⚙️ AWIBI MEDTECH - Backend Development Guide

Version: 1.0.0
Date: July 9, 2025
Author: Manus AI
Status: Draft

🎯 Purpose of this Document

This comprehensive guide is designed to take you on a complete journey through the backend development of the AWIBI MEDTECH application. As a beginner-friendly resource, it will explain every concept, every line of code, and every architectural decision made in building the Node.js/Express backend. You will learn not just 'what' the code does, but 'why' it's written that way, 'how' it contributes to the overall application security and functionality, and 'what' you achieve by implementing each feature. By the end of this guide, you will have a deep understanding of modern backend development, including API design, authentication, authorization, database management, security, middleware, CORS, rate limiting, and much more. This is not just a code walkthrough—it's a complete learning experience that will enable you to build robust, secure, and scalable backend applications from scratch.

🏗️ Backend Architecture Overview

The AWIBI MEDTECH backend is built using Node.js with the Express.js framework, creating a RESTful API that serves data to the React frontend. This architecture follows the principles of separation of concerns, scalability, and security-first design. The backend acts as the brain of the application, handling business logic, data persistence, user authentication, and providing secure endpoints for the frontend to consume. [1]

Why Node.js and Express.js?

Node.js was chosen as the runtime environment for several compelling reasons that make it ideal for modern web applications:

1. JavaScript Everywhere: Node.js allows us to use JavaScript on both the frontend and backend, creating a unified development experience. This means developers can work across the entire stack without switching programming languages, reducing context switching and improving productivity. The shared language also enables code reuse between frontend and backend, such as validation schemas and utility functions.

2. Event-Driven, Non-Blocking I/O: Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. This is particularly beneficial for applications that handle many concurrent connections, such as web APIs that serve multiple frontend clients simultaneously. Unlike traditional server models that create a new thread for each request, Node.js handles all requests in a single thread using an event loop, making it highly scalable. [2]

3. Rich Ecosystem (npm): Node.js has access to the largest package ecosystem in the world through npm (Node Package Manager). This means we can leverage thousands of pre-built modules for common functionality like authentication (Passport.js), database connectivity (Mongoose), security (Helmet), and much more, significantly accelerating development.

4. Express.js Framework: Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It simplifies the process of building APIs by providing: - Robust routing system - Middleware support for cross-cutting concerns - Template engine integration - Static file serving - Error handling mechanisms

5. JSON-First Approach: Node.js and Express.js work seamlessly with JSON, which is the standard data format for modern web APIs. This natural affinity for JSON makes it easy to build RESTful APIs that communicate efficiently with frontend applications.

Backend Architecture Principles

The AWIBI MEDTECH backend follows several key architectural principles that ensure maintainability, scalability, and security:

1. RESTful API Design: The backend implements a RESTful (Representational State Transfer) API, which is an architectural style for designing networked applications. REST uses standard HTTP methods (GET, POST, PUT, DELETE) and status codes to create a predictable and intuitive API interface. This approach makes the API easy to understand, test, and consume by frontend applications and third-party integrations.

2. Layered Architecture: The backend is organized into distinct layers, each with specific responsibilities: - **Routes Layer:** Handles HTTP requests and responses - **Middleware Layer:** Processes requests before they reach route handlers - **Business Logic Layer:** Contains application-specific logic - **Data Access Layer:** Manages database interactions - **Configuration Layer:** Handles environment-specific settings

3. Security-First Design: Security is built into every layer of the application, not added as an afterthought. This includes input validation, authentication, authorization, rate limiting, CORS configuration, and protection against common web vulnerabilities.

4. Environment-Based Configuration: The application uses environment variables to manage configuration, allowing it to behave differently in development, testing, and production environments without code changes.

📁 Detailed Backend Structure

Let's examine the backend structure in detail, understanding the purpose and logic behind each directory and file:

```
backend/
├── config/                   # Configuration files for database,
authentication, etc.
│   ├── database.js           # MongoDB connection configuration
│   └── passport.js           # Passport.js authentication strategies
├── middleware/               # Express middleware functions
│   ├── auth.js               # Authentication middleware
│   ├── rbac.js               # Role-based access control middleware
│   └── security.js           # Security-related middleware
├── models/                   # Mongoose data models (database schemas)
│   ├── User.js               # User data model and schema
│   ├── Chapter.js            # Chapter data model and schema
│   ├── Event.js              # Event data model and schema
│   ├── Badge.js              # Badge system data model
│   └── Analytics.js          # Analytics and metrics data model
├── routes/                   # API route definitions and handlers
│   ├── auth.js               # Authentication routes (login, register, OAuth)
│   ├── users.js              # User management routes
│   ├── chapters.js           # Chapter-related routes
│   ├── events.js             # Event management routes
│   ├── badges.js             # Badge system routes
│   ├── dashboard.js          # Dashboard data routes
│   └── analytics.js          # Analytics and reporting routes
├── .env                      # Development environment variables
├── .env.production           # Production environment variables
├── package.json              # Project metadata, dependencies, and scripts
├── server-final-production.js # Main server entry point for production
└── server-test.js            # Simplified server for testing purposes
```

Understanding the Directory Structure:

config/ Directory: This directory contains configuration files that set up various aspects of the application. Configuration files are separated from the main application logic to promote modularity and make it easier to manage different environments (development, testing, production).

middleware/ Directory: Middleware functions are pieces of code that execute during the request-response cycle. They have access to the request object, response object, and the next middleware function in the application's request-response cycle. Middleware can execute code, modify request and response objects, end the request-response cycle, or call the next middleware in the stack.

models/ Directory: This directory contains Mongoose models that define the structure of data stored in MongoDB. Models serve as the interface between the application and the database, providing methods for creating, reading, updating, and deleting data while enforcing data validation and business rules.

routes/ Directory: Route files define the API endpoints and their corresponding handler functions. Each route file typically handles a specific domain of functionality (authentication, users, events, etc.), promoting organization and maintainability.

📦 Understanding package.json

The `package.json` file is the heart of any Node.js project, serving as a manifest that describes the project and its dependencies. Let's examine the backend's `package.json` in detail:

```json
{
  "name": "awibi-medtech-backend",
  "version": "3.0.0",
  "main": "server-final-production.js",
  "scripts": {
    "start": "node server-final-production.js",
    "dev": "nodemon server-final-production.js",
    "test": "node server-test.js"
  },
  "dependencies": {
    "express": "^4.18.2",
    "mongoose": "^7.5.0",
    "cors": "^2.8.5",
    "helmet": "^7.0.0",
    "bcryptjs": "^2.4.3",
    "jsonwebtoken": "^9.0.2",
    "passport": "^0.6.0",
    "passport-google-oauth20": "^2.0.0",
    "express-rate-limit": "^6.10.0",
    "dotenv": "^16.3.1",
    "express-validator": "^7.0.1",
    "express-mongo-sanitize": "^2.2.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.1"
  }
}
```

Breaking Down Each Dependency:

Core Framework Dependencies:

express (^4.18.2): Express.js is the web application framework that provides the foundation for building the API. It handles HTTP requests, routing, middleware integration, and response generation. Express is minimal and unopinionated, allowing developers to structure applications according to their needs while providing essential web server functionality.

mongoose (^7.5.0): Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It provides a schema-based solution to model application data, including built-in type casting, validation, query building, and business logic hooks. Mongoose makes it easier to work with MongoDB by providing a more structured approach to data modeling compared to the native MongoDB driver.

Security Dependencies:

cors (^2.8.5): Cross-Origin Resource Sharing (CORS) is a security feature implemented by web browsers to restrict web pages from making requests to a different domain than the one serving the web page. The cors middleware configures the server to allow or deny cross-origin requests based on specified rules, enabling the React frontend (running on a different port or domain) to communicate with the backend API.

helmet (^7.0.0): Helmet helps secure Express applications by setting various HTTP headers that protect against common web vulnerabilities. It's not a silver bullet for security, but it provides a good foundation by setting headers like Content Security Policy, X-Frame-Options, and X-XSS-Protection to mitigate various attack vectors.

express-rate-limit (^6.10.0): This middleware implements rate limiting to prevent abuse of the API by limiting the number of requests a client can make within a specified time window. Rate limiting helps protect against denial-of-service attacks, brute force attacks, and general API abuse.

express-mongo-sanitize (^2.2.0): This middleware sanitizes user input to prevent NoSQL injection attacks. It removes any keys that start with '$' or contain '.' from user input, which are characters that have special meaning in MongoDB queries and could be used maliciously.

Authentication Dependencies:

bcryptjs (^2.4.3): Bcrypt is a password hashing function designed to be slow and computationally expensive, making it resistant to brute force attacks. It automatically handles salt generation and provides methods for hashing passwords and comparing plain text passwords with hashed versions.

jsonwebtoken (^9.0.2): JSON Web Tokens (JWT) are a compact, URL-safe means of representing claims to be transferred between two parties. This library provides methods for creating, signing, and verifying JWTs, which are used for stateless authentication in the application.

passport (^0.6.0): Passport is authentication middleware for Node.js that supports over 500 authentication strategies, including local username/password authentication and OAuth providers like Google, Facebook, and Twitter. It provides a consistent API for authentication regardless of the underlying strategy.

passport-google-oauth20 (^2.0.0): This is a Passport strategy for authenticating with Google using OAuth 2.0. It allows users to log in using their Google accounts, providing a seamless authentication experience without requiring users to create new accounts.

