Incident Report

Raymond Choy

CPAN-212-0NA
Professor Vitalii Bohudskyi
February 8 2026

1. Introduction

The purpose of this lab was to design and implement a full-stack incident reporting and tracking system called *IncidentTracker*. The application consists of an Express.js backend that exposes REST API endpoints and a Next.js frontend that consumes these APIs.

In the original version, the system used in-memory storage. For Lab 3, the system was modified to use JSON file-based persistence so that data remains stored even after restarting the server. Additional features were implemented, including an ARCHIVED status, controlled status transitions, CSV bulk upload functionality, and a configuration file for centralized settings.

This project demonstrates REST API design, frontend-backend integration, file persistence, input validation, and proper error handling.

2. System Architecture

The system follows a simple three-layer architecture:

Frontend (Next.js) → Backend API (Express.js) → JSON File Storage (incidents.json)

The frontend sends HTTP requests to the backend. The backend processes the request, applies validation and business rules, and then reads or writes data to a JSON file located at:

backend/data/incidents.json

No database was used, as required by the lab instructions.

3. Backend Implementation

3.1 server.js

Purpose:

- Entry point of the backend application.
- Starts the Express server.
- Uses configuration settings from config.js.

This file launches the server using a configurable port.

3.2 src/app.js

Purpose:

- Creates the Express application instance.
- Configures middleware (JSON parsing, error handling).
- Connects route files to the application.

This file ensures all API routes are properly registered and that errors are returned in JSON format.

3.3 src/routes/incidents.routes.js

Purpose:

- Defines all REST API endpoints for incidents.

Endpoints implemented:

- GET /health
- GET /api/incidents
- GET /api/incidents/:id
- POST /api/incidents
- PATCH /api/incidents/:id/status
- POST /api/incidents/bulk-upload

This file receives requests, validates inputs, calls store functions, and returns appropriate HTTP responses.

3.4 src/store/incidents.store.js

Purpose:

- Handles file-based data persistence.
- Reads and writes to incidents.json.

This file replaces the original in-memory array. It:

- Loads data from file
- Saves new incidents
- Updates incident status
- Applies archive filtering
- Writes changes back to file

Data persists even after server restart.

3.5 src/utils/validate.js

Purpose:

- Validates incident data before storing or updating.

Validations include:

- Required fields
- Minimum length requirements
- Valid category values
- Valid severity values
- Valid status transitions

If validation fails, the backend returns HTTP 400 with a JSON error message.

3.6 src/utils/csv.js

Purpose:

- Handles CSV file uploads.
- Parses CSV rows.
- Validates each row.

- Creates incidents for valid rows.
- Skips invalid rows.

Returns a summary in the following format:

```
{
 "totalRows": number,
 "created": number,
 "skipped": number
 }
```

3.7 config.js

Purpose:

- Central configuration file.
- Similar to Lab 2 structure.

Contains:

- Server port
- Archive behavior configuration
- Other reusable constants

This improves maintainability and organization.

3.8 data/incidents.json

Purpose:

- Stores all incident records.
- Provides file-based persistence.

Example structure:

```
[
 {
```

"id": "unique-id",
"title": "Network outage",
"description": "Main office network is down",
"category": "IT",
"severity": "HIGH",
"status": "OPEN",
"reportedAt": "ISO timestamp"
}
]

The file is updated every time an incident is created or modified.

4. Frontend Implementation

4.1 Incident List Page (/incidents)

Features:

- Fetches incidents from the backend.
- Displays incidents in a list or table.
- Contains a “Show Archived” checkbox.
- Archived incidents are hidden by default.
- When the checkbox is selected, archived incidents are displayed.

4.2 Create Incident Page (/incidents/create)

Features:

- Form inputs for title, description, category, severity.
- Client-side validation.
- Send POST request to backend.
- Redirects to the incident list on success.

4.3 Incident Details Page (/incidents/[id])

Features:

- Fetches specific incidents by ID.
- Displays all fields.
- Dropdown to update status.
- Supports ARCHIVED status.
- Sends PATCH requests to the backend.
- Displays error messages if transition is invalid.

4.4 CSV Upload Page (/bulk-upload)

Features:

- File input accepting CSV only.
- Upload button.
- Sends multipart/form-data requests.
- Displays summary of totalRows, created, and skipped.

5. Business Rules Implemented

5.1 Status Transitions

Allowed transitions:

- OPEN → INVESTIGATING
- INVESTIGATING → RESOLVED
- OPEN → ARCHIVED
- RESOLVED → ARCHIVED
- ARCHIVED → OPEN

Invalid transitions return HTTP 400.

5.2 Archive Behavior

- ARCHIVED incidents are hidden by default.
- A checkbox allows viewing archived incidents.
- Archive filtering is configurable.
- ARCHIVED incidents can only be reset to OPEN.

6. Error Handling

Backend:

- Uses proper HTTP status codes (400, 404).
- Returns errors in JSON format.
- Prevents server crashes on invalid input.

Frontend:

- Displays backend error messages.
- Shows alerts for invalid transitions.
- Prevents application crashes.

7. Data Persistence

The system uses JSON file persistence instead of in-memory storage.

Process:

1. Incidents are written to incidents.json.
2. When the server restarts, data is loaded from the file.
3. No database is required.

This ensures data remains stored after restart.

8. Data Workflows

8.1 Create Incident Workflow

1. User fills form.
2. Frontend sends POST requests.
3. Backend validates data.
4. Data is written to a JSON file.
5. Response returned.
6. Frontend redirects to the list page.

8.2 Update Incident Workflow

1. User selects new status.
2. Frontend sends PATCH requests.
3. Backend validates transition.
4. JSON file is updated.
5. Updated data returned.

8.3 Read Incident Workflow

1. Frontend sends GET requests.
2. The backend reads JSON files.
3. Archive filter applied.
4. Data returned.

8.4 Bulk Upload Workflow

1. User uploads CSV file.
2. Backend parses file.
3. Each row validated.
4. Valid rows saved.
5. Summary returned.

9. Conclusion

The IncidentTracker application successfully implements:

- RESTful API design
- JSON file-based persistence
- ARCHIVED status functionality
- Controlled status transitions
- CSV bulk upload
- Frontend-backend integration
- Proper validation and error handling

All coding requirements for Lab 3 were completed successfully. The application runs locally without additional configuration and meets all specified functional requirements.