

# Incident Management System

## 1. Project Overview

---

The Incident Management System is a full-stack web application designed to track and manage incidents across different categories including IT, Safety, Facilities, and Other. The system provides a complete workflow from incident creation through investigation to resolution and archival.

The system is built on a Node.js and Express.js backend that exposes a REST API, paired with a Next.js (React) frontend for a rich user experience. Data is persisted using a JSON file-based storage strategy, which ensures durability across server restarts without requiring a dedicated database server.

### Key Features

The system uses JSON-based file storage to ensure data durability across sessions. Incidents follow a well-defined status workflow moving through OPEN, INVESTIGATING, RESOLVED, and ARCHIVED states. Users can archive and restore incidents as needed, and a bulk upload feature allows importing multiple incidents at once from CSV files. A real-time dashboard provides KPI summaries and status-based views, while a complete REST API enables all CRUD operations programmatically.

### Technology Stack

The backend is built on Node.js with the Express.js framework, which handles REST API development and request routing. The frontend uses Next.js, a React framework, which provides server-side rendering and client-side routing. Data storage relies on a JSON file system, with Multer middleware handling CSV file uploads. The project intentionally keeps external dependencies minimal to aid maintainability.

## 2. System Architecture

---

The Incident Management System follows a three-tier architecture with a clear separation of concerns between the presentation layer, business logic layer, and data layer.

The Presentation Layer is handled by Next.js (React) and is responsible for the user interface, forms, dashboards, and client-side routing. The API Layer, powered by Express.js, manages REST endpoints, request validation, and business logic. The Data Access Layer consists of Node.js modules that perform CRUD operations, handle JSON file I/O, and manage data persistence. Finally, the Storage Layer is made up of plain JSON files that store incident records and serve as the application's file-based database.

## Project Structure

The backend is organized under a root `backend/` directory. At the top level sits `config.js` for centralized configuration and `server.js` as the application entry point. A `data/` subdirectory holds `incidents.json`, the JSON database file. The `src/` directory contains `app.js` for Express setup, a `routes/` subdirectory with `incidents.routes.js` defining all API endpoints, a `store/` subdirectory with `incidents.store.js` as the data layer, and a `utils/` subdirectory housing `csv.js` for CSV parsing and `validate.js` for input validation.

The frontend is organized under a `frontend/` directory. The `pages/` directory uses Next.js file-based routing: `index.js` is the home page that redirects to the dashboard, `dashboard.js` renders the main dashboard view, `bulk-upload.js` handles CSV uploads, and an `incidents/` subdirectory contains `index.js` for the list view, `create.js` for the creation form, and `[id].js` for individual incident detail and editing. Supporting this are a `components/` directory with `Layout.js` and `ErrorBanner.js`, a `services/` directory with `api.js` as the HTTP client, and a `styles/` directory with `globals.css` for application-wide styling.

## 3. Backend Documentation

---

### 3.1 Configuration File (`config.js`)

The configuration file serves as the central location for all backend settings, including server parameters, storage paths, validation rules, and business logic configurations. This approach ensures that key values are defined in one place and can be adjusted without touching multiple files.

```
export const config = {
  // Server Configuration
  server: {
    port: 3001,
    host: "localhost",
    corsOrigin: "http://localhost:3000",
  },
}
```

Server configuration specifies the HTTP port (default 3001), the host (localhost), and the allowed CORS origin. Storage configuration defines the file path for the JSON database, whether auto-save is enabled, and backup settings. Incident configuration holds the lists of valid statuses, allowed status transitions, and accepted categories. Validation rules set character length limits for fields like title and description. Bulk upload configuration specifies file size limits and allowed MIME types for CSV imports.

The status transition rules define the allowed workflow paths. An incident in the OPEN state may transition to either INVESTIGATING or ARCHIVED. An INVESTIGATING incident may only move to RESOLVED. A RESOLVED incident may only be moved to ARCHIVED. An ARCHIVED incident may be reset back to OPEN, but this is only

possible via the dedicated reset endpoint rather than the general status update endpoint.

## 3.2 Server Entry Point (server.js)

The server entry point is the main file that bootstraps the entire backend application. Its primary responsibility is to initialize the data store and then start the HTTP server.

The startup sequence begins by importing the Express application and loading the configuration. It then calls `initializeStore()` to load the existing incidents from the JSON file into memory. Once the store is ready, the HTTP server starts listening on the configured port (default 3001) and logs its status. If initialization fails for any reason, an error is logged and the process exits with code 1 to signal failure clearly.

```
✓ async function startServer() {
✓   try {
      // Initialize store - loads incidents from JSON file
      console.log("Initializing incidents store...")
      await initializeStore()
      console.log("Store initialized successfully")

      // Start HTTP server
      const PORT = process.env.PORT || config.server.port
      app.listen(PORT, () => {
        console.log(
          `IncidentTracker API running on http://${config.server.host}:${PORT}`,
        )
        console.log(`Data file: ${config.storage.incidentsFilePath}`)
      })
    } catch (error) {
      console.error("Failed to start server:", error)
      process.exit(1)
    }
  }
}
```

## 3.3 Express Application Setup (app.js)

The `app.js` file configures the Express application by assembling the middleware chain, registering route handlers, and setting up error handling. Each piece of middleware performs a specific role in processing incoming requests.

