IOT Smart Tile System - Design and Implementation

Executive Summary

This document provides a comprehensive account of how the IOT Smart Tile System was designed, planned, and implemented. It covers the architectural decisions, implementation strategies, key code examples, development workflows, and analysis of results including successes and challenges encountered.

Table of Contents

---

1. Design Planning

Before writing any code, we established clear design principles and planned the architecture for each major component. This section documents our planning process and the rationale behind our design decisions.

1.1 Embedded Software Design

Design Goals

The embedded layer needed to achieve several critical objectives:

Goal	Requirement	Design Decision
Low Latency	<100ms response to user input	Use ESP-NOW instead of WiFi for local communication

Battery Efficiency	Months of operation on battery	Deep sleep modes, wake-on-interrupt, minimal WiFi usage
Reliability	Continue working if cloud is unavailable	Local-first architecture, offline fallback modes
Scalability	Support 10+ nodes per coordinator	Efficient message protocol, NVS-based registry
Maintainability	Easy to add new features	Manager pattern with single responsibility

Planned Architecture

We designed a two-tier embedded architecture:

```
┌──────────────────────────────────────────────────────────────────────┐
│                         DESIGN BLUEPRINT                             │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  TIER 1: COORDINATOR (ESP32-S3)                                      │
│  ┌──────────────────────────────────────────────────────────────┐    │
│  │  Responsibilities:                                            │   │
│  │  • Bridge ESP-NOW mesh ↔ MQTT cloud                          │    │
│  │  • Host environmental sensors (light, mmWave, temp)          │    │
│  │  • Manage node registry and pairing                          │    │
│  │  • Provide local intelligence (presence → lighting)          │    │
│  │                                                               │   │
│  │  Design Pattern: Manager Pattern                              │   │
│  │  Each subsystem is isolated in its own manager class:         │   │
│  │  • EspNow → handles ESP-NOW protocol                         │    │
│  │  • Mqtt → handles MQTT connection and messaging              │    │
│  │  • WifiManager → handles WiFi provisioning and reconnection │    │
│  │  • NodeRegistry → handles node storage and pairing           │    │
│  │  • ZoneControl → handles lighting logic                      │    │
│  │  • MmWave → handles radar sensor                             │    │
│  │  • AmbientLightSensor → handles light sensor                 │    │
│  │  • ThermalControl → handles temperature monitoring           │    │
│  │  • ButtonControl → handles physical button input             │    │
│  └──────────────────────────────────────────────────────────────┘    │
│                                                                      │
│  TIER 2: NODE (ESP32-C3)                                            │
│  ┌──────────────────────────────────────────────────────────────┐    │
│  │  Responsibilities:                                            │   │
│  │  • Control RGBW LED strips                                   │    │
│  │  • Report local temperature                                  │    │
│  │  • Handle button input for local control                     │    │
│  │  • Manage battery and deep sleep                             │    │
│  │                                                               │   │
│  │  Design Pattern: State Machine                                │   │
│  │  Node operates in distinct states:                            │   │
│  │  • UNPAIRED → scanning for coordinator                       │    │
│  │  • PAIRING → handshake in progress                           │    │
```

```
│  │  • NORMAL → operational, receiving commands             │    │
│  │  • DEEP_SLEEP → battery conservation                    │    │
│  └─────────────────────────────────────────────────────────┘    │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Planned Message Protocol

We designed a JSON-based message protocol for ESP-NOW communication:

Message Type	Direction	Purpose
`join_request`	Node → Coordinator	Node requests to join network
`join_accept`	Coordinator → Node	Coordinator accepts node, assigns ID
`set_light`	Coordinator → Node	Set LED color/brightness
`status`	Node → Coordinator	Node reports sensor values
`ping` / `pong`	Bidirectional	Keepalive heartbeat

1.2 MQTT Communication Design

Connection Strategy

We planned a hierarchical MQTT topic structure that would:

1. **Support multi-site deployments** - Topic prefix `site/{siteId}/`
2. **Distinguish device types** - Separate paths for `coord/` and `node/`
3. **Enable selective subscriptions** - Backend can subscribe to specific sites or wildcards
4. **Support bidirectional commands** - Telemetry on `.../telemetry`, commands on `.../cmd`

Planned Topic Hierarchy