Validation and Utility Dependencies:

express-validator (^7.0.1): Express-validator is a set of express.js middlewares that wraps validator.js, providing comprehensive input validation and sanitization. It allows for declarative validation rules and provides detailed error messages for invalid input.

dotenv (^16.3.1): Dotenv loads environment variables from a .env file into process.env, providing a convenient way to manage configuration and sensitive information like database connection strings and API keys without hardcoding them in the source code.

Development Dependencies:

nodemon (^3.0.1): Nodemon is a development tool that automatically restarts the Node.js application when file changes are detected. This eliminates the need to manually restart the server during development, significantly improving the development experience.

Scripts Explanation:

"start": "node server-final-production.js": This script starts the production server using the standard Node.js runtime. It's used in production environments where the application should run with maximum performance and stability.

"dev": "nodemon server-final-production.js": This script starts the development server using nodemon, which automatically restarts the server when files change. This is used during development to provide a smooth development experience with automatic reloading.

"test": "node server-test.js": This script runs a simplified test server that can be used for testing purposes without requiring a full database connection or all production dependencies.

🔧 Main Server Entry Point: server-final-production.js

The main server file is the entry point of the backend application. It orchestrates all the components, sets up middleware, configures security, and starts the HTTP server. Let's examine this file in detail:

```
const express = require('express');
const dotenv = require('dotenv');
const connectDB = require('./config/database');
const cors = require('cors');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');
const mongoSanitize = require('express-mongo-sanitize');
const passport = require('passport');

// Load environment variables
dotenv.config({ path: './.env.production' });

// Connect to database
connectDB();

const app = express();

// CORS Configuration
const corsOptions = {
  origin: function (origin, callback) {
    const allowedOrigins = process.env.ALLOWED_ORIGINS?.split(',') || [];
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error('Not allowed by CORS'));
    }
  },
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS', 'PATCH'],
  allowedHeaders: ['Content-Type', 'Authorization', 'X-Requested-With',
'Accept', 'Origin'],
  optionsSuccessStatus: 200
};
app.use(cors(corsOptions));

// Security Middleware
app.use(helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'"],
      scriptSrc: ["'self'"],
      imgSrc: ["'self'", "data:", "https:"],
    },
  },
  crossOriginEmbedderPolicy: false
}));

// Body parsing middleware
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));

// Data sanitization against NoSQL query injection
app.use(mongoSanitize());

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: {
    error: 'Too many requests from this IP, please try again later.',
    retryAfter: 15 * 60 * 1000
  },
  standardHeaders: true,
  legacyHeaders: false,
});
app.use('/api/', limiter);

// Passport middleware
app.use(passport.initialize());
require('./config/passport')(passport);

// Import Routes
const authRoutes = require('./routes/auth');
const userRoutes = require('./routes/users');
const chapterRoutes = require('./routes/chapters');
const eventRoutes = require('./routes/events');
const badgeRoutes = require('./routes/badges');
const dashboardRoutes = require('./routes/dashboard');
const analyticsRoutes = require('./routes/analytics');

// Use Routes
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/chapters', chapterRoutes);
app.use('/api/events', eventRoutes);
app.use('/api/badges', badgeRoutes);
app.use('/api/dashboard', dashboardRoutes);
app.use('/api/analytics', analyticsRoutes);

// Health Check Endpoint
app.get('/health', (req, res) => {
  res.json({
    status: 'OK',
    message: 'AWIBI MEDTECH API is running',
    timestamp: new Date().toISOString(),
    uptime: process.uptime(),
    environment: process.env.NODE_ENV || 'development',
    version: '3.0.0'
  });
});

// 404 handler
app.use('*', (req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found',
    path: req.originalUrl
  });
});

// Global error handling middleware
app.use((err, req, res, next) => {
  console.error('Error:', err.stack);

  // Mongoose validation error
  if (err.name === 'ValidationError') {
    const errors = Object.values(err.errors).map(e => e.message);
    return res.status(400).json({
      success: false,
      message: 'Validation Error',
      errors
    });
  }

  // JWT errors
  if (err.name === 'JsonWebTokenError') {
    return res.status(401).json({
      success: false,
      message: 'Invalid token'
    });
  }

  if (err.name === 'TokenExpiredError') {
    return res.status(401).json({
      success: false,
      message: 'Token expired'
    });
  }

  // CORS errors
  if (err.message === 'Not allowed by CORS') {
    return res.status(403).json({
      success: false,
      message: 'CORS policy violation'
    });
  }

  // Default error
  res.status(err.status || 500).json({
    success: false,
    message: process.env.NODE_ENV === 'production'
      ? 'Internal server error'
      : err.message
  });
});

const PORT = process.env.PORT || 5000;

app.listen(PORT, '0.0.0.0', () => {
  console.log(`🚀 Server running on port ${PORT}`);
  console.log(`📊 Health check: http://localhost:${PORT}/health`);
  console.log(`🌍 Environment: ${process.env.NODE_ENV || 'development'}`);
});
```

Understanding the Server Setup:

Environment Configuration:

```
dotenv.config({ path: './.env.production' });
```

This loads environment variables from the specified file, allowing the application to access configuration values like database URLs, API keys, and other sensitive information without hardcoding them in the source code.

Database Connection:

```
connectDB();
```

This establishes a connection to the MongoDB database using the configuration defined in the database config file. The connection is established early in the application lifecycle to ensure the database is available before handling requests.

Express Application Initialization:

```
const app = express();
```

This creates an Express application instance that will handle HTTP requests and responses. The app object provides methods for configuring middleware, defining routes, and starting the server.

CORS Configuration Deep Dive:

Cross-Origin Resource Sharing (CORS) is a critical security feature that controls which domains can access the API. Let's examine the CORS configuration in detail:

```
const corsOptions = {
  origin: function (origin, callback) {
    const allowedOrigins = process.env.ALLOWED_ORIGINS?.split(',') || [];
    if (!origin || allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      callback(new Error('Not allowed by CORS'));
    }
  },
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS', 'PATCH'],
  allowedHeaders: ['Content-Type', 'Authorization', 'X-Requested-With',
'Accept', 'Origin'],
  optionsSuccessStatus: 200
};
```

Dynamic Origin Validation: The origin function dynamically validates incoming requests based on environment variables. This allows different allowed origins for development and production environments. The function checks if the request origin is in the list of allowed origins and either permits or denies the request accordingly.

Credentials Support:

```
credentials: true
```

This enables the server to accept credentials (cookies, authorization headers) in cross-origin requests, which is essential for authentication to work properly between the frontend and backend running on different domains or ports.

Allowed Methods: The methods array specifies which HTTP methods are allowed for cross-origin requests. This includes all standard REST methods plus OPTIONS, which is used for preflight requests.

Allowed Headers: The allowedHeaders array specifies which headers can be included in cross-origin requests. This includes standard headers like Content-Type and Authorization, which are essential for API communication.

Security Middleware Configuration:

Helmet Security Headers:

```
 app.use(helmet({
   contentSecurityPolicy: {
     directives: {
       defaultSrc: ["'self'"],
       styleSrc: ["'self'", "'unsafe-inline'"],
       scriptSrc: ["'self'"],
       imgSrc: ["'self'", "data:", "https:"],
     },
   },
   crossOriginEmbedderPolicy: false
}));
```

Helmet sets various HTTP headers to protect against common web vulnerabilities: - **Content Security Policy (CSP):** Prevents XSS attacks by controlling which resources can be loaded - **X-Frame-Options:** Prevents clickjacking attacks - **X-XSS-Protection:** Enables browser XSS filtering - **X-Content-Type-Options:** Prevents MIME type sniffing

Body Parsing Middleware:

```
 app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true, limit: '10mb' }));
```

These middlewares parse incoming request bodies: - **express.json():** Parses JSON payloads with a 10MB limit - **express.urlencoded():** Parses URL-encoded payloads with extended syntax support

NoSQL Injection Protection:

```
app.use(mongoSanitize());
```

This middleware removes any keys that start with '$' or contain '.' from user input, preventing NoSQL injection attacks that could manipulate database queries.

Rate Limiting:

```
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // limit each IP to 100 requests per windowMs
  message: {
    error: 'Too many requests from this IP, please try again later.',
    retryAfter: 15 * 60 * 1000
  },
  standardHeaders: true,
  legacyHeaders: false,
});
```

Rate limiting protects the API from abuse by limiting the number of requests per IP address within a time window. This configuration allows 100 requests per 15-minute window, which is sufficient for normal usage while protecting against automated attacks.

Route Organization:

```
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/chapters', chapterRoutes);
app.use('/api/events', eventRoutes);
app.use('/api/badges', badgeRoutes);
app.use('/api/dashboard', dashboardRoutes);
app.use('/api/analytics', analyticsRoutes);
```

Routes are organized by functionality and mounted under the `/api` prefix. This creates a clear API structure where related endpoints are grouped together, making the API intuitive to use and maintain.

Error Handling:

The server includes comprehensive error handling for different types of errors:

Mongoose Validation Errors:

```
if (err.name === 'ValidationError') {
  const errors = Object.values(err.errors).map(e => e.message);
  return res.status(400).json({
    success: false,
    message: 'Validation Error',
    errors
  });
}
```

JWT Authentication Errors:

```
if (err.name === 'JsonWebTokenError') {
  return res.status(401).json({
    success: false,
    message: 'Invalid token'
  });
}
```

CORS Policy Violations:

```
if (err.message === 'Not allowed by CORS') {
  return res.status(403).json({
    success: false,
    message: 'CORS policy violation'
  });
}
```

This comprehensive error handling ensures that clients receive meaningful error messages and appropriate HTTP status codes, making debugging easier and improving the overall API experience.

🗄️ Database Configuration and Connection

The database configuration is a critical component that establishes the connection between the application and MongoDB. Let's examine the database configuration file:

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

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
      maxPoolSize: 10, // Maintain up to 10 socket connections
      serverSelectionTimeoutMS: 5000, // Keep trying to send operations for 5
seconds
      socketTimeoutMS: 45000, // Close sockets after 45 seconds of inactivity
      family: 4 // Use IPv4, skip trying IPv6
    });

    console.log(`✅ MongoDB Connected: ${conn.connection.host}`);

    // Handle connection events
    mongoose.connection.on('error', (err) => {
      console.error('❌ MongoDB connection error:', err);
    });

    mongoose.connection.on('disconnected', () => {
      console.log('⚠ MongoDB disconnected');
    });

    mongoose.connection.on('reconnected', () => {
      console.log('✅ MongoDB reconnected');
    });

    // Graceful shutdown
    process.on('SIGINT', async () => {
      await mongoose.connection.close();
      console.log('🔒 MongoDB connection closed through app termination');
      process.exit(0);
    });

  } catch (error) {
    console.error('❌ Database connection failed:', error.message);
    process.exit(1);
  }
};

module.exports = connectDB;
```

Understanding Database Connection:

Mongoose Connection Options:

useNewUrlParser and useUnifiedTopology: These options enable the new MongoDB driver's connection management and topology engine, providing better connection handling and monitoring capabilities.

maxPoolSize:

```
maxPoolSize: 10
```

This sets the maximum number of connections in the connection pool. Connection pooling improves performance by reusing existing connections rather than creating new ones for each request.

serverSelectionTimeoutMS:

```
serverSelectionTimeoutMS: 5000
```

This sets how long the driver will wait to find an available server before timing out. A 5-second timeout provides a good balance between responsiveness and allowing for temporary network issues.

socketTimeoutMS:

```
socketTimeoutMS: 45000
```

This sets how long a socket will stay open during inactivity. A 45-second timeout prevents hanging connections while allowing for longer-running operations.

Connection Event Handling: The configuration includes event listeners for various connection states: - **error:** Logs connection errors for debugging - **disconnected:** Notifies when the connection is lost - **reconnected:** Confirms when connection is restored

Graceful Shutdown:

```
process.on('SIGINT', async () => {
  await mongoose.connection.close();
  console.log('🔒 MongoDB connection closed through app termination');
  process.exit(0);
});
```

This ensures that database connections are properly closed when the application shuts down, preventing connection leaks and ensuring clean termination.

Why MongoDB?

MongoDB was chosen for the AWIBI MEDTECH application for several reasons:

1. Document-Oriented Storage: MongoDB stores data in flexible, JSON-like documents, which aligns well with JavaScript applications. This eliminates the object-relational impedance mismatch common with SQL databases.

2. Schema Flexibility: MongoDB's flexible schema allows for easy evolution of data models as the application grows and requirements change, without requiring complex migrations.

3. Horizontal Scalability: MongoDB supports horizontal scaling through sharding, allowing the application to handle increased load by distributing data across multiple servers.

4. Rich Query Language: MongoDB provides a powerful query language that supports complex queries, indexing, and aggregation operations.

5. Cloud Integration: MongoDB Atlas provides a fully managed cloud database service with automatic scaling, backup, and monitoring capabilities.

📊 Data Models and Schemas

Data models define the structure and validation rules for data stored in the database. Mongoose provides a schema-based solution for modeling application data. Let's examine the key data models used in the AWIBI MEDTECH application:

User Model (models/User.js)

The User model is the foundation of the authentication and authorization system:

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

const userSchema = new mongoose.Schema({
  firstName: {
    type: String,
    required: [true, 'First name is required'],
    trim: true,
    maxlength: [50, 'First name cannot exceed 50 characters']
  },
  lastName: {
    type: String,
    required: [true, 'Last name is required'],
    trim: true,
    maxlength: [50, 'Last name cannot exceed 50 characters']
  },
  email: {
    type: String,
    required: [true, 'Email is required'],
    unique: true,
    lowercase: true,
    trim: true,
    match: [
      /^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/,
      'Please enter a valid email address'
    ]
  },
  password: {
    type: String,
    required: function() {
      return !this.googleId; // Password required only if not using Google
OAuth
    },
    minlength: [6, 'Password must be at least 6 characters'],
    select: false // Don't include password in queries by default
  },
  googleId: {
    type: String,
    sparse: true // Allow multiple null values but unique non-null values
  },
  role: {
    type: String,
    enum: ['member', 'leader', 'admin', 'superadmin'],
    default: 'member'
  },
  profilePicture: {
    type: String,
    default: null
  },
  bio: {
    type: String,
    maxlength: [500, 'Bio cannot exceed 500 characters']
  },
  location: {
    type: String,
    maxlength: [100, 'Location cannot exceed 100 characters']
  },
  skills: [{
    type: String,
    trim: true
  }],
  interests: [{
    type: String,
    trim: true
  }],
  chapters: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Chapter'
  }],
  badges: [{
    badge: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Badge'
    },
    earnedAt: {
      type: Date,
      default: Date.now
    }
  }],
  isActive: {
    type: Boolean,
    default: true
  },
  lastLogin: {
    type: Date,
    default: Date.now
  },
  emailVerified: {
    type: Boolean,
    default: false
  },
  emailVerificationToken: {
    type: String,
    select: false
  },
  passwordResetToken: {
    type: String,
    select: false
  },
  passwordResetExpires: {
    type: Date,
    select: false
  }
}, {
  timestamps: true, // Automatically add createdAt and updatedAt fields
  toJSON: { virtuals: true },
  toObject: { virtuals: true }
});

// Virtual for full name
userSchema.virtual('fullName').get(function() {
  return `$`{this.firstName} `${this.lastName}`;
});

// Index for better query performance
userSchema.index({ email: 1 });
userSchema.index({ googleId: 1 });
userSchema.index({ role: 1 });

// Pre-save middleware to hash password
userSchema.pre('save', async function(next) {
  // Only hash the password if it has been modified (or is new)
  if (!this.isModified('password')) return next();
  try {
    // Hash password with cost of 12
    const salt = await bcrypt.genSalt(12);
    this.password = await bcrypt.hash(this.password, salt);
    next();
  } catch (error) {
    next(error);
  }
});

// Instance method to check password
userSchema.methods.comparePassword = async function(candidatePassword) {
  if (!this.password) return false;
  return await bcrypt.compare(candidatePassword, this.password);
};

// Instance method to generate JWT token
userSchema.methods.generateAuthToken = function() {
  const jwt = require('jsonwebtoken');
  return jwt.sign(
    {
      id: this._id,
      email: this.email,
      role: this.role
    },
    process.env.JWT_SECRET,
    { expiresIn: '7d' }
  );
};

// Static method to find user by email
userSchema.statics.findByEmail = function(email) {
  return this.findOne({ email: email.toLowerCase() });
};

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

Understanding the User Schema:

Field Validation: Each field includes comprehensive validation rules: - **required:** Ensures the field is present - **trim:** Removes whitespace from string fields - **maxlength/minlength:** Enforces length constraints - **match:** Uses regex for email validation - **enum:** Restricts values to predefined options

Conditional Requirements:

```
required: function() {
  return !this.googleId; // Password required only if not using Google OAuth
}
```

This demonstrates conditional validation where password is only required for users not using Google OAuth authentication.

Indexes for Performance:

```
 userSchema.index({ email: 1 });
userSchema.index({ googleId: 1 });
userSchema.index({ role: 1 });
```

Indexes improve query performance by creating efficient lookup structures for frequently queried fields.

Virtual Fields:

```
 userSchema.virtual('fullName').get(function() {
  return `$`{this.firstName} `${this.lastName}`;
});
```

Virtual fields are computed properties that don't exist in the database but can be accessed like regular fields.

Pre-save Middleware:

```
 userSchema.pre('save', async function(next) {
  if (!this.isModified('password')) return next();

  const salt = await bcrypt.genSalt(12);
  this.password = await bcrypt.hash(this.password, salt);
  next();
});
```

This middleware automatically hashes passwords before saving to the database, ensuring passwords are never stored in plain text.

Instance Methods:

```
 userSchema.methods.comparePassword = async function(candidatePassword) {
  if (!this.password) return false;
  return await bcrypt.compare(candidatePassword, this.password);
};
```

Instance methods are available on individual document instances and provide reusable functionality like password comparison.

Static Methods:

