Security Documentation

Overview

This document describes the security mechanisms, patterns, and practices implemented in the POC (FormSuite) application. The application employs multiple layers of defense to protect user data, authentication sessions, and API endpoints.

Table of Contents

Authentication & Authorization

Authentication Middleware

The application implements JWT-based authentication through the `authenticate` middleware in server/src/middlewares/auth.middleware.ts:

- **Token Verification**: Access tokens are extracted from HTTP-only cookies and verified using `HS256` algorithm
- **User Validation**: Active user status is verified on each authenticated request by querying the User model with status `ACTIVE`
- **Error Handling**: Invalid or missing tokens return 401 Unauthorized responses

Key Implementation:

- Tokens are validated against `JWT_SECRET` environment variable
- User session is confirmed in the database, ensuring revoked users cannot access the API even with a valid token

Role-Based Access Control (RBAC)

Two authorization middlewares enforce permission hierarchies:

`authorize` Middleware:

- Validates that the authenticated user's role is included in a required roles list
- Returns 403 Forbidden for insufficient permissions

`authorizeModule` **Middleware**:

- Implements granular module-level permissions for `users` and `forms` modules
- **SUPERADMIN** role bypasses all module checks
- **ADMIN** role requires explicit module permissions via `modulePermissions` object containing:
  - `users: boolean` - Access to user management features
  - `forms: boolean` - Access to form management features
  - Admin can be allowed to access `both` the modules as well.
- Returns 403 Forbidden if the user lacks module-specific permissions

Protection Pattern:

```
SUPERADMIN → All access (unrestricted)
ADMIN → Module-specific access (users/forms/all)
USER → No admin access
```

Registration & Module Permission Assignment

The register endpoint enforces that:

- Only **SUPERADMIN** users can assign `modulePermissions` during admin creation
- Non-admin users attempting to set permissions receive a 403 Forbidden error
- Module permissions are validated as a boolean object before accessing the database

---

Token Management

JWT Token Generation

Tokens are generated in server/src/utils/jwt.utils.ts with distinct purposes and expiration:

Access Token:

- **Expiration**: 15 minutes
- **Algorithm**: HS256
- **Payload**: `userId`, `role`, `modulePermissions`
- **Secret**: `JWT_SECRET` environment variable

Refresh Token:

- **Expiration**: 7 days
- **Algorithm**: HS256
- **Payload**: `userId`
- **Secret**: `JWT_REFRESH_SECRET` environment variable (separate from access token secret)

Separation of Concerns: Using distinct secrets for access and refresh tokens limits damage if one secret is compromised.

Token Verification

- Tokens are verified using explicit algorithm specification (`HS256`) to prevent algorithm confusion attacks
- Token verification throws errors on:
  - Expired tokens
  - Invalid signatures
  - Wrong algorithms

Refresh Token Endpoint

The refresh endpoint:

- Accepts refresh tokens from HTTP-only cookies
- Verifies the refresh token and validates user status in the database
- Updates `lastHeartbeat` timestamp to track user activity
- Generates a new access token and optional new refresh token
- Requires successful token refresh to access protected resources

Client-Side Token Handling

The axios config implements automatic token refresh:

- When a 401 response is received, the client automatically attempts refresh
- Failed refresh redirects to login and clears authentication state
- Prevents replay of expired tokens

CSRF Protection

Double-Submit Token Pattern

CSRF protection is implemented in server/src/utils/csrf.utils.ts using the double-submit cookie pattern:

Token Generation:

- Generated using cryptographically secure `crypto.randomBytes(32)` converted to hex
- 256 bits of entropy per token

Token Attachment:

- Tokens are set in a non-httpOnly cookie (`csrf_token`)
- Client JavaScript can read the token from cookies
- Token persists for 24 hours

Token Verification:

- Applies to all state-changing methods: POST, PUT, PATCH, DELETE
- GET, HEAD, OPTIONS requests skip verification
- Compares cookie token with `X-CSRF-Token` header value
- Uses timing-safe comparison (`crypto.timingSafeEqual`) to prevent timing attacks

- Validates token presence and equality before processing requests

Security Properties:

- Timing-safe comparison prevents attackers from guessing tokens
- Length check before comparison prevents DoS via exception
- SameSite cookie attribute (lax) provides additional cross-site request protection

Client Token Retrieval

The axios interceptor handles token management:

- Fetches token from `/auth/csrf-token` endpoint on first state-changing request
- Implements token caching with promise deduplication to avoid race conditions
- Retries token fetch on 403 CSRF errors
- Attaches token to all non-GET requests via `X-CSRF-Token` header

Rate Limiting

Rate limiting is configured in server/src/middlewares/ratelimit.middleware.ts using `express-rate-limit`:

Authentication Rate Limiter

Purpose: Prevent brute-force attacks on login endpoints

Configuration:

- **Window**: 15 minutes
- **Limit**:
  - Production: 20 attempts per window
  - Development: 100 attempts per window
- **Skip Successful Requests**: Enabled
  - Successful authentications don't count against limit
  - Failed attempts consume limit capacity
- **Response Headers**: Standard RateLimit headers returned for client awareness

General API Rate Limiter

Purpose: Protect general API endpoints from abuse

Configuration:

- **Window**: 15 minutes
- **Limit**:
  - Production: 100 requests per window
  - Development: 1000 requests per window
- **Applied to**: All `/api` routes globally

Trust Proxy Configuration

The server sets `trust proxy: 1` to correctly identify client IP addresses in reverse proxy environments (Render, Heroku), ensuring rate limits apply per-client rather than per-server.

