Traffic News App
3-Layered Architecture Analysis

CPS731 - Software Engineering I (Team 15)

Arshia Rahim (500994106) Vraj Patel (501243245) Lei Singha (500960134)

Prepared for CPS731 – F25, Section 021-022
Instructor: Dr. Soheila Bashardoust-Tajali, TA: Eamon Earl

November 15, 2025

Abstract

This document presents a comprehensive architectural analysis of the Traffic News App, demonstrating the transformation from a flat, unstructured class design to a well-organized 3-layered architecture. The analysis includes detailed class diagrams for both approaches, eight sequence diagrams showing component interactions for all major use cases, identification of problems with flat architecture, and explanation of benefits achieved through proper layering. This architectural pattern provides clear separation of concerns, improved maintainability, enhanced testability, and better scalability for the Traffic News App system.

Contents

1 Introduction

1.1 Purpose

This document analyzes the architectural design of the Traffic News App, a web-based system that enables users to view and report real-time traffic incidents. The primary objective is to demonstrate the application of the **3-Layered Architecture Pattern** to improve system organization, maintainability, and scalability.

1.2 System Overview

The Traffic News App is designed to improve road safety awareness by allowing users to:

- View live traffic incidents (accidents, construction, closures, hazards)
- Report new incidents with validation and confirmation
- Filter and search incidents by type, severity, and location
- Visualize incidents on an interactive map
- Manage saved routes and receive notifications for nearby incidents
- Access the system offline with automatic synchronization

1.3 Document Structure

This analysis is organized into the following sections:

1. **Before:** Flat class structure and its problems
2. **After:** 3-Layered architecture solution
3. **Sequence Diagrams:** Detailed component interactions for all use cases
4. **Benefits:** Advantages of the layered approach
5. **Traceability:** Mapping of layers to requirements
6. **Implementation:** Technology stack and best practices
7. **Conclusion:** Summary and recommendations

2 Before: Flat Class Structure

2.1 Overview

In the initial design, all system components (C01–C12) were organized at the same architectural level without clear separation of concerns. This flat structure treats presentation, business logic, and data access components equally, resulting in a tangled web of dependencies.

2.2 Class Diagram

This section presents two views of the flat class structure:

- **Simplified View** (Figure 1): High-level overview showing components and key relationships
- **Detailed View** (Figure 11): Complete implementation with attributes and methods

2.2.1 Simplified View

Figure 1 provides a clear overview of the flat structure, showing how all components exist at the same level without architectural boundaries.

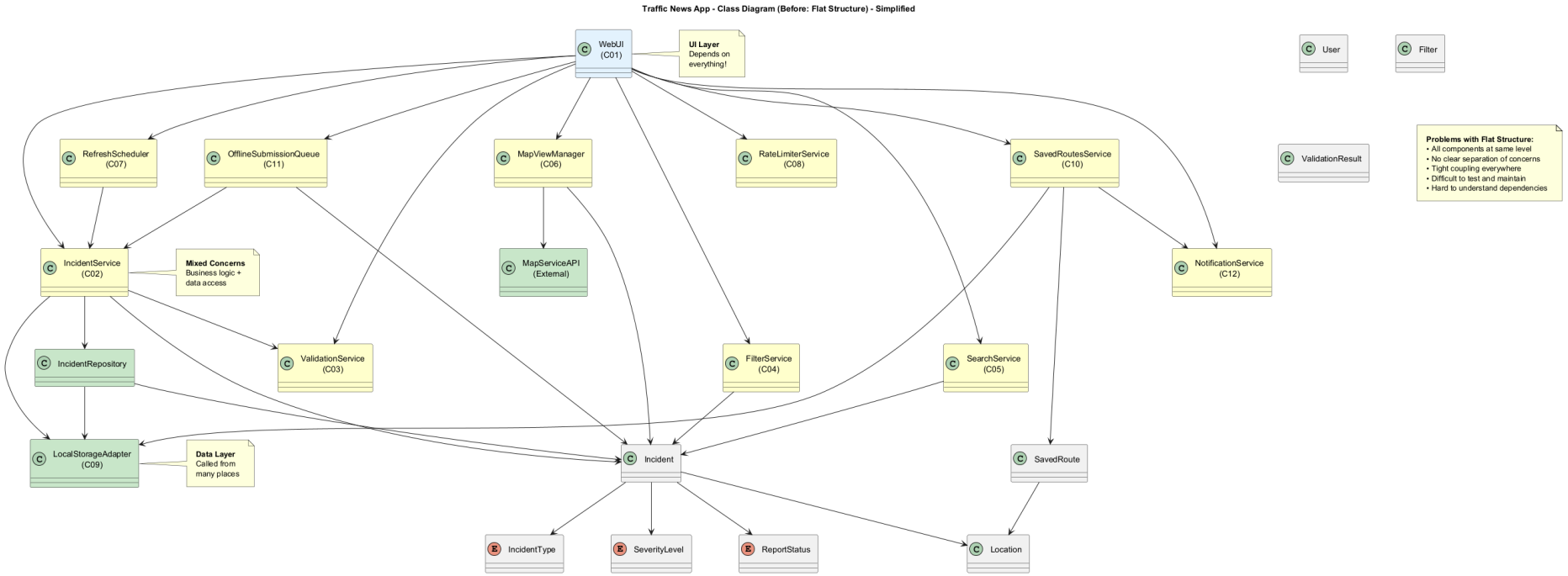

Figure 1: Class Diagram – Before (Flat Structure) – Simplified View

2.2.2 Detailed View

For reference, Figure 11 shows the complete class structure with all attributes, methods, and detailed relationships. This detailed view is included in the appendix for implementation reference.

Note: See Appendix A for the detailed class diagram with full implementation specifications.

2.3 Key Characteristics

The flat architecture exhibits the following characteristics:

- **No Layer Separation:** All 12 components (C01–C12) exist at the same level
- **Mixed Concerns:** Presentation, business logic, and data access are intermingled
- **Bidirectional Dependencies:** Components reference each other in both directions
- **Tight Coupling:** Changes to one component ripple through many others
- **Unclear Responsibilities:** Component roles and boundaries are ambiguous

2.4 Problems with Flat Structure

2.4.1 1. Tight Coupling

Components are tightly coupled with multiple interdependencies. For example:

- `WebUI` directly depends on 10+ other components
- `IncidentService` is accessed by UI, schedulers, and offline queue
- `ValidationService` is called from multiple unrelated components

Impact: Changes to one component require modifications to many others, increasing maintenance cost and risk of introducing bugs.

2.4.2 2. Difficult to Test

Without clear boundaries, unit testing becomes challenging:

- Cannot test business logic independently from UI
- Cannot mock data layer without affecting business logic
- Integration tests become overly complex

Impact: Poor test coverage, harder to identify bugs, longer testing cycles.

2.4.3 3. Poor Maintainability

The lack of structure makes the system hard to understand and modify:

- New developers struggle to understand component relationships
- No clear entry points for implementing new features
- Difficult to locate where specific functionality resides

Impact: Increased onboarding time, slower feature development, higher bug rates.

2.4.4 4. Limited Scalability

The flat structure hinders system evolution:

- Cannot easily replace data storage mechanism
- Cannot swap UI framework without affecting business logic
- Cannot distribute components across multiple servers

Impact: System cannot adapt to changing requirements or scale with user growth.

2.4.5 5. Violation of Software Engineering Principles

The flat architecture violates fundamental principles:

- **Separation of Concerns:** Presentation, logic, and data are mixed
- **Single Responsibility:** Components have multiple, unrelated responsibilities
- **Dependency Inversion:** High-level modules depend on low-level details

3 After: 3-Layered Architecture

3.1 Overview

The 3-Layered Architecture organizes the system into three distinct layers, each with well-defined responsibilities and clear boundaries. This architectural pattern is one of the most fundamental and widely-used software design patterns for complex applications.

3.2 Class Diagram

This section presents two views of the 3-layered architecture:

- **Simplified View** (Figure 2): Clean architectural overview showing layer organization
- **Detailed View** (Figure 16): Complete implementation specifications

3.2.1 Simplified View

Figure 2 clearly illustrates the reorganized class structure with distinct layer boundaries and one-way dependencies.

