Event Logging Architecture

Architecture Diagrams

Document Version: 1.0 **Last Updated:** 2025-10-26 **Status:** Implemented - Phase 5 **Priority:** CRITICAL - Required for FDA audit trail

---

Table of Contents

Overview

Purpose

TOSCA implements a comprehensive event logging system that provides an **immutable audit trail** for all safety-critical and operational events, meeting FDA 21 CFR Part 11 requirements.

Regulatory Requirements: - **FDA 21 CFR Part 11:** Electronic records and audit trails - **IEC 62304:** Software lifecycle documentation - **ISO 14971:** Risk management traceability

Logging Objectives

1. **Audit Trail:** Complete, immutable record of all system events
2. **Debugging:** Troubleshoot issues using detailed event history
3. **Compliance:** Meet FDA regulatory requirements for electronic records
4. **Real-Time Monitoring:** Live event display during treatment
5. **Post-Analysis:** Review treatment sessions for quality assurance

---

Two-Tier Logging Strategy

Architecture Overview

```
                     EventLogger
                 (src/core/event_logger.py)


                    log_event(type, severity, desc)




            ▼                              ▼

    JSONL File Log                SQLite Database
   (High Frequency)              (Event-Based Only)

  - All events                  - Safety events
  - Timestamped                 - Treatment events
  - Append-only                 - Hardware events
  - 10-100 events/sec           - User actions
  - Backup/recovery             - System events

  File: events.jsonl            Table: safety_log
```

Why Two Tiers?

Problem: Different logging needs require different storage strategies

Requirement	JSONL File	SQLite Database
High frequency	[DONE] Excellent (append-only)	[FAILED] Slow (transaction overhead)
Queryable	[FAILED] Linear scan	[DONE] Excellent (indexed queries)
FDA audit	[DONE] Immutable text	[DONE] Structured + signatures
Real-time	[DONE] Fast writes	WARNING: Slower inserts
Post-analysis	[FAILED] Hard to query	[DONE] SQL queries
Backup	[DONE] Copy file	[DONE] Export + verify

Solution: Use both for different purposes

Event Types & Severity

Event Categories

File: `src/core/event_logger.py`

Enum: `EventType`

```python
class EventType(Enum):
    # Safety events (CRITICAL)
    SAFETY_EMERGENCY_STOP = "e_stop_pressed"
    SAFETY_EMERGENCY_CLEAR = "e_stop_released"
    SAFETY_INTERLOCK_FAIL = "interlock_failure"
    SAFETY_INTERLOCK_OK = "interlock_recovery"
    SAFETY_POWER_LIMIT = "power_limit_exceeded"
    SAFETY_GPIO_FAIL = "gpio_interlock_failure"
    SAFETY_GPIO_OK = "gpio_interlock_ok"

    # Hardware events
    HARDWARE_CAMERA_CONNECT = "camera_connected"
    HARDWARE_LASER_CONNECT = "laser_connected"
    HARDWARE_ACTUATOR_HOME_START = "actuator_homing_started"
    HARDWARE_ERROR = "hardware_error"

    # Treatment events (AUDIT REQUIRED)
    TREATMENT_SESSION_START = "session_started"
    TREATMENT_SESSION_END = "session_ended"
    TREATMENT_LASER_ON = "laser_enabled"
    TREATMENT_LASER_OFF = "laser_disabled"
    TREATMENT_POWER_CHANGE = "laser_power_changed"

    # User events (AUDIT REQUIRED)
    USER_LOGIN = "user_login"
    USER_ACTION = "user_action"
    USER_OVERRIDE = "user_override"

    # System events
    SYSTEM_STARTUP = "system_startup"
    SYSTEM_SHUTDOWN = "system_shutdown"
    SYSTEM_ERROR = "system_error"
```

Severity Levels

Enum: `EventSeverity`

Level	Usage	Example	Color (UI)
INFO	Normal operation	"Session started"	Blue
WARNING	Attention needed	"Interlock temporarily failed"	◍ Yellow
CRITICAL	Safety issue	"Emergency stop activated"	● Orange
EMERGENCY	Immediate danger	"Laser power limit exceeded"	Red