CORS middleware is configured to allow cross-origin requests from `http://localhost:3000` (the frontend), enabling the two services to communicate across different ports. The `express.json()` middleware parses incoming JSON request bodies and makes them available as `req.body`. The incidents router is mounted at `/api/incidents`, handling all CRUD-related endpoints. A 404 handler catches any requests to undefined routes and returns an appropriate error. A global error handler sits at the end of the chain to catch any unhandled exceptions and return a 500 response. Additionally, a GET `/health` endpoint returns a simple JSON object of `{"status": "ok"}` for monitoring and health check purposes.

### 3.4 Routes Module (incidents.routes.js)

The routes module defines all HTTP endpoints for incident management and is responsible for request validation, delegating to the store layer, and formatting responses. There are seven endpoints in total.

GET /api/incidents accepts an optional query parameter includeArchived (boolean, default false) and returns an array of all matching incidents. GET /api/incidents/:id takes a UUID in the URL path and returns either the matching incident object or a 404 if not found. POST /api/incidents accepts a JSON body with title, description, category, and severity fields, and returns the newly created incident with a 201 status. PATCH /api/incidents/:id/status accepts a JSON body with a status field and updates the incident if the transition is valid. POST /api/incidents/:id/archive archives an incident, and POST /api/incidents/:id/reset resets an archived incident back to OPEN. Finally, POST /api/incidents/bulk-upload accepts a multipart form submission with a CSV file and returns a summary of how many rows were processed, created, and skipped.

### 3.5 Store Module — Data Layer (incidents.store.js)

The store module is the data access layer for the application and manages all CRUD operations for incident records with JSON file persistence. It maintains an in-memory JavaScript array of incidents for fast read access, while automatically synchronizing changes back to the JSON file after each modification (when auto-save is enabled).

```
export async function initializeStore() {
  if (isInitialized) return

  await ensureDataDirectory()

  try {
    const data = await fs.readFile(config.storage.incidentsFilePath, "utf-8")
    incidents = JSON.parse(data)
    console.log(`Loaded ${incidents.length} incidents from file`)
  } catch (error) {
    if (error.code === "ENOENT") {
      // File doesn't exist, start with empty array
      incidents = []
      await saveToFile()
      console.log("Created new incidents data file")
    } else {
      console.error("Error reading incidents file:", error)
      incidents = []
    }
  }
}
```

The module exposes several functions. initializeStore() loads all incidents from the JSON file into memory at startup, creating the file if it does not already exist. listAll(includeArchived) returns all incidents from the memory array, optionally filtering out archived ones. findById(id) searches the in-memory array for an incident by UUID. createIncident(data) creates a new incident by generating a UUID, setting the initial status to OPEN, adding a timestamp, and saving to the file. updateStatus(id, status) updates an incident's status after validating the transition rules. archiveIncident(id) sets status to ARCHIVED for incidents in OPEN or RESOLVED states.

resetArchivedIncident(id) resets an ARCHIVED incident back to OPEN. save() provides a manual trigger for file persistence when auto-save is disabled.

## 3.6 Utility Modules

### 3.6.1 CSV Parser (csv.js)

The CSV parser utility provides the parseCsvBuffer(buffer) function, which takes a Buffer of CSV file data from Multer and returns a Promise that resolves to an array of row objects. It uses the csv-parse library, configured with columns set to true so that the first row is treated as headers, and trim set to true so that whitespace is stripped from each value.

```
export function parseCsvBuffer(buffer) {
  return new Promise((resolve, reject) => {
    const rows = [];

    const parser = parse({
      columns: true,
      trim: true
    });
```

### 3.6.2 Validation (validate.js)

The validation module provides four functions that ensure data integrity before any database operations occur. validateCreateIncident(body) validates all fields of a new incident, checking that the title is between 5 and 200 characters, the description is between 10 and 2000 characters, and that the category and severity values are among the accepted options. It returns an object with ok, errors, and value properties. validateStatusChange(current, next) checks whether the proposed status transition is permitted according to the configuration, returning ok, error, and next. validateArchive(currentStatus) returns ok only if the current status is OPEN or RESOLVED, since only those states may be archived. validateReset(currentStatus) returns ok only if the current status is ARCHIVED, since reset is only meaningful for archived incidents.

```
export function validateCreateIncident(body) {
  const errors = []

  // Validate title
  if (!body.title || typeof body.title !== "string") {
    errors.push("Title is required")
  } else if (body.title.length < config.validation.title.minLength) {
    errors.push(
      `Title must be at least ${config.validation.title.minLength} characters`,
    )
  } else if (body.title.length > config.validation.title.maxLength) {
    errors.push(
      `Title must not exceed ${config.validation.title.maxLength} characters`,
    )
  }
}
```

## 4. Frontend Documentation

---

The frontend is built with Next.js, a React framework, and provides a user-friendly interface for managing incidents. It uses client-side routing, React hooks for state management, and calls the backend REST API for all data operations.

### 4.1 API Service (api.js)

The API service is a centralized HTTP client responsible for all communication with the backend. Every network call in the application flows through this module, which handles constructing fetch requests, managing headers, and parsing responses. A shared `handleJson()` function processes every response and throws a descriptive error (including the HTTP status code and backend error message) for any non-2xx response, ensuring consistent error handling across the application.

```
export async function health() {  
  const res = await fetch(`${BASE}/health`)  
  return handleJson(res)  
}
```

The module exposes the following functions: `health()` performs a GET `/health` request to verify the backend is running. `listIncidents(includeArchived)` calls GET `/api/incidents` with the appropriate query parameter. `getIncident(id)` fetches a single incident by UUID. `createIncident(payload)` sends a POST request with the new incident payload. `changeIncidentStatus(id, status)` sends a PATCH request with the new status. `archiveIncident(id)` and `resetIncident(id)` send POST requests to their respective endpoints. `bulkUploadCsv(file)` constructs a multipart form submission and posts it to the bulk-upload endpoint, returning the creation summary.

### 4.2 React Components

The application has two shared React components. `Layout.js` acts as a page wrapper that provides a consistent structure across all pages, including a sidebar with application branding and navigation links to Create, Dashboard, Incidents, and Bulk Upload. It accepts a `title` prop for the page heading and a `children` prop for the main content. `ErrorBanner.js` is a simple component that displays error messages and accepts a `message` prop, used throughout the application to show feedback when API calls fail.

## 5. Application Workflows

---

## 5.1 Application Startup Workflow

1. Execute: `node server.js`
2. Import Express app from `app.js`
3. Import store functions from `incidents.store.js`
4. Call `initializeStore()`
  - └ Create data directory if needed
  - └ Attempt to read `incidents.json`
  - └ If file exists: parse JSON and load into memory
  - └ If file doesn't exist: create empty array
5. Start HTTP server on configured port
6. Log "Server running" message
7. Server ready to accept requests

## 5.2 Frontend Application Load Workflow

1. User navigates to `http://localhost:3000`
2. `Next.js` serves index page
3. User clicks "Dashboard"
4. Dashboard component mounts
5. `useEffect` triggers `loadIncidents()`
6. Fetch GET `/api/incidents?includeArchived=false`
7. Backend returns filtered incidents array
8. Component updates state with incidents
9. `useMemo` calculates statistics
10. Render KPI cards and status columns

## 5.3 Status Update Workflow

1. User opens incident detail page
2. Component fetches incident data
3. Component determines allowed next statuses
4. User selects new status from dropdown
5. User clicks "Update" button
6. `onClick` handler validates selection
7. Call `changeIncidentStatus(id, newStatus)`
8. API sends PATCH to `/api/incidents/:id/status`
9. Backend validates status transition
10. Backend updates status in memory

11. Backend saves to JSON file
12. Backend returns updated incident
13. Frontend updates local state
14. UI re-renders with new status

## 5.4 Archive Workflow

1. User viewing incident in OPEN or RESOLVED status
2. UI shows "Archive" button
3. User clicks "Archive"
4. onClick handler calls archiveIncident(id)
5. API sends POST to /api/incidents/:id/archive
6. Backend validates current status (must be OPEN or RESOLVED)
7. Backend changes status to ARCHIVED
8. Backend saves to JSON file
9. Backend returns archived incident
10. Frontend updates local state
11. UI re-renders showing ARCHIVED status
12. UI shows "Reset to Open" button instead

## 5.5 Reset Workflow

1. User viewing archived incident
2. UI shows "Reset to Open" button
3. User clicks "Reset to Open"
4. onClick handler calls resetIncident(id)
5. API sends POST to /api/incidents/:id/reset
6. Backend validates status is ARCHIVED
7. Backend changes status to OPEN
8. Backend saves to JSON file
9. Backend returns reset incident
10. Frontend updates local state
11. UI re-renders showing OPEN status
12. UI shows status update and archive options again

# Data Workflows

---

## Create Incident Data Flow

User Input → Frontend Validation → API Request → Backend Validation  
→

Store Creation → File Save → Response → UI Update

Details:

1. User fills form with title, description, category, severity
2. Frontend validates required fields
3. Frontend calls createIncident(payload)
4. API sends POST with JSON body
5. Backend validates using validateCreateIncident()
6. If valid: Backend calls createIncident()
7. Store generates UUID, adds OPEN status, adds timestamp
8. Store pushes to incidents array
9. Store writes array to incidents.json
10. Backend returns 201 with created incident
11. Frontend redirects to incident list

## **Read Incident Data Flow**

Page Load → API Request → Backend Query → Memory Lookup →  
Response → UI Render

Details:

1. Dashboard component mounts
2. Frontend calls listIncidents(includeArchived)
3. API sends GET /api/incidents?includeArchived=true/false
4. Backend calls listAll(includeArchived)
5. Store filters incidents array based on archived flag
6. Store returns filtered array
7. Backend sends JSON response
8. Frontend updates state
9. Component calculates statistics
10. UI renders KPI cards and columns

## **Update Incident Data Flow**

User Action → Frontend Validation → API Request → Backend Validation  
→

Store Update → File Save → Response → UI Update

Details:

1. User selects new status and clicks Update
2. Frontend validates selection exists
3. Frontend calls `changeIncidentStatus(id, status)`
4. API sends PATCH with status in body
5. Backend finds incident by ID
6. Backend validates transition using `validateStatusChange()`
7. If valid: Backend calls `updateStatus()`
8. Store finds incident, updates status property
9. Store writes array to `incidents.json`
10. Backend returns updated incident
11. Frontend updates local state
12. UI re-renders with new status

## Bulk Upload Data Flow

File Selection → Frontend Upload → Backend Parse → Validation Loop →  
Batch Creation → File Save → Summary Response → UI Update

Details:

1. User selects CSV file
2. User clicks Upload
3. Frontend creates `FormData` with file
4. Frontend calls `bulkUploadCsv(file)`
5. API sends POST with multipart form data
6. Multer middleware stores buffer in memory
7. Backend calls `parseCsvBuffer()`
8. Parser reads CSV, returns array of row objects
9. For each row:
  - └─ Validate using `validateCreateIncident()`
  - └─ If valid: create incident
  - └─ If invalid: increment skipped counter
10. Store saves all new incidents to file
11. Backend returns `{ totalRows, created, skipped }`
12. Frontend displays summary

## Archive/Reset Data Flow

User Action → API Request → Backend Validation → Store Update →  
File Save → Response → UI Update

Archive:

1. User clicks Archive button
2. Frontend calls archiveIncident(id)
3. API sends POST to /archive endpoint
4. Backend validates status is OPEN or RESOLVED
5. Backend calls archiveIncident()
6. Store changes status to ARCHIVED
7. Store saves to file
8. Backend returns archived incident
9. Frontend updates state
10. UI shows archived status and reset button

Reset:

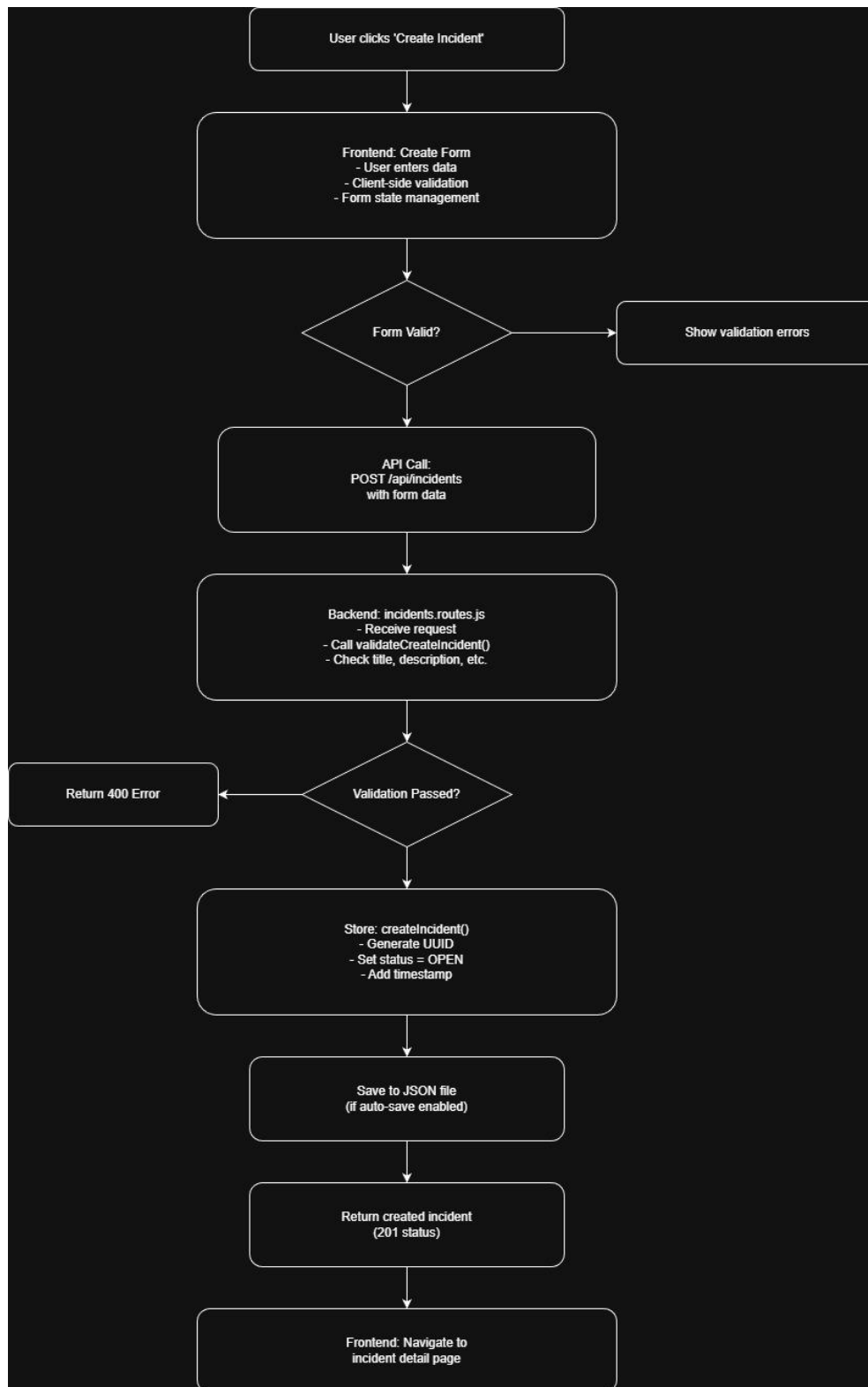
1. User clicks Reset button
2. Frontend calls resetIncident(id)
3. API sends POST to /reset endpoint
4. Backend validates status is ARCHIVED
5. Backend calls resetArchivedIncident()
6. Store changes status to OPEN
7. Store saves to file
8. Backend returns reset incident
9. Frontend updates state
10. UI shows OPEN status and normal workflow