```
site/{siteId}/
├── coord/{coordId}/
│   ├── telemetry      → Coordinator publishes sensor data
│   ├── mmwave         → Coordinator publishes radar frames
│   ├── status         → Coordinator publishes connection state
│   └── cmd            ← Backend publishes commands TO coordinator
│
├── node/{nodeId}/
│   ├── telemetry      → Coordinator publishes node status (relayed from ESP-NOW)
│   └── cmd            ← Backend publishes commands TO node (via coordinator)
│
└── zone/{zoneId}/
    ├── presence       → Aggregated occupancy for automation
    └── cmd            ← Zone-wide commands
```

QoS and Reliability

Topic Type	QoS Level	Rationale
Telemetry	QoS 1	At-least-once delivery; duplicates are acceptable

Commands	QoS 1	Ensure command reaches device
Status	QoS 0	Fire-and-forget; stale status is quickly replaced

1.3 Backend Design

Design Pattern: Dependency Injection with Uber fx

We chose to structure the Go backend using **Uber fx** for dependency injection:

```
┌─────────────────────────────────────────────────────────────────────┐
│                     BACKEND DESIGN BLUEPRINT                        │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  ┌──────────────────────────────────────────────────────────────┐   │
│  │                      main.go (Entry)                         │   │
│  │  fx.New(                                                     │   │
│  │      http.Module,      // REST API + WebSocket               │   │
│  │      mqtt.Module,      // MQTT client                        │   │
│  │      googlehome.Module // Smart home integration             │   │
│  │      fx.Provide(                                             │   │
│  │          db.NewMongoDB,                                      │   │
│  │          repository.NewMongoRepository,                      │   │
│  │          config.NewConfig,                                   │   │
│  │      )                                                       │   │
│  │  ).Run()                                                     │   │
│  └──────────────────────────────────────────────────────────────┘   │
│                               │                                     │
│               ┌───────────────┼───────────────┐                     │
│               ▼               ▼               ▼                     │
│  ┌───────────────┐  ┌───────────────┐  ┌───────────────┐            │
│  │ HTTP Module   │  │ MQTT Module   │  │ Repository    │            │
│  │ • REST API    │  │ • Subscribe   │  │ • MongoDB     │            │
│  │ • WebSocket   │  │ • Publish     │  │ • CRUD ops    │            │
│  │ • Handlers    │  │ • Handlers    │  │ • Upserts     │            │
│  └───────────────┘  └───────────────┘  └───────────────┘            │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

Data Flow Design

```
MQTT Message Received
         │
         ▼
┌─────────────────┐
│ Parse JSON      │
│ Extract IDs     │
└────────┬────────┘
         │
         ▼
┌─────────────────┐
│ Upsert to       │
```

```
│ MongoDB        │
└───────┬────────┘
        │
        ▼
┌────────────────┐
│ Broadcast via  │
│ WebSocket      │
└────────────────┘
```

1.4 Frontend Design

Design Pattern: Reactive Signals with Angular

We planned the frontend around Angular's new **Signals API** for state management:

```
┌──────────────────────────────────────────────────────────────────────┐
│                    FRONTEND DESIGN BLUEPRINT                         │
├──────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  Architecture: Feature-based modules                                 │
│                                                                      │
│  src/app/                                                            │
│  ├── core/                  # Shared services and models             │
│  │   ├── services/                                                   │
│  │   │   ├── data.service.ts      # Central data store (signals)    │
│  │   │   ├── websocket.service.ts # Real-time connection            │
│  │   │   ├── api.service.ts       # REST API calls                  │
│  │   │   └── environment.service.ts                                 │
│  │   └── models/                                                     │
│  │       └── api.models.ts        # TypeScript interfaces           │
│  │                                                                   │
│  └── features/              # Feature modules                        │
│      ├── dashboard/         # Main monitoring UI                     │
│      │   ├── components/    # Reusable dashboard components         │
│      │   └── tabs/          # Tab content components                │
│      ├── debug/             # Developer tools                        │
│      └── login/             # Authentication                         │
│                                                                      │
│  State Management: Angular Signals                                   │
│  ┌──────────────────────────────────────────────────────────────┐   │
│  │  DataService (Singleton)                                      │   │
│  │  • nodes = signal<Map<string, Node>>()                       │   │
│  │  • coordinators = signal<Map<string, Coordinator>>()         │   │
│  │  • sites = signal<Site[]>([])                                │   │
│  │                                                               │   │
│  │  Components subscribe via computed() and effect()             │   │
│  └──────────────────────────────────────────────────────────────┘   │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

---

2. Implementation Details

This section describes how we implemented each component, following the design plans outlined above.

2.1 Embedded Software Implementation

Coordinator Implementation

The Coordinator was implemented in C++ using the Arduino framework on PlatformIO. The core implementation follows the Manager pattern exactly as planned.

File Structure:

```
coordinator/src/
├── main.cpp              # Entry point, calls Coordinator::begin() and loop()
├── core/
│   ├── Coordinator.cpp   # Main orchestrator class
│   └── Coordinator.h
├── comm/
│   ├── EspNow.cpp        # ESP-NOW protocol handler
│   ├── Mqtt.cpp          # MQTT client wrapper
│   └── WifiManager.cpp   # WiFi provisioning and reconnection
├── nodes/
│   └── NodeRegistry.cpp  # NVS-backed node storage
├── sensors/
│   ├── MmWave.cpp        # LD2450 radar driver
│   ├── AmbientLightSensor.cpp
│   └── ThermalControl.cpp
├── input/
│   └── ButtonControl.cpp # Touch button handler
└── zones/
    └── ZoneControl.cpp   # Lighting automation logic
```

Initialization Sequence:

The Coordinator initializes managers in dependency order:

1. **ESP-NOW first** - Must be ready before WiFi connects (channel sync)
2. **WiFi second** - Connects to AP, syncs ESP-NOW channel
3. **MQTT third** - Depends on WiFi being connected
4. **Sensors** - Can initialize in parallel
5. **Node Registry** - Loads paired nodes from NVS

Node Implementation

The Node firmware implements a state machine as planned:

```cpp
enum class NodeState {
    UNPAIRED,      // Scanning for coordinator
    PAIRING,       // Handshake in progress
    NORMAL,        // Operational
    DEEP_SLEEP     // Power saving
};
```

Key Implementation Details:

- **LED Control**: Uses NeoPixel library for SK6812 RGBW LEDs with PWM fading
- **Button Input**: Interrupt-driven with debounce, supports short press and long press
- **Deep Sleep**: Configured for wake-on-button with 5-minute auto-sleep timer
- **ESP-NOW**: Listens on fixed channel, switches to coordinator's channel after pairing

2.2 MQTT Implementation

Coordinator MQTT Client

The coordinator uses the PubSubClient library wrapped in our `Mqtt` class:

Connection Strategy:

1. Load broker address from NVS (`ConfigManager`)
2. If not configured, prompt via Serial for interactive setup
3. Connect with auto-reconnect logic (exponential backoff)
4. Subscribe to `site/{siteId}/coord/{coordId}/cmd` for incoming commands

Publishing Telemetry:

Every 5 seconds, the coordinator publishes a snapshot:

```
void Mqtt::publishCoordinatorTelemetry(const CoordinatorSensorSnapshot& snapshot) {
    StaticJsonDocument<512> doc;
    doc["ts"] = snapshot.timestamp;
    doc["light_lux"] = snapshot.lightLux;
    doc["temp_c"] = snapshot.tempC;
    doc["mmwave_presence"] = snapshot.mmwavePresence;
    doc["mmwave_confidence"] = snapshot.mmwaveConfidence;
    doc["wifi_rssi"] = snapshot.wifiRssi;
    doc["wifi_connected"] = snapshot.wifiConnected;

    String payload;
    serializeJson(doc, payload);

    String topic = "site/" + siteId + "/coord/" + coordId + "/telemetry";
    mqttClient.publish(topic.c_str(), payload.c_str());
}
```

Backend MQTT Handler

The Go backend uses Paho MQTT client with Uber fx lifecycle hooks:

Subscription Setup:

```
func (h *Handler) connectHandler(client mqtt.Client) {
    // Subscribe to all relevant topics on connect
    client.Subscribe("site/+/node/+/telemetry", 1, h.handleNodeTelemetry)
    client.Subscribe("site/+/coord/+/telemetry", 1, h.handleCoordTelemetry)
    client.Subscribe("site/+/coord/+/mmwave", 1, h.handleCoordMMWave)
}
```

2.3 Backend Implementation

Dependency Injection with fx

The backend uses Uber fx for clean dependency management:

```go
func main() {
    fx.New(
        http.Module,           // REST API + WebSocket
        mqtt.Module,           // MQTT client
        googlehome.Module,     // Smart home integration
        fx.Provide(
            zap.NewProduction,                   // Logger
            db.NewMongoDB,                       // Database connection
            mux.NewRouter,                       // HTTP router
            config.NewConfig,                    // Configuration
            fx.Annotate(
                repository.NewMongoRepository,
                fx.As(new(repository.Repository)), // Interface binding
            ),
        ),
    ).Run()
}
```

Benefits Realized:

- Automatic startup/shutdown ordering
- Easy testing with mock repositories
- Clear dependency graph
- No global state

Repository Pattern

All database operations go through the Repository interface:

```go
type Repository interface {
    // Sites
    GetSites() ([]types.Site, error)
    UpsertSite(ctx context.Context, site *types.Site) error

    // Coordinators
    GetCoordinatorById(id string) (*types.Coordinator, error)
    UpsertCoordinator(ctx context.Context, coord *types.Coordinator) error

    // Nodes
    GetNodeById(id string) (*types.Node, error)
    UpsertNode(ctx context.Context, node *types.Node) error