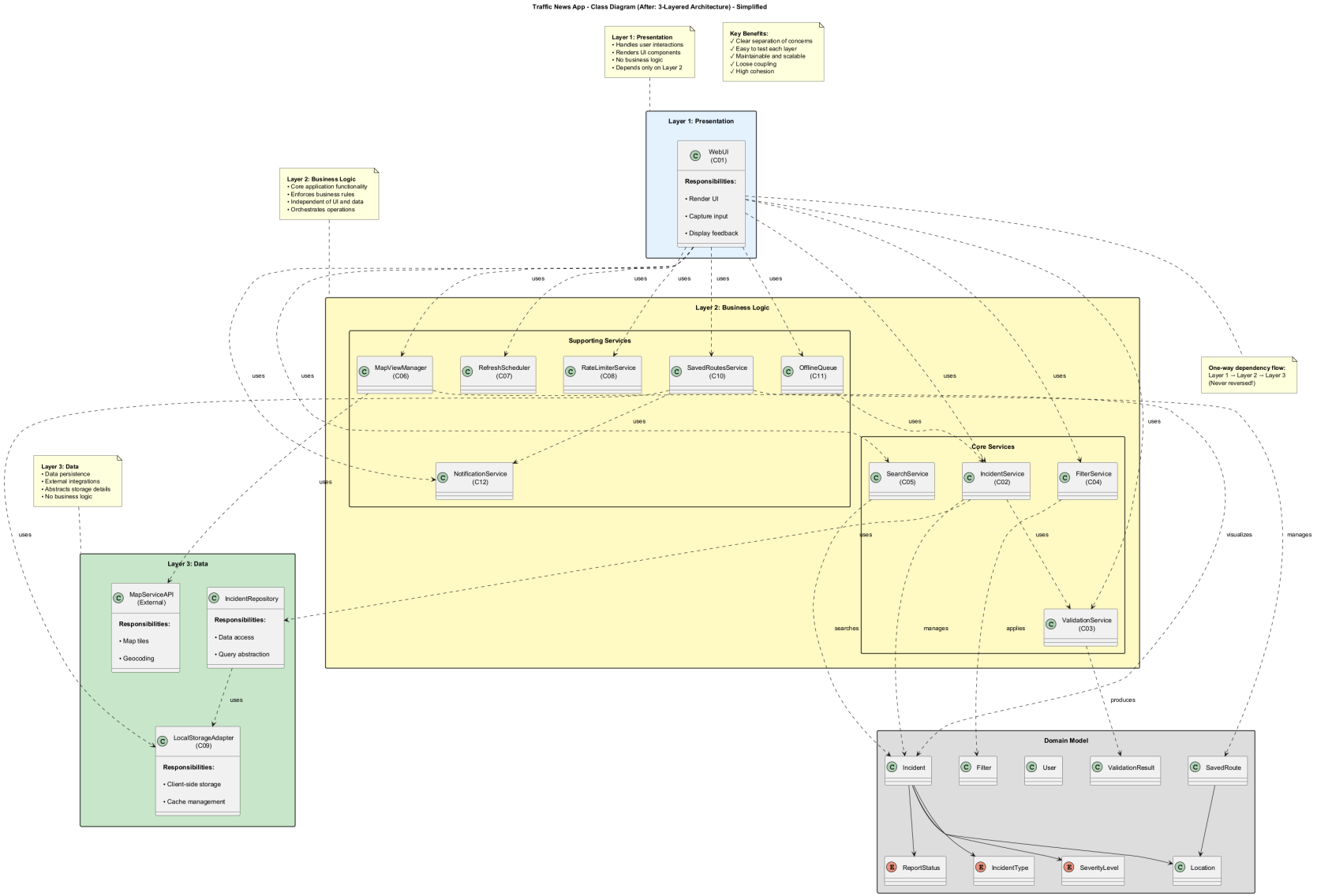

Figure 2: Class Diagram – After (3-Layered Architecture) – Simplified View

3.2.2 Detailed View

The detailed implementation view with all class members is provided in Appendix B for development reference.

Note: See Appendix B for the detailed class diagram with complete method signatures and attributes.

3.3 Architecture Description

3.3.1 Layer 1: Presentation Layer (View)

Purpose: Handles all user interactions and visual presentation.

Responsibilities:

- Render user interface components (lists, maps, forms)
- Capture user input (clicks, form submissions, gestures)
- Display system feedback (errors, confirmations, notifications)
- Delegate business operations to the Business Logic Layer

Components:

- `WebUI` (C01) – Main UI controller

Design Rules:

- Never contains business logic or data access code
- Only communicates with Business Logic Layer (never Data Layer)
- Presents data but does not process or validate it

3.3.2 Layer 2: Business Logic Layer (Application/Service)

Purpose: Implements core application functionality and enforces business rules.

Responsibilities:

- Execute business operations (CRUD, filtering, searching)
- Validate input data and enforce constraints
- Coordinate between presentation and data layers
- Implement application workflows (reporting, moderation)
- Manage application state and caching

Components:

- `IncidentService` (C02) – Core incident management
- `ValidationService` (C03) – Input validation and error handling
- `FilterService` (C04) – Type and severity filtering logic
- `SearchService` (C05) – Keyword and location search
- `MapViewManager` (C06) – Map rendering and pin management
- `RefreshScheduler` (C07) – Auto-refresh timing logic
- `RateLimiterService` (C08) – Submission throttling
- `SavedRoutesService` (C10) – Route management
- `OfflineSubmissionQueue` (C11) – Offline synchronization
- `NotificationService` (C12) – Alert and notification logic

Design Rules:

- Never directly accesses UI components
- Uses Data Layer for persistence (no direct database/storage access)
- Remains independent of presentation and data technologies

3.3.3 Layer 3: Data Layer (Persistence)

Purpose: Manages data storage, retrieval, and external API integration.

Responsibilities:

- Persist and retrieve application data
- Interface with external services (Map API)
- Manage local storage and caching
- Abstract data access implementation details

Components:

- `LocalStorageAdapter` (C09) – Client-side storage
- `IncidentRepository` – Data access abstraction
- `MapServiceAPI` – External map service integration

Design Rules:

- Never calls business logic or UI components
- Provides technology-agnostic interface to upper layers
- Handles all data source complexity (REST, local storage, external APIs)

3.4 Dependency Flow

The 3-Layered Architecture enforces a strict **one-way dependency rule**:

Presentation Layer → Business Logic Layer → Data Layer

Key Principles:

- Each layer only depends on the layer directly below it
- Lower layers never depend on upper layers
- Business Logic Layer acts as a bridge between UI and data
- Changes to lower layers do not affect upper layers

4 Sequence Diagrams

4.1 Overview

This section presents eight sequence diagrams (SD01–SD08) that demonstrate how the 3-layered architecture supports each major use case. Each sequence diagram shows the detailed interactions between components across the Presentation, Business Logic, and Data layers, illustrating the one-way dependency flow and clear separation of concerns achieved through proper layering.

4.2 Mapping to Use Cases

Each sequence diagram corresponds to a specific activity diagram (AD) from Phase I and covers one or more use cases:

Table 1: Sequence Diagram to Use Case Mapping

SD	Main UC	Description
SD01	UC01	View, filter, search, and sort incidents
SD02	UC05	Interactive map visualization with pins
SD03	UC08	Manage saved routes and notifications
SD04	UC10	Adjust auto-refresh interval settings
SD05	UC12	Report new traffic incidents
SD06	UC13	Edit or withdraw submitted reports
SD07	UC14	View submission status and feedback
SD08	UC20	Moderate pending incident reports

4.3 SD01: View Incidents

Primary Use Case: UC01 – View Incidents
Related Use Cases: UC02 (Filter), UC03 (Search), UC04 (Clear Filters), UC11 (Sort)

Description: This sequence diagram illustrates the core workflow for viewing traffic incidents. It demonstrates how the WebUI (Presentation Layer) coordinates with multiple services in the Business Logic Layer (IncidentService, FilterService, SearchService, RefreshScheduler) and how those services interact with the Data Layer (IncidentRepository) to retrieve and present incident data.

Key Interactions:

- User triggers incident list display
- WebUI requests data from IncidentService
- IncidentService retrieves from IncidentRepository
- User applies filters via FilterService

- User performs searches via SearchService
- RefreshScheduler automatically updates data periodically

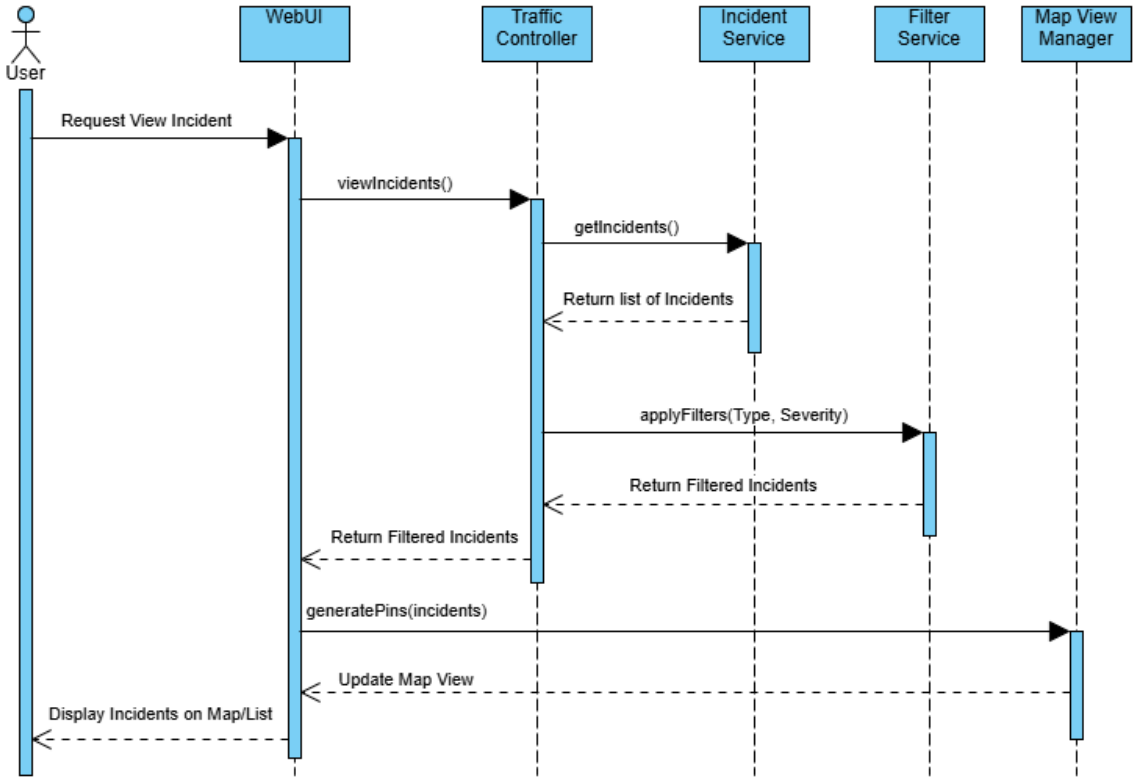

Figure 3: SD01: View Incidents – Shows filtering, searching, and auto-refresh interactions

4.4 SD02: Map Visualization

Primary Use Case: UC05 – Map Visualization
Related Use Cases: UC06 (View Details), UC19 (Fetch Map Tiles)

Description: This diagram shows how the map-based incident visualization is implemented. It demonstrates the interaction between MapViewManager (Business Logic Layer), the external MapServiceAPI (Data Layer), and how incident pins are rendered on the map with popup details.