```
 userSchema.statics.findByEmail = function(email) {
  return this.findOne({ email: email.toLowerCase() });
};
```

Static methods are available on the model itself and provide reusable query functionality.

Chapter Model (models/Chapter.js)

The Chapter model represents local community groups:

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

const chapterSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'Chapter name is required'],
    trim: true,
    maxlength: [100, 'Chapter name cannot exceed 100 characters']
  },
  description: {
    type: String,
    required: [true, 'Chapter description is required'],
    maxlength: [1000, 'Description cannot exceed 1000 characters']
  },
  location: {
    city: {
      type: String,
      required: [true, 'City is required'],
      trim: true
    },
    state: {
      type: String,
      required: [true, 'State is required'],
      trim: true
    },
    country: {
      type: String,
      required: [true, 'Country is required'],
      trim: true
    },
    coordinates: {
      latitude: {
        type: Number,
        min: -90,
        max: 90
      },
      longitude: {
        type: Number,
        min: -180,
        max: 180
      }
    }
  },
  leaders: [{
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User',
      required: true
    },
    role: {
      type: String,
      enum: ['president', 'vice-president', 'secretary', 'treasurer'],
      required: true
    },
    appointedAt: {
      type: Date,
      default: Date.now
    }
  }],
  members: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'User'
  }],
  events: [{
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Event'
  }],
  isActive: {
    type: Boolean,
    default: true
  },
  establishedDate: {
    type: Date,
    required: [true, 'Established date is required']
  },
  contactEmail: {
    type: String,
    required: [true, 'Contact email is required'],
    match: [
      /^\w+([.-]?\w+)*@\w+([.-]?\w+)*(\.\w{2,3})+$/,
      'Please enter a valid email address'
    ]
  },
  socialMedia: {
    website: String,
    linkedin: String,
    twitter: String,
    facebook: String
  },
  meetingSchedule: {
    frequency: {
      type: String,
      enum: ['weekly', 'biweekly', 'monthly', 'quarterly'],
      default: 'monthly'
    },
    dayOfWeek: {
      type: String,
      enum: ['monday', 'tuesday', 'wednesday', 'thursday', 'friday',
'saturday', 'sunday']
    },
    time: String,
    location: String
  }
}, {
  timestamps: true,
  toJSON: { virtuals: true },
  toObject: { virtuals: true }
});

// Virtual for member count
chapterSchema.virtual('memberCount').get(function() {
  return this.members ? this.members.length : 0;
});

// Virtual for full location string
chapterSchema.virtual('fullLocation').get(function() {
  return `$`{this.location.city}, `${this.location.state},
${this.location.country}`;
});

// Index for location-based queries
chapterSchema.index({ 'location.city': 1, 'location.state': 1 });
chapterSchema.index({ 'location.coordinates': '2dsphere' }); // Geospatial
```

```
index

// Static method to find chapters by location
chapterSchema.statics.findByLocation = function(city, state) {
  return this.find({
    'location.city': new RegExp(city, 'i'),
    'location.state': new RegExp(state, 'i'),
    isActive: true
  });
};

// Static method to find nearby chapters (requires coordinates)
chapterSchema.statics.findNearby = function(longitude, latitude, maxDistance =
50000) {
  return this.find({
    'location.coordinates': {
      $near: {
        $geometry: {
          type: 'Point',
          coordinates: [longitude, latitude]
        },
        $maxDistance: maxDistance // in meters
      }
    },
    isActive: true
  });
};

module.exports = mongoose.model('Chapter', chapterSchema);
```

Understanding the Chapter Schema:

Embedded Documents:

```
location: {
  city: { type: String, required: true },
  state: { type: String, required: true },
  country: { type: String, required: true },
  coordinates: {
    latitude: { type: Number, min: -90, max: 90 },
    longitude: { type: Number, min: -180, max: 180 }
  }
}
```

Embedded documents allow related data to be stored together, improving query performance and maintaining data consistency.

Array of References:

```
members: [{
  type: mongoose.Schema.Types.ObjectId,
  ref: 'User'
}]
```

This creates a many-to-many relationship between chapters and users, allowing efficient queries for chapter membership.

Geospatial Indexing:

```
chapterSchema.index({ 'location.coordinates': '2dsphere' });
```

The 2dsphere index enables geospatial queries, allowing the application to find chapters near a specific location.

Complex Validation:

```
coordinates: {
  latitude: { type: Number, min: -90, max: 90 },
  longitude: { type: Number, min: -180, max: 180 }
}
```

Numeric validation ensures that coordinates are within valid ranges for latitude and longitude.

Event Model (models/Event.js)

The Event model manages community events and activities:

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

const eventSchema = new mongoose.Schema({
  title: {
    type: String,
    required: [true, 'Event title is required'],
    trim: true,
    maxlength: [200, 'Title cannot exceed 200 characters']
  },
  description: {
    type: String,
    required: [true, 'Event description is required'],
    maxlength: [2000, 'Description cannot exceed 2000 characters']
  },
  category: {
    type: String,
    required: [true, 'Event category is required'],
    enum: ['workshop', 'seminar', 'networking', 'conference', 'hackathon',
'social', 'training']
  },
  dateTime: {
    start: {
      type: Date,
      required: [true, 'Start date and time is required']
    },
    end: {
      type: Date,
      required: [true, 'End date and time is required'],
      validate: {
        validator: function(value) {
          return value > this.dateTime.start;
        },
        message: 'End date must be after start date'
      }
    }
  },
  location: {
    type: {
      type: String,
      enum: ['physical', 'virtual', 'hybrid'],
      required: true
    },
    venue: {
      name: String,
      address: String,
      city: String,
      state: String,
      country: String,
      coordinates: {
        latitude: Number,
        longitude: Number
      }
    },
    virtualLink: {
      type: String,
      validate: {
        validator: function(value) {
          if (this.location.type === 'virtual' || this.location.type ===
'hybrid') {
            return value && value.length > 0;
          }
          return true;
        },
        message: 'Virtual link is required for virtual or hybrid events'
      }
    }
  },
  organizer: {
    chapter: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Chapter',
      required: true
    },
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User',
      required: true
    }
  },
  speakers: [{
    name: {
      type: String,
      required: true,
      trim: true
    },
    bio: String,
    title: String,
    company: String,
    profilePicture: String,
    socialMedia: {
      linkedin: String,
      twitter: String
    }
  }],
  attendees: [{
    user: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User',
      required: true
    },
    registeredAt: {
      type: Date,
      default: Date.now
    },
    attended: {
      type: Boolean,
      default: false
    },
    feedback: {
      rating: {
        type: Number,
        min: 1,
        max: 5
      },
      comment: String,
      submittedAt: Date
    }
  }],
  capacity: {
    type: Number,
    required: [true, 'Event capacity is required'],
    min: [1, 'Capacity must be at least 1']
  },
  registrationDeadline: {
    type: Date,
    required: [true, 'Registration deadline is required'],
    validate: {
      validator: function(value) {
        return value <= this.dateTime.start;
      },
      message: 'Registration deadline must be before event start time'
    }
  },
  tags: [{
    type: String,
    trim: true,
    lowercase: true
  }],
  isPublic: {
    type: Boolean,
    default: true
  },
  status: {
    type: String,
    enum: ['draft', 'published', 'cancelled', 'completed'],
    default: 'draft'
  },
  resources: [{
    name: String,
    url: String,
    type: {
      type: String,
      enum: ['presentation', 'document', 'video', 'link', 'other']
    }
  }]
}, {
  timestamps: true,
  toJSON: { virtuals: true },
  toObject: { virtuals: true }
});

// Virtual for available spots
eventSchema.virtual('availableSpots').get(function() {
  const registeredCount = this.attendees ? this.attendees.length : 0;
  return Math.max(0, this.capacity - registeredCount);
});

// Virtual for registration status
eventSchema.virtual('isRegistrationOpen').get(function() {
  const now = new Date();
  return this.status === 'published' &&
         this.registrationDeadline > now &&
         this.availableSpots > 0;
});

// Virtual for event duration in hours
eventSchema.virtual('durationHours').get(function() {
  if (this.dateTime.start && this.dateTime.end) {
    const diffMs = this.dateTime.end - this.dateTime.start;
    return Math.round(diffMs / (1000 * 60 * 60) * 10) / 10; // Round to 1
decimal
  }
  return 0;
});
```

```
// Indexes for efficient queries
eventSchema.index({ 'dateTime.start': 1 });
eventSchema.index({ category: 1 });
eventSchema.index({ status: 1 });
eventSchema.index({ 'organizer.chapter': 1 });
eventSchema.index({ tags: 1 });

// Compound index for location-based queries
eventSchema.index({ 'location.venue.city': 1, 'location.venue.state': 1 });

// Text index for search functionality
eventSchema.index({
  title: 'text',
  description: 'text',
  tags: 'text'
});

// Static method to find upcoming events
eventSchema.statics.findUpcoming = function(limit = 10) {
  return this.find({
    'dateTime.start': { $gte: new Date() },
    status: 'published'
  })
  .sort({ 'dateTime.start': 1 })
  .limit(limit)
  .populate('organizer.chapter', 'name location')
  .populate('organizer.user', 'firstName lastName');
};

// Static method to find events by category
eventSchema.statics.findByCategory = function(category) {
  return this.find({
    category: category,
    status: 'published',
    'dateTime.start': { $gte: new Date() }
  })
  .sort({ 'dateTime.start': 1 })
  .populate('organizer.chapter', 'name location');
};

// Instance method to check if user is registered
eventSchema.methods.isUserRegistered = function(userId) {
  return this.attendees.some(attendee =>
    attendee.user.toString() === userId.toString()
  );
};

// Instance method to register user
eventSchema.methods.registerUser = function(userId) {
  if (this.isUserRegistered(userId)) {
    throw new Error('User is already registered for this event');
  }

  if (this.availableSpots <= 0) {
    throw new Error('Event is full');
  }

  if (!this.isRegistrationOpen) {
    throw new Error('Registration is closed for this event');
  }

  this.attendees.push({ user: userId });
  return this.save();
};

module.exports = mongoose.model('Event', eventSchema);
```

Understanding the Event Schema:

Complex Validation:

