**Secure Programming**

**Secure SDLC Technical Report**

Software Support Ticket System (Secure Ticketing Platform)

**Members** : Lavaraju ADARAPU  
 Akash PRAKASHAN  
 Bibin VELLANGOTT CHERUVADI  
 Nikitha ANTONY Ramakrishna PALLAPU

**Repository**: <https://github.com/bibinvc/Software-Support-Ticket-System>  
Date: 12 Jan 2026

# Table of Contents

Note: In Word, right-click here and choose 'Update Field' to generate the table of contents.

1. Executive Summary

2. System Description and Scope

3. Functional Requirements Mapping

4. Architecture and Data Flow

5. Database Design

6. Secure SDLC Implementation

7. Threat Modeling (STRIDE)

8. Security Controls Implementation

9. Testing and Validation

10. Deployment and Configuration

Appendix A. Key Code Excerpts

Appendix B. Repository Structure

# 1. Executive Summary

This document presents the design and implementation of a secure Software Support Ticket System developed as a full-stack web application. The platform supports three roles (Client, Support Agent, Administrator) and implements a controlled ticket lifecycle (Open → In Progress → Resolved → Closed). Security is integrated throughout the Secure SDLC, including STRIDE threat modeling, MFA (TOTP), RBAC enforcement, input validation, rate limiting, audit logging, encryption of sensitive data, and secure deployment practices.

# 2. System Description and Scope

The platform enables clients to browse and ticket tickets offered by agents. Support Agents manage their ticket listings and update ticket status. Administrators manage users, categories, moderation, and audit logs.

## 2.1 User Roles

Client: Registers and logs in, browses tickets, places tickets, tracks ticket status, and submits ratings/reviews after completion.

Support Agent: Registers and logs in, creates and manages ticket listings, confirms and fulfills tickets, and interacts with clients.

Administrator: Manages users and categories, moderates tickets, monitors platform statistics, and reviews audit logs.

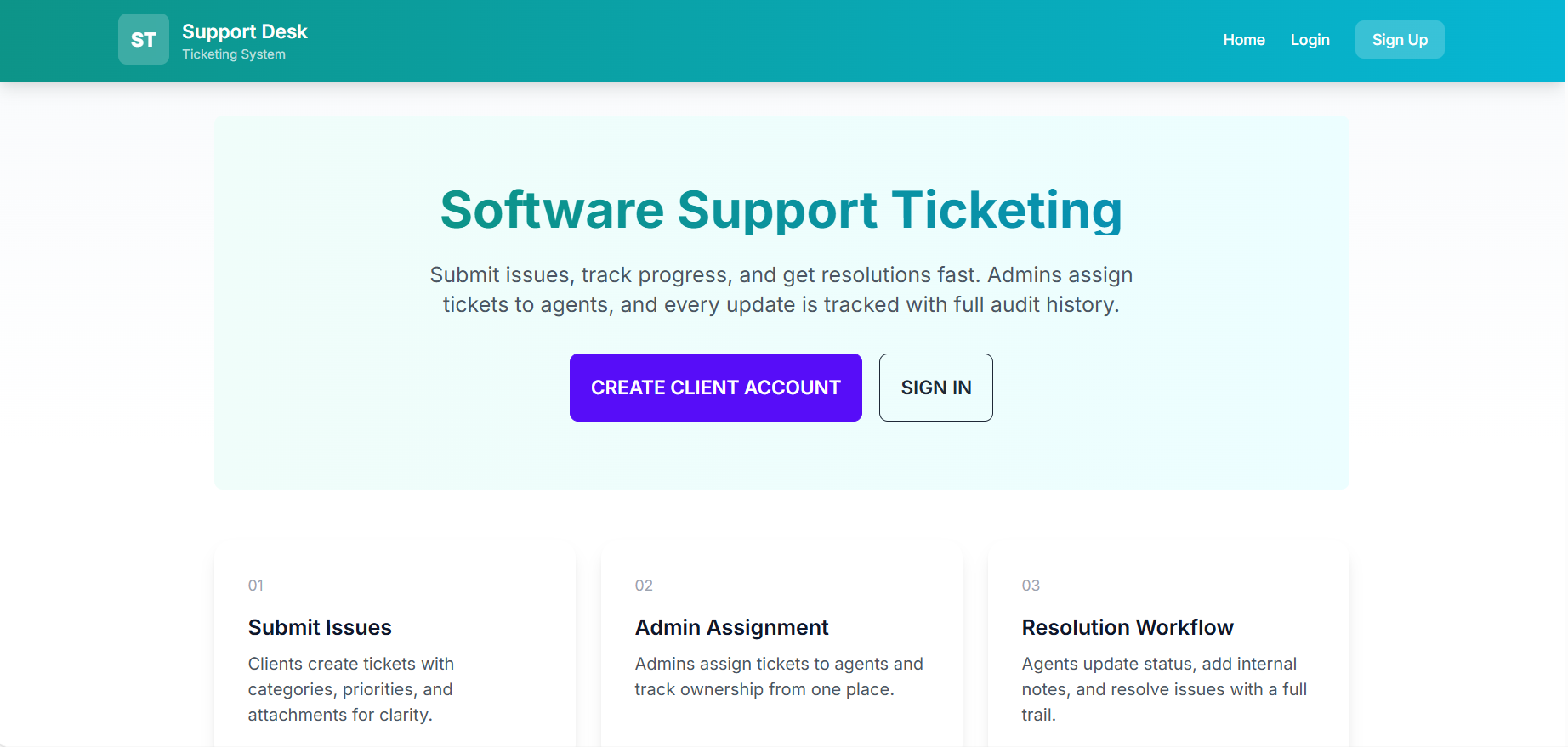
## 2.2 In-Scope Features

* Authentication and authorization with multi-role RBAC
* Multi-Factor Authentication (MFA) using TOTP
* Ticket listing (search and filtering)
* Ticket lifecycle workflow (Open -> In Progress -> Resolved -> Closed; cancellations with restrictions)
* Administrator dashboard (user management, category management, platform stats, audit logs)
* Security controls across the SDLC (validation, encryption, logging, rate limiting, TLS guidance)

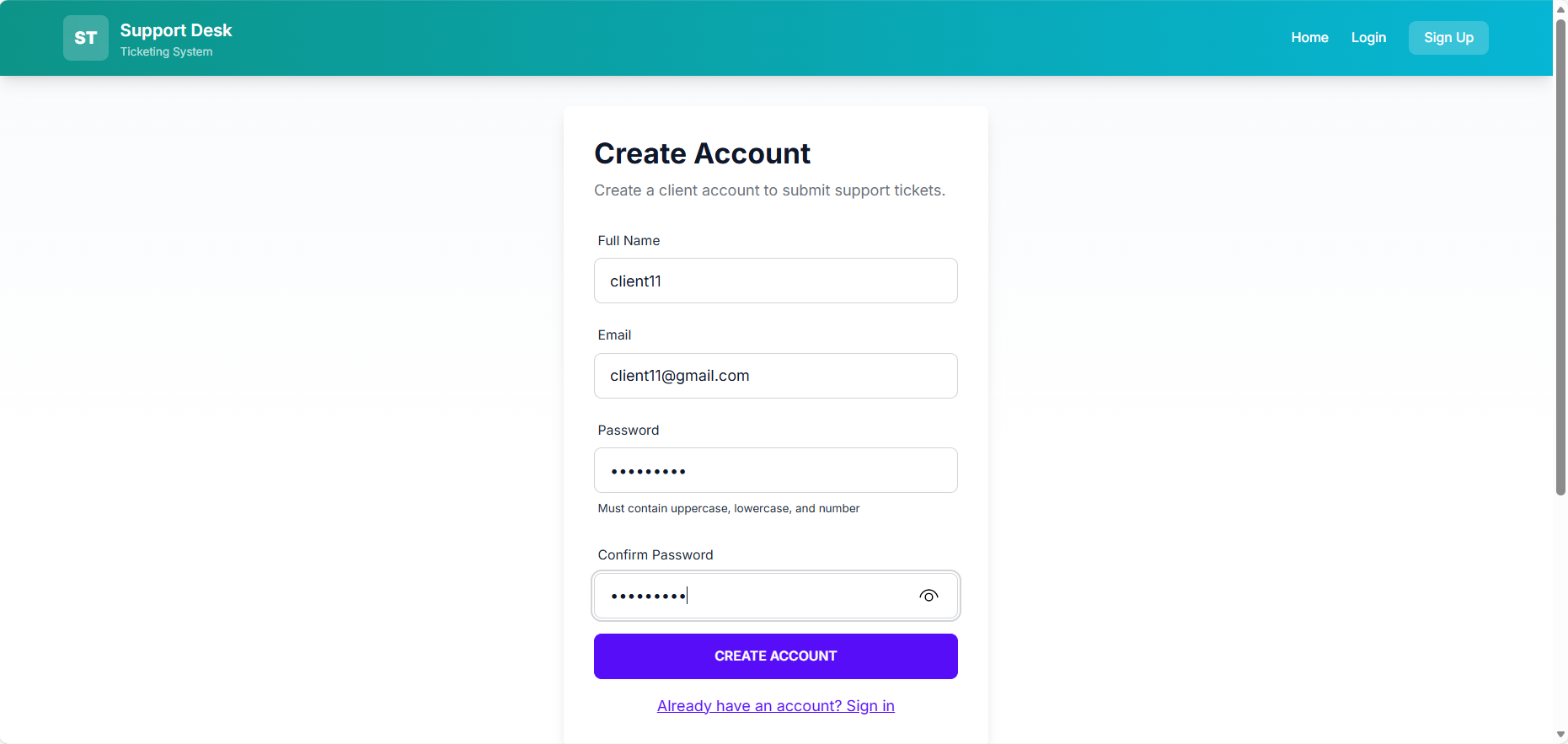
# 3. Functional Requirements Mapping

This section maps required platform functionality to implemented modules (backend routes/middleware and frontend pages). Use this as evidence during the presentation/video.