Key Interactions:

- User switches to map view
- MapViewManager requests map tiles from external API
- Incidents are plotted as pins on the map
- User clicks pin to view incident details
- Map updates when user pans or zooms
- External MapServiceAPI provides geocoding services

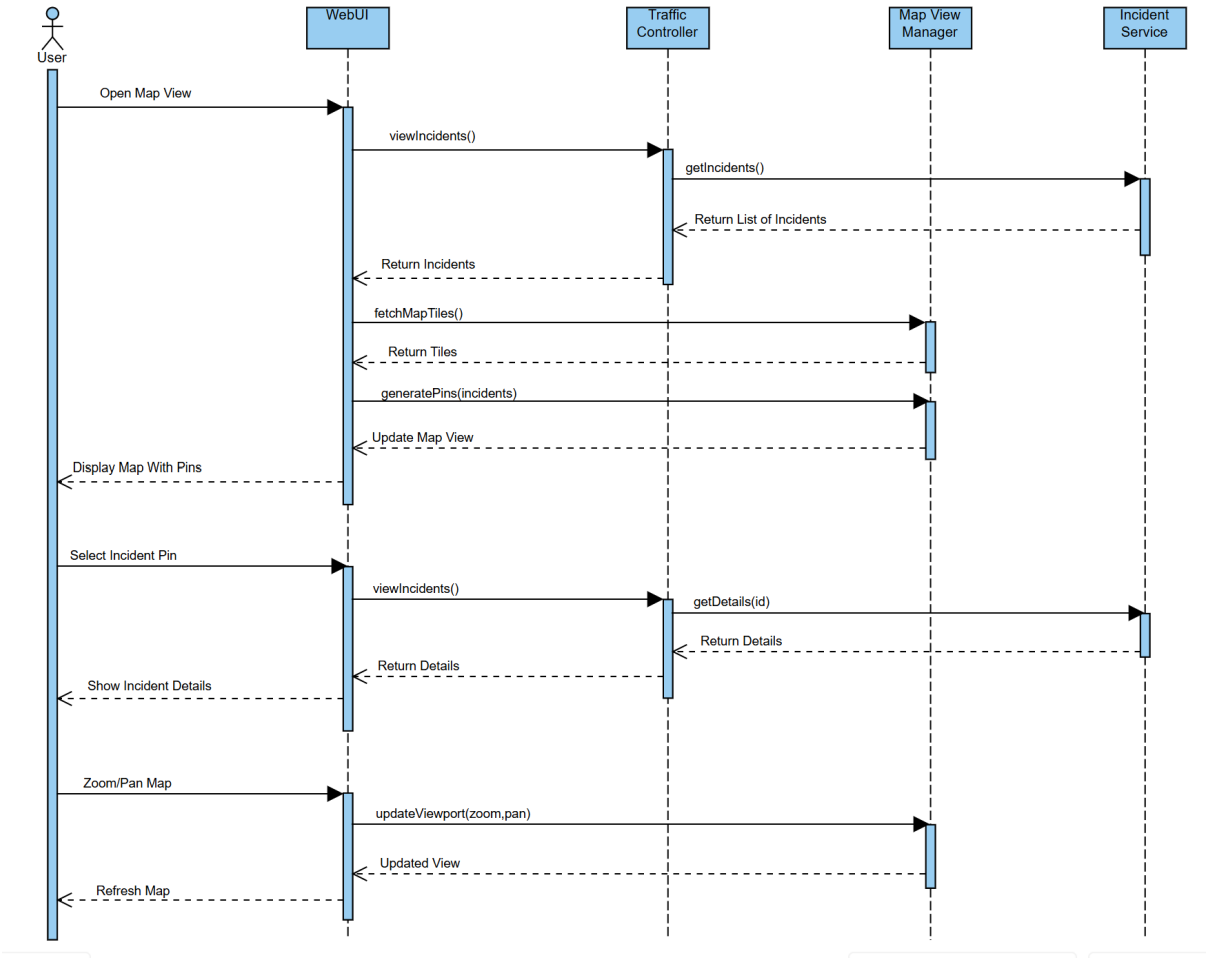

Figure 4: SD02: Map Visualization – External API integration and pin management

4.5 SD03: Manage Saved Routes

Primary Use Case: UC08 – Manage Saved Routes
Related Use Cases: UC07 (Save Route), UC09 (Receive Notifications)

Description: This sequence diagram illustrates how users can save frequently traveled routes and receive notifications when incidents occur nearby. It shows the coordination between SavedRoutesService, LocalStorageAdapter for persistence, and NotificationService for alerts.

Key Interactions:

- User saves a new route or area
- SavedRoutesService validates and stores via LocalStorageAdapter
- User enables notifications for a route
- IncidentService monitors for nearby incidents
- NotificationService triggers alerts when incidents detected
- User can delete saved routes

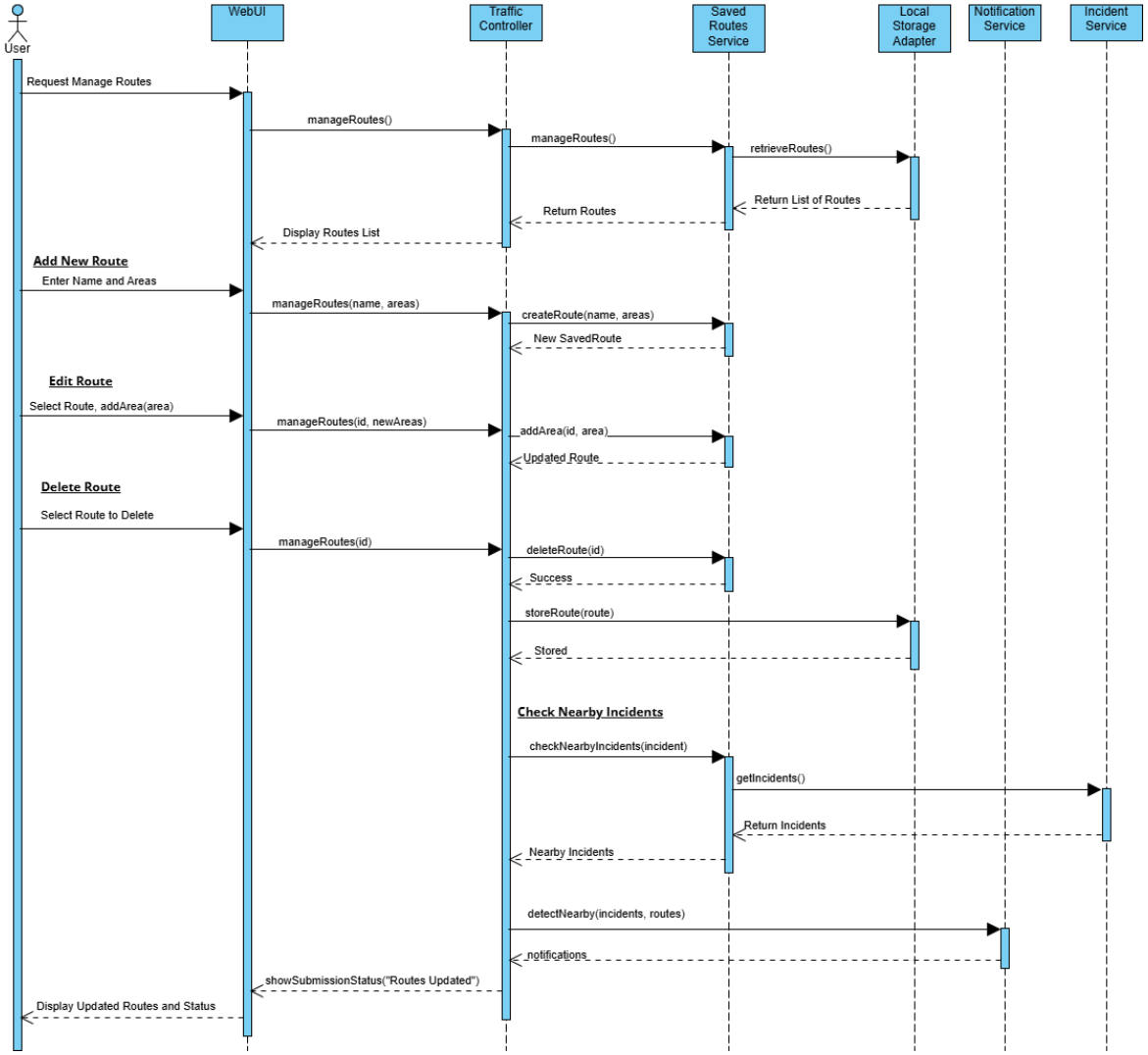

Figure 5: SD03: Manage Saved Routes – Route persistence and notification setup

4.6 SD04: Adjust Refresh Interval

Primary Use Case: UC10 – Adjust Refresh Interval

Description: This diagram shows how users can customize the auto-refresh interval for incident data. It demonstrates the RefreshScheduler lifecycle management (stop, update, restart) and how user preferences are persisted.

