RESOURCE MONITORING AND MANAGEMENT SYSTEM

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING Database Management System – CD252IA Experiential Learning (Lab)

Design of Forms, Security, and Validation Report

Submitted by [Your Name] [Roll Number]

==========

A. Design of Forms & Frontend Screenshots

The Resource Monitoring and Management System is built using React.js for the frontend dashboard and Python/FastAPI for the backend API, providing a modern, responsive, and efficient interface for system administrators. The UI is designed to be intuitive for IT staff, providing real-time monitoring and management of computer systems across a network.

Below are descriptions of the key forms and interfaces:

1. Login Screen

Purpose: The primary entry point for the application, ensuring only authenticated administrators can access the system monitoring dashboard.

Design: A clean authentication form with fields for "Email" and "Password". The login screen uses modern Material-UI components for a professional appearance. It includes JWT-based authentication with secure token storage. A clear "Login" button initiates the authentication process. The design is centered and responsive, adapting to different screen sizes.

Location: - Frontend: dashboard/src/pages/Login.jsx - Backend API: backend/app/api/auth.py (lines 24-31)

Technical Implementation: - Uses React state management for form handling - Implements client-side validation - Stores JWT tokens securely in browser localStorage - API endpoint: POST /api/auth/login

1. Main Dashboard Screen

Purpose: The central command center of the application. It displays a comprehensive real-time overview of all monitored systems in the network.

Design: Uses a grid-based layout with Material-UI Card components for a modern look. The dashboard displays: - Total number of systems being monitored - System health indicators (CPU, Memory, Disk usage) - Active alerts and warnings - Quick statistics on system performance - Interactive charts showing resource usage trends

Features: - Auto-refresh functionality (polling every 5 seconds) - Manual refresh button - Responsive grid layout adapting to screen size - Color-coded status indicators (green/yellow/red) - Navigation sidebar for accessing different sections

Location: - Frontend: dashboard/src/pages/Dashboard.jsx - Backend API: backend/app/api/endpoints.py

Technical Implementation: - Real-time data fetching using Axios - Chart.js for visualization - Material-UI for consistent design - API endpoints: GET /api/systems, GET /api/metrics

1. System Detail Screen

Purpose: Provides detailed breakdown of a specific system, showing comprehensive information including hardware specs, current metrics, and historical performance data.

Design: This screen displays comprehensive system information:

System Information Section: - Hostname, IP Address, MAC Address - Operating System details (Windows edition, build number) - CPU information (name, cores, threads) - Memory specifications (total RAM) - Disk information (capacity, model) - GPU details - Manufacturer and model information

Real-Time Metrics Section: - Current CPU usage (%) - Current Memory usage (%) - Disk usage and free space - Network statistics (sent/received) - Running process count - System uptime - Battery status (for laptops)

Performance History Section: - Interactive time-series charts showing: * CPU usage over time * Memory usage trends * Disk I/O statistics * Network traffic patterns

Alert History Section: - List of all alerts triggered for this system - Alert severity levels - Timestamps and resolution status

Location: - Frontend: dashboard/src/pages/SystemDetail.jsx - Backend API: backend/app/api/endpoints.py

Technical Implementation: - Dynamic routing with React Router - Real-time metric updates - Historical data visualization - API endpoints: * GET /api/systems/{system_id} * GET /api/systems/{system_id}/metrics * GET /api/systems/{system_id}/alerts

1. Alert Configuration Form

Purpose: Allows administrators to configure threshold values for automated alerts on a per-system or global basis.

Design: A form-based interface with clearly labeled input fields for: - CPU usage threshold (default: 90%) - Memory usage threshold (default: 90%) - Disk usage threshold (default: 90%) - Alert notification preferences

The form includes: - Number input fields with validation - Save/Cancel action buttons - Reset to default button - Visual feedback on save success/failure

Location: - Backend Model: backend/app/models/models.py (lines 107-116) - Backend API: backend/app/api/endpoints.py

Technical Implementation: - Form validation (0-100 range for percentages) - RESTful API integration - Immediate application of new thresholds - API endpoint: PUT /api/alert-settings

==========

B. Security and Validation Proof

Security:

1. Authentication System The system implements robust multi-user authentication using JSON Web Tokens (JWT).

Implementation Details: - Users must register and log in to access the dashboard - JWT tokens are generated upon successful authentication - Tokens are validated on every API request - Token expiration is enforced

Code Reference: File: backend/app/api/auth.py

Registration Endpoint (lines 10-22):

```
@router.post("/register", response_model=schemas.User)
def register(user: schemas.UserCreate, db: Session = Depends(get_db)):
    # Check if user exists
    existing = db.query(models.User).filter(models.User.email == user.email).first()
    if existing:
        raise HTTPException(status_code=400, detail="Email already registered")

    hashed = get_password_hash(user.password)
    db_user = models.User(email=user.email, hashed_password=hashed)
    db.add(db_user)
    db.commit()
    db.refresh(db_user)
    return db_user
```

Login Endpoint (lines 24-31):

```
@router.post("/login", response_model=schemas.Token)
def login(user: schemas.UserLogin, db: Session = Depends(get_db)):
    db_user = db.query(models.User).filter(models.User.email == user.email).first()
    if not db_user or not verify_password(user.password, db_user.hashed_password):
        raise HTTPException(status_code=401, detail="Invalid credentials")

    access_token = create_access_token(data={"sub": db_user.email})
    return {"access_token": access_token, "token_type": "bearer"}
```

1. Password Hashing User passwords are never stored in plain text. They are securely hashed using the PBKDF2-SHA256 algorithm via the passlib library before being stored in the database.

Code Reference: File: backend/app/core/security.py

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["pbkdf2_sha256"], deprecated="auto")

def verify_password(plain_password, hashed_password):
    return pwd_context.verify(plain_password, hashed_password)

def get_password_hash(password):
    return pwd_context.hash(password)
```

Database Proof: File: backend/resource_monitor.db Table: users

Current User Records: | ID | Email | Hashed Password | Status | Created | |----|-------|-----------------|--------|---------| | 1 | nthy2355@gmail.com | $pbkdf2-sha256$29000$WKv1nhOCM... | Active | 2025-12-25 | | 2 | Admin1@gmail.com | $pbkdf2-sha256$29000$GmOM0boX... | Active | 2025-12-27 |

As visible, passwords are stored as cryptographic hashes, making them irreversible.

1. Data Integrity and Foreign Key Relationships Every data-related table in the database has proper foreign key relationships to ensure data integrity and referential consistency.

Database Schema: File: backend/app/models/models.py

Key Tables and Relationships:

a) Systems Table (lines 15-49): - Stores all monitored system information - Has relationships to metrics, alerts, and tickets

b) Metrics Table (lines 68-93): - Foreign key: system_id → systems.id - Stores time-series performance data - Cascade delete when system is removed

c) Alerts Table (lines 55-66): - Foreign key: system_id → systems.id - Stores alert history - Cascade delete when system is removed

d) Tickets Table (lines 95-105): - Foreign key: system_id → systems.id - Stores support tickets - Cascade delete when system is removed

Example Schema Definition:

```
class Metric(Base):
    __tablename__ = "metrics"

    id = Column(Integer, primary_key=True, index=True)
    system_id = Column(Integer, ForeignKey("systems.id"))  # Foreign key constraint
    timestamp = Column(DateTime(timezone=True), server_default=func.now(), index=True)
    cpu_usage = Column(Float)
    memory_percent = Column(Float)
    # ... other fields

    system = relationship("System", back_populates="metrics")  # Relationship
```

1. API Endpoint Protection All sensitive API endpoints are protected and require authentication.

Implementation: - Rate limiting to prevent abuse - CORS configuration for cross-origin security - Input sanitization - SQL injection prevention through ORM (SQLAlchemy)

Code Reference: File: backend/app/core/rate_limiter.py File: backend/app/main.py

1. Agent-Server Communication Security The monitoring agents running on client systems communicate with the server using secure protocols.

Implementation Details: - Configurable server IP and port - Automated IP detection for deployment - Secure metric transmission - System identification using hostname and MAC address

Code Reference: File: agent/main.py File: agent/gui.py Build Configuration: setup_lab.py

==========

Validation:

1. Frontend Validation The React dashboard implements client-side validation for all user inputs:
2. Email format validation for login/registration
3. Required field validation
4. Number range validation for alert thresholds (0-100)
5. Form state management using React hooks

Location: dashboard/src/ (various component files)

1. Backend Validation The FastAPI backend uses Pydantic for powerful, automatic data validation.

Every incoming request is validated against a schema. If a client sends data of the wrong type or with missing required fields, the request is automatically rejected with a 422 Unprocessable Entity error.

Code Reference: File: backend/app/schemas/schemas.py

Example Schema Definitions:

User Registration Schema (lines 7-9):

```
class UserCreate(BaseModel):
    email: str  # Must be string
    password: str  # Must be string
```

User Login Schema (lines 11-13):

```
class UserLogin(BaseModel):
    email: str
    password: str
```

System Input Schema (lines 27-59):

```
class SystemCreate(BaseModel):
    hostname: str  # Required
    ip_address: Optional[str] = None
    mac_address: Optional[str] = None
    os_info: Optional[str] = None
    # ... all fields with type constraints
```

Metric Input Schema (lines 91-112):

```
class MetricCreate(BaseModel):
    cpu_usage: float  # Must be float
    memory_percent: float
    memory_used: Optional[int] = None
    disk_free: Optional[int] = None
    disk_usage: Optional[float] = None
    # ... validated types prevent invalid data
```

1. Database-Level Validation SQLAlchemy models enforce database constraints:

Constraints: - NOT NULL constraints for required fields - UNIQUE constraints for emails, hostnames - PRIMARY KEY constraints - FOREIGN KEY constraints - DEFAULT values for boolean and timestamp fields - Data type enforcement at database level

Example:

```
class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True, index=True)  # Primary key
    email = Column(String, unique=True, index=True, nullable=False)  # Unique, not null
    hashed_password = Column(String, nullable=False)  # Required
    is_active = Column(Boolean, default=True)  # Default value
    created_at = Column(DateTime(timezone=True), server_default=func.now())  # Auto timestamp
```

1. Input Sanitization All user inputs are sanitized before processing:
2. SQL Injection Prevention: Using SQLAlchemy ORM with parameterized queries
3. XSS Prevention: React automatically escapes HTML in JSX
4. Data type validation: Pydantic schemas enforce correct types
5. Length limits: Database column constraints

==========

C. Innovative Experiment: Automated System Discovery and Monitoring

The most innovative component of the Resource Monitoring and Management System is the Automated Agent Deployment and Real-Time Monitoring architecture. This goes beyond simple system monitoring by implementing an intelligent, self-contained agent that can be rapidly deployed across an entire network.

Concept:

The primary challenge in enterprise IT management is rapid deployment and configuration of monitoring solutions across diverse systems. Traditional monitoring tools require complex setup, manual configuration, and constant maintenance. Our innovative approach was to create a self-contained, automatically configurable monitoring agent that requires zero manual configuration.

Working:

1. Automated IP Detection and Configuration File: setup_lab.py

The build system automatically detects the server's local IP address and embeds it directly into the agent executable during the build process.

```
def get_local_ip():
    """Detect the local IP address of the server"""
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        s.connect(('8.8.8.8', 80))
        ip = s.getsockname()[0]
    finally:
        s.close()
    return ip

# Automatically update agent files with server IP
local_ip = get_local_ip()
update_agent_files_with_ip(local_ip)
```

1. One-Click Deployment File: build_lab_agent.bat

IT administrators can build deployment-ready agents with a single command: - Agent is compiled to a standalone executable - No Python installation required on client systems - Pre-configured with correct server address - Includes system tray GUI for user visibility

1. Real-Time Metrics Collection File: agent/main.py

The agent runs continuously on client systems, collecting comprehensive metrics:

Hardware Information Collected: - CPU name, cores, threads, architecture - Total memory (RAM) - Disk capacity and model - GPU information - Manufacturer and model details - BIOS version - Serial number

Runtime Metrics Collected: - CPU usage percentage - Memory usage (percent and bytes) - Disk usage and free space - Disk I/O (read/write bytes) - Network statistics (sent/received) - Process count - System uptime - Boot time - Top processes by resource usage - Battery status (for laptops) - Driver information

1. Centralized Database Architecture File: backend/app/models/models.py

All metrics are centralized in a SQLite database with optimized schema design:

Time-Series Data Storage: - Indexed timestamps for fast querying - Efficient storage of metrics history - Support for large-scale data (BigInteger for cumulative values) - Automatic cleanup of old metrics

1. Intelligent Alert System File: backend/app/core/alerts.py

The system automatically monitors thresholds and generates alerts:

Alert Generation Process: - Continuous threshold monitoring - Automatic alert creation when thresholds are exceeded - Configurable severity levels - Alert resolution tracking - Per-system and global configuration

1. System Discovery File: backend/app/core/discovery.py

New systems are automatically discovered and registered: - First connection creates system record - Subsequent connections update "last_seen" timestamp - Inactive systems are flagged - Hardware changes are tracked

Architecture Benefits:

1. Zero-Configuration Deployment: IT staff can deploy agents to hundreds of systems without manual IP configuration
2. Self-Healing: Agents automatically reconnect if the connection is lost
3. Minimal Overhead: Efficient data collection with minimal impact on system performance
4. Scalability: SQLite database can handle thousands of systems and millions of metrics
5. Historical Analysis: Complete time-series data for performance trend analysis
6. Comprehensive Monitoring: Single agent collects all necessary system information

This architecture creates an enterprise-grade monitoring solution that rivals commercial products while maintaining simplicity of deployment. The automated configuration and robust data collection ensure that administrators can focus on analysis and response rather than system management.

Technical Innovation:

The key innovation is the integration of build-time configuration with runtime flexibility. By embedding the server IP during the build process (setup_lab.py), we eliminate the most common deployment error—incorrect server configuration—while still maintaining a clean, professional agent interface (agent/gui.py) that provides users with visibility into the monitoring process.

==========

D. Database Structure and Verification

Database File Location: backend/resource_monitor.db

Total Tables: 6

1. users
2. Stores administrator accounts
3. Columns: id, email, hashed_password, is_active, created_at
4. systems
5. Stores monitored system information
6. Columns: id, hostname, ip_address, mac_address, os_info, cpu_name, total_memory_gb, and 30+ other hardware/software fields
7. metrics
8. Stores time-series performance data
9. Columns: id, system_id, timestamp, cpu_usage, memory_percent, disk_usage, network_sent, network_recv, and more

10. alerts
11. Stores alert history
12. Columns: id, system_id, alert_type, severity, message, is_resolved, created_at
13. tickets
14. Stores support tickets
15. Columns: id, system_id, message, status, resolved_at, created_at
16. alert_settings
17. Stores threshold configurations
18. Columns: id, system_id, cpu_threshold, memory_threshold, disk_threshold

Verification Scripts: - inspect_db.py - View user table - view_all_user_data.py - Detailed user information - open_database.bat - Open database in DB Browser for SQLite

==========

E. System Architecture Overview

Components:

1. Backend Server (FastAPI)
2. RESTful API
3. JWT Authentication
4. Database management
5. Alert generation
6. Real-time metrics processing
7. Frontend Dashboard (React.js)
8. Material-UI components
9. Real-time data visualization
10. Interactive charts
11. System management interface
12. Monitoring Agent (Python + GUI)
13. System metrics collection
14. Continuous background operation
15. System tray interface
16. Automatic reconnection
17. Database (SQLite)
18. Centralized data storage
19. Efficient queries

20. Foreign key relationships
21. Data integrity constraints
22. Build System (Automation)
23. Automated IP detection
24. Agent compilation
25. Configuration management
26. One-click deployment

Technology Stack: - Backend: Python 3.x, FastAPI, SQLAlchemy, Passlib - Frontend: React.js, Material-UI, Axios, Chart.js - Agent: Python 3.x, psutil, PyQt5/tkinter - Database: SQLite - Build: PyInstaller, Batch scripts

==========

Conclusion:

The Resource Monitoring and Management System demonstrates professional implementation of database management principles, security best practices, and innovative system architecture. The combination of robust authentication, comprehensive validation, and intelligent automation creates a production-ready monitoring solution suitable for enterprise deployment.

Key Achievements: ✓ Secure multi-user authentication with JWT ✓ Password hashing using industry-standard algorithms ✓ Comprehensive input validation at all layers ✓ Foreign key relationships ensuring data integrity ✓ Automated agent deployment system ✓ Real-time monitoring and alerting ✓ Professional, responsive user interface ✓ Scalable architecture supporting large deployments

==========