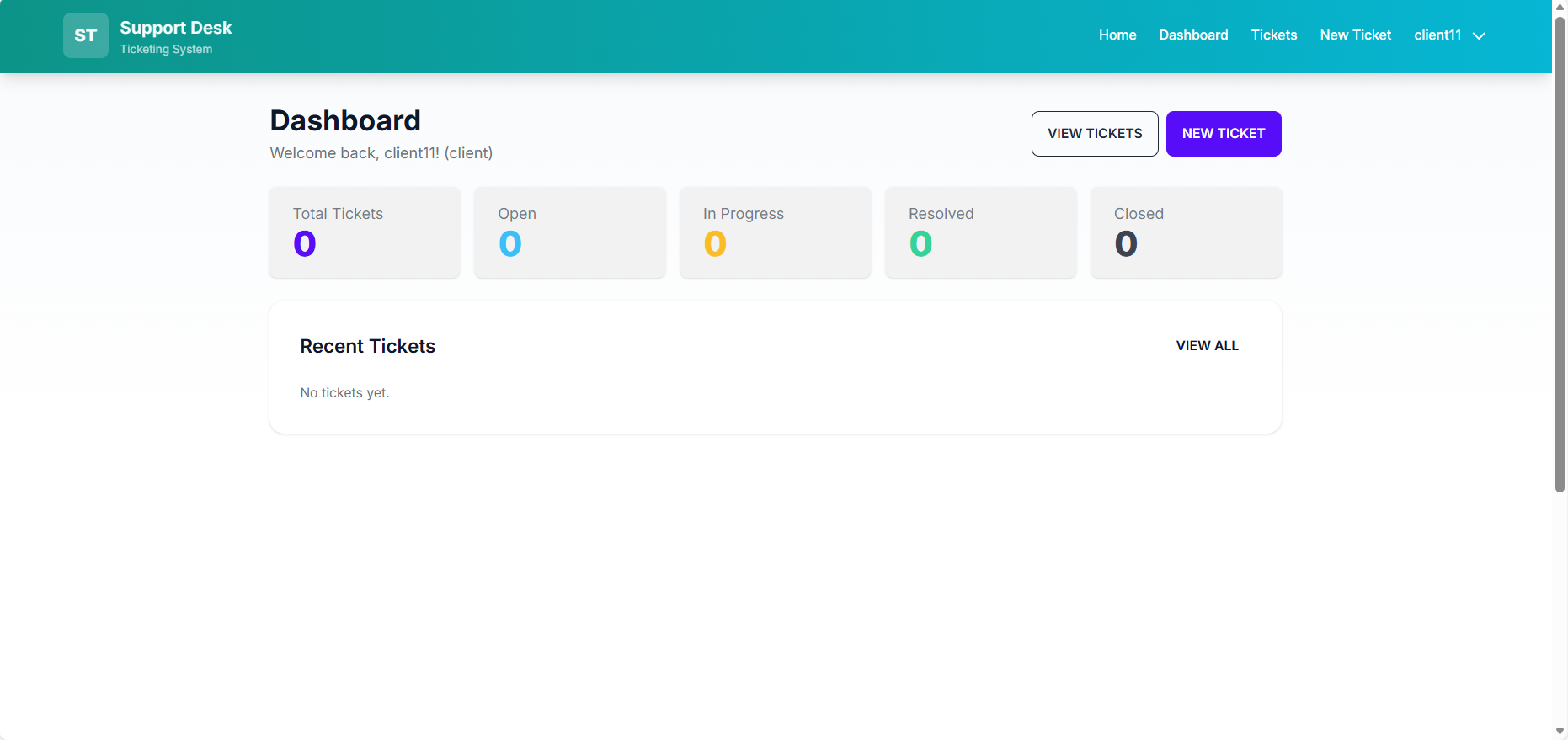
**Home / landing page and navigation**



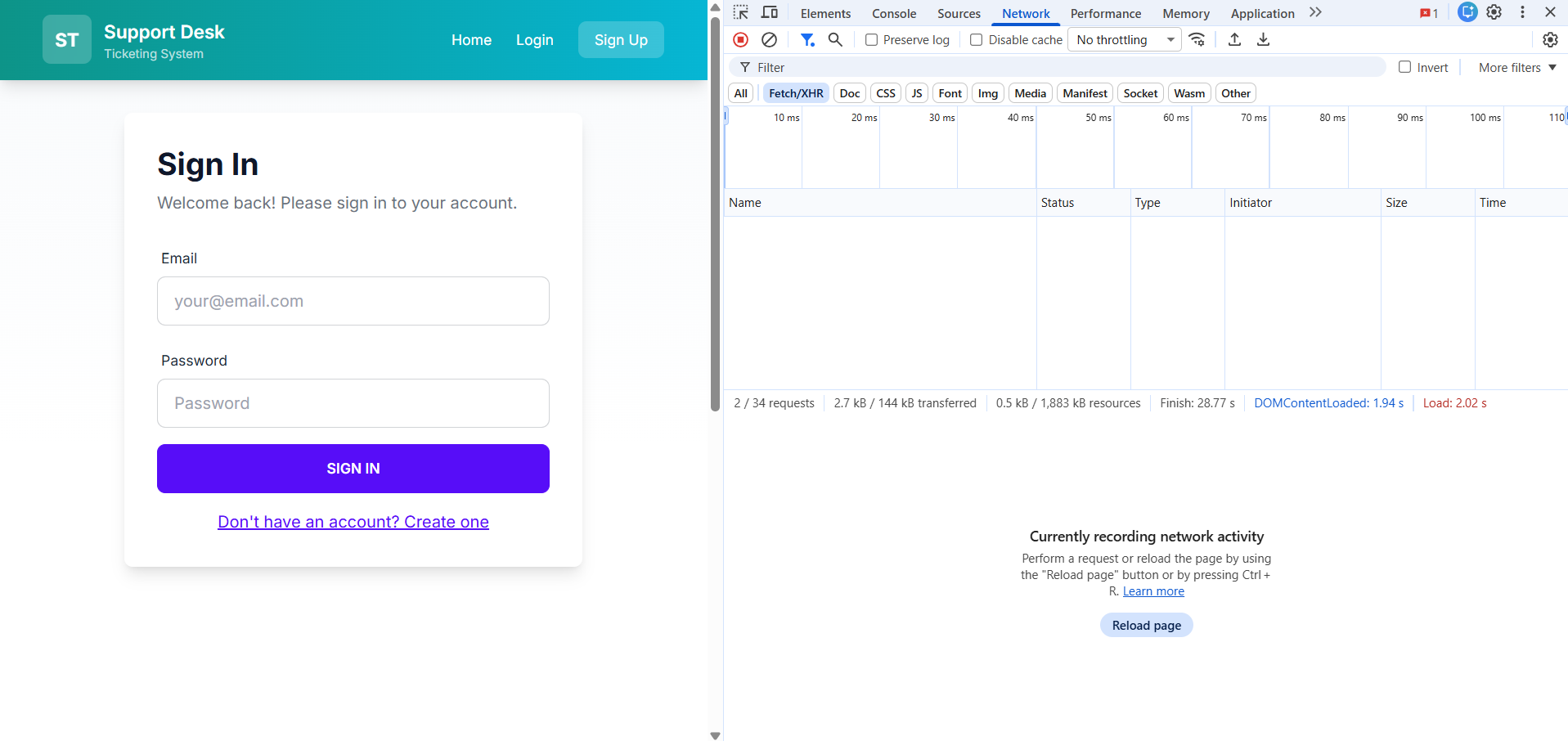
**Registration with role selection (client)**

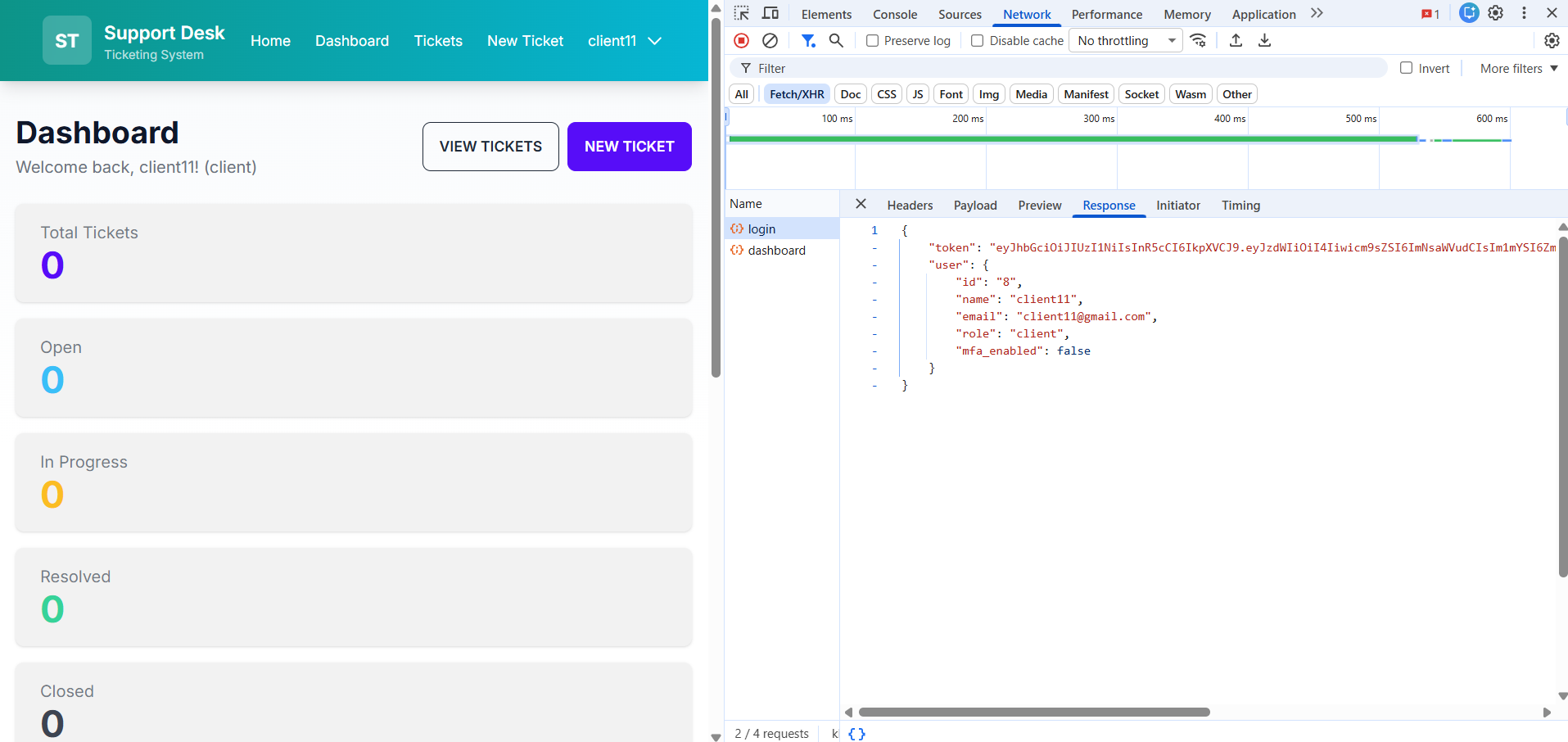


**Client Dashboard**

****

**Login + JWT received (network tab) + redirect to dashboard**





**Token response**

{

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI4Iiwicm9sZSI6ImNsaWVudCIsIm1mYSI6ZmFsc2UsImlhdCI6MTc2ODM1MjQ3OCwiZXhwIjoxNzY4OTU3Mjc4fQ.HAAe5Wp-Qcnc541fhkeA8DfanbW-0j4JwpDCsXLbgvs",