Key Interactions:

- User opens settings panel
- System displays current refresh interval
- User selects new interval (5–120 seconds)
- ValidationService ensures value is within valid range
- RefreshScheduler is stopped and reconfigured
- LocalStorageAdapter saves user preference
- RefreshScheduler restarts with new interval

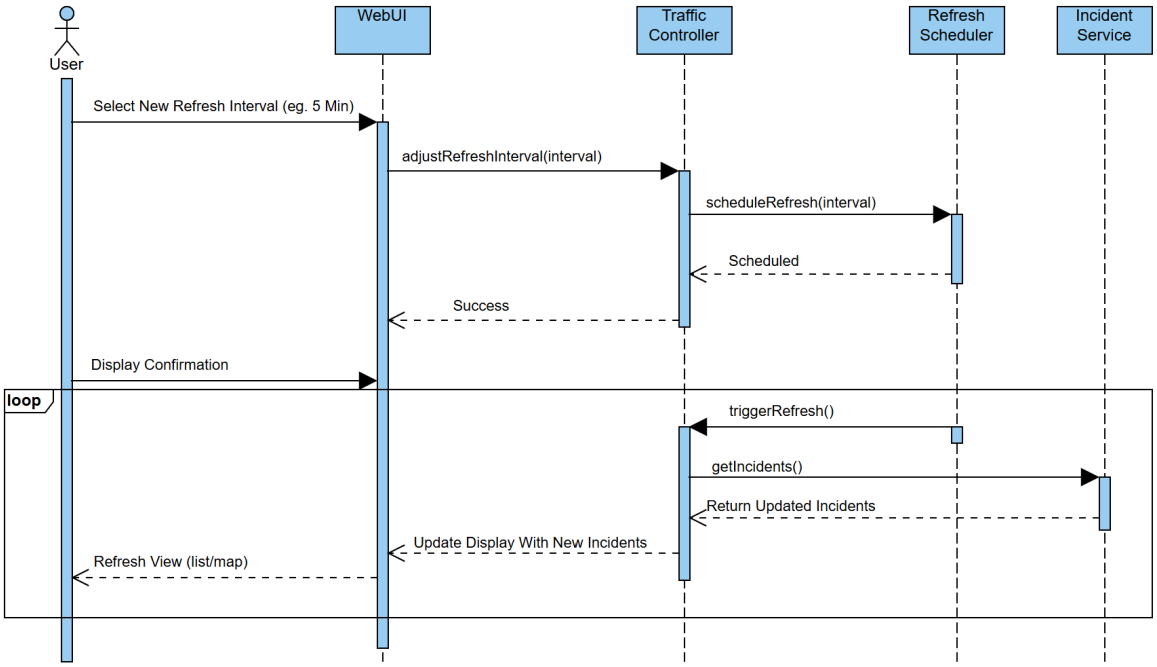

Figure 6: SD04: Adjust Refresh Interval – Scheduler configuration and preference storage

4.7 SD05: Report Incident

Primary Use Case: UC12 – Report Incident
Related Use Cases: UC15 (Validate), UC16 (Rate Limit), UC17 (Geocode), UC18 (Offline Queue)

Description: This is one of the most complex sequence diagrams, showing the complete workflow for reporting a new traffic incident. It includes rate limiting, validation, geocoding, and offline submission queue handling for when the user has no internet connection.

Key Interactions:

- User clicks "Report Incident" button
- WebUI displays incident report form
- User fills in required fields (type, severity, location, description)
- RateLimiterService checks if user exceeded submission limit (5/hour)
- ValidationService validates all form fields
- MapViewManager geocodes location text to coordinates
- If online: IncidentRepository saves immediately
- If offline: OfflineSubmissionQueue stores for later sync
- System confirms submission to user
- When connection restored, queue automatically submits pending reports

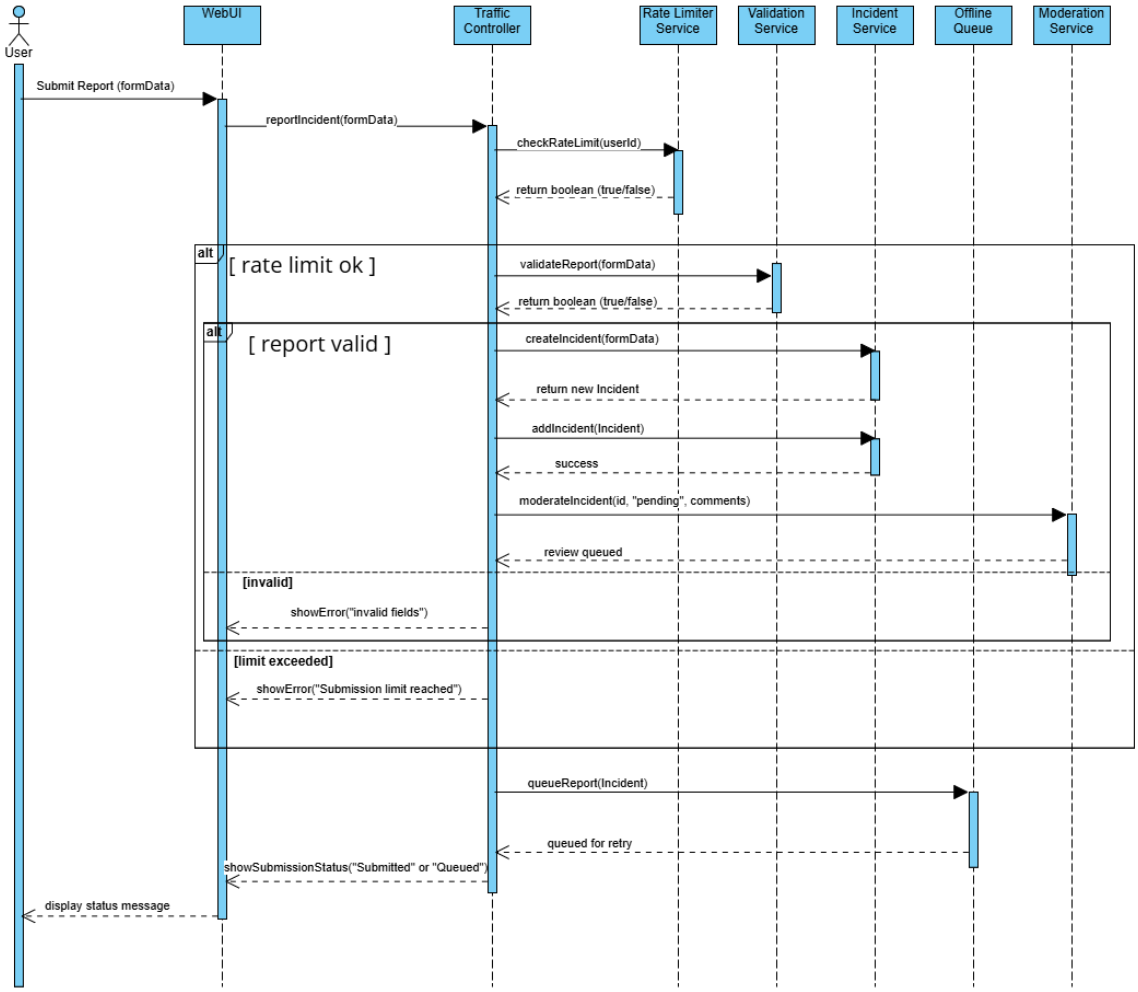

Figure 7: SD05: Report Incident – Rate limiting, validation, and offline queue handling

4.8 SD06: Edit or Withdraw Report

Primary Use Case: UC13 – Edit or Withdraw Report
Related Use Cases: UC15 (Validate)

Description: This diagram shows how users can modify or delete their previously submitted incident reports. It includes ownership verification to ensure users can only edit their own reports, re-validation on edits, and status change logic.

Key Interactions:

- User opens "My Reports" section
- IncidentService retrieves user's submitted reports
- User selects a report to edit
- System verifies ownership (users can only edit their own reports)
- WebUI pre-fills form with existing data
- User modifies fields
- ValidationService re-validates changed data
- IncidentRepository updates report (status changes to "Pending" for re-moderation)
- Alternatively, user can withdraw (delete) report
- System confirms action

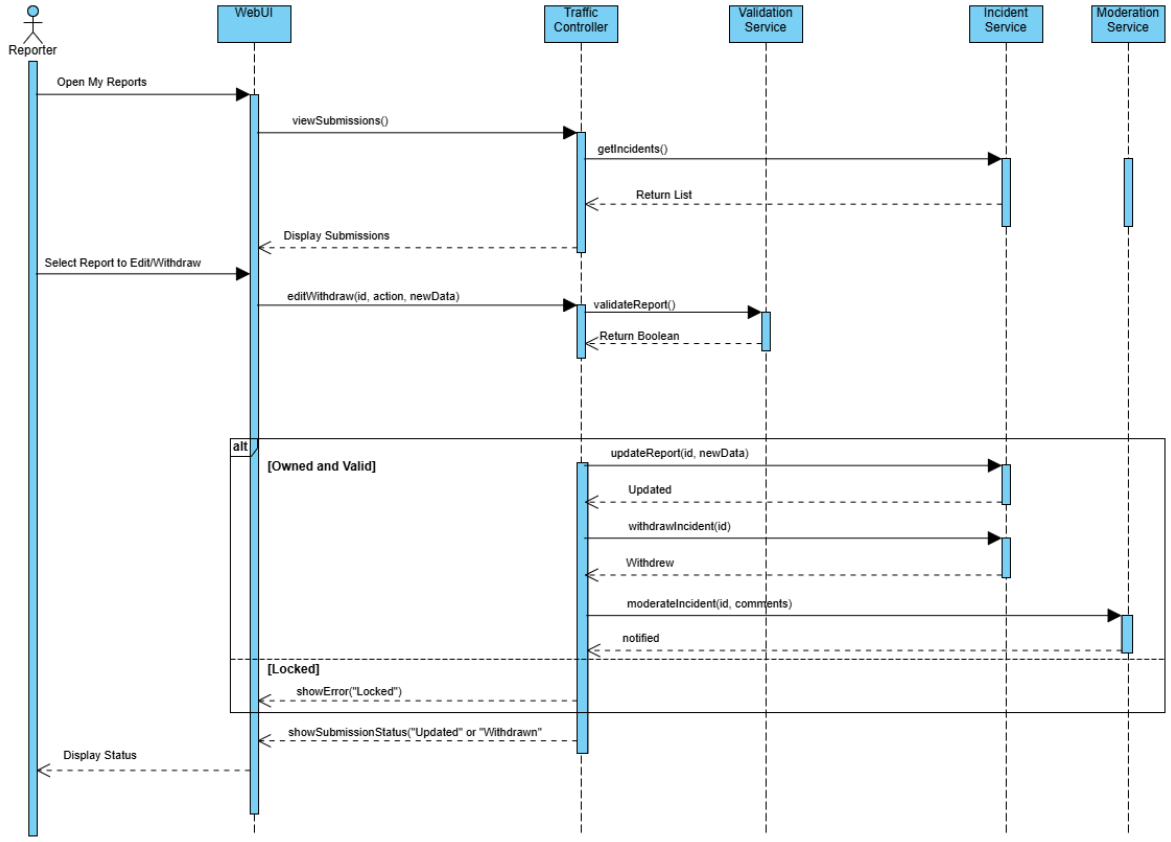

Figure 8: SD06: Edit or Withdraw Report – Ownership verification and re-validation

4.9 SD07: View Submission Status

Primary Use Case: UC14 – View Submission Status

Description: This sequence diagram illustrates how users can track the status of their submitted incident reports. It shows the different status states (Pending, Approved, Rejected, Needs Changes) and how moderator feedback is displayed to users.