```
validate: {
  validator: function(value) {
    return value > this.dateTime.start;
  },
  message: 'End date must be after start date'
}
```

Custom validators can access other fields in the document to perform complex validation logic.

Conditional Validation:

```
validate: {
  validator: function(value) {
    if (this.location.type === 'virtual' || this.location.type === 'hybrid') {
      return value && value.length > 0;
    }
    return true;
  },
  message: 'Virtual link is required for virtual or hybrid events'
}
```

This demonstrates conditional validation where certain fields are required based on the values of other fields.

Text Search Index:

```
eventSchema.index({
  title: 'text',
  description: 'text',
  tags: 'text'
});
```

Text indexes enable full-text search capabilities, allowing users to search events by keywords in multiple fields.

Complex Instance Methods:

```
eventSchema.methods.registerUser = function(userId) {
  if (this.isUserRegistered(userId)) {
    throw new Error('User is already registered for this event');
  }

  if (this.availableSpots <= 0) {
    throw new Error('Event is full');
  }

  if (!this.isRegistrationOpen) {
    throw new Error('Registration is closed for this event');
  }

  this.attendees.push({ user: userId });
  return this.save();
};
```

Instance methods can implement complex business logic that operates on individual documents.

What Data Models Achieve:

1. **Data Integrity:** Validation rules ensure data quality and consistency
2. **Performance:** Indexes optimize query performance for common access patterns
3. **Relationships:** References and embedded documents model complex data relationships
4. **Business Logic:** Methods encapsulate domain-specific functionality
5. **Flexibility:** Schema design accommodates evolving requirements
6. **Security:** Validation prevents invalid or malicious data from entering the system

The data models form the foundation of the application's data layer, providing a robust and flexible structure for storing and manipulating application data while maintaining integrity and performance.

🔐 Authentication and Authorization

Authentication and authorization are fundamental security concepts that control access to the application. Authentication verifies who a user is, while authorization determines what they can do. The AWIBI MEDTECH backend implements a comprehensive authentication and authorization system using JSON Web Tokens (JWT), Passport.js for OAuth integration, and role-based access control (RBAC).

Understanding Authentication vs Authorization

Authentication answers the question "Who are you?" It's the process of verifying the identity of a user, typically through credentials like username/password or third-party services like Google OAuth. Once authenticated, the system knows who the user is.

Authorization answers the question "What can you do?" It's the process of determining whether an authenticated user has permission to access a specific resource or perform a particular action. Authorization happens after authentication and depends on the user's role, permissions, or other attributes.

JWT (JSON Web Tokens) Authentication

JSON Web Tokens are a compact, URL-safe means of representing claims to be transferred between two parties. In the context of web applications, JWTs are used for stateless authentication, meaning the server doesn't need to store session information. [3]

JWT Structure and Benefits

A JWT consists of three parts separated by dots: 1. **Header:** Contains metadata about the token type and signing algorithm 2. **Payload:** Contains the claims (user information and permissions) 3. **Signature:** Ensures the token hasn't been tampered with

Benefits of JWT: - **Stateless:** No need to store session data on the server - **Scalable:** Works well in distributed systems - **Cross-domain:** Can be used across different domains - **Self-contained:** Contains all necessary information - **Secure:** Cryptographically signed to prevent tampering

Authentication Middleware (middleware/auth.js)

The authentication middleware verifies JWT tokens and extracts user information from requests:

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

// Middleware to verify JWT token
const authenticateToken = async (req, res, next) => {
  try {
    // Get token from header
    const authHeader = req.headers.authorization;
    const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN

    if (!token) {
      return res.status(401).json({
        success: false,
        message: 'Access token is required'
      });
    }

    // Verify token
    const decoded = jwt.verify(token, process.env.JWT_SECRET);

    // Get user from database
    const user = await User.findById(decoded.id).select('-password');

    if (!user) {
      return res.status(401).json({
        success: false,
        message: 'User not found'
      });
    }

    if (!user.isActive) {
      return res.status(401).json({
        success: false,
        message: 'Account is deactivated'
      });
    }

    // Add user to request object
    req.user = user;
    next();
  } catch (error) {
    if (error.name === 'JsonWebTokenError') {
      return res.status(401).json({
        success: false,
        message: 'Invalid token'
      });
    }

    if (error.name === 'TokenExpiredError') {
      return res.status(401).json({
        success: false,
        message: 'Token expired'
      });
    }

    console.error('Authentication error:', error);
    res.status(500).json({
      success: false,
      message: 'Authentication failed'
    });
  }
};

// Middleware for optional authentication (doesn't fail if no token)
const optionalAuth = async (req, res, next) => {
  try {
    const authHeader = req.headers.authorization;
    const token = authHeader && authHeader.split(' ')[1];

    if (token) {
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      const user = await User.findById(decoded.id).select('-password');

      if (user && user.isActive) {
        req.user = user;
      }
    }

    next();
  } catch (error) {
    // Continue without authentication for optional auth
    next();
  }
};

module.exports = {
  authenticateToken,
  optionalAuth
};
```

Understanding Authentication Middleware:

Token Extraction:

```
 const authHeader = req.headers.authorization;
const token = authHeader && authHeader.split(' ')[1]; // Bearer TOKEN
```

The middleware extracts the JWT token from the Authorization header, which follows the format "Bearer ". The `&&` operator provides safe navigation, preventing errors if the header is missing.

Token Verification:

```
 const decoded = jwt.verify(token, process.env.JWT_SECRET);
```

The `jwt.verify()` method validates the token signature and decodes the payload. If the token is invalid or expired, it throws an error that's caught and handled appropriately.

User Lookup:

```
const user = await User.findById(decoded.id).select('-password');
```

After verifying the token, the middleware fetches the complete user record from the database. The `.select('-password')` excludes the password field for security.

Request Enhancement:

```
 req.user = user;
next();
```

The middleware adds the authenticated user to the request object, making it available to subsequent middleware and route handlers.

Error Handling: The middleware handles different types of JWT errors: - **JsonWebTokenError:** Invalid token format or signature - **TokenExpiredError:** Token has expired - **General errors:** Database connection issues or other unexpected errors

Role-Based Access Control (RBAC) Middleware

RBAC is a security model that restricts access based on user roles. The AWIBI MEDTECH application implements four roles: member, leader, admin, and superadmin.

```
 // middleware/rbac.js
const authorize = (...roles) => {
  return (req, res, next) => {
    // Check if user is authenticated
    if (!req.user) {
      return res.status(401).json({
        success: false,
        message: 'Authentication required'
      });
    }

    // Check if user has required role
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        success: false,
        message: 'Insufficient permissions',
        required: roles,
        current: req.user.role
      });
    }

    next();
  };
};

// Role hierarchy check
const authorizeHierarchy = (minimumRole) => {
  const roleHierarchy = {
    'member': 1,
    'leader': 2,
    'admin': 3,
    'superadmin': 4
  };

  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        message: 'Authentication required'
      });
    }

    const userRoleLevel = roleHierarchy[req.user.role] || 0;
    const requiredLevel = roleHierarchy[minimumRole] || 0;

    if (userRoleLevel < requiredLevel) {
      return res.status(403).json({
        success: false,
        message: 'Insufficient permissions',
        required: minimumRole,
        current: req.user.role
      });
    }

    next();
  };
};

// Resource ownership check
const authorizeOwnership = (resourceField = 'user') => {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({
        success: false,
        message: 'Authentication required'
      });
    }

    // Allow admins and superadmins to access any resource
    if (['admin', 'superadmin'].includes(req.user.role)) {
      return next();
    }

    // Check if user owns the resource
    const resourceUserId = req.params.userId || req.body[resourceField] ||
req.user.id;

    if (resourceUserId !== req.user.id.toString()) {
      return res.status(403).json({
        success: false,
        message: 'Access denied: You can only access your own resources'
      });
    }

    next();
  };
};

module.exports = {
  authorize,
  authorizeHierarchy,
  authorizeOwnership
};
```

Understanding RBAC Middleware:

Role-Based Authorization:

```
const authorize = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role)) {
      return res.status(403).json({
        success: false,
        message: 'Insufficient permissions'
      });
    }
    next();
  };
};
```

This middleware factory creates authorization middleware that checks if the user's role is in the list of allowed roles. The spread operator (`...roles`) allows multiple roles to be passed as arguments.

Hierarchical Authorization:

```
const authorizeHierarchy = (minimumRole) => {
  const roleHierarchy = {
    'member': 1,
    'leader': 2,
    'admin': 3,
    'superadmin': 4
  };
  // ...
};
```

Hierarchical authorization assumes that higher-level roles inherit permissions from lower-level roles. For example, an admin can perform all actions that a leader can perform.

Ownership-Based Authorization:

```
const authorizeOwnership = (resourceField = 'user') => {
  return (req, res, next) => {
    // Allow admins and superadmins to access any resource
    if (['admin', 'superadmin'].includes(req.user.role)) {
      return next();
    }

    // Check if user owns the resource
    const resourceUserId = req.params.userId || req.body[resourceField] ||
req.user.id;

    if (resourceUserId !== req.user.id.toString()) {
      return res.status(403).json({
        success: false,
        message: 'Access denied: You can only access your own resources'
      });
    }
    next();
  };
};
```

Ownership-based authorization ensures users can only access resources they own, while allowing administrators to access any resource.