Input Validation & Sanitization

Regex Injection Prevention

The helper.utils.ts implements safe regex construction:

```
export const escapeRegex = (str: string): string =>
  str.replace(/[.*+?^${}()|[\]\\]/g, "\\$&");

export const buildSafeRegex = (search: string, maxLength = 128): RegExp =>
{
  const trimmed = search.slice(0, maxLength);
  return new RegExp(escapeRegex(trimmed), "i");
};
```

Protection Against:

- Regular Expression Denial of Service (ReDoS) attacks
- Unintended regex metacharacter behavior
- Search strings are truncated to 128 characters maximum

Date Validation

Date range filters in search endpoints validate:

- ISO 8601 date string format
- Logical date ranges (start ≤ end)
- Returns 400 Bad Request for invalid dates
- Throws explicit error messages for validation failures

Short Answer Length Validation

The response controller validates form responses:

- Short answer responses limited to 255 characters
- Prevents excessive payload sizes
- Returns 400 Bad Request if answer exceeds limit

Payload Structure Validation

The response submission sanitizes nested payload structures to prevent nesting bugs:

- Flattens potentially double-nested `answers` objects
- Validates answer structure before database insertion

Google Sheets URL Validation

Google Sheet integration includes:

- URL parsing to extract sheet ID using regex
- Fallback to direct ID if URL parsing fails
- Sheet access validation through Google API calls
- Returns 403 Forbidden if service account lacks permissions

Error Handling & Information Disclosure

Centralized Error Handler

The error middleware implements:

Error Classification:

- **Operational Errors**: Expected errors (validation, not found, unauthorized)
  - Log level: WARN
  - Full error details sent to client
  - Examples: 400, 401, 403, 404 responses
- **Non-Operational Errors**: Programming bugs or unknown errors
  - Log level: ERROR (includes stack trace)
  - Generic response to client in production
  - Stack traces only shown in development

Special Error Handling:

- **MongoDB Validation Errors**: 400 Bad Request
- **Invalid ObjectId (CastError)**: 400 Bad Request with "Invalid ID format"
- **Duplicate Key Errors**: 409 Conflict with "Duplicate entry"

Information Disclosure Prevention:

- Production environment: Stack traces not sent to client
- Development environment: Stack traces provided for debugging
- Consistent error response structure: `{ success: false, message }`
- Client IP and request metadata logged server-side for investigation

AppError Custom Class

The AppError class provides:

- Standardized error structure with status codes
- Factory methods for common HTTP errors
- Distinction between operational and non-operational errors
- Automatic stack trace capture for debugging

Security Headers

The application uses Helmet.js middleware to set HTTP security headers:

```
app.use(helmet());
```

Headers Applied (default Helmet configuration):

- `Strict-Transport-Security`: HSTS for HTTPS enforcement
- `X-Frame-Options`: Clickjacking protection
- `X-Content-Type-Options`: MIME type sniffing prevention
- `X-XSS-Protection`: Legacy XSS filter
- `Content-Security-Policy`: Restricts resource loading
- Additional security headers per Helmet defaults

Database Security

MongoDB Connection

The database connection:

- Reads `MONGODB_URI` from environment variables
- Connection parameters specified at URI level (authentication, database selection)
- Automatic reconnection handling
- Exits process on connection failure (prevents silent failures)

User Model Security

The User model:

- Passwords stored using bcrypt (salt rounds: 10 by default in bcryptjs)
- Unique index on email field to prevent duplicate accounts
- User status field (`ACTIVE`, `INACTIVE`) allows account deactivation without deletion
- Module permissions stored as a nested schema with proper structure validation
- Database indexes optimized for query performance:
  - `createdAt` (descending) for sorting
  - `lastHeartbeat` (descending) for session monitoring
  - `(status, createdAt)` composite index
  - `(status, lastHeartbeat)` composite index

Query Security

Aggregation pipelines in controllers:

- Use `$match` with properly constructed filters
- Limit result sets before `$lookup` operations to prevent excessive data transfer
- Use `$expr` for conditional matching in pipeline stages
- Validate numeric inputs (page, limit) before database queries

---

Logging & Monitoring

Winston Logger Configuration

The logger provides:

Log Levels (npm standard):

- `error`: Non-operational errors with stack traces
- `warn`: Operational errors and warnings
- `info`: General application information
- `http`: HTTP request details
- `debug`: Detailed debugging information

Environment-Based Configuration:

- **Production**: Only `error` and `warn` levels
- **Development**: All levels including `debug`