"user": {

"id": "8",

"name": "client11",

"email": "client11@gmail.com",

"role": "client",

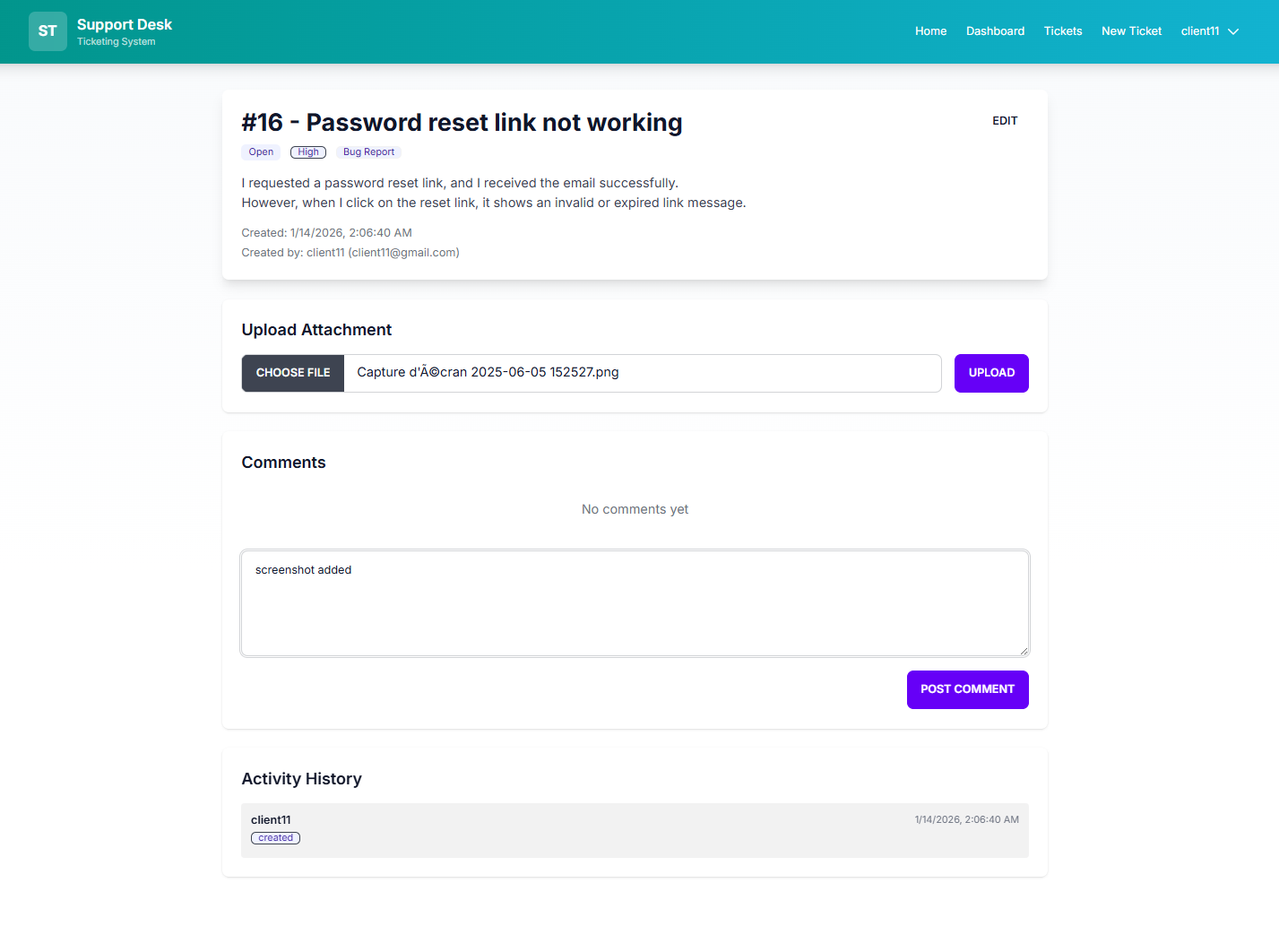
"mfa\_enabled": false

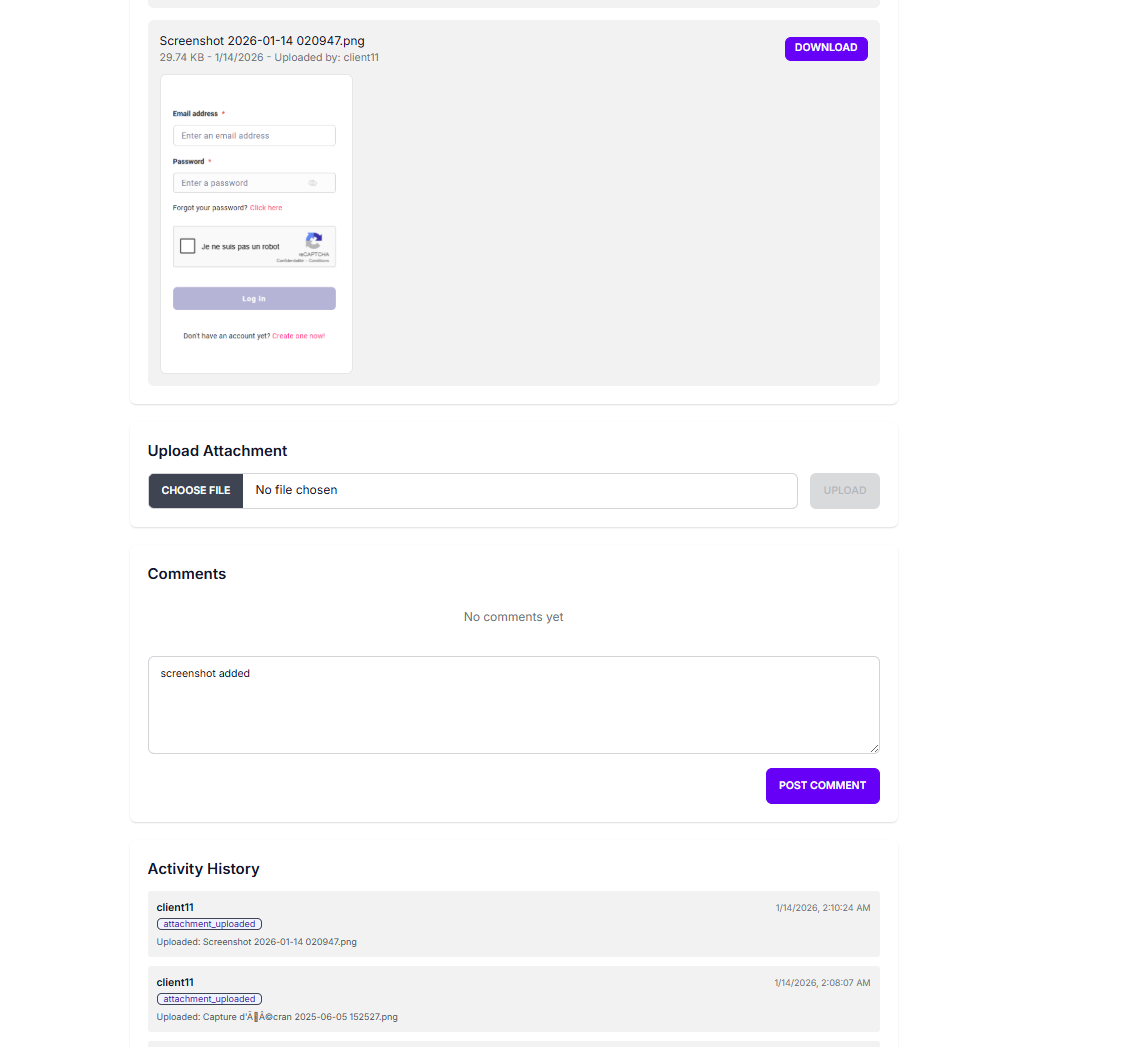
}

}

**Ticket listing + search/filter results  
Create new ticket**

****



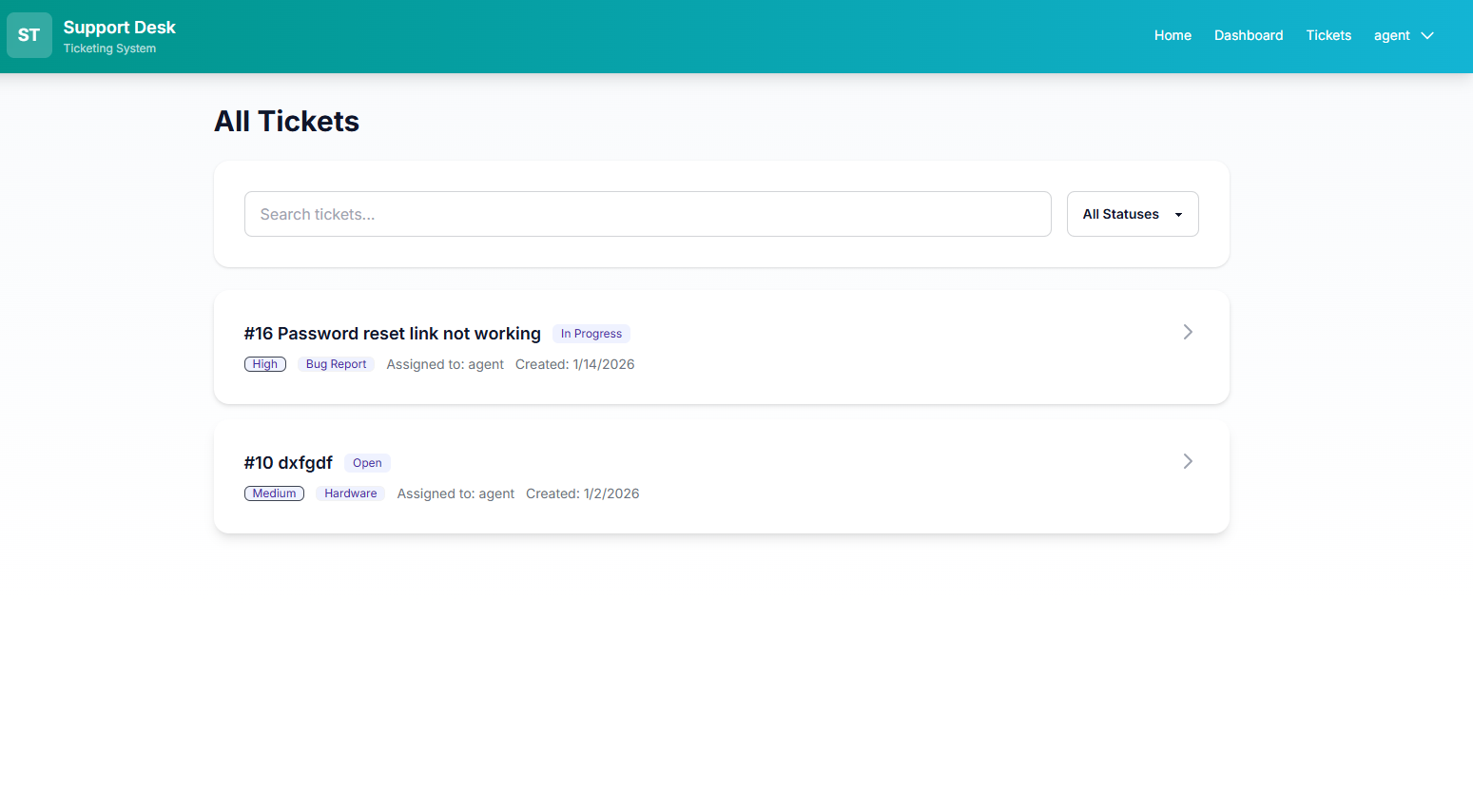


**Created ticket**

Client places an ticket (open)



**Support Agent confirms ticket and updates to in\_progress/completed**



**Admin dashboard: users list and audit logs**

# 

# 4. Architecture and Data Flow

High-level architecture uses a React SPA frontend, an Express.js REST API backend, and PostgreSQL for persistence.

## 4.1 System Architecture

[Browser / React Frontend]

|

| HTTPS (TLS)

v

[Express.js API Server]

- Routes (Auth, Tickets, Tickets, Admin)

- Middleware (JWT auth, RBAC, validation, rate limiting, security headers)

|

v

[PostgreSQL Database]

- Users, Tickets, Tickets, Categories, Sessions, Audit Logs, Attachments

## 4.2 Data Flow (Request Lifecycle)

User action -> Frontend validation -> API request -> Rate limiting -> Authentication (JWT) -> Authorization (RBAC) -> Input validation/sanitization -> Business logic -> ORM query (Sequelize) -> Audit log -> JSON response -> Frontend rendering

# 5. Database Design

The database is PostgreSQL and accessed via Sequelize ORM. Tables include users, tickets, tickets, categories, sessions, attachments, and audit logs.

## 5.1 Key Entities

**users:** Stores account data (role, password hash, MFA secret (encrypted), status).

**tickets**: Support Agent-created listings with pricing, availability, and status.

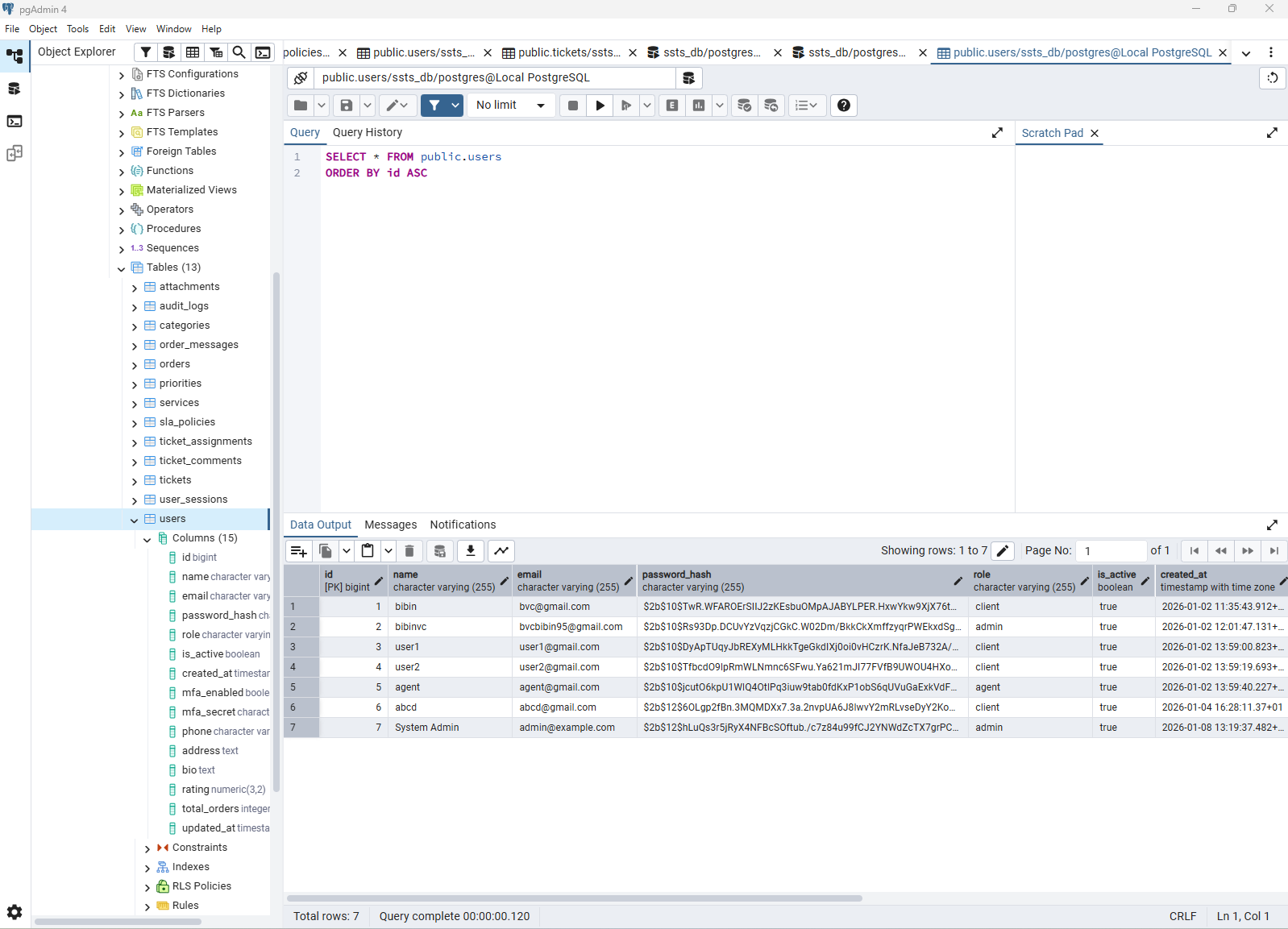
**tickets**: Client tickets referencing ticket and agent; status transitions enforced.