Passport.js Configuration for OAuth

Passport.js is authentication middleware for Node.js that supports over 500 authentication strategies. The AWIBI MEDTECH application uses Passport for Google OAuth 2.0 authentication.

```javascript
// config/passport.js
const GoogleStrategy = require('passport-google-oauth20').Strategy;
const User = require('../models/User');

module.exports = function(passport) {
  passport.use(new GoogleStrategy({
    clientID: process.env.GOOGLE_CLIENT_ID,
    clientSecret: process.env.GOOGLE_CLIENT_SECRET,
    callbackURL: "/api/auth/google/callback"
  },
  async (accessToken, refreshToken, profile, done) => {
    try {
      // Check if user already exists with this Google ID
      let user = await User.findOne({ googleId: profile.id });

      if (user) {
        // User exists, update last login
        user.lastLogin = new Date();
        await user.save();
        return done(null, user);
      }

      // Check if user exists with same email
      user = await User.findOne({ email: profile.emails[0].value });

      if (user) {
        // Link Google account to existing user
        user.googleId = profile.id;
        user.emailVerified = true;
        user.lastLogin = new Date();

        // Update profile picture if not set
        if (!user.profilePicture && profile.photos[0]) {
          user.profilePicture = profile.photos[0].value;
        }

        await user.save();
        return done(null, user);
      }

      // Create new user
      user = new User({
        googleId: profile.id,
        firstName: profile.name.givenName,
        lastName: profile.name.familyName,
        email: profile.emails[0].value,
        profilePicture: profile.photos[0]?.value,
        emailVerified: true,
        lastLogin: new Date()
      });

      await user.save();
      done(null, user);
    } catch (error) {
      console.error('Google OAuth error:', error);
      done(error, null);
    }
  }));

  // Serialize user for session
  passport.serializeUser((user, done) => {
    done(null, user.id);
  });

  // Deserialize user from session
  passport.deserializeUser(async (id, done) => {
    try {
      const user = await User.findById(id).select('-password');
      done(null, user);
    } catch (error) {
      done(error, null);
    }
  });
};
```

Understanding Passport Configuration:

Google OAuth Strategy:

```
passport.use(new GoogleStrategy({
  clientID: process.env.GOOGLE_CLIENT_ID,
  clientSecret: process.env.GOOGLE_CLIENT_SECRET,
  callbackURL: "/api/auth/google/callback"
}, async (accessToken, refreshToken, profile, done) => {
  // Strategy implementation
}));
```

The Google OAuth strategy handles the OAuth 2.0 flow with Google. The callback function receives the user's Google profile information and handles user creation or linking.

User Linking Logic: The strategy implements sophisticated user linking logic: 1. **Existing Google User:** If a user with the Google ID exists, update their last login 2. **Email Match:** If a user with the same email exists, link the Google account 3. **New User:** Create a new user with Google profile information

Profile Data Extraction:

```
user = new User({
  googleId: profile.id,
  firstName: profile.name.givenName,
  lastName: profile.name.familyName,
  email: profile.emails[0].value,
  profilePicture: profile.photos[0]?.value,
  emailVerified: true
});
```

The strategy extracts relevant information from the Google profile and creates a user record with appropriate defaults.

Authentication Routes (routes/auth.js)

The authentication routes handle user registration, login, and OAuth flows:

```
 const express = require('express');
const bcrypt = require('bcryptjs');
const jwt = require('jsonwebtoken');
const passport = require('passport');
const { body, validationResult } = require('express-validator');
const User = require('../models/User');
const { authenticateToken } = require('../middleware/auth');

const router = express.Router();

// Register new user
router.post('/register', [
  body('firstName')
    .trim()
    .isLength({ min: 2, max: 50 })
    .withMessage('First name must be between 2 and 50 characters'),
  body('lastName')
    .trim()
    .isLength({ min: 2, max: 50 })
    .withMessage('Last name must be between 2 and 50 characters'),
  body('email')
    .isEmail()
    .normalizeEmail()
    .withMessage('Please provide a valid email'),
  body('password')
    .isLength({ min: 6 })
    .withMessage('Password must be at least 6 characters')
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
    .withMessage('Password must contain at least one lowercase letter, one
uppercase letter, and one number')
], async (req, res) => {
  try {
    // Check for validation errors
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        success: false,
        message: 'Validation failed',
        errors: errors.array()
      });
    }

    const { firstName, lastName, email, password } = req.body;

    // Check if user already exists
    const existingUser = await User.findOne({ email });
    if (existingUser) {
      return res.status(400).json({
        success: false,
        message: 'User already exists with this email'
      });
    }

    // Create new user
    const user = new User({
      firstName,
      lastName,
      email,
      password // Will be hashed by pre-save middleware
    });
    await user.save();

    // Generate JWT token
    const token = user.generateAuthToken();

    // Remove password from response
    const userResponse = user.toObject();
    delete userResponse.password;

    res.status(201).json({
      success: true,
      message: 'User registered successfully',
      data: {
        user: userResponse,
        token
      }
    });
  } catch (error) {
    console.error('Registration error:', error);
    res.status(500).json({
      success: false,
      message: 'Registration failed'
    });
  }
});

// Login user
router.post('/login', [
  body('email')
    .isEmail()
    .normalizeEmail()
    .withMessage('Please provide a valid email'),
  body('password')
    .notEmpty()
    .withMessage('Password is required')
], async (req, res) => {
  try {
    // Check for validation errors
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({
        success: false,
        message: 'Validation failed',
        errors: errors.array()
      });
    }

    const { email, password } = req.body;

    // Find user and include password for comparison
    const user = await User.findOne({ email }).select('+password');
    if (!user) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Check if account is active
    if (!user.isActive) {
      return res.status(401).json({
        success: false,
        message: 'Account is deactivated'
      });
    }

    // Compare password
    const isPasswordValid = await user.comparePassword(password);
    if (!isPasswordValid) {
      return res.status(401).json({
        success: false,
        message: 'Invalid credentials'
      });
    }

    // Update last login
    user.lastLogin = new Date();
    await user.save();

    // Generate JWT token
    const token = user.generateAuthToken();

    // Remove password from response
    const userResponse = user.toObject();
    delete userResponse.password;

    res.json({
      success: true,
      message: 'Login successful',
      data: {
        user: userResponse,
        token
      }
    });
  } catch (error) {
    console.error('Login error:', error);
    res.status(500).json({
      success: false,
      message: 'Login failed'
    });
  }
});

// Google OAuth routes
router.get('/google',
  passport.authenticate('google', { scope: ['profile', 'email'] })
);

router.get('/google/callback',
  passport.authenticate('google', { session: false }),
  async (req, res) => {
    try {
      // Generate JWT token for the authenticated user
      const token = req.user.generateAuthToken();

      // Redirect to frontend with token
      const frontendURL = process.env.FRONTEND_URL || 'http://localhost:5173';
      res.redirect(`$`{frontendURL}/auth/callback?token=`${token}`);
    } catch (error) {
      console.error('Google OAuth callback error:', error);
      const frontendURL = process.env.FRONTEND_URL || 'http://localhost:5173';
      res.redirect(`${frontendURL}/auth/callback?error=oauth_failed`);
    }
  }
```

```
);

// Verify token
router.get('/verify', authenticateToken, (req, res) => {
  res.json({
    success: true,
    message: 'Token is valid',
    data: {
      user: req.user
    }
  });
});

// Logout (client-side token removal)
router.post('/logout', authenticateToken, async (req, res) => {
  try {
    // In a JWT-based system, logout is typically handled client-side
    // by removing the token from storage. However, we can update
    // the user's last activity or perform other cleanup here.

    res.json({
      success: true,
      message: 'Logout successful'
    });
  } catch (error) {
    console.error('Logout error:', error);
    res.status(500).json({
      success: false,
      message: 'Logout failed'
    });
  }
});

// Test login endpoint for development
router.post('/test-login', async (req, res) => {
  try {
    const { email, password } = req.body;

    // For testing purposes, create a test user if it doesn't exist
    let user = await User.findOne({ email });

    if (!user) {
      user = new User({
        firstName: 'Test',
        lastName: 'User',
        email: email,
        password: password,
        role: 'member',
        emailVerified: true
      });
      await user.save();
    }

    // Generate token
    const token = user.generateAuthToken();

    res.json({
      success: true,
      message: 'Test login successful',
      data: {
        user: {
          id: user._id,
          firstName: user.firstName,
          lastName: user.lastName,
          email: user.email,
          role: user.role
        },
        token
      }
    });
  } catch (error) {
    console.error('Test login error:', error);
    res.status(500).json({
      success: false,
      message: 'Test login failed'
    });
  }
});

module.exports = router;
```

Understanding Authentication Routes:

Input Validation:

```
 body('password')
   .isLength({ min: 6 })
   .withMessage('Password must be at least 6 characters')
   .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)/)
   .withMessage('Password must contain at least one lowercase letter, one
uppercase letter, and one number')
```

Express-validator provides comprehensive input validation with custom error messages. The password validation ensures strong passwords with multiple character types.

Secure Password Handling:

```
 // Find user and include password for comparison
const user = await User.findOne({ email }).select('+password');

// Compare password
const isPasswordValid = await user.comparePassword(password);
```

Passwords are excluded from queries by default (using `select: false` in the schema). When needed for authentication, they're explicitly included with `select('+password')`.