## 6. System Flowcharts

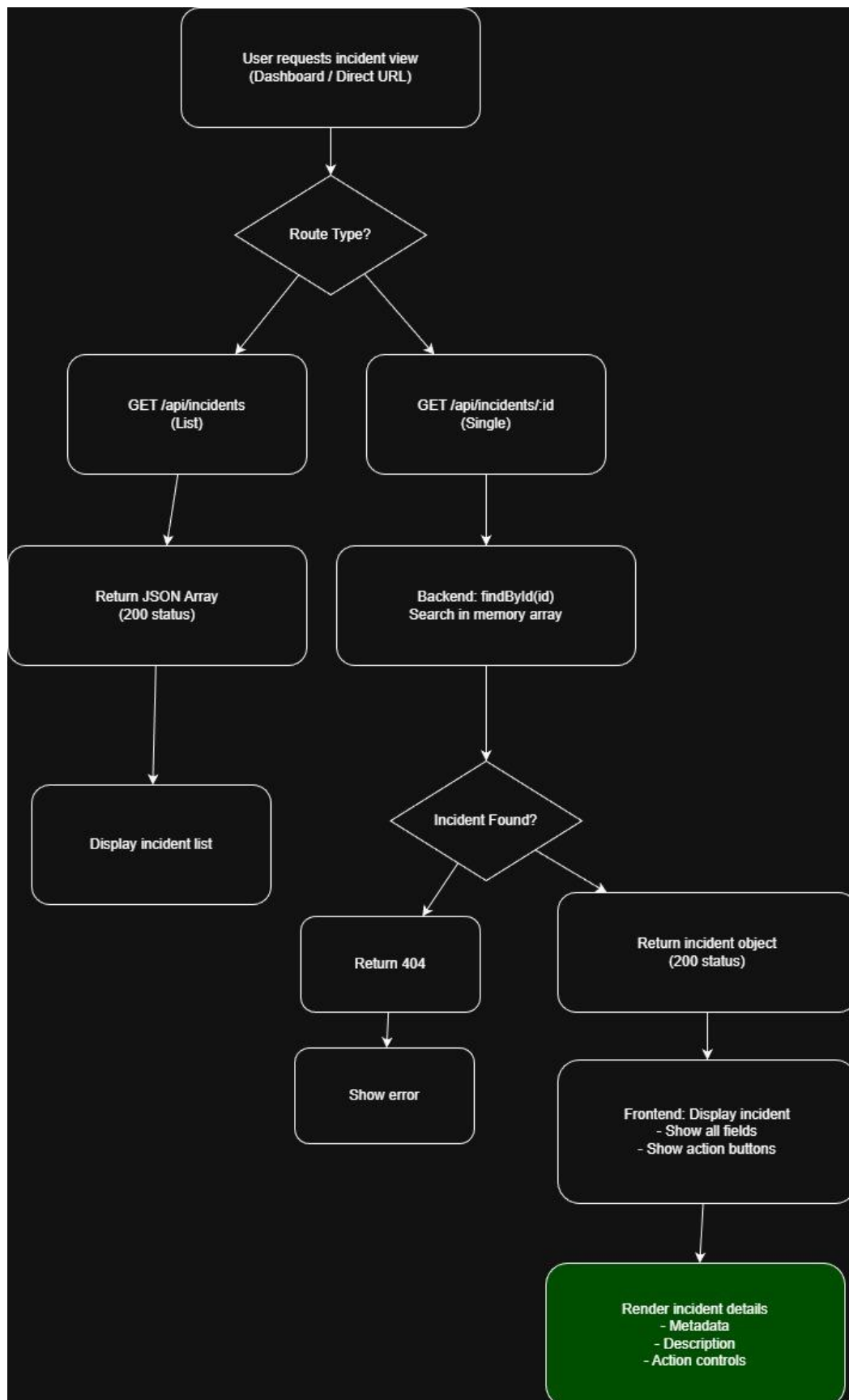
---

This section describes the detailed flows for the four main operations in the Incident Management System: Create, Read, Update Status, and Bulk Upload. Each flow covers the complete path from user action through frontend processing, API calls, backend validation, data operations, and response handling.