EventLogger Implementation

Core Class

File: `src/core/event_logger.py`

Class: `EventLogger(QObject)`

Key Features:

1. **Dual Logging:**

```python
def log_event(
    self,
    event_type: EventType | str,
    severity: EventSeverity | str,
    description: str,
    metadata: Optional[dict] = None
) -> None:
    """Log event to both JSONL file and database."""
    timestamp = datetime.now().isoformat()

    # 1. Write to JSONL file (fast, append-only)
    self._write_to_file(timestamp, event_type, severity, description, metadata)

    # 2. Write to database (structured, queryable)
    self._write_to_database(timestamp, event_type, severity, description, metadata)

    # 3. Emit signal for real-time display
    self.event_logged.emit(str(event_type), str(severity), description)
```

2. **JSONL File Format:**

```
{"timestamp": "2025-10-26T14:32:15.123456", "type": "session_started", "severity": "info", "session_id": 42, "tech_id": 5, "description": "Treatment session started"}
{"timestamp": "2025-10-26T14:32:16.789012", "type": "laser_enabled", "severity": "info", "power_w": 5.0, "description": "Laser enabled at 5.0W"}
{"timestamp": "2025-10-26T14:32:45.234567", "type": "e_stop_pressed", "severity": "emergency", "description": "Emergency stop activated by operator"}
```

Benefits: - [DONE] One event per line (easy to parse) - [DONE] Append-only (immutable) - [DONE] Human-readable (can grep/tail) - [DONE] Timestamped (high precision)

3. **Database Schema:**

```
CREATE TABLE safety_log (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    timestamp TEXT NOT NULL,
    event_type TEXT NOT NULL,
    severity TEXT NOT NULL,
    session_id INTEGER,
    tech_id INTEGER,
    description TEXT NOT NULL,
    metadata TEXT,  -- JSON-encoded additional data
    FOREIGN KEY (session_id) REFERENCES sessions(id),
    FOREIGN KEY (tech_id) REFERENCES technicians(id)
);

CREATE INDEX idx_safety_log_timestamp ON safety_log(timestamp);
CREATE INDEX idx_safety_log_event_type ON safety_log(event_type);
CREATE INDEX idx_safety_log_session_id ON safety_log(session_id);
```

Benefits: - [DONE] Fast queries (indexed timestamps, event types) - [DONE] Relational (links to sessions, technicians) - [DONE] Structured (SQL schema validation)

Real-Time Display

PyQt6 Signal Integration

EventLogger emits signal on every event:

```
class EventLogger(QObject):
    # Signal: (event_type, severity, description)
    event_logged = pyqtSignal(str, str, str)

    def log_event(self, type, severity, desc, metadata=None):
        # ... write to file and database ...

        # Emit signal for real-time UI update
        self.event_logged.emit(str(type), str(severity), desc)
```

UI widget connects to signal:

```
class SafetyWidget(QWidget):
    def __init__(self, event_logger: EventLogger):
        super().__init__()
        self.event_log_display = QTextEdit()
        # Connect to event logger signal
        event_logger.event_logged.connect(self.display_event)

    def display_event(self, event_type: str, severity: str, desc: str):
        """Display event in real-time log."""
        timestamp = datetime.now().strftime("%H:%M:%S")
        color = self._severity_color(severity)

        html = f'<span style="color:{color}">[{timestamp}] {desc}</span><br>'
        self.event_log_display.append(html)
```

Result: All events appear in UI in real-time during treatment

---

FDA Compliance

21 CFR Part 11 Requirements

§11.10(e) - Audit Trail:

> Use of secure, computer-generated, time-stamped audit trails to independently record the date and time of operator entries and actions that create, modify, or delete electronic records.

TOSCA Compliance:

[DONE] **Secure:** JSONL file is append-only (cannot modify past events)

[DONE] **Computer-Generated:** Automatic logging (no manual entry)

[DONE] **Time-Stamped:** ISO 8601 timestamps with microsecond precision

[DONE] **Independent:** EventLogger runs independently of business logic

[DONE] **Record Actions:** All operator actions logged (user_action events)

[DONE] **Create/Modify/Delete:** Session start/end, parameter changes logged

Immutability

JSONL File: - Append-only writes (no modification of past lines) - File permissions: Read-only after session ends - Backup: Copied to archive on session completion

Database:

```
-- No UPDATE or DELETE operations on safety_log
-- Only INSERT allowed

-- Audit trail integrity verification (future)
SELECT COUNT(*) FROM safety_log WHERE session_id = 42;
-- Should match JSONL line count for session
```

Future Enhancement (Phase 6): - HMAC signatures per event (tamper detection) - Cryptographic chain (each event signs previous event) - Verification tool (validate audit trail integrity)

Retention & Export

Retention Policy: - Event logs retained for 7 years (FDA requirement) - JSONL files archived on session completion - Database events never deleted

Export for Audit:

```
# Export session events to PDF
def export_session_audit(session_id: int) -> Path:
    """Export complete audit trail for session."""
    events = db.query(
        "SELECT * FROM safety_log WHERE session_id = ? ORDER BY timestamp",
        (session_id,)
    )

    # Generate PDF with:
    # - Session metadata
    # - Complete event timeline
    # - Operator actions
    # - Safety events highlighted

    return pdf_path
```

Usage Examples

Example 1: Log Safety Event

```
event_logger.log_event(
    event_type=EventType.SAFETY_EMERGENCY_STOP,
    severity=EventSeverity.EMERGENCY,
    description="Emergency stop activated by operator",
    metadata={
        "operator_id": current_user.id,
        "laser_power_at_stop": laser.get_power(),
        "session_duration_s": session.elapsed_time()
    }
)
```

Result: - [DONE] Written to JSONL: `data/logs/events.jsonl` - [DONE] Written to database: `safety_log` table - [DONE] Displayed in UI: SafetyWidget event log - [DONE] Audit trail: Immutable record created

Example 2: Log Treatment Event

```
event_logger.log_event(
    event_type=EventType.TREATMENT_LASER_ON,
    severity=EventSeverity.INFO,
    description=f"Laser enabled at {power:.1f}W",
    metadata={
        "power_w": power,
        "wavelength_nm": 1064,
        "protocol_id": protocol.id,
        "pulse_number": pulse_count
    }
)
```

Example 3: Query Events

```
# Find all safety events in last hour
one_hour_ago = (datetime.now() - timedelta(hours=1)).isoformat()

events = db.query("""
    SELECT timestamp, event_type, description
    FROM safety_log
    WHERE timestamp > ? AND event_type LIKE 'SAFETY_%'
    ORDER BY timestamp DESC
""", (one_hour_ago,))
```

Testing

Event Logging Tests

Test 1: Verify Dual Logging

```
def test_event_logged_to_both_tiers():
    """Verify event written to JSONL and database."""
    event_logger.log_event(
        EventType.SYSTEM_STARTUP,
        EventSeverity.INFO,
        "System started"
    )

    # Check JSONL file
    with open(event_logger.log_file) as f:
        lines = f.readlines()
        last_event = json.loads(lines[-1])
        assert last_event["type"] == "system_startup"

    # Check database
    events = db.query("SELECT * FROM safety_log WHERE event_type = 'system_startup'")
    assert len(events) > 0
```

Test 2: Verify Signal Emission

```
def test_event_signal_emitted():
    """Verify event_logged signal is emitted."""
    signals_received = []

    def capture_signal(type, severity, desc):
        signals_received.append((type, severity, desc))

    event_logger.event_logged.connect(capture_signal)
    event_logger.log_event(EventType.TREATMENT_SESSION_START, EventSeverity.INFO, "Test")

    assert len(signals_received) == 1
    assert signals_received[0][0] == "session_started"
```

References

Standards

- **FDA 21 CFR Part 11:** Electronic Records; Electronic Signatures
- **ISO 14971:** Medical devices — Application of risk management
- **IEC 62304:** Medical device software — Software life cycle processes

File Formats

- **JSONL:** https://jsonlines.org/
- **ISO 8601:** Date and time format

Document Owner: Software Architect **Last Updated:** 2025-10-26 **Next Review:** Before Phase 6 (HMAC signatures) **Status:** Implemented - Two-tier logging operational