    // Telemetry
    InsertTelemetry(ctx context.Context, t *types.Telemetry) error
}
```

2.4 Frontend Implementation

WebSocket Service

The WebSocket service manages real-time updates using RxJS:

```
@Injectable({ providedIn: 'root' })
export class WebSocketService {
    // Signals for connection state
    public readonly connected = signal<boolean>(false);
    public readonly connecting = signal<boolean>(false);

    // Observable streams for different message types
    public readonly telemetry$ = this.telemetrySubject.asObservable();
    public readonly presence$ = this.presenceSubject.asObservable();

    connect(): void {
        this.ws = new WebSocket(this.env.wsUrl);
        this.ws.onmessage = (event) => this.handleMessage(event);
        this.ws.onclose = () => this.handleDisconnect();
    }

    private handleMessage(event: MessageEvent): void {
        const msg = JSON.parse(event.data) as WSMessage;
        switch (msg.type) {
            case 'telemetry':
                this.telemetrySubject.next(msg.payload);
                break;
            case 'presence':
                this.presenceSubject.next(msg.payload);
                break;
        }
    }
}
```

Dashboard Component

The dashboard uses Angular Signals for reactive state:

```
@Component({
    selector: 'app-dashboard',
    templateUrl: './dashboard.component.html'
})
export class DashboardComponent implements OnInit {
    data = inject(DataService);

    // Computed signals derive from data service
    registeredNodes = computed(() =>
        Array.from(this.data.nodes().values())
    );

    registeredCoordinators = computed(() =>
        Array.from(this.data.coordinators().values())
    );
```

```
    ngOnInit() {
        // Load initial data
        this.data.loadSites().then(() => {
            const sites = this.data.sites();
            if (sites.length > 0) {
                this.data.loadSite(sites[0]._id);
            }
        });
    }
}
```

---

3. Key Code Examples

This section highlights the most important and interesting code from our implementation.

3.1 ESP-NOW Message Protocol

Why This Code Is Important:

The ESP-NOW message protocol is the foundation of our local mesh network. This code defines how nodes and coordinators communicate, including the pairing handshake that establishes trust between devices.

File: `shared/src/EspNowMessage.cpp`

```
// JOIN REQUEST - Sent by node to coordinator during pairing
// This message advertises the node's capabilities so the coordinator
// knows what features are available (RGBW LEDs, temperature sensor, etc.)

JoinRequestMessage::JoinRequestMessage() {
    type = MessageType::JOIN_REQUEST;
    msg = "join_request";
    ts = millis();
}

String JoinRequestMessage::toJson() const {
    DynamicJsonDocument doc(512);
    doc["msg"] = msg;
    doc["mac"] = mac;           // Node's MAC address for identification
    doc["fw"] = fw;             // Firmware version for compatibility checks

    // Capability advertisement - coordinator needs to know what the node supports
    doc["caps"]["rgbw"] = caps.rgbw;           // Has RGBW LED?
    doc["caps"]["led_count"] = caps.led_count; // How many LEDs?
    doc["caps"]["temp_i2c"] = caps.temp_i2c;   // Has I2C temperature sensor?
    doc["caps"]["deep_sleep"] = caps.deep_sleep;
    doc["caps"]["button"] = caps.button;

    doc["token"] = token;       // Security token for pairing verification

    String out;
```

```cpp
    serializeJson(doc, out);
    return out;
}

// JOIN ACCEPT - Coordinator's response to approve pairing
// Assigns the node a unique ID and provides configuration

JoinAcceptMessage::JoinAcceptMessage() {
    type = MessageType::JOIN_ACCEPT;
    msg = "join_accept";
    ts = millis();
    wifi_channel = 1;  // Node will switch to this channel

    // Default configuration values
    cfg.pwm_freq = 0;
    cfg.rx_window_ms = 20;   // How long to listen for messages
    cfg.rx_period_ms = 100;  // How often to wake up and check
}

String JoinAcceptMessage::toJson() const {
    DynamicJsonDocument doc(256);
    doc["msg"] = msg;
    doc["node_id"] = node_id;      // Assigned unique ID
    doc["light_id"] = light_id;    // Logical light group
    doc["lmk"] = lmk;              // Local Master Key for encryption
    doc["wifi_channel"] = wifi_channel;

    // Operating parameters for power management
    doc["cfg"]["pwm_freq"] = cfg.pwm_freq;
    doc["cfg"]["rx_window_ms"] = cfg.rx_window_ms;
    doc["cfg"]["rx_period_ms"] = cfg.rx_period_ms;

    String out;
    serializeJson(doc, out);
    return out;
}
```

What Makes This Interesting:

1. **Capability Advertisement**: