

return token ? { 'Authorization': `Bearer ${token}` } : {};

};

const handleResponse = async (response) => {

if (!response.ok) {

const errorData = await response.json().catch(() => ({ message: 'Something went wrong.' }));

throw new Error(errorData.message || `HTTP error! Status: ${response.status}`);

}

return response.json();

};

export const fetchUrls = async () => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls`, {

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const addUrl = async (urlData) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls`, {

method: 'POST',

headers: {

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

...getAuthHeaders(),

},

body: JSON.stringify(urlData),

});

return handleResponse(response);

};

export const updateUrl = async (id, urlData) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls/${id}`, {

method: 'PUT',

headers: {

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

...getAuthHeaders(),

},

body: JSON.stringify(urlData),

});

return handleResponse(response);

};

export const deleteUrl = async (id) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls/${id}`, {

method: 'DELETE',

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const fetchMonitoringLogs = async (urlId, limit = 100) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls/${urlId}/logs?limit=${limit}`, {

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const fetchCertificateInfo = async (urlId) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/urls/${urlId}/certificate`, {

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const fetchProxyConfigs = async () => {

const response = await fetch(`${API\_BASE\_URL}/monitor/proxy-configs`, {

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const addProxyConfig = async (configData) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/proxy-configs`, {

method: 'POST',

headers: {

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

...getAuthHeaders(),

},

body: JSON.stringify(configData),

});

return handleResponse(response);

};

export const updateProxyConfig = async (id, configData) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/proxy-configs/${id}`, {

method: 'PUT',

headers: {

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

...getAuthHeaders(),

},

body: JSON.stringify(configData),

});

return handleResponse(response);

};

export const deleteProxyConfig = async (id) => {

const response = await fetch(`${API\_BASE\_URL}/monitor/proxy-configs/${id}`, {

method: 'DELETE',

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const fetchAlertConfig = async () => {

const response = await fetch(`${API\_BASE\_URL}/alerts/config`, {

headers: {

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

...getAuthHeaders(),

},

});

return handleResponse(response);

};

export const saveAlertConfig = async (configData) => {

const response = await fetch(`${API\_BASE\_URL}/alerts/config`, {

method: 'PUT',

headers: {

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

...getAuthHeaders(),

},

body: JSON.stringify(configData),

});

return handleResponse(response);

};

**4.4. Frontend Styling**

The frontend uses **Tailwind CSS** for styling. Tailwind is a utility-first CSS framework that allows building designs directly in your HTML/JSX without writing custom CSS, by composing small, single-purpose utility classes.

* **frontend/src/index.css**: Imports Tailwind's base, components, and utilities, and includes some custom global styles (e.g., font family, default border-radius).
* **tailwind.config.js**: Configures Tailwind, allowing for customization of colors, spacing, fonts, etc.
* **Inline Classes:** Most styling is applied directly in the JSX using Tailwind classes (e.g., bg-blue-600, p-4, rounded-lg).

This approach ensures a responsive and consistent design across the application.

**5. Database Schema**

The MySQL database monitoring\_tool uses the following tables:

* **Users**: Stores user authentication details (id, username, email, hashed password, role).
* **ProxyConfigs**: Stores configurations for proxy servers (name, host, port, protocol, optional credentials, enabled status). Linked to Users.
* **Urls**: Stores details of monitored URLs (name, URL string, type (API/DOMAIN), monitoring interval, association with a ProxyConfig, active status, last check status, last latency, last checked timestamp, certificate status, days remaining for cert, last error message). Linked to Users and ProxyConfigs. Includes a unique constraint on (user\_id, url(255)) to prevent duplicate URLs per user.
* **MonitoringLogs**: Stores historical monitoring results for each URL (status, latency, HTTP status code, error message, timestamp). Linked to Urls.
* **AlertConfigs**: Stores user-specific alert settings (email recipient, SNMP receiver host/community/OIDs, certificate warning threshold). Linked to Users with a one-to-one relationship.

**backend/src/db/models/sql\_schema.sql**:

-- backend/src/db/models/sql\_schema.sql

-- This file contains the SQL DDL (Data Definition Language) statements

-- to create the necessary tables for the monitoring tool.

-- These commands should be run directly in your MySQL client.

-- Create the database if it does not exist

CREATE DATABASE IF NOT EXISTS `monitoring\_tool`;

-- Use the newly created database

USE `monitoring\_tool`;

-- IMPORTANT: For a clean slate during development/initial setup,

-- drop tables if they already exist. This implicitly drops associated indexes.

-- In production, you would typically use a more sophisticated migration tool.