Key Interactions:

- User navigates to "My Reports" section
- IncidentService retrieves user's reports with status information
- WebUI displays reports with color-coded status icons:
  - **Approved** – Green checkmark + publication timestamp
  - **Pending** – Yellow hourglass + estimated review time
  - **Rejected** – Red X + rejection reason from moderator
  - **Needs Changes** – Orange warning + specific feedback
- User clicks report for detailed status information
- System shows moderator notes and feedback
- User can filter reports by status
- NotificationService alerts user when status changes

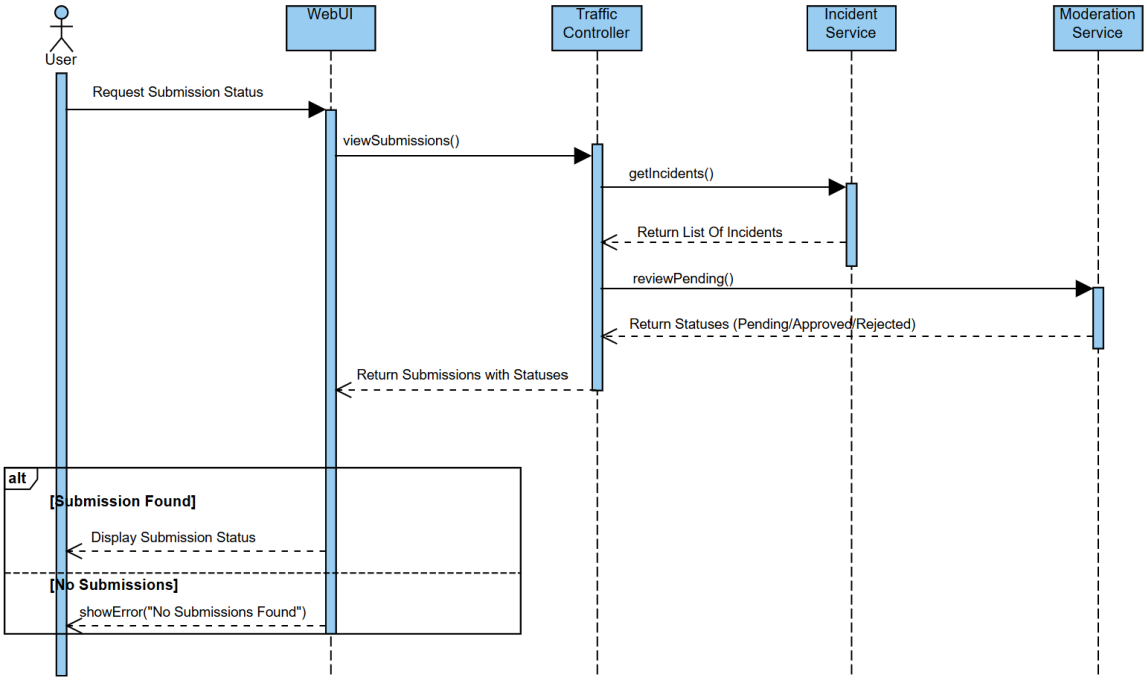

Figure 9: SD07: View Submission Status – Status tracking and moderator feedback display

4.10 SD08: Moderate Incidents

Primary Use Case: UC20 – Moderate Incidents
Related Use Cases: UC15 (Validate)

Description: This diagram shows the moderation workflow where authorized moderators review pending incident reports and decide whether to approve, reject, or request changes. This is a critical quality control process to prevent spam and ensure data accuracy.

Key Interactions:

- Moderator authenticates with admin credentials
- System displays queue of pending incident reports
- Moderator selects a report to review
- IncidentService retrieves full report details
- ValidationService automatically flags potential issues
- Moderator makes decision:
  - **Approve** – Report becomes publicly visible
  - **Reject** – Report removed (reason required)
  - **Request Changes** – Reporter notified with specific feedback
- IncidentRepository updates report status
- NotificationService alerts original reporter of decision
- System logs moderation action for audit trail
- Moderator can process reports in bulk for efficiency

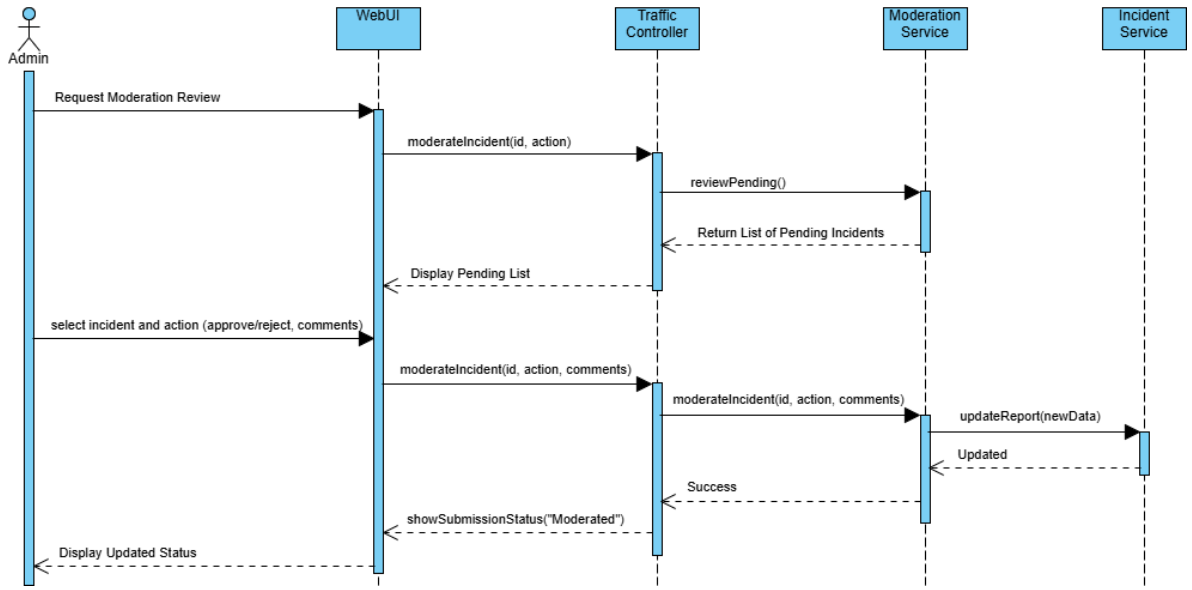

Figure 10: SD08: Moderate Incidents – Admin moderation workflow with decision branches

4.11 Sequence Diagram Analysis

4.11.1 Architectural Validation

The sequence diagrams validate that the 3-layered architecture is correctly implemented:

Dependency Flow Verification:

- All diagrams show **one-way dependencies**: Presentation → Business Logic → Data
- No lower layers call upper layers (no circular dependencies)
- Business Logic Layer acts as intermediary between UI and data

Layer Responsibilities Confirmed:

- **Presentation Layer (WebUI):** Only handles user interaction, no business logic
- **Business Logic Layer (Services):** Implements all business rules, validation, coordination
- **Data Layer (Repository/API):** Only handles persistence and external integration

4.11.2 Design Patterns in Sequence Diagrams

The sequence diagrams illustrate several design patterns in action:

1. Repository Pattern (SD01, SD05, SD06, SD08):

- IncidentRepository abstracts data access
- Business Logic services call repository methods (fetch, persist, update)
- Implementation details of storage hidden from upper layers

2. Service Pattern (All SDs):

- Each business capability encapsulated in dedicated service
- Services provide cohesive, reusable operations
- Clear interfaces between components

3. Adapter Pattern (SD02, SD05):

- MapServiceAPI adapts external map service to internal interface
- LocalStorageAdapter wraps browser storage API
- Shields business logic from external API changes

4. Facade Pattern (All SDs):

- WebUI acts as facade, simplifying access to complex subsystems
- Coordinates multiple services for single user action
- Hides complexity from user

5. Queue Pattern (SD05):

- OfflineSubmissionQueue handles asynchronous operations
- Buffers requests when network unavailable
- Automatic retry mechanism

4.11.3 Traceability: Requirements to Implementation

The sequence diagrams provide complete traceability from Phase I requirements to implementation:

```
Functional Requirement → Use Case → Activity Diagram → Sequence Diagram → Code
```

Example Traceability Chain:

- **FR12:** "Users shall be able to view incident details on map"
- → **UC06:** View Incident Details
- → **AD02:** Map Visualization activity flow
- → **SD02:** Detailed component interactions
- → **Implementation:** `MapViewManager.showPopup(incidentId)`

This complete traceability ensures all requirements are implemented and testable.

5 Benefits and Justification

The 3-Layered Architecture addresses the problems identified in the flat structure (Section 2.3) and provides significant advantages for the Traffic News App:

5.1 Key Benefits

1. Separation of Concerns: Each layer has a single, well-defined responsibility (Presentation for UI, Business Logic for rules/workflows, Data for persistence), enabling developers to work independently on each layer without detailed knowledge of the others.

2. Improved Testability: Clear layer boundaries enable comprehensive testing strategies: unit tests for individual services with mocked dependencies, integration tests for layer interactions, UI tests with mocked services, and end-to-end tests across all layers. This results in higher test coverage and easier bug isolation.