OAuth Flow Handling:

```
router.get('/google/callback',
  passport.authenticate('google', { session: false }),
  async (req, res) => {
    const token = req.user.generateAuthToken();
    const frontendURL = process.env.FRONTEND_URL || 'http://localhost:5173';
    res.redirect(`$`{frontendURL}/auth/callback?token=`${token}`);
  }
);
```

The OAuth callback generates a JWT token and redirects to the frontend with the token as a query parameter, allowing the frontend to store the token and authenticate the user.

Security Best Practices: 1. **Generic Error Messages:** "Invalid credentials" instead of "User not found" or "Wrong password" 2. **Account Status Checks:** Verify account is active before authentication 3. **Rate Limiting:** Applied at the server level to prevent brute force attacks 4. **Token Expiration:** JWTs have expiration times to limit exposure 5. **Secure Headers:** Helmet middleware adds security headers

What Authentication and Authorization Achieve:

1. **Identity Verification:** Ensures users are who they claim to be
2. **Access Control:** Restricts access to resources based on user roles
3. **Data Protection:** Prevents unauthorized access to sensitive information
4. **Audit Trail:** Tracks user actions for security and compliance
5. **User Experience:** Provides seamless login with multiple authentication methods
6. **Scalability:** Stateless JWT authentication scales well across multiple servers

The authentication and authorization system forms the security backbone of the application, ensuring that only authorized users can access appropriate resources while providing a smooth user experience.

🛡️ Security Implementation

Security is a critical aspect of backend development that must be considered at every layer of the application. The AWIBI MEDTECH backend implements multiple security measures to protect against common web vulnerabilities and ensure data integrity. Security is not a single feature but a comprehensive approach that includes input validation, output encoding, authentication, authorization, rate limiting, and protection against various attack vectors.

Security Middleware (middleware/security.js)

The security middleware implements various protective measures:

```javascript
const rateLimit = require('express-rate-limit');
const helmet = require('helmet');
const mongoSanitize = require('express-mongo-sanitize');
const { body, validationResult } = require('express-validator');

// Rate limiting configurations for different endpoints
const createRateLimiter = (windowMs, max, message) => {
  return rateLimit({
    windowMs,
    max,
    message: {
      error: message,
      retryAfter: windowMs
    },
    standardHeaders: true,
    legacyHeaders: false,
    handler: (req, res) => {
      res.status(429).json({
        success: false,
        message: message,
        retryAfter: Math.ceil(windowMs / 1000)
      });
    }
  });
};

// Strict rate limiting for authentication endpoints
const authLimiter = createRateLimiter(
  15 * 60 * 1000, // 15 minutes
  5, // 5 attempts
  'Too many authentication attempts, please try again later'
);

// General API rate limiting
const apiLimiter = createRateLimiter(
  15 * 60 * 1000, // 15 minutes
  100, // 100 requests
  'Too many requests from this IP, please try again later'
);

// Strict rate limiting for password reset
const passwordResetLimiter = createRateLimiter(
  60 * 60 * 1000, // 1 hour
  3, // 3 attempts
  'Too many password reset attempts, please try again later'
);

// Input sanitization middleware
const sanitizeInput = (req, res, next) => {
  // Remove any keys that start with '$' or contain '.'
  mongoSanitize()(req, res, () => {
    // Additional sanitization for specific fields
    if (req.body) {
      // Trim whitespace from string fields
      Object.keys(req.body).forEach(key => {
        if (typeof req.body[key] === 'string') {
          req.body[key] = req.body[key].trim();
        }
      });

      // Remove potentially dangerous HTML tags
      const dangerousFields = ['bio', 'description', 'comment'];
      dangerousFields.forEach(field => {
        if (req.body[field]) {
          req.body[field] = req.body[field]
            .replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, '')
            .replace(/<iframe\b[^<]*(?:(?!<\/iframe>)<[^<]*)*<\/iframe>/gi, '')
            .replace(/javascript:/gi, '')
            .replace(/on\w+\s*=/gi, '');
        }
      });
    }

    next();
  });
};

// Security headers configuration
const securityHeaders = helmet({
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'", "https://fonts.googleapis.com"],
      fontSrc: ["'self'", "https://fonts.gstatic.com"],
      scriptSrc: ["'self'"],
      imgSrc: ["'self'", "data:", "https:", "blob:"],
      connectSrc: ["'self'", "https://api.awibi-medtech.com"],
      frameSrc: ["'none'"],
      objectSrc: ["'none'"],
      mediaSrc: ["'self'"],
      workerSrc: ["'self'"],
      childSrc: ["'self'"],
      formAction: ["'self'"],
      upgradeInsecureRequests: [],
    },
  },
  crossOriginEmbedderPolicy: false,
  crossOriginOpenerPolicy: { policy: "same-origin" },
  crossOriginResourcePolicy: { policy: "cross-origin" },
  dnsPrefetchControl: { allow: false },
  frameguard: { action: 'deny' },
  hidePoweredBy: true,
  hsts: {
    maxAge: 31536000,
    includeSubDomains: true,
    preload: true
  },
  ieNoOpen: true,
  noSniff: true,
  originAgentCluster: true,
  permittedCrossDomainPolicies: false,
  referrerPolicy: { policy: "no-referrer" },
  xssFilter: true,
});

// Request logging for security monitoring
const securityLogger = (req, res, next) => {
  const startTime = Date.now();

  // Log suspicious patterns
  const suspiciousPatterns = [
    /(\<|\%3C)script/i,
    /javascript:/i,
    /vbscript:/i,
    /onload=/i,
    /onerror=/i,
    /\.\./,
    /\/etc\/passwd/i,
    /\/proc\//i,
    /cmd\.exe/i,
    /powershell/i
  ];

  const requestString = JSON.stringify({
    url: req.url,
    body: req.body,
    query: req.query,
    headers: req.headers
  });

  const isSuspicious = suspiciousPatterns.some(pattern =>
    pattern.test(requestString)
  );

  if (isSuspicious) {
    console.warn('🚨 Suspicious request detected:', {
      ip: req.ip,
      userAgent: req.get('User-Agent'),
      url: req.url,
      method: req.method,
      timestamp: new Date().toISOString()
    });
  }

  // Log response time and status
  res.on('finish', () => {
    const duration = Date.now() - startTime;

    if (duration > 5000) { // Log slow requests
      console.warn('⏱️ Slow request detected:', {
        url: req.url,
        method: req.method,
        duration: `${duration}ms`,
        status: res.statusCode
      });
    }
  });

  next();
};

// File upload security
const validateFileUpload = (allowedTypes = [], maxSize = 5 * 1024 * 1024) => {
  return (req, res, next) => {
    if (!req.file && !req.files) {
      return next();
    }

    const files = req.files || [req.file];

    for (const file of files) {
      // Check file type
      if (allowedTypes.length > 0 && !allowedTypes.includes(file.mimetype)) {
        return res.status(400).json({
          success: false,
          message: `File type ${file.mimetype} is not allowed`,
          allowedTypes
        });
      }

      // Check file size
      if (file.size > maxSize) {
        return res.status(400).json({
          success: false,
          message: `File size exceeds limit of ${maxSize / (1024 * 1024)}MB`
        });
      }

      // Check for executable file extensions
      const dangerousExtensions = ['.exe', '.bat', '.cmd', '.scr', '.pif',
'.com'];
      const fileExtension = file.originalname.toLowerCase().split('.').pop();

      if (dangerousExtensions.includes(`.${fileExtension}`)) {
        return res.status(400).json({
          success: false,
          message: 'Executable files are not allowed'
        });
      }
    }

    next();
  };
};

module.exports = {
  authLimiter,
  apiLimiter,
  passwordResetLimiter,
  sanitizeInput,
  securityHeaders,
  securityLogger,
  validateFileUpload
};
```

Understanding Security Measures:

Rate Limiting Strategy:

```
const authLimiter = createRateLimiter(
  15 * 60 * 1000, // 15 minutes
  5, // 5 attempts
  'Too many authentication attempts, please try again later'
);
```

Different endpoints have different rate limits based on their sensitivity: - **Authentication endpoints:** 5 attempts per 15 minutes (strict) - **General API:** 100 requests per 15 minutes (moderate) - **Password reset:** 3 attempts per hour (very strict)

Input Sanitization:

```
const sanitizeInput = (req, res, next) => {
  mongoSanitize()(req, res, () => {
    // Remove potentially dangerous HTML tags
    const dangerousFields = ['bio', 'description', 'comment'];
    dangerousFields.forEach(field => {
      if (req.body[field]) {
        req.body[field] = req.body[field]
          .replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, '')
          .replace(/<iframe\b[^<]*(?:(?!<\/iframe>)<[^<]*)*<\/iframe>/gi, '')
          .replace(/javascript:/gi, '')
          .replace(/on\w+\s*=/gi, '');
      }
    });
  });
};
```

Input sanitization removes dangerous content: - **NoSQL Injection:** Removes MongoDB operators (`$`, `.`) - **XSS Prevention:** Removes script tags and event handlers - **Data Cleaning:** Trims whitespace and normalizes input

Content Security Policy (CSP):

```
contentSecurityPolicy: {
  directives: {
    defaultSrc: ["'self'"],
    scriptSrc: ["'self'"],
    styleSrc: ["'self'", "'unsafe-inline'"],
    imgSrc: ["'self'", "data:", "https:"],
    connectSrc: ["'self'", "https://api.awibi-medtech.com"],
    frameSrc: ["'none'"],
    objectSrc: ["'none'"]
  }
}
```