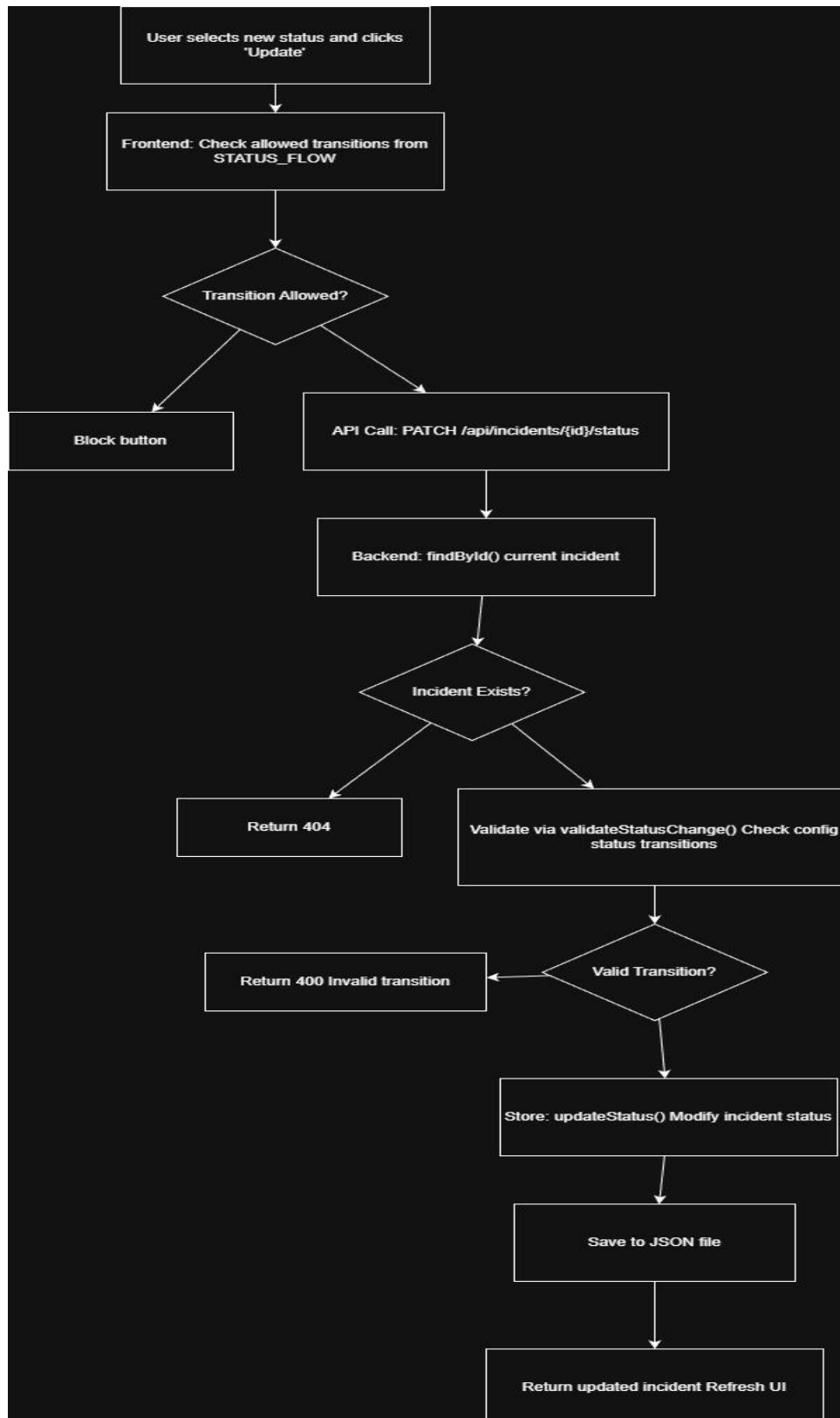
### 6.1 Create Incident Workflow



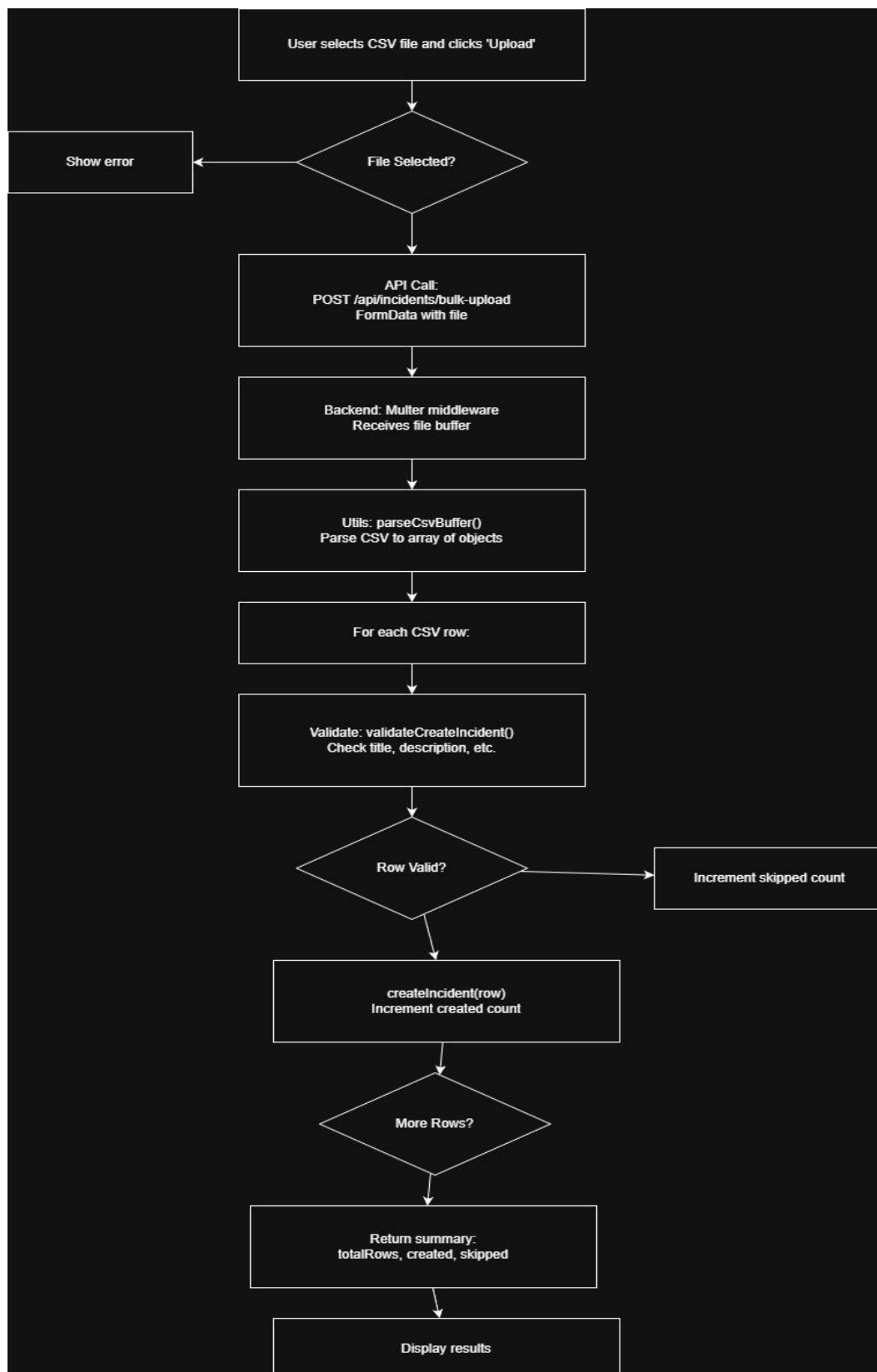
## 6.2 Read Incident Workflow



## 6.3 Update Incident Status Workflow



## 6.4 Bulk Upload CSV Workflow



## 7. Function Reference

---

This section provides a detailed reference of all major functions across the system, organized by module.

### 7.1 Backend Functions

#### Store Module (`incidents.store.js`)

**initializeStore()** → **Promise<void>**: Initializes the data store by loading incidents from the JSON file into memory. Creates the data file if it doesn't exist. Called once at startup before the server begins accepting requests.

**listAll(includeArchived: boolean)** → **Array<Incident>**: Returns all incidents from the in-memory array. When `includeArchived` is false (the default), archived incidents are filtered out. This is a synchronous in-memory read and does not access the file system.

**findById(id: string)** → **Incident | undefined**: Searches the incidents array for an incident with the specified UUID. Returns the incident object if found, or undefined if no match exists.

**createIncident(data: Object)** → **Promise<Incident>**: Creates a new incident with the provided data (title, description, category, severity). Automatically generates a UUID, sets status to OPEN, and adds a creation timestamp. Saves to the JSON file and returns the created incident.

**updateStatus(id: string, status: string)** → **Promise<Incident | null>**: Updates the status field of an existing incident after finding it by ID and modifying the in-memory record. Triggers a file save and returns the updated incident, or null if not found.