**categories**: Admin-managed ticket categories.

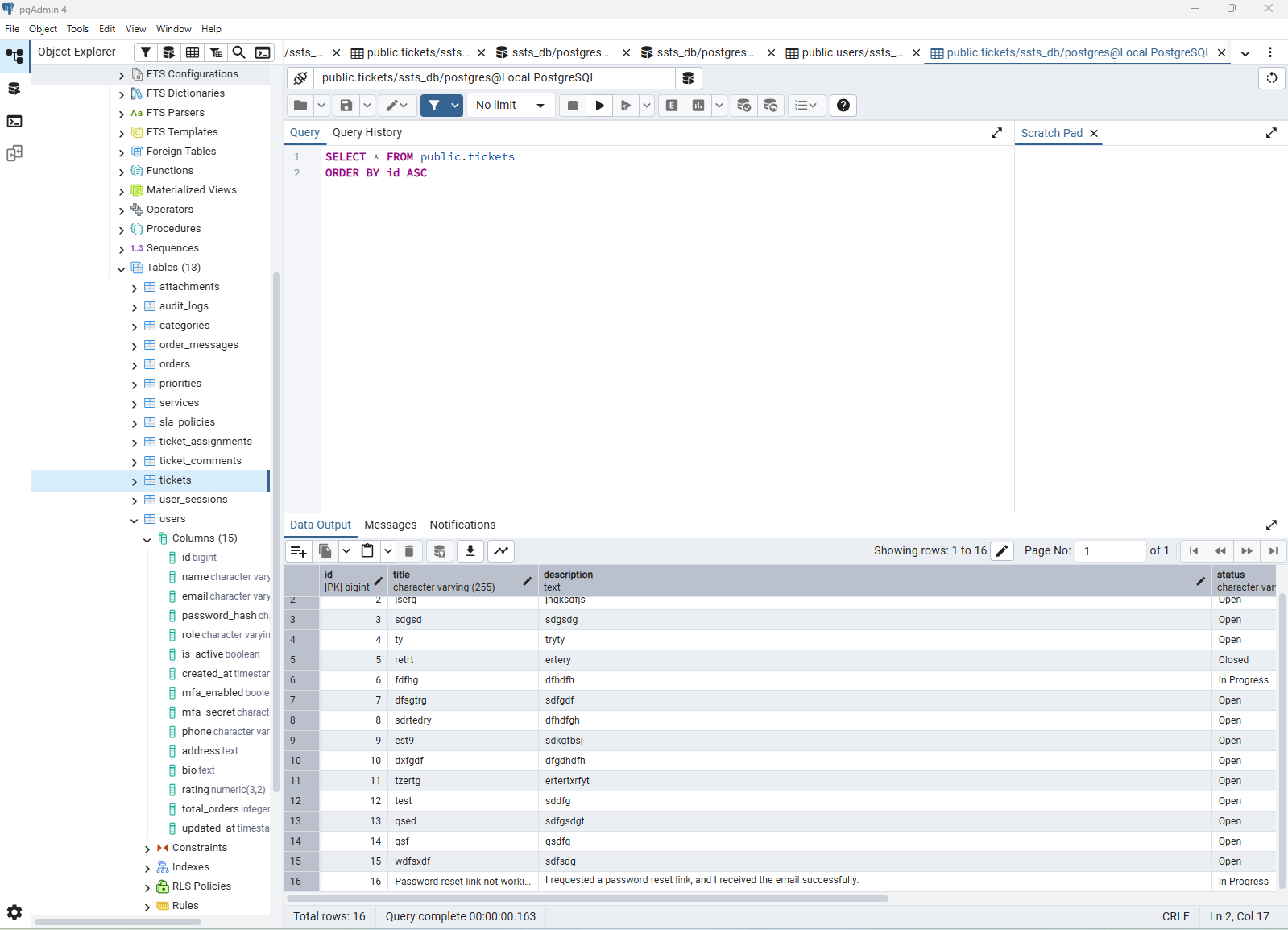
**user\_sessions:** Tracks active JWT sessions via token hash and revocation/expiry.

**audit\_logs:** Append-only audit events for critical actions.

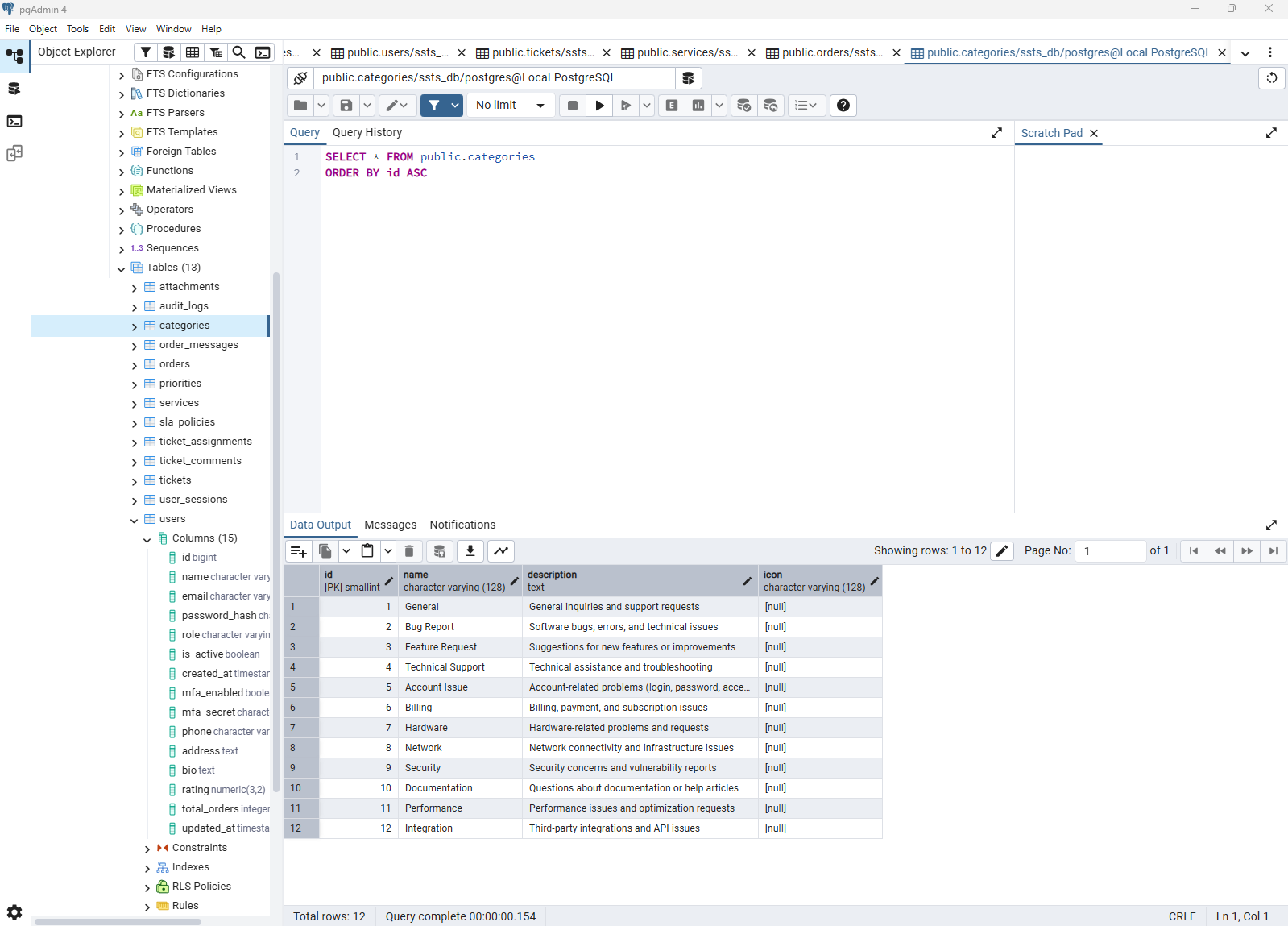
**Database schema/ER diagram**



**Tickets DB**

****

**Services DB**

****

**Migrations or schema.sql**

# 

# 6. Secure SDLC Implementation

Security was applied across the SDLC: requirements, architecture/design, threat modeling, secure coding, testing, and deployment guidance.

## 6.1 Requirements

Security requirements include MFA, secure password handling, RBAC, secure session management, input validation, protections against injections, HTTPS/TLS, encryption and secret management, and secure backend API practices.

## 6.2 Design

Design includes an API-first architecture, centralized middleware for security controls, and STRIDE threat modeling.

## 6.3 Implementation

Implementation uses well-established libraries for authentication, validation, encryption, and security headers; database access uses ORM parameterization.

## 6.4 Verification

Verification includes manual security test scenarios (SQLi, XSS, RBAC bypass, rate limiting, MFA) and functional end-to-end tests.

## 6.5 Deployment

Deployment guidance includes environment variables for secrets, TLS setup (self-signed acceptable for demo), and hosting recommendations (frontend + backend + database).