CSP prevents various attacks by controlling resource loading: - **XSS Prevention:** Restricts script execution to trusted sources - **Clickjacking Protection:** Prevents framing by other sites - **Data Exfiltration:** Controls where data can be sent

Security Monitoring:

```
const securityLogger = (req, res, next) => {
  const suspiciousPatterns = [
    /(\<|\%3C)script/i,
    /javascript:/i,
    /\.\./,
    /\/etc\/passwd/i,
    /cmd\.exe/i
  ];

  const isSuspicious = suspiciousPatterns.some(pattern =>
    pattern.test(requestString)
  );

  if (isSuspicious) {
    console.warn('🚨 Suspicious request detected:', {
      ip: req.ip,
      userAgent: req.get('User-Agent'),
      url: req.url
    });
  }
};
```

Security monitoring detects and logs suspicious activity: - **Attack Pattern Detection:** Identifies common attack signatures - **Performance Monitoring:** Logs slow requests that might indicate attacks - **Audit Trail:** Creates logs for security analysis

CORS (Cross-Origin Resource Sharing) Deep Dive

CORS is a critical security feature that controls which domains can access the API. Let's examine the CORS configuration in detail:

```
// Enhanced CORS configuration
const corsOptions = {
  origin: function (origin, callback) {
    // Allow requests with no origin (mobile apps, Postman, etc.)
    if (!origin) {
      return callback(null, true);
    }

    const allowedOrigins = process.env.ALLOWED_ORIGINS?.split(',') || [];

    // Development environment - allow localhost with any port
    if (process.env.NODE_ENV === 'development') {
      const localhostPattern = /^https?:\/\/localhost(:\d+)?$/;
      if (localhostPattern.test(origin)) {
        return callback(null, true);
      }
    }

    // Check against allowed origins
    if (allowedOrigins.includes(origin)) {
      callback(null, true);
    } else {
      console.warn('🚨 CORS violation attempt:', {
        origin,
        allowedOrigins,
        timestamp: new Date().toISOString()
      });
      callback(new Error('Not allowed by CORS'));
    }
  },
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS', 'PATCH'],
  allowedHeaders: [
    'Content-Type',
    'Authorization',
    'X-Requested-With',
    'Accept',
    'Origin',
    'Cache-Control',
    'X-File-Name'
  ],
  exposedHeaders: [
    'X-Total-Count',
    'X-Page-Count',
    'X-Current-Page'
  ],
  optionsSuccessStatus: 200,
  maxAge: 86400 // 24 hours
};
```

Understanding CORS Configuration:

Dynamic Origin Validation: The origin function provides flexible origin validation: - **Development Mode:** Allows any localhost origin for development convenience - **Production Mode:** Strictly validates against environment-configured origins - **No Origin Requests:** Allows requests without origin (mobile apps, server-to-server)

Credentials Support:

```
credentials: true
```

Enables sending cookies and authorization headers with cross-origin requests, essential for authentication.

Preflight Optimization:

```
maxAge: 86400 // 24 hours
```

Caches preflight responses for 24 hours, reducing unnecessary OPTIONS requests and improving performance.

Password Security Best Practices

Password security is implemented through multiple layers:

```
// Password validation rules
const passwordValidation = [
  body('password')
    .isLength({ min: 8, max: 128 })
    .withMessage('Password must be between 8 and 128 characters'),
  body('password')
    .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$`!%*?&])[A-Za-z\d@`$!%*?
&]/)
    .withMessage('Password must contain at least one lowercase letter, one
uppercase letter, one number, and one special character'),
  body('password')
    .not()
    .matches(/(.)\1{2,}/)
    .withMessage('Password cannot contain more than 2 consecutive identical
characters'),
  body('password')
    .custom((value, { req }) => {
      const commonPasswords = [
        'password', '123456', 'password123', 'admin', 'qwerty',
        'letmein', 'welcome', 'monkey', '1234567890'
      ];

      if (commonPasswords.includes(value.toLowerCase())) {
        throw new Error('Password is too common');
      }

      // Check if password contains user information
      if (req.body.email &&
value.toLowerCase().includes(req.body.email.split('@')[0].toLowerCase())) {
        throw new Error('Password cannot contain email username');
      }

      if (req.body.firstName &&
value.toLowerCase().includes(req.body.firstName.toLowerCase())) {
        throw new Error('Password cannot contain first name');
      }

      return true;
    })
];

// Password hashing with bcrypt
const hashPassword = async (password) => {
  const saltRounds = 12; // Higher than default for better security
  const salt = await bcrypt.genSalt(saltRounds);
  return await bcrypt.hash(password, salt);
};

// Secure password comparison
const comparePassword = async (plainPassword, hashedPassword) => {
  return await bcrypt.compare(plainPassword, hashedPassword);
};
```

Understanding Password Security:

Complexity Requirements: - **Minimum Length:** 8 characters (industry standard) - **Character Diversity:** Requires uppercase, lowercase, numbers, and special characters

- **Pattern Prevention:** Prevents repetitive patterns - **Common Password Prevention:** Blocks commonly used passwords

Personal Information Protection: The validation prevents passwords that contain: - User's email username - User's first or last name - Other personal information

Secure Hashing:

```
const saltRounds = 12;
```

Uses bcrypt with 12 salt rounds, which is computationally expensive enough to resist brute force attacks while remaining practical for legitimate authentication.

API Security Headers

Security headers provide additional protection against various attacks:

```
const securityHeaders = helmet({
  // Prevents XSS attacks
  contentSecurityPolicy: {
    directives: {
      defaultSrc: ["'self'"],
      scriptSrc: ["'self'"],
      styleSrc: ["'self'", "'unsafe-inline'"],
      imgSrc: ["'self'", "data:", "https:"],
      connectSrc: ["'self'"],
      frameSrc: ["'none'"],
      objectSrc: ["'none'"]
    }
  },

  // Prevents clickjacking
  frameguard: { action: 'deny' },

  // Enforces HTTPS
  hsts: {
    maxAge: 31536000, // 1 year
    includeSubDomains: true,
    preload: true
  },

  // Prevents MIME type sniffing
  noSniff: true,

  // Hides server information
  hidePoweredBy: true,

  // Controls referrer information
  referrerPolicy: { policy: "no-referrer" },

  // Enables XSS filtering
  xssFilter: true
});
```

Understanding Security Headers:

Content Security Policy (CSP): Controls which resources can be loaded, preventing XSS attacks by restricting script execution to trusted sources.

HTTP Strict Transport Security (HSTS): Forces browsers to use HTTPS, preventing man-in-the-middle attacks and protocol downgrade attacks.

X-Frame-Options: Prevents the page from being embedded in frames, protecting against clickjacking attacks.

X-Content-Type-Options: Prevents browsers from MIME-sniffing responses, reducing the risk of drive-by downloads and other attacks.

Error Handling Security

Secure error handling prevents information disclosure:

```
const secureErrorHandler = (err, req, res, next) => {
  // Log full error details for debugging
  console.error('Error details:', {
    message: err.message,
    stack: err.stack,
    url: req.url,
    method: req.method,
    ip: req.ip,
    userAgent: req.get('User-Agent'),
    timestamp: new Date().toISOString()
  });

  // Determine error type and appropriate response
  let statusCode = 500;
  let message = 'Internal server error';

  if (err.name === 'ValidationError') {
    statusCode = 400;
    message = 'Validation failed';
  } else if (err.name === 'CastError') {
    statusCode = 400;
    message = 'Invalid ID format';
  } else if (err.code === 11000) {
    statusCode = 400;
    message = 'Duplicate field value';
  } else if (err.name === 'JsonWebTokenError') {
    statusCode = 401;
    message = 'Invalid token';
  } else if (err.name === 'TokenExpiredError') {
    statusCode = 401;
    message = 'Token expired';
  }

  // Never expose sensitive information in production
  const response = {
    success: false,
    message: message
  };

  // Include additional details only in development
  if (process.env.NODE_ENV === 'development') {
    response.error = err.message;
    response.stack = err.stack;
  }

  res.status(statusCode).json(response);
};
```

Understanding Secure Error Handling:

Information Disclosure Prevention: Production error responses only include generic messages, preventing attackers from gaining insights into the system architecture or database structure.

Comprehensive Logging: Full error details are logged for debugging while keeping sensitive information out of client responses.

Error Classification: Different error types receive appropriate HTTP status codes and messages, providing useful feedback without exposing system internals.

What Security Implementation Achieves:

1. **Attack Prevention:** Protects against common web vulnerabilities (XSS, CSRF, injection attacks)
2. **Data Protection:** Ensures sensitive information remains secure
3. **Access Control:** Prevents unauthorized access to resources
4. **Audit Trail:** Provides logging for security monitoring and compliance
5. **Performance Protection:** Rate limiting prevents abuse and DoS attacks
6. **Compliance:** Meets security standards and best practices
7. **User Trust:** Builds confidence through robust security measures

The comprehensive security implementation creates multiple layers of protection, ensuring that even if one security measure fails, others remain in place to protect the application and its users.

📚 References

[1] Node.js Foundation. (2023). *About Node.js*. Retrieved from https://nodejs.org/en/about/ [2] Node.js Foundation. (2023). *The Node.js Event Loop*. Retrieved from https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/ [3] Auth0. (2023). *JSON Web Tokens Introduction*. Retrieved from https://jwt.io/introduction

---

Next: Final Review and Delivery of Documentation