3. Enhanced Maintainability: The layered structure confines modifications to specific layers, provides clear entry points for new features, reduces complexity by breaking the system into manageable parts, and makes the architecture self-documenting. This leads to faster bug fixes and reduced technical debt.

4. Better Scalability: The architecture supports horizontal scaling (Business Logic Layer on multiple servers), technology replacement (swap UI or database independently), easy feature addition (new services integrate seamlessly), and independent performance optimization per layer.

5. Reusability and Team Collaboration: Business Logic services can support multiple interfaces (web, mobile, API), promoting code reuse and consistent behavior. Teams can work in parallel on different layers, coordinating only through well-defined interfaces, resulting in faster development and fewer conflicts.

6 Traceability Matrix

6.1 Layer to Component Mapping

Table 2 maps each system component to its architectural layer.

Table 2: Component Assignment to Layers

Layer	ID	Component Name
1*Presentation	C01	WebUI
2*Business Logic	C02	IncidentService
	C03	ValidationService
	C04	FilterService
	C05	SearchService
	C06	MapViewManager
	C07	RefreshScheduler
	C08	RateLimiterService
	C10	SavedRoutesService
	C11	OfflineSubmissionQueue
	C12	NotificationService
3*Data	C09	LocalStorageAdapter
	—	IncidentRepository
	—	MapServiceAPI (External)

6.2 Use Case Support by Layer

Table 3 shows which layers support each use case, with sub-use cases marked as s_uc.

Table 3: Use Case Support by Layer

UC	Use Case Name	Presentation	Business	Data
UC01	View Incidents	X	X	X
s_uc02	Filter Incidents	X	X	
s_uc03	Search Incidents	X	X	
s_uc04	Clear Filters	X	X	
UC05	Map Visualization	X	X	X
s_uc06	View Incident Details	X	X	
s_uc07	Save Route / Area	X	X	X
UC08	Manage Saved Routes	X	X	X
s_uc09	Receive Notifications	X	X	
UC10	Adjust Refresh Interval	X	X	
s_uc11	Sort Incidents	X	X	
UC12	Report Incident	X	X	X
UC13	Edit or Withdraw Report	X	X	X
UC14	View Submission Status	X	X	X
s_uc15	Validate Report		X	
s_uc16	Check Rate Limit		X	
s_uc17	Geocode Location		X	X
s_uc18	Queue Offline		X	X
s_uc19	Fetch Map Tiles		X	X
UC20	Moderate Incidents	X	X	X

7 Implementation Considerations

7.1 Technology Stack Recommendations

Presentation Layer:

- HTML5 + CSS3 for structure and styling

- JavaScript/TypeScript with React or Vue.js
- Leaflet.js or Google Maps API for map visualization

Business Logic Layer:

- Node.js with Express (for web server)
- Service-oriented architecture with clear interfaces
- Dependency injection for loose coupling

Data Layer:

- Browser LocalStorage for client-side persistence
- REST API for server communication
- JSON for data serialization
- External Map Service API (Google Maps / Leaflet)

7.2 Design Patterns Integration

The 3-Layered Architecture works well with other patterns:

- **Repository Pattern:** Used in Data Layer (IncidentRepository)
- **Service Pattern:** Used throughout Business Logic Layer
- **Adapter Pattern:** Used for external API integration (MapServiceAPI)
- **Facade Pattern:** WebUI provides simplified interface to services

7.3 Best Practices

7.3.1 Interface-Based Communication

Define clear interfaces between layers:

```
// Business Logic Layer exposes:
interface IIncidentService {
  getAll(): List<Incident>
  create(incident: Incident): Incident
  filterBy(filter: Filter): List<Incident>
}

// Data Layer exposes:
interface IIncidentRepository {
  fetch(): List<Incident>
  persist(incident: Incident): void
}
```

7.3.2 Error Handling

Each layer handles errors at its level:

- **Data Layer:** Network errors, storage failures
- **Business Logic:** Validation errors, business rule violations
- **Presentation:** User-friendly error display

7.3.3 Logging and Monitoring

Implement logging at each layer for observability:

- UI interactions and errors

- Business logic operations and timing
- Data access queries and performance

8 Conclusion

8.1 Summary

This analysis demonstrates the significant benefits of applying the 3-Layered Architecture pattern to the Traffic News App. By organizing components into Presentation, Business Logic, and Data layers, the system achieves:

1. Clear separation of concerns with well-defined responsibilities
2. Improved testability through layer isolation
3. Enhanced maintainability with localized changes
4. Better scalability supporting system growth
5. Increased reusability across different contexts

8.2 Comparison

Quality Attribute	**Before (Flat)**	**After (3-Layer)**
Modularity	Poor	Excellent
Testability	Difficult	Easy
Maintainability	Low	High
Scalability	Limited	Good
Understandability	Hard	Clear
Coupling	Tight	Loose
Cohesion	Low	High

8.3 Recommendations

For successful implementation of the 3-Layered Architecture:

1. **Enforce Layer Boundaries:** Use code reviews and automated checks to prevent layer violations
2. **Define Clear Interfaces:** Document all inter-layer communication contracts
3. **Start with Business Logic:** Implement core services before UI or data access
4. **Write Tests Early:** Leverage testability to catch issues during development
5. **Monitor Dependencies:** Use tools to visualize and track component relationships
6. **Refactor Incrementally:** Migrate from flat to layered structure one component at a time

8.4 Future Work

The 3-Layered Architecture provides a solid foundation for:

- Adding mobile application support (reuse Business Logic and Data Layers)
- Implementing microservices architecture (split Business Logic Layer)
- Adding real-time features with WebSockets (minimal changes to architecture)
- Scaling horizontally (deploy Business Logic Layer on multiple servers)

8.5 Final Statement

The 3-Layered Architecture is a proven, industry-standard pattern that transforms the Traffic News App from a tangled flat structure into a well-organized, maintainable, and scalable system. This architectural foundation ensures the application can evolve to meet future requirements while remaining robust, testable, and easy to understand.

References

1. Sommerville, I. (2016). *Software Engineering* (10th ed.). Pearson.
2. Pressman, R. S., & Maxim, B. R. (2014). *Software Engineering: A Practitioner's Approach* (8th ed.). McGraw-Hill.
3. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
4. Traffic News App – Phase I (Requirements Analysis). CPS731 Team 15, Fall 2025.
5. CPS731 Lecture 11: System Architecture and Design Patterns. S. Tajali, 2025.

A Appendix A: Detailed Class Diagram (Before)

This appendix provides the complete detailed class diagram for the flat structure, including all attributes, methods, and relationships.

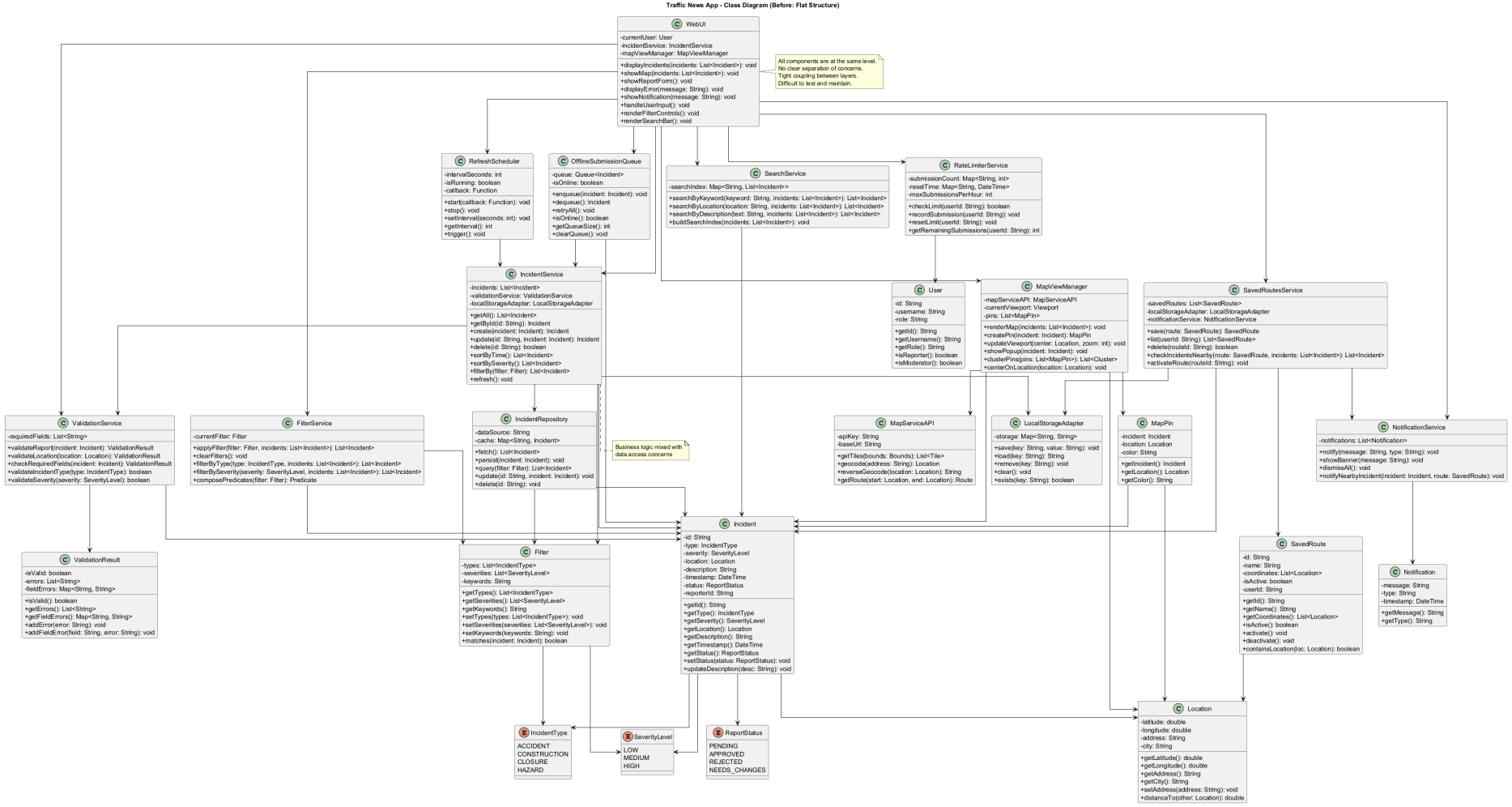

Figure 11: Detailed Class Diagram – Before (Flat Structure)

Purpose: This detailed view shows complete implementation specifications including:

- All class attributes with types
- All method signatures with parameters and return types
- All relationships and dependencies
- Complete domain model with enumerations

Use Case: Reference this diagram when implementing the system or understanding detailed component interactions.