# 7. Threat Modeling (STRIDE)

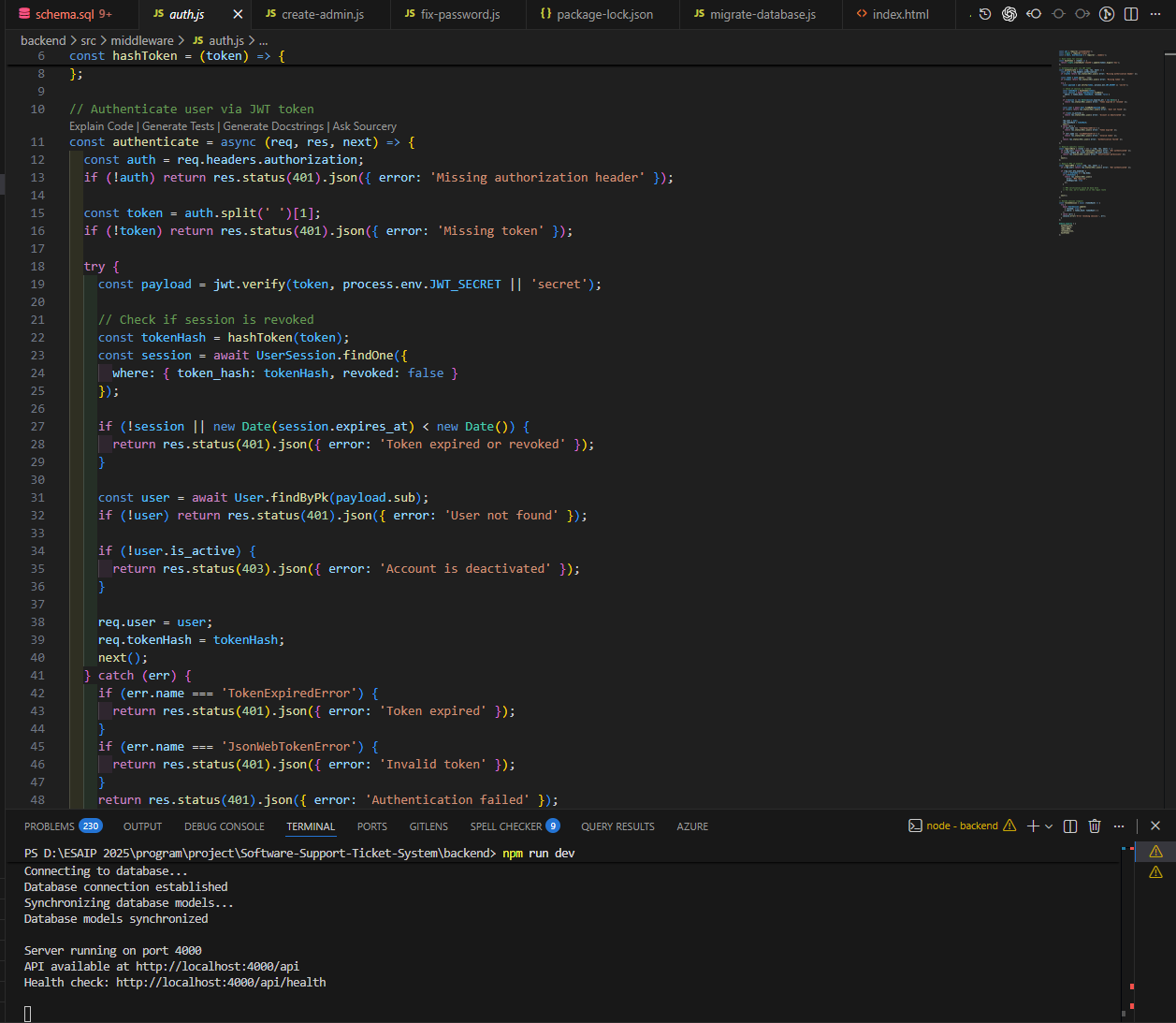
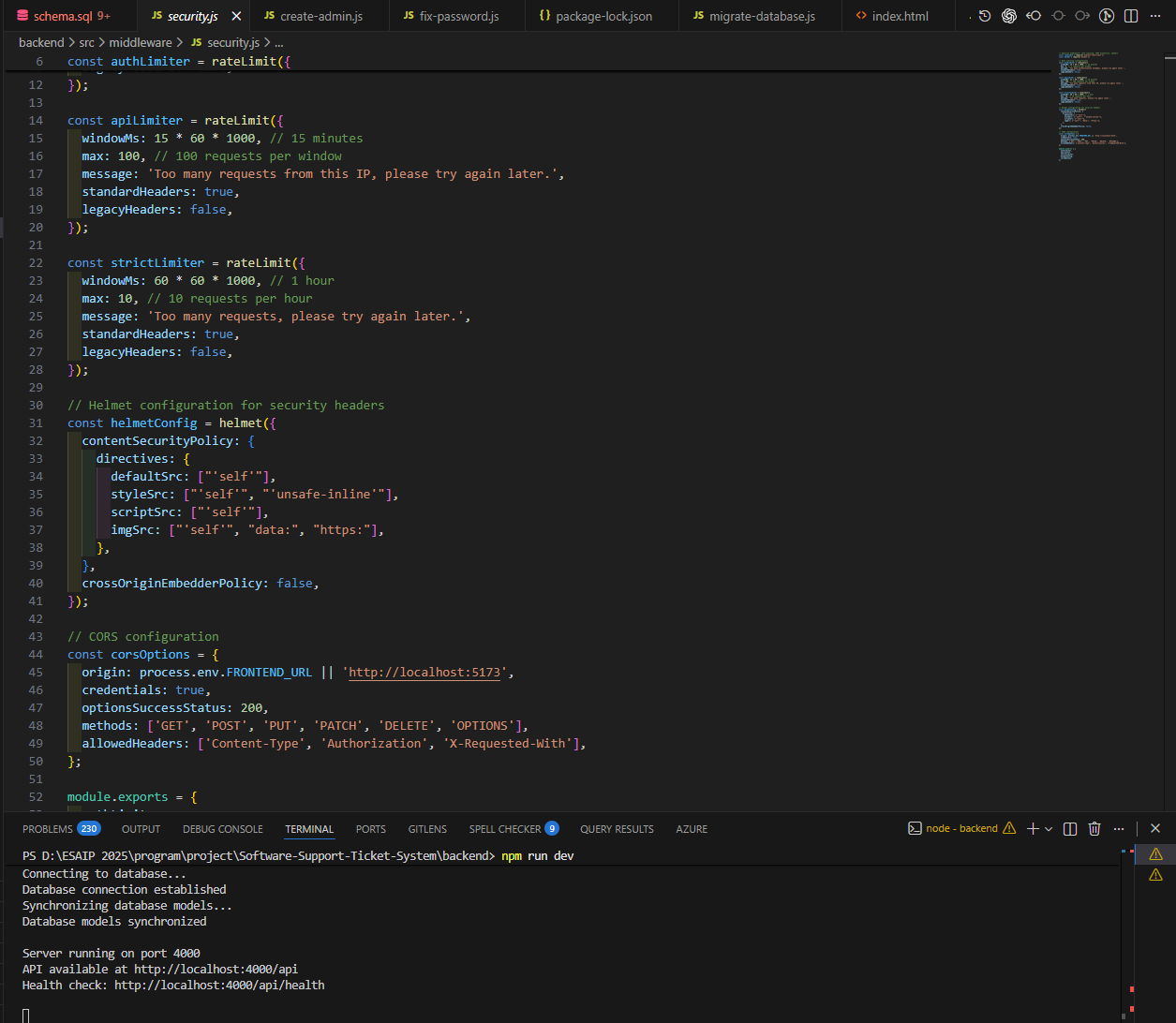
One representative threat per STRIDE category is identified, with practical mitigations and implementation evidence.

## 7.1 Spoofing

**Threat:** Session token theft (JWT) to impersonate a user.

**Mitigations:**

* HTTPS/TLS to prevent MITM
* Short-lived tokens + session tracking and revocation
* CSP and input sanitization to reduce XSS risk
* Audit logging for authentication events

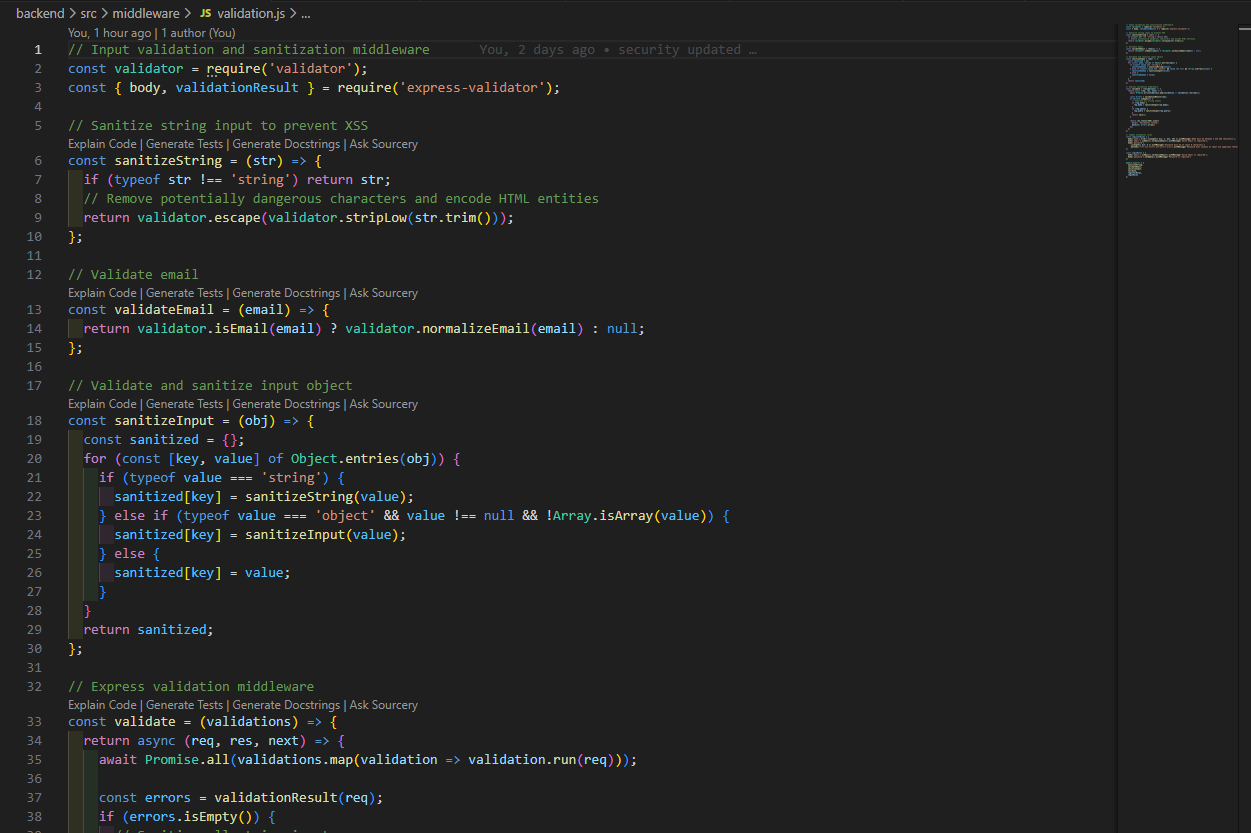
 

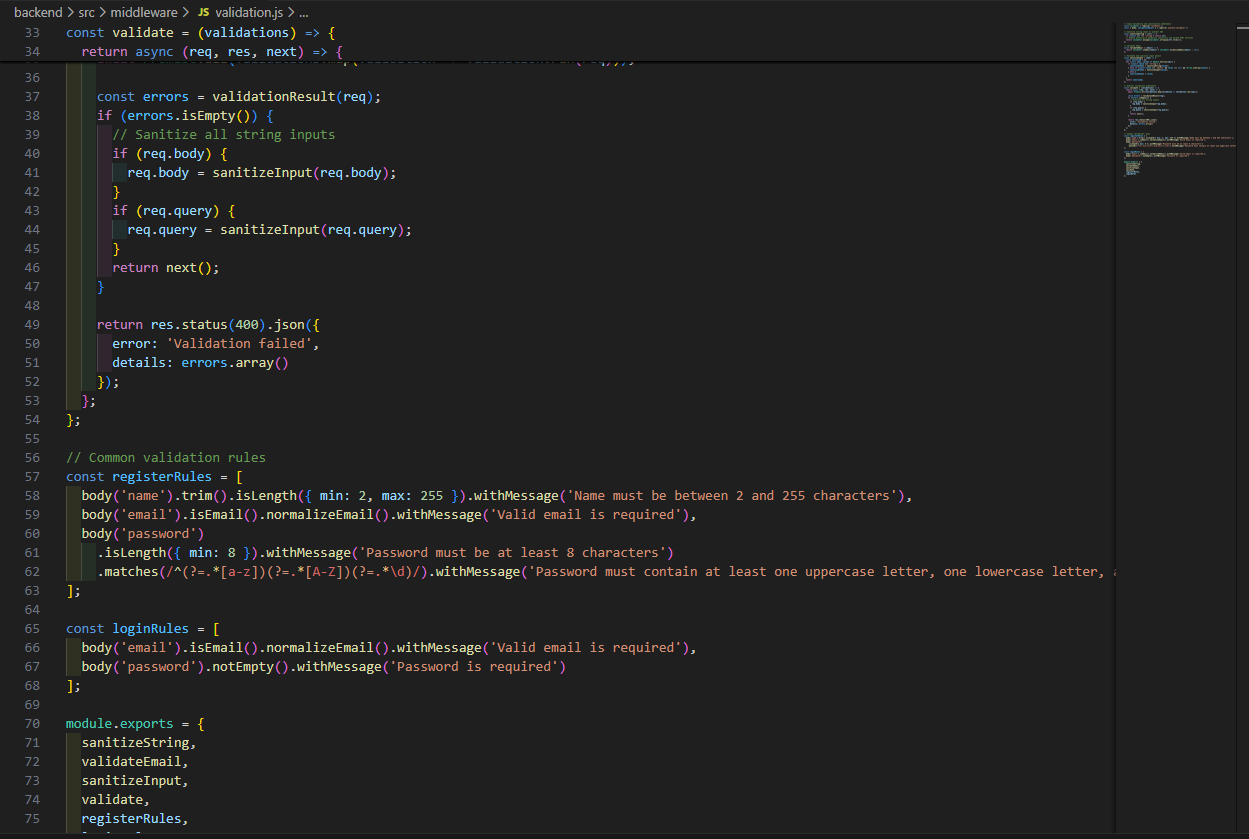
## 7.2 Tampering

**Threat:** SQL injection via malicious input in search or identifiers.

**Mitigations:**

* Sequelize ORM with parameterized queries
* Strict input validation (types, lengths, patterns)
* Sanitization before processing