**archiveIncident(id: string)** → **Promise<Incident | null>**: Archives an incident by changing its status to ARCHIVED. Only works for incidents currently in OPEN or RESOLVED status. Returns the updated incident or null if not found.

**resetArchivedIncident(id: string)** → **Promise<Incident | null>**: Resets an ARCHIVED incident back to OPEN status. Only works for incidents with status ARCHIVED. Returns the updated incident or null if not found.

**save()** → **Promise<void>**: Manually triggers a save of the in-memory incidents array to the JSON file. Useful when auto-save is disabled and an explicit persist is needed.

#### Validation Module (`validate.js`)

**validateCreateIncident(body: Object)** → **{ok, errors, value}**: Validates incident creation data, checking that title is 5–200 characters, description is 10–2000 characters, and that category and severity are valid enumerated values. Returns an object with `ok` (boolean), `errors` (array of messages), and `value` (the cleaned data).

**validateStatusChange(current: string, next: string)** → **{ok, error, next}**: Validates a status transition against the allowed transitions defined in configuration. Returns `ok`, a descriptive error message if the transition is invalid, and the sanitized next status.

**validateArchive(currentStatus: string) → {ok, error}**: Validates whether an incident can be archived. Returns ok only if the current status is OPEN or RESOLVED, since only those states may be archived.