B Appendix B: Detailed Class Diagram (After)

This appendix provides the complete detailed class diagram for the 3-layered architecture, with full implementation specifications.

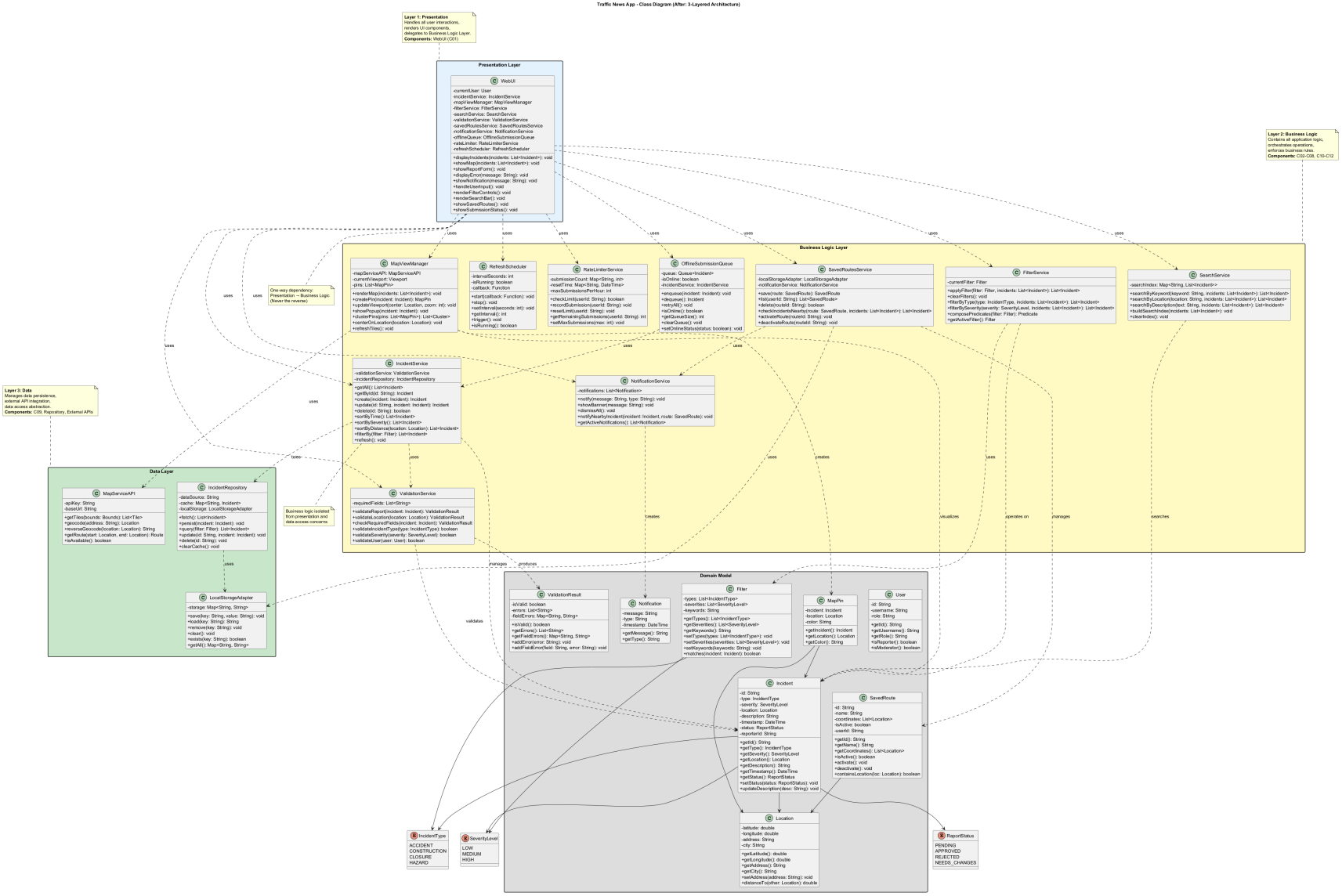

Figure 12: Detailed Class Diagram – After (3-Layered Architecture)

Purpose: This detailed view includes:

- Complete method signatures for all services
- All attributes with proper types and visibility
- Clear layer boundaries with package organization
- Detailed dependency relationships
- Interface contracts between layers

Use Case: Use this diagram as a blueprint for implementation, ensuring all methods and attributes are correctly defined according to the architecture.

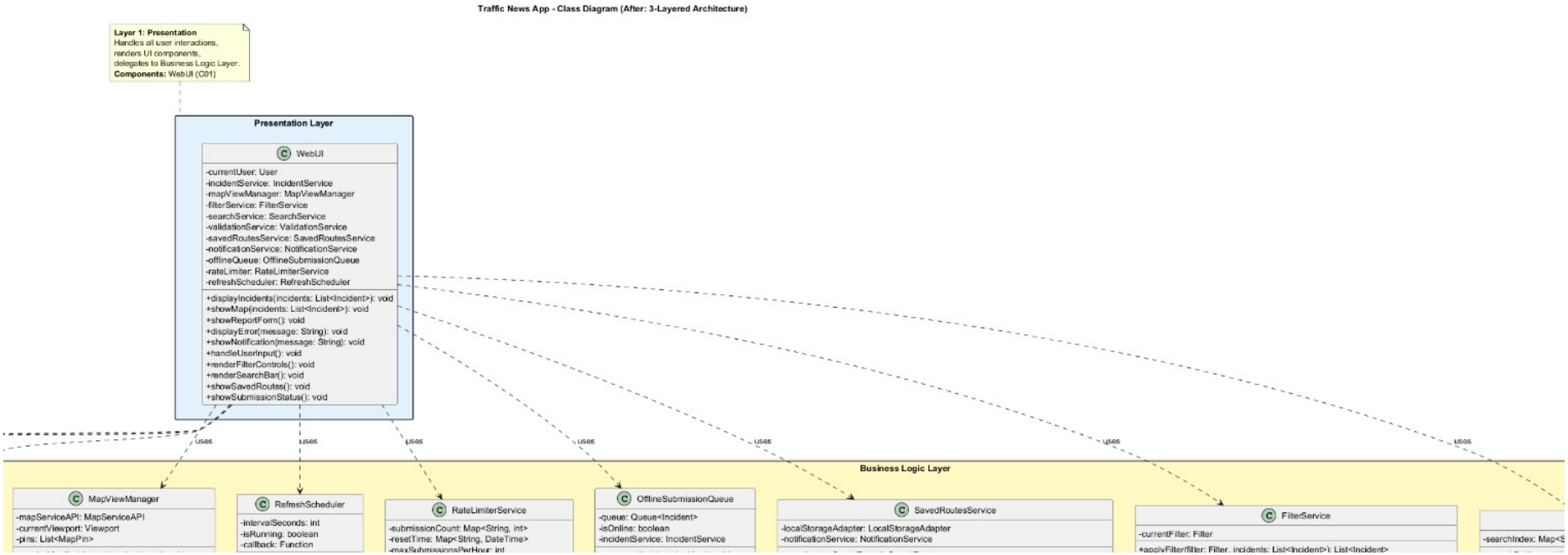

Figure 13: Business Layer Class Diagram – After (3-Layered Architecture)

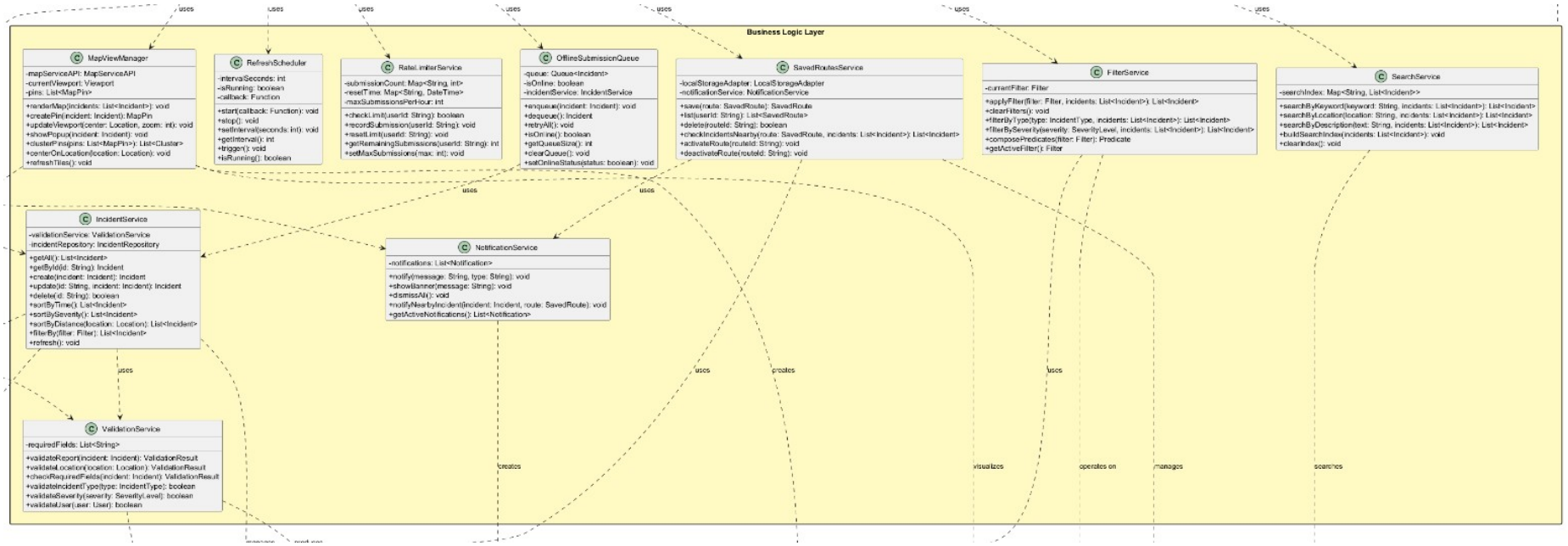

Figure 14: Business Layer Class Diagram – After (3-Layered Architecture)