DROP TABLE IF EXISTS `AlertConfigs`;

DROP TABLE IF EXISTS `MonitoringLogs`;

DROP TABLE IF EXISTS `Urls`;

DROP TABLE IF EXISTS `ProxyConfigs`;

DROP TABLE IF EXISTS `Users`;

-- Table for Users

-- Stores user authentication details

CREATE TABLE IF NOT EXISTS `Users` (

`id` INT AUTO\_INCREMENT PRIMARY KEY,

`username` VARCHAR(255) UNIQUE NOT NULL,

`email` VARCHAR(255) UNIQUE NOT NULL,

`password` VARCHAR(255) NOT NULL, -- Hashed password

`role` ENUM('user', 'admin') DEFAULT 'user' NOT NULL, -- Added role column

`created\_at` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

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

);

-- Table for Proxy Configurations

-- Stores details about proxy servers that can be used for monitoring

CREATE TABLE IF NOT EXISTS `ProxyConfigs` (

`id` INT AUTO\_INCREMENT PRIMARY KEY,

`user\_id` INT NOT NULL, -- Link proxy config to a specific user

`name` VARCHAR(255) NOT NULL,

`host` VARCHAR(255) NOT NULL,

`port` INT NOT NULL,

`protocol` ENUM('http', 'https', 'socks4', 'socks5') DEFAULT 'http',

`username` VARCHAR(255) NULL, -- Optional proxy authentication

`password` VARCHAR(255) NULL, -- Optional proxy authentication

`enabled` BOOLEAN DEFAULT TRUE,

`created\_at` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

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

FOREIGN KEY (`user\_id`) REFERENCES `Users`(`id`) ON DELETE CASCADE,

UNIQUE (`user\_id`, `name`) -- Prevent duplicate proxy names per user

);

-- Table for Monitored URLs

-- Stores the URLs that are to be monitored (API endpoints or domains for certificates)

CREATE TABLE IF NOT EXISTS `Urls` (

`id` INT AUTO\_INCREMENT PRIMARY KEY,

`user\_id` INT NOT NULL,

`name` VARCHAR(255) NOT NULL, -- User-friendly name for the URL

`url` VARCHAR(2048) NOT NULL, -- The actual URL/endpoint to monitor

`type` ENUM('API', 'DOMAIN') NOT NULL, -- Type of monitoring

`monitoring\_interval\_minutes` INT DEFAULT 5, -- How often to monitor this URL

`proxy\_config\_id` INT NULL, -- Optional foreign key to ProxyConfigs

`is\_active` BOOLEAN DEFAULT TRUE, -- Whether monitoring is currently active for this URL

`last\_status` ENUM('up', 'down') NULL, -- Last recorded status (added)

`last\_latency` INT NULL, -- Last recorded latency in ms (added)

`last\_checked\_at` TIMESTAMP NULL, -- Timestamp of last check (added)

`certificate\_status` ENUM('valid', 'warning', 'expired', 'unavailable', 'error', 'not\_applicable', 'not\_reachable') NULL, -- Cert status (added)

`days\_remaining` INT NULL, -- Days until cert expiry (added)

`last\_error` TEXT NULL, -- Stores the last error message from monitoring (added)

`created\_at` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

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

FOREIGN KEY (`user\_id`) REFERENCES `Users`(`id`) ON DELETE CASCADE,

FOREIGN KEY (`proxy\_config\_id`) REFERENCES `ProxyConfigs`(`id`) ON DELETE SET NULL,

-- FIX: Use a prefix for the 'url' column in the unique index to avoid 'Specified key was too long' error

UNIQUE KEY `idx\_user\_url` (`user\_id`, `url`(255))

);

-- Table for Monitoring Logs

-- Stores historical uptime/response data for API monitoring

CREATE TABLE IF NOT EXISTS `MonitoringLogs` (

`id` INT AUTO\_INCREMENT PRIMARY KEY,

`url\_id` INT NOT NULL,

`status` ENUM('up', 'down') NOT NULL,

`latency` INT NULL, -- Response time in milliseconds

`status\_code` INT NULL, -- HTTP status code

`error` TEXT NULL, -- Error message if status is 'down'

`created\_at` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

FOREIGN KEY (`url\_id`) REFERENCES `Urls`(`id`) ON DELETE CASCADE

);

-- Table for Alert Configurations

-- Stores user-specific alert settings (email, SNMP details)

CREATE TABLE IF NOT EXISTS `AlertConfigs` (

`id` INT AUTO\_INCREMENT PRIMARY KEY,

`user\_id` INT NOT NULL UNIQUE, -- One-to-one relationship with Users table

`email\_recipient` VARCHAR(255) NULL, -- Email address to send alerts

`snmp\_receiver\_host` VARCHAR(255) NULL,

`snmp\_community` VARCHAR(255) NULL,

`snmp\_api\_down\_oid` VARCHAR(255) NULL,

`snmp\_cert\_expiry\_oid` VARCHAR(255) NULL,

`cert\_warning\_days` INT DEFAULT 30, -- Days before expiry to trigger certificate warning