## 

## 7.3 Repudiation

**Threat:** Users deny actions performed; attackers attempt to alter logs.

**Mitigations:**

* Append-only audit logs for critical actions
* Admin-only access to view logs
* Include metadata (user, IP, timestamp, payload)



## 

## 

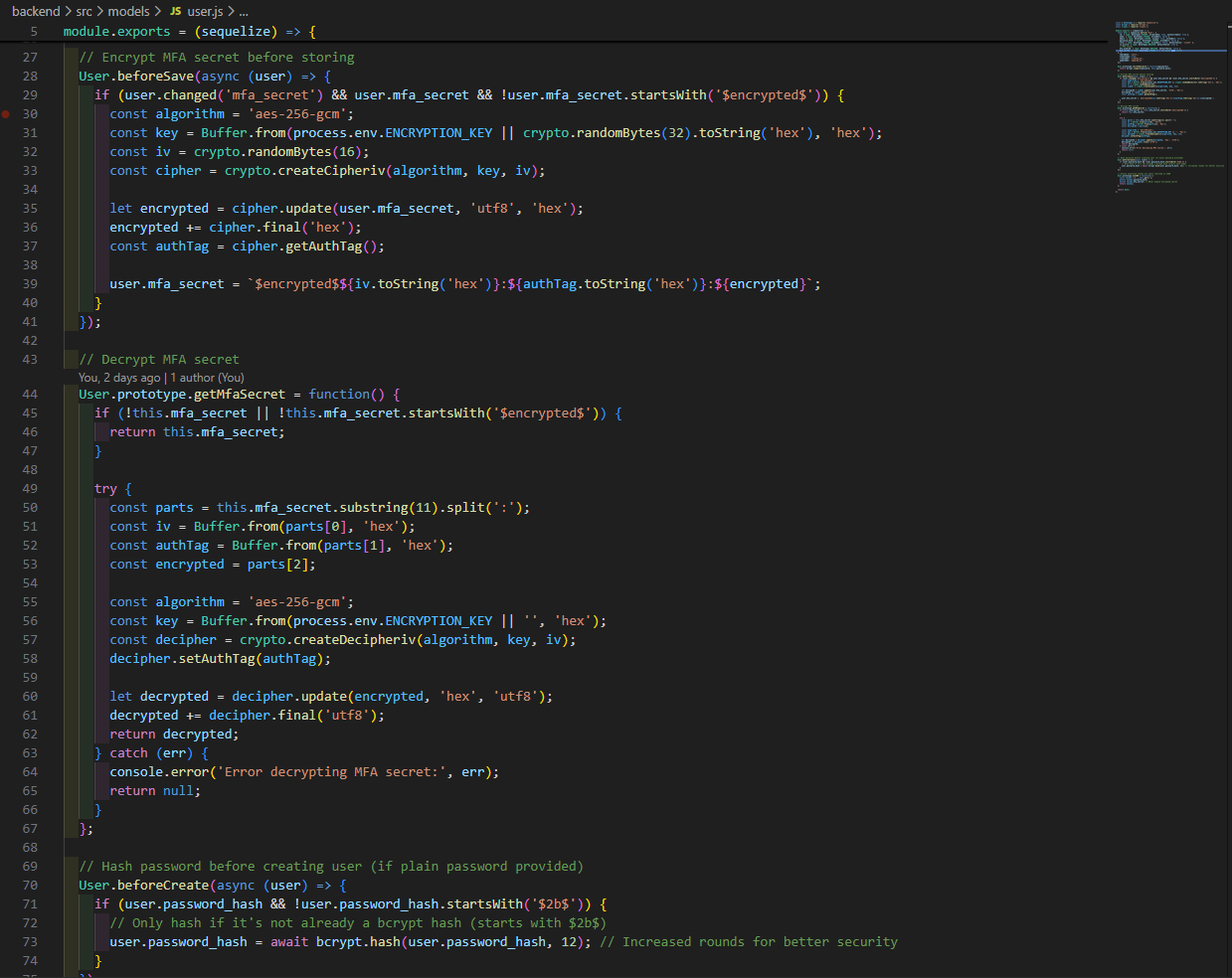
## 7.4 Information Disclosure

**Threat:** Sensitive fields exposed via API responses or logs (password hashes, MFA secret).

**Mitigations:**

* Exclude sensitive fields from API responses
* Encrypt MFA secrets at rest (AES-256-GCM)
* Environment variables for keys
* Generic error handling

**Implementation evidence:**



## 

## 7.5 Denial of Ticket

**Threat:** Brute force and request flooding to exhaust resources.

**Mitigations:**

* Rate limiting (auth stricter than general API)
* Request size limits and file upload constraints

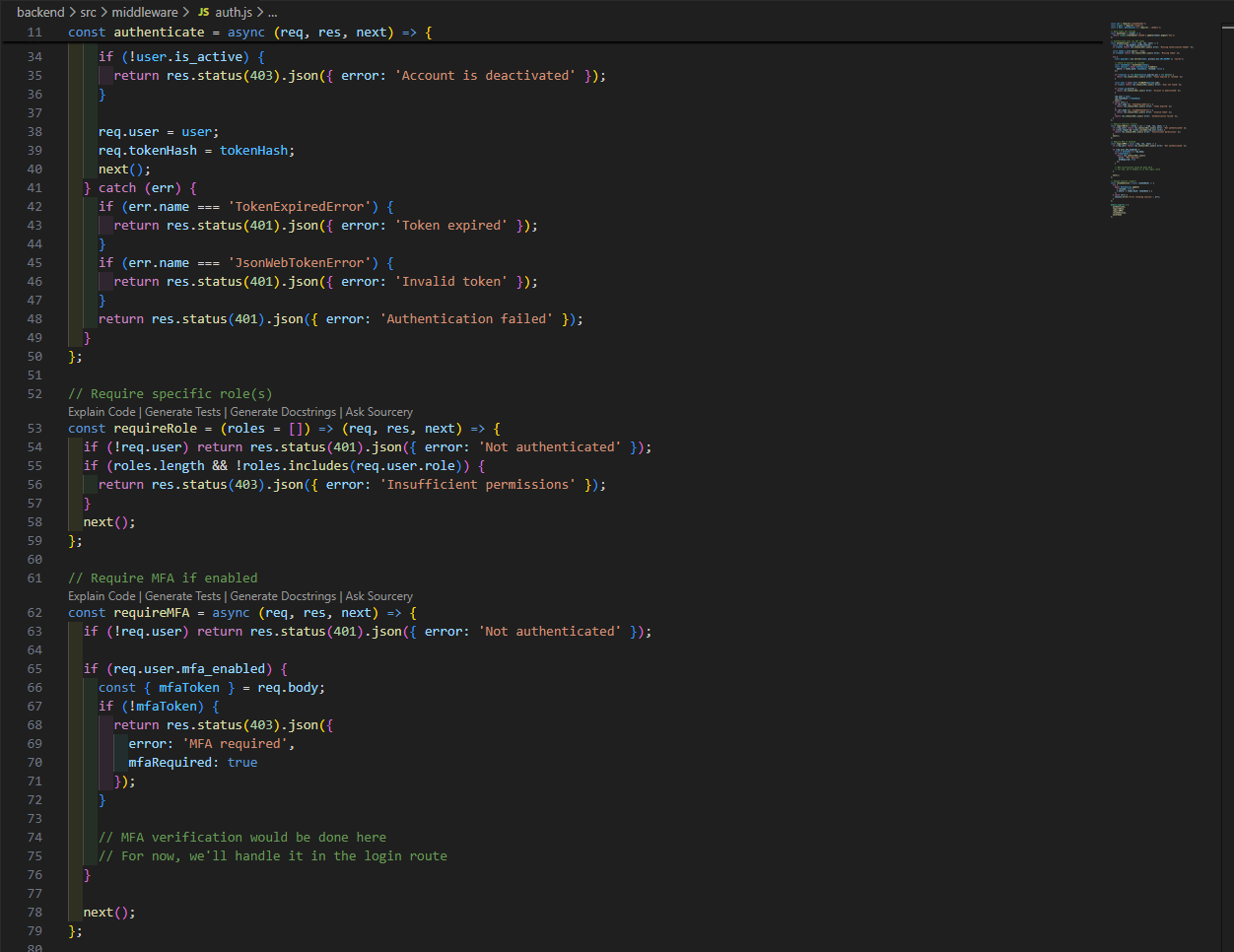
## 

## 7.6 Elevation of Privilege

**Threat:** RBAC bypass (direct API call or manipulated token).

**Mitigations:**

* Server-side RBAC enforcement on every protected route
* JWT signature verification and role checks
* Ownership checks on resources



# 8. Security Controls Implementation

## 8.1 Authentication Security

Authentication uses secure password hashing (bcrypt), JWT tokens, session tracking with revocation, and optional MFA (TOTP).

## 8.2 Authorization (RBAC)

RBAC is enforced server-side using middleware guards. Admin-only endpoints require the admin role; agent/client actions include ownership checks.

## 8.3 Input Validation and Injection Protections

All API inputs are validated and sanitized. ORM-based database access provides parameterized queries by default, reducing SQL injection risk.

## 8.4 Secure Communication (HTTPS/TLS)

TLS guidance is provided for deployment; self-signed certificates are acceptable for demo environments.