**validateReset(currentStatus: string) → {ok, error}**: Validates whether an incident can be reset. Returns ok only if the current status is ARCHIVED, as reset is only applicable to archived incidents.

## 7.2 Frontend Functions

### API Service (api.js)

**health() → Promise<{status}>**: Performs a GET /health request to the backend server. Returns an object with a status field indicating server health.

**listIncidents(includeArchived: boolean) → Promise<Array<Incident>>**: Retrieves all incidents by calling GET /api/incidents with the includeArchived query parameter. Returns an array of incident objects.

**getIncident(id: string) → Promise<Incident>**: Retrieves a single incident by UUID via GET /api/incidents/:id. Returns the incident object or throws an error if not found.

**createIncident(payload: Object) → Promise<Incident>**: Creates a new incident by sending a POST request to /api/incidents with a JSON payload containing title, description, category, and severity.

**changeIncidentStatus(id: string, status: string) → Promise<Incident>**: Updates an incident's status by sending a PATCH request to /api/incidents/:id/status with the new status in the request body.

**archiveIncident(id: string) → Promise<Incident>**: Archives an incident by sending a POST request to /api/incidents/:id/archive. Returns the updated incident with status ARCHIVED.

**resetIncident(id: string) → Promise<Incident>**: Resets an archived incident by sending a POST request to /api/incidents/:id/reset. Returns the updated incident with status OPEN.

**bulkUploadCsv(file: File) → Promise<{totalRows, created, skipped}>**: Uploads a CSV file for bulk incident creation by sending a POST request to /api/incidents/bulk-upload as a multipart form submission. Returns a summary of the upload operation.

## 8. API Endpoints Reference

---

This section provides a complete reference of all REST API endpoints, including request format, response format, status codes, and error handling details.

### GET /api/incidents

This endpoint lists all incidents. It accepts an optional boolean query parameter `includeArchived` (defaulting to `false`), and requires no request body. On success it returns HTTP 200 with an array of incident objects. Archived incidents are excluded by default unless `includeArchived=true` is specified. A server failure returns HTTP 500. Example: `GET /api/incidents?includeArchived=true`

## **GET /api/incidents/:id**

This endpoint retrieves a specific incident by its UUID, supplied as a URL path parameter. No request body is needed. On success it returns HTTP 200 with the incident object. If no incident with the given ID exists, HTTP 404 is returned. A server failure returns HTTP 500. Example: `GET /api/incidents/550e8400-e29b-41d4-a716-446655440000`

## **POST /api/incidents**

This endpoint creates a new incident. The request body must be a JSON object containing all four required fields: `title` (5–200 characters), `description` (10–2000 characters), `category` (one of `IT`, `SAFETY`, `FACILITIES`, or `OTHER`), and `severity` (one of `LOW`, `MEDIUM`, or `HIGH`). On success it returns HTTP 201 with the created incident object including its auto-generated `id`, initial status of `OPEN`, and `reportedAt` timestamp. A validation failure returns HTTP 400, and a server error returns HTTP 500.

## **PATCH /api/incidents/:id/status**

This endpoint updates an incident's status. The URL path parameter is the incident UUID, and the request body must be a JSON object with a `status` field containing the desired new status. Allowed transitions are: `OPEN` to `INVESTIGATING` or `ARCHIVED`, `INVESTIGATING` to `RESOLVED`, and `RESOLVED` to `ARCHIVED`. The `ARCHIVED` to `OPEN` transition is only permitted via the `reset` endpoint, not this one. On success it returns HTTP 200 with the updated incident object. HTTP 400 is returned if the transition is invalid, HTTP 404 if the incident is not found, and HTTP 500 for server errors.

## **POST /api/incidents/:id/archive**

This endpoint archives an incident. The URL path parameter is the incident UUID and no request body is required. The incident must currently be in `OPEN` or `RESOLVED` status to be eligible for archiving. On success it returns HTTP 200 with the incident object showing status `ARCHIVED`. HTTP 400 is returned if the current status does not allow archiving, HTTP 404 if the incident is not found, and HTTP 500 for server errors.

## **POST /api/incidents/:id/reset**

This endpoint resets an archived incident back to `OPEN`. The URL path parameter is the incident UUID and no request body is required. The incident must currently be in `ARCHIVED` status for this operation to succeed. On success it returns HTTP 200 with

the incident object showing status OPEN. HTTP 400 is returned if the incident is not in ARCHIVED status, HTTP 404 if not found, and HTTP 500 for server errors.

## **POST /api/incidents/bulk-upload**

This endpoint handles bulk creation of incidents from a CSV file. The request must use the Content-Type of multipart/form-data, with the CSV file attached under the field name file. The CSV format requires a header row with the columns title, description, category, and severity, with each subsequent row representing one incident to create. The backend validates each row individually: valid rows are created as incidents and counted, while invalid rows are skipped and counted separately. On success it returns HTTP 200 with a JSON object containing totalRows (the number of rows in the CSV), created (the number successfully created), and skipped (the number that failed validation). HTTP 400 is returned if no file is provided, and HTTP 500 for server errors.