`created\_at` TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,

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

FOREIGN KEY (`user\_id`) REFERENCES `Users`(`id`) ON DELETE CASCADE

);

-- Optional: Add indexes for frequently queried columns to improve performance

CREATE INDEX idx\_monitoringlogs\_url\_id ON `MonitoringLogs` (`url\_id`);

CREATE INDEX idx\_urls\_user\_id ON `Urls` (`user\_id`);

CREATE INDEX idx\_proxyconfigs\_user\_id ON `ProxyConfigs` (`user\_id`);

**6. How to Run the Application**

To run the full monitoring tool locally:

1. **Open two separate terminal windows/tabs.**
2. **In the first terminal, start the Backend:**
   * Navigate to the backend/ directory:
   * cd backend
   * Start the server:
   * npm start
   * You should see logs indicating database connection and server listening on port 3000.
3. **In the second terminal, start the Frontend:**
   * Navigate to the frontend/ directory:
   * cd frontend
   * Start the React development server:
   * npm start
   * This will usually open your default web browser to http://localhost:3001 (or another available port).

You can now interact with the application through your browser. Register a new user, log in, and start adding URLs to monitor!

**7. Troubleshooting**

* **Error: Cannot find module**:
  + Ensure you have run npm install in both backend/ and frontend/ directories.
  + Verify file paths and casing in require() or import statements are correct.
  + Try deleting node\_modules and package-lock.json and running npm install again.
* **TypeError: Router.use() requires a middleware function**:
  + This is almost always due to the authMiddleware.js file not exporting a function correctly, or an outdated version of it being loaded.
  + **Solution:** Manually verify backend/src/middleware/authMiddleware.js matches the correct content provided earlier in the conversation. Then, perform a deep clean: rm -rf node\_modules && rm package-lock.json && npm cache clean --force && npm install in the backend directory.
* **DB error updating... Incorrect arguments to mysqld\_stmt\_execute**:
  + Ensure that numerical values passed to SQL queries (like LIMIT) are correctly parsed as integers using parseInt().
* **Specified key was too long; max key length is 3072 bytes**:
  + This error occurs during database initialization. It means a unique index (like UNIQUE (user\_id, url)) is too long for MySQL's limits, especially with VARCHAR(2048) and multi-byte character sets.
  + **Solution:** Update your sql\_schema.sql to use a column prefix for the unique index on url, e.g., UNIQUE KEY \idx\_user\_url` (`user\_id`, `url`(255))`. Then, **re-initialize your database.**
* **URLs showing "Status: N/A" or "Never Checked"**:
  + Verify your backend/.env configuration, especially DB\_HOST, DB\_USER, DB\_PASSWORD, and DB\_DATABASE.
  + Ensure your MySQL server is running and accessible from your backend.
  + Check backend console logs for errors from monitoringScheduler.js, apiMonitorService.js, or certMonitorService.js. Errors here prevent status updates.
  + Ensure monitoringScheduler.start() is called in server.js.
* **"Is active must be a boolean" validation error**:
  + Ensure your frontend is sending true/false booleans for is\_active, or 1/0 numbers. The updated utils/validation.js should handle 1/0 coercion.
* **"Cannot read properties of undefined (reading 'notBefore')" related to node-forge**:
  + This indicates node-forge is still involved.
  + **Solution:** The certMonitorService.js has been updated to remove node-forge entirely. Perform a **deep clean** (rm -rf node\_modules, npm cache clean --force, npm install) in your backend directory to ensure no remnants of node-forge are loaded.

**8. Future Enhancements**

This project provides a solid foundation. Here are some potential future enhancements:

* **Detailed Monitoring Graphs:** Implement recharts or d3.js in the frontend to display historical latency and uptime trends for each URL.
* **User-Specific Dashboards:** Allow users to customize their dashboard layout and widgets.
* **Advanced Alerting:**
  + Integrate with more notification channels (e.g., Slack, PagerDuty, SMS).
  + Allow more granular alert conditions (e.g., alert if latency exceeds X ms for Y minutes).
  + Alert throttling to prevent alert storms.
* **API Monitoring with Custom Payloads:** Allow users to define custom HTTP methods (POST, PUT), headers, and request bodies for API checks.
* **Location-Based Monitoring:** Deploy multiple monitoring agents in different geographical locations to check reachability from various points.
* **Public Status Pages:** Allow users to generate public status pages for their monitored services.
* **User Roles and Permissions:** Implement more detailed access control beyond basic user/admin roles.
* **Load Testing Integration:** Ability to run simple load tests against monitored APIs.
* **Dockerization:** Provide Docker containers for easier deployment and environment consistency.