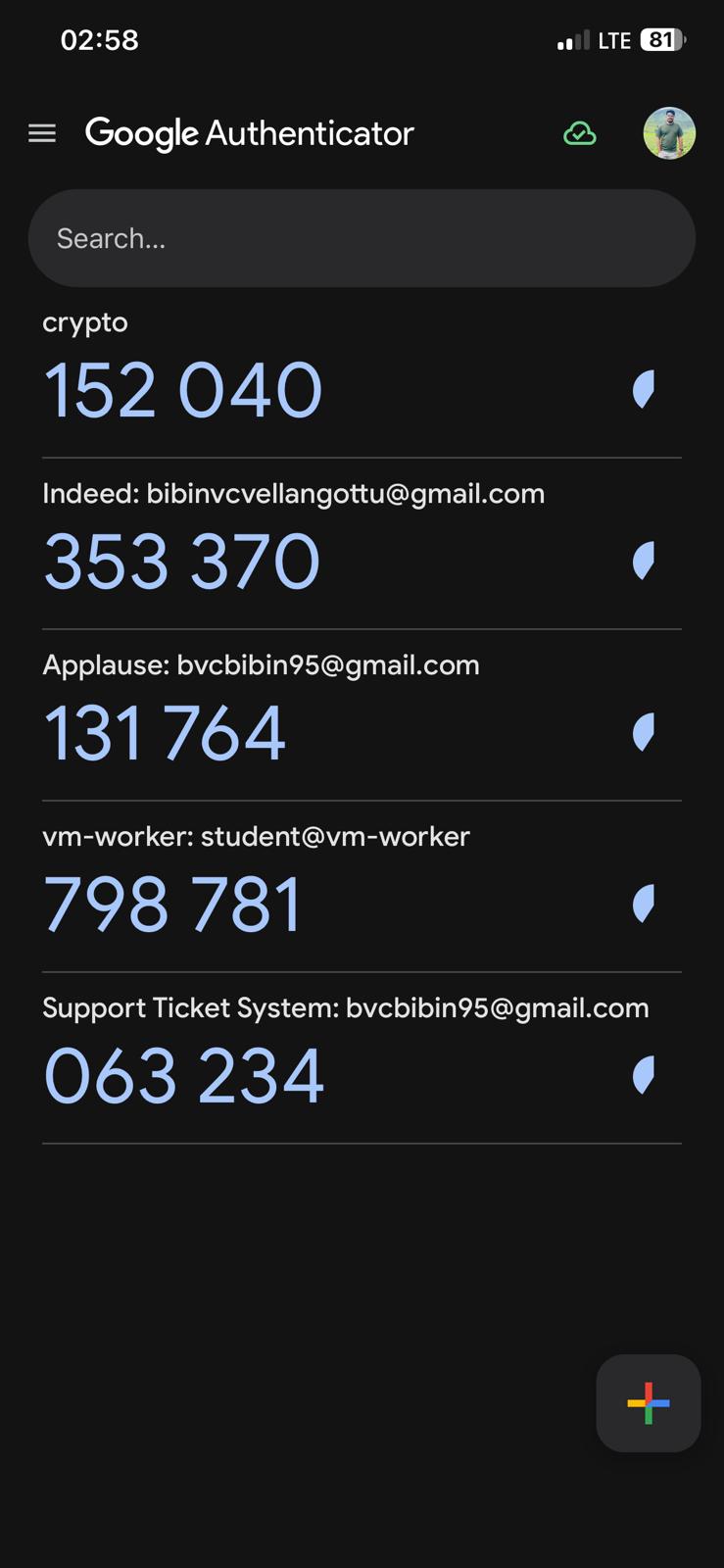
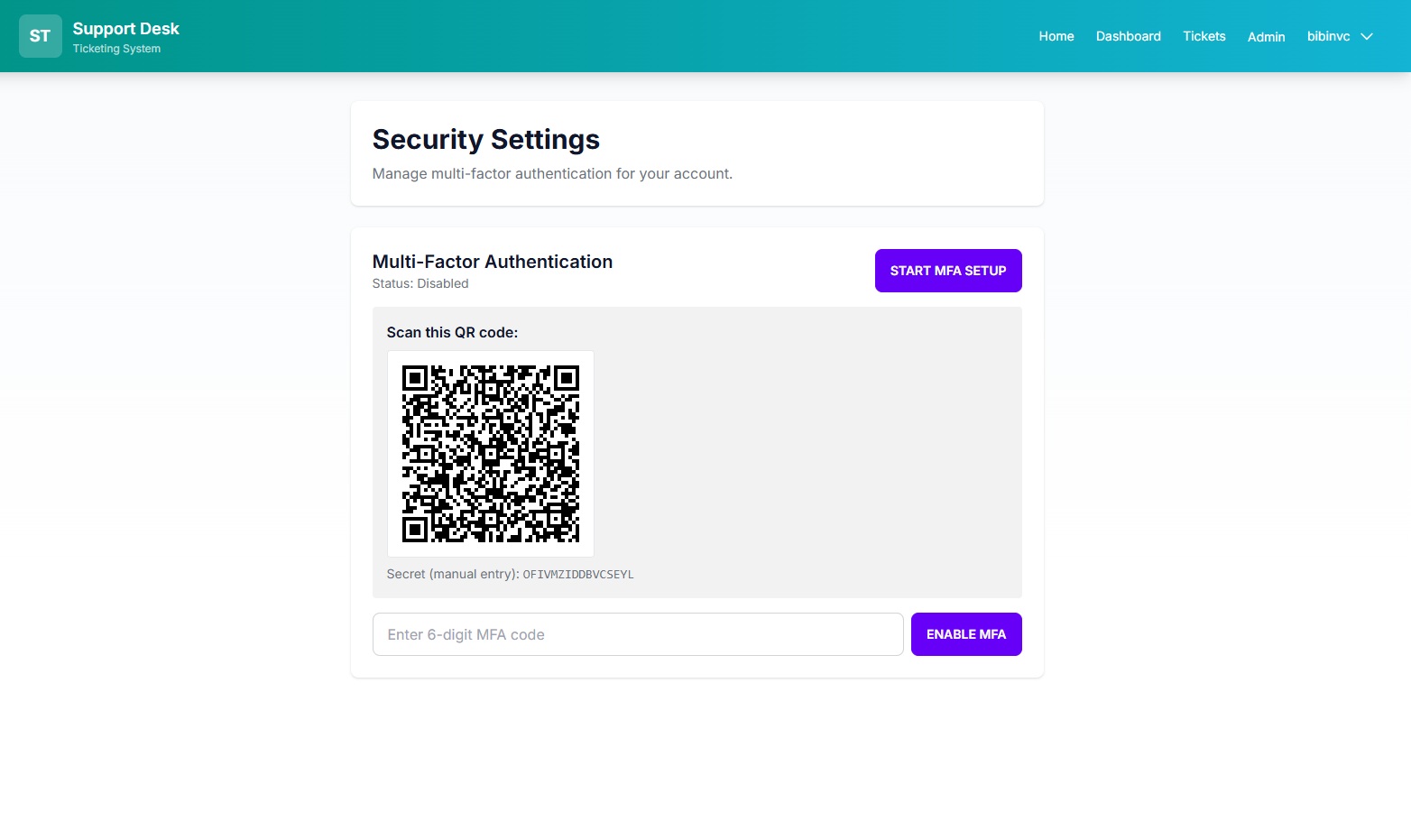
## 8.5 Encryption and Secret Management

Secrets are stored in environment variables. MFA secrets are encrypted at rest using authenticated encryption (AES-256-GCM).

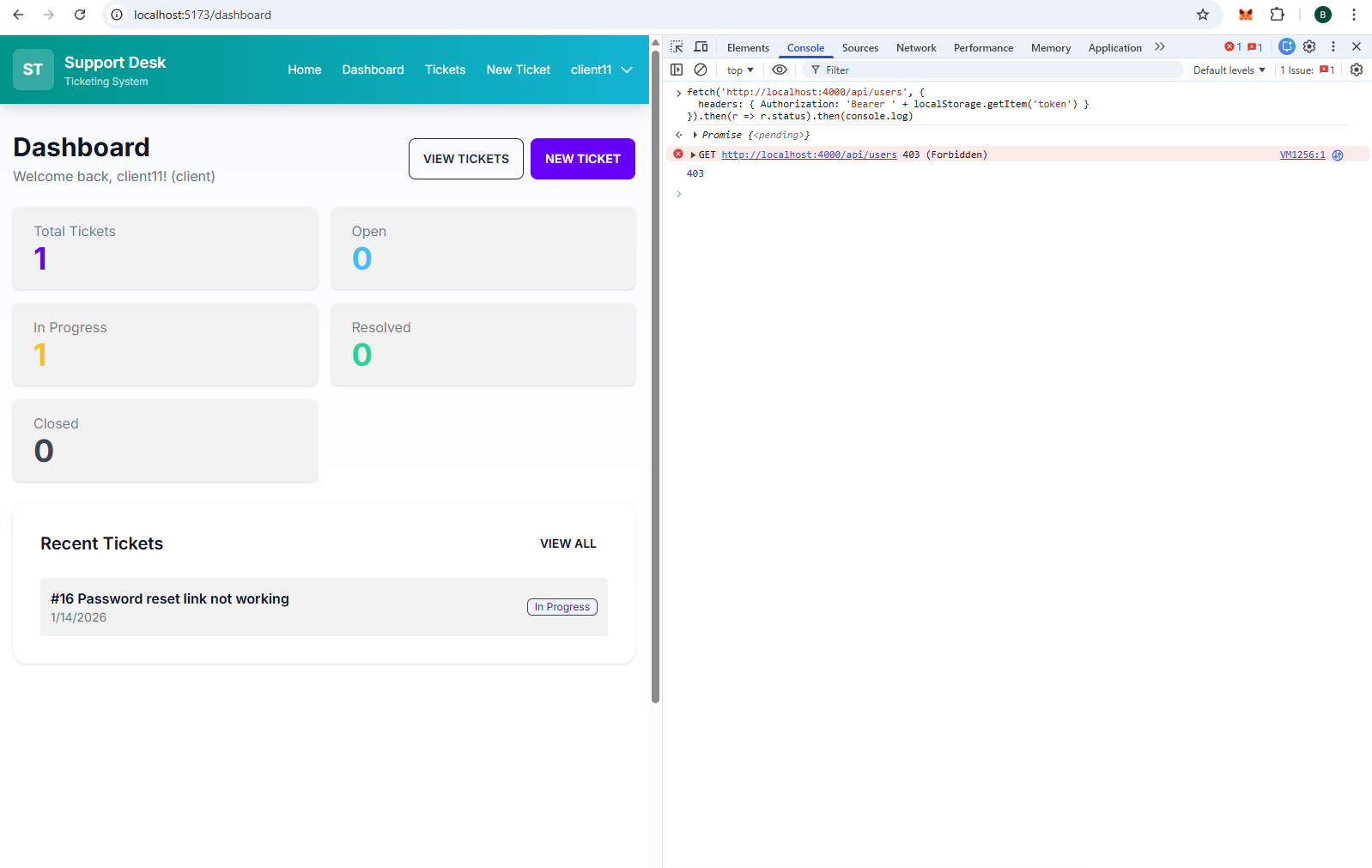
## 8.6 Secure Backend Practices

Security headers (Helmet), CORS restrictions, rate limiting, file upload constraints, and audit logs harden the API against common web threats.

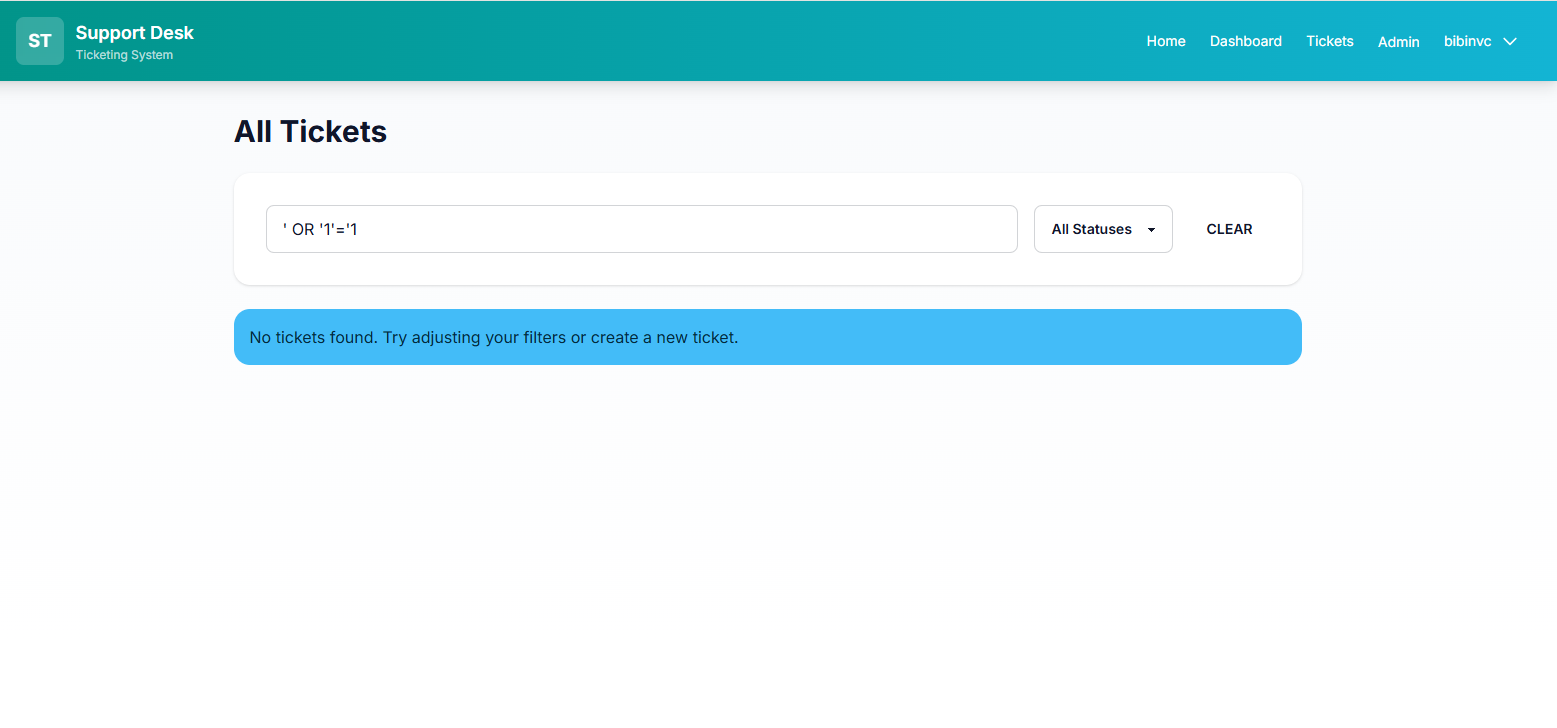
**MFA setup (QR code / secret shown) and successful MFA login**