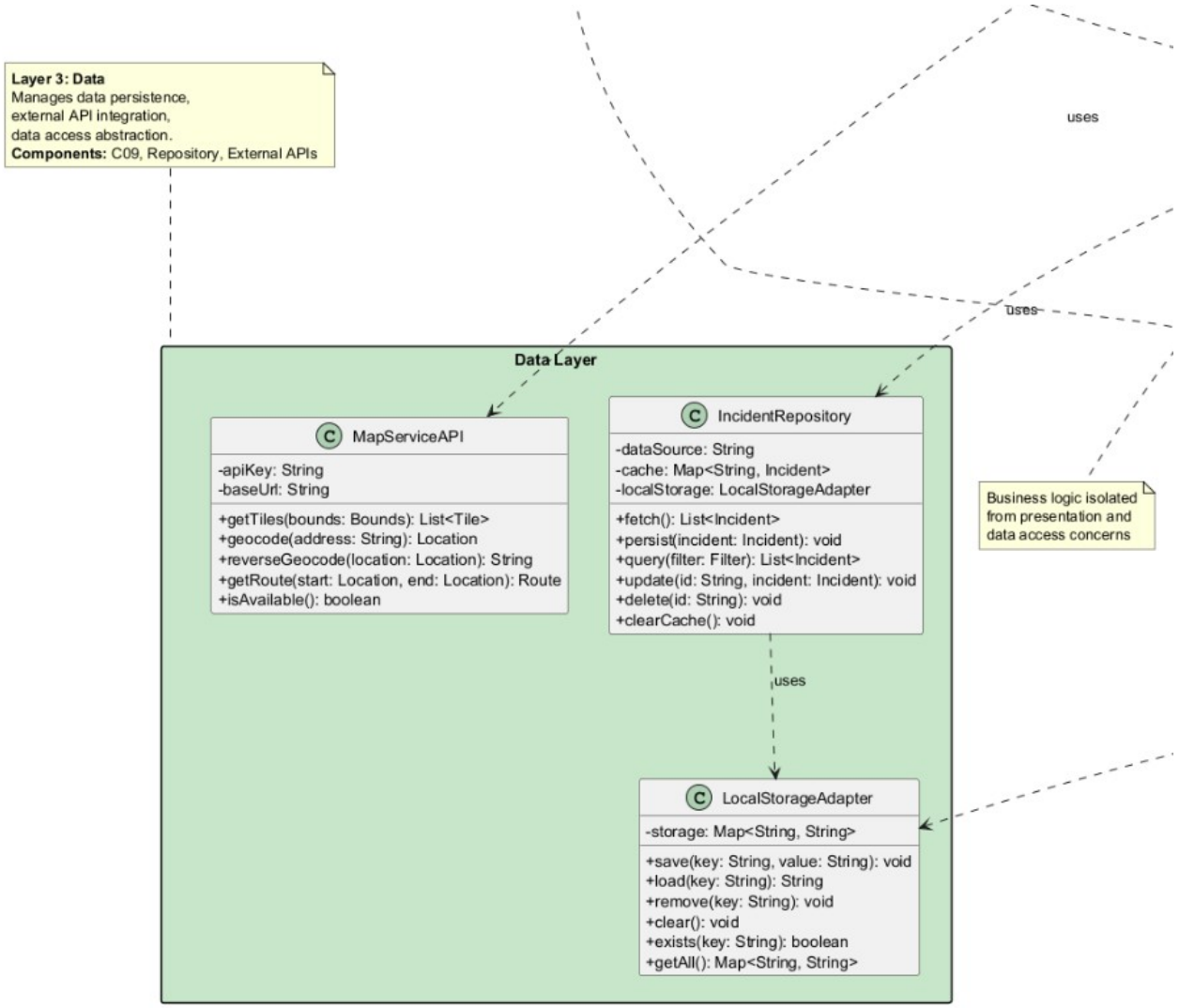

Figure 15: Business Layer Class Diagram – After (3-Layered Architecture)

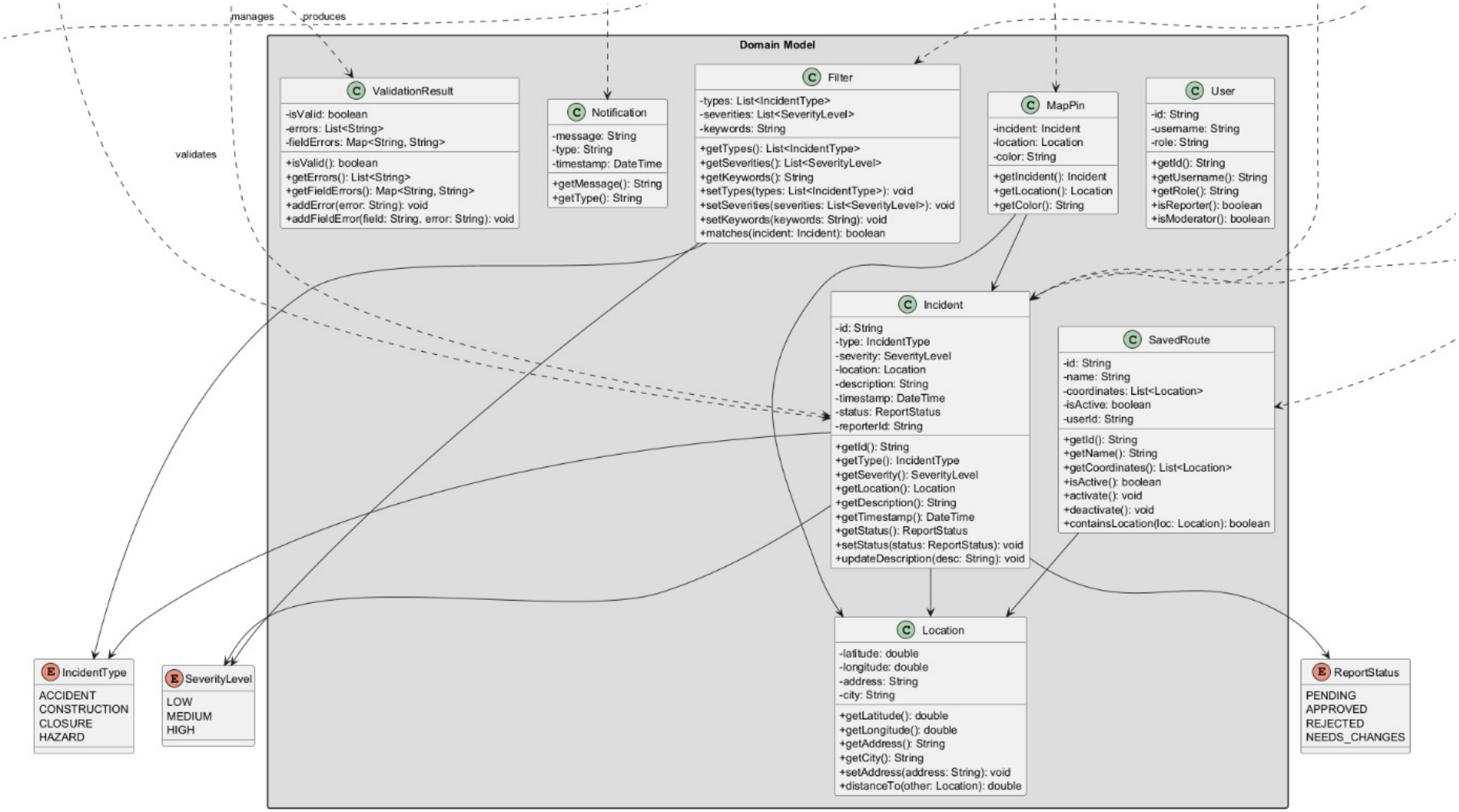

Figure 16: Domain Layer Class Diagram – After (3-Layered Architecture)

C Appendix C: Diagram Comparison Guide

C.1 When to Use Each Diagram Type

Purpose	Simplified Diagram	Detailed Diagram
Presentations	x Use this	
Documentation		x Use this
Team Discussions	x Use this	
Implementation		x Use this
Architecture Review	x Use this	
Code Generation		x Use this
Training	x Start here	Then refer to detailed
Maintenance		x Use this

C.2 Key Differences

Simplified Diagrams:

- Show only class names and component IDs
- Display key architectural relationships
- Focus on layer organization
- Easy to understand at a glance
- Suitable for presentations and high-level discussions

Detailed Diagrams:

- Include all attributes with types
- Show complete method signatures
- Display all relationships and dependencies
- Provide implementation specifications
- Reference for developers during coding

C.3 Recommendation

For Phase I (Requirements Analysis): Use simplified diagrams in the main report to clearly communicate architectural decisions.

For Phase II (Design): Reference detailed diagrams when creating component specifications and interface definitions.

For Phase III (Implementation): Use detailed diagrams as implementation blueprints, ensuring all specified methods and attributes are coded correctly.