Transport Configuration:

- **Console**: Colorized output for development
- **File**: JSON format for parsing and archival (only for development)
- **MongoDB**: Persistent error logging via MongoDB transport

Logged Information:

- Timestamp, level, message, stack trace (for errors)
- Request context: method, URL, IP address, response status
- Error-specific metadata for investigation

Activity Tracking

The heartbeat middleware:

- Updates `lastHeartbeat` on authenticated requests
- Non-blocking async update (doesn't delay request response)
- Tracks user activity patterns for security analysis
- Configurable window period via SystemSettings

Third-Party Integration Security

Google Sheets Integration

The Google Sheets service:

Authentication:

- Uses Google Service Account with separate private key
- Scoped to Sheets API only: `https://www.googleapis.com/auth/spreadsheets`
- Credentials loaded from `GOOGLE_SERVICE_ACCOUNT_JSON` environment variable

- JSON parsing with error handling for invalid credentials

Authorization Checks:

- Validates spreadsheet access before allowing form creation
- Returns 403 Forbidden if service account lacks permissions
- Returns 404 Not Found if spreadsheet doesn't exist
- Error logging for troubleshooting

Sheet Validation:

- Parses sheet URLs to extract sheet ID
- Supports both URL(preferred) and direct ID formats
- Initializes required headers (ID, NAME, EMAIL) on first use
- Validates before form association

Considerations:

- Service account email must be shared on target sheets
- Google API quota limits apply to validation and sync operations
- Sheet sync to responses handled by MongoDB Atlas Triggers (separate from main application)

Session Management

Cookie-Based Sessions

The application uses HTTP-only cookies for both authentication and CSRF tokens:

Access Token Cookie:

- **Name**: `access_token`
- **Expiration**: 15 minutes
- **HttpOnly**: true (JavaScript cannot access)
- **Secure**: true in production, false in development
- **SameSite**: lax
- **Path**: /

Refresh Token Cookie:

- **Name**: `refresh_token`
- **Expiration**: 7 days
- **HttpOnly**: true
- **Secure**: true in production, false in development
- **SameSite**: lax
- **Path**: /

CSRF Token Cookie:

- **Name**: `csrf_token`
- **Expiration**: 24 hours

- **HttpOnly**: false (client JavaScript must read)
- **Secure**: true in production, false in development
- **SameSite**: lax
- **Path**: /

SameSite Cookie Attribute

All cookies use `SameSite: lax`:

- Allows cookies on initial navigation (user clicking links from external sites)
- Blocks cookies on cross-site form submissions
- Provides additional CSRF protection layer

Session Validation

Each authenticated request:

- Verifies access token signature and expiration
- Confirms user exists and is in ACTIVE status
- Rejects tokens from inactive/deleted users
- Requires token refresh for expired tokens

Development vs Production Differences

Aspect	Development	Production
Secure Cookie Flag	false	true
CORS Enabled	yes	no
Error Stack Traces	yes	no
Rate Limit (Auth)	100/15min	20/15min
Rate Limit (API)	1000/15min	100/15min

Cookie Security

Cookie Configuration Strategy

HttpOnly Cookies (Access & Refresh Tokens):

- Protected from JavaScript XSS attacks
- Automatically sent with every request to the domain
- Cannot be stolen via document.cookie

Non-HttpOnly Cookies (CSRF Token):

- Must be readable by JavaScript for double-submit pattern
- Contains random, unpredictable value
- Paired with header validation for CSRF protection

Secure Flag

- Enabled in production to enforce HTTPS-only transmission
- Disabled in development to allow HTTP testing
- Prevents transmission over unencrypted channels in production

Path & Domain Scope

All cookies use:

- **Path**: `/` - Accessible to entire application
- **Domain**: Not explicitly set - Defaults to current domain
- Prevents unauthorized domain access via cookie scope

Further Considerations

Authentication & Access Control

- **Password Creation** :
  - Remove default/admin configured passwords.
  - Create Password creation flow for new users using email (preferred).
  - Or atleast send password to emails on registration.
- **Password Reset Mechanism**: Ensure password reset flows use temporary, expiring tokens sent via email (if implemented).

Cryptography & Secrets

- **JWT Secret Rotation**: Currently using static secrets from environment variables. Consider implementing key rotation policies.
- **Secret Exposure Risk**: GOOGLE_SERVICE_ACCOUNT_JSON and JWT secrets stored in `.env` file. Ensure:
  - `.env` files are never committed to version control
  - Secrets managed via secure secret management service in production
  - Regular audits of who has access to secrets

API Security

- **Request Size Limits**: Body parser configured with 2MB limit. Verify this aligns with expected payloads.

Third-Party Dependencies

- **Dependency Updates**: Ensure regular security updates for:
  - Express.js
  - Mongoose
  - JWT library
  - Bcrypt
  - Helmet
  - All other npm packages

- **Vulnerability Scanning**: Use `npm audit` regularly and consider automated scanning in CI/CD.

Last Updated: December 29, 2025