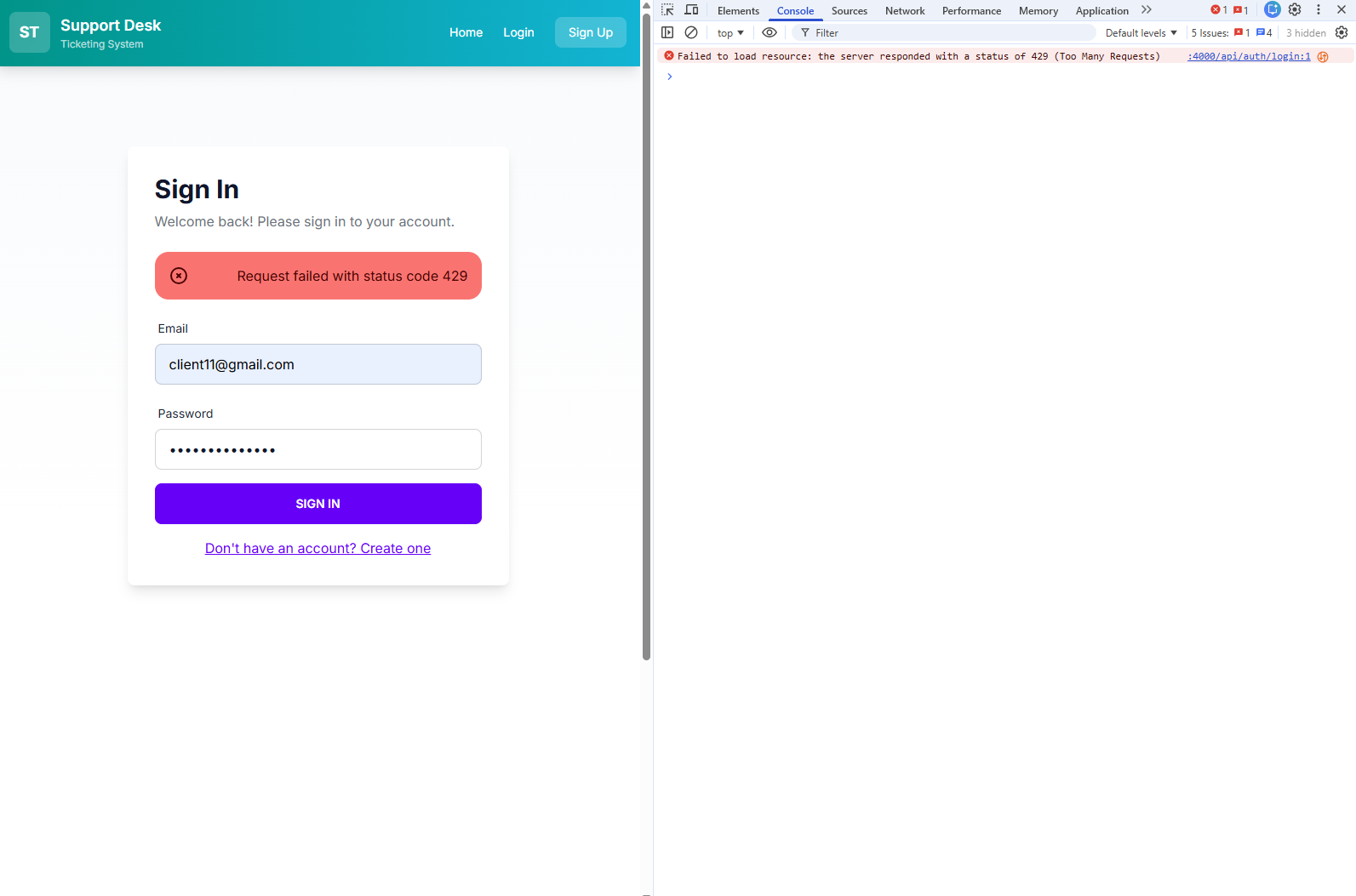
**RBAC test: access admin route as client**



**SQL injection attempt in search field returns safe results**



**Rate limiting test: repeated login attempts return 429**



**Audit log entry after critical action**

**HTTPS/TLS enabled**

**backend**

# 

**Frontend**

# 9. Testing and Validation

## 9.1 Functional Test Scenarios

1. Register a client and agent; verify role-based dashboards.
2. Support Agent creates a ticket; client can see it in the public listing.
3. Client places an ticket; agent can confirm and complete it; client can track status.
4. Admin can list users, deactivate a user, manage categories, and review audit logs.

## 9.2 Security Test Scenarios

1. RBAC bypass attempt: call admin endpoints with a client token -> must be denied (403).
2. SQL injection attempt in query fields -> must not execute; no DB errors or sensitive leakage.
3. XSS attempt by injecting script tags into text fields -> must be sanitized/escaped.
4. Brute force login -> rate limiter returns 429 after threshold.
5. MFA enforcement -> when enabled, login must require valid TOTP code.

# 10. Deployment and Configuration

## 10.1 Prerequisites

* Node.js v16+
* PostgreSQL v12+
* npm or yarn

## 10.2 Environment Variables

|  |  |
| --- | --- |
| Variable | Purpose |
| DATABASE\_URL | PostgreSQL connection string |
| JWT\_SECRET | JWT signing secret (>= 32 chars recommended) |
| ENCRYPTION\_KEY | Hex key for MFA secret encryption |
| PORT | Backend listening port (default 4000) |
| FRONTEND\_URL | Allowed origin for CORS |
| UPLOAD\_DIR | Uploads directory path |
| NODE\_ENV | Environment (development/production) |

## 10.3 Local Run Steps (Summary)

**Commands**

# Backend

cd backend

cp .env.example .env

npm install

npm run setup-db

npm run dev

# Frontend

cd frontend

cp .env.example .env

npm install

npm run dev

**Backend running (terminal showing server started + port)**

**Frontend running**

**Database setup script success output**

# Appendix A. Key Code Excerpts

Include only the most important excerpts in the main document. For grading, these excerpts show the implementation of security controls.

## A.1 Authentication Middleware (JWT + session revocation + RBAC)

**backend/src/middleware/auth.js (excerpt)**

// backend/src/middleware/auth.js

const authenticate = async (req, res, next) => {

const auth = req.headers.authorization;

if (!auth) return res.status(401).json({ error: 'Missing authorization header' });

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

const payload = jwt.verify(token, process.env.JWT\_SECRET || 'secret');

const session = await UserSession.findOne({ where: { token\_hash: hashToken(token), revoked: false } });

if (!session) return res.status(401).json({ error: 'Token expired or revoked' });

req.user = await User.findByPk(payload.sub);

next();

};

const requireRole = (roles = []) => (req, res, next) => {

if (!req.user) return res.status(401).json({ error: 'Not authenticated' });

if (roles.length && !roles.includes(req.user.role)) {

return res.status(403).json({ error: 'Insufficient permissions' });

}

next();

};

## A.2 Registration Endpoint (role validation)

**backend/src/routes/auth.js (excerpt)**

// backend/src/routes/auth.js

router.post('/register', authLimiter, validate(registerRules), async (req, res) => {

const { name, email, password, role = 'client' } = req.body;

if (!['client', 'agent'].includes(role)) {

return res.status(400).json({ error: 'Invalid role. Must be client or agent' });

}

const user = await User.create({ name, email: email.trim().toLowerCase(), password\_hash: password, role });

return res.status(201).json({ id: user.id, email: user.email, role: user.role });

});

## A.3 Input Validation and Sanitization

**backend/src/middleware/validation.js (excerpt 1)**

const sanitizeString = (str) => {

if (typeof str !== 'string') return str;

return validator.escape(validator.stripLow(str.trim()));

};

**backend/src/middleware/validation.js (excerpt 2)**

const validate = (validations) => async (req, res, next) => {

await Promise.all(validations.map(v => v.run(req)));

if (validationResult(req).isEmpty()) {

if (req.body) req.body = sanitizeInput(req.body);

if (req.query) req.query = sanitizeInput(req.query);

return next();

}

return res.status(400).json({ error: 'Validation failed', details: errors.array() });

};

## A.4 Rate Limiting

**backend/src/middleware/security.js (excerpt)**

const authLimiter = rateLimit({ windowMs: 15 \* 60 \* 1000, max: 5 });

const apiLimiter = rateLimit({ windowMs: 15 \* 60 \* 1000, max: 100 });

## A.5 Ticket Creation and Status Transition Rules

backend/src/routes/tickets.js (excerpt - create ticket)

router.post('/', authenticate, requireRole(['client', 'admin']), apiLimiter, validate(ticketRules), async (req, res) => {

const ticket = await Ticket.findByPk(req.body.ticket\_id);

if (!ticket || ticket.status !== 'active') {

return res.status(400).json({ error: 'Ticket is not available' });

}

const ticket = await Ticket.create({

ticket\_id: req.body.ticket\_id,

client\_id: req.user.id,

agent\_id: ticket.agent\_id,

status: 'pending'

});

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

});

backend/src/routes/tickets.js (excerpt - update status)

router.patch('/:id/status', authenticate, apiLimiter, async (req, res) => {

if (status === 'confirmed' && currentStatus === 'pending' && !isSupport Agent) {

return res.status(403).json({ error: 'Only agent can confirm tickets' });

}

if (status === 'completed' && currentStatus === 'in\_progress' && !isSupport Agent) {

return res.status(403).json({ error: 'Only agent can complete tickets' });

}

...

});

## A.6 Admin User Listing (exclude sensitive fields)

**backend/src/routes/users.js (excerpt)**

router.get('/', authenticate, requireRole(['admin']), apiLimiter, async (req, res) => {

const users = await User.findAll({ attributes: { exclude: ['password\_hash', 'mfa\_secret'] } });

res.json(users);

});

## A.7 MFA (TOTP) – setup and verification (conceptual excerpt)

Files: backend/src/routes/auth.js, backend/src/models/user.js

// Example flow used in this project (see auth.js + user model):  
// 1) User requests MFA setup -> server generates TOTP secret (base32) + QR payload  
// 2) Server encrypts and stores the MFA secret (AES-256-GCM) in DB  
// 3) User confirms setup by submitting a 6-digit TOTP code  
// 4) On login, if MFA enabled -> require TOTP code before issuing final JWT/session

## A.8 Audit Logging Middleware (who did what, when)

Files: backend/src/middleware/audit.js, backend/src/routes/audit.js

// Minimal pattern implemented:  
// - Attach audit logger to req (req.audit = { action, targetType, targetId, meta })  
// - After successful request (or in route handlers) persist audit log:  
// AuditLog.create({ actorUserId: req.user.id, action, targetType, targetId, ip: req.ip, userAgent: req.headers['user-agent'] })

# Appendix B. Repository Structure

**Repository structure**

Software-Support-Ticket-System/

backend/

src/

middleware/

models/

routes/

frontend/

src/

pages/

components/

db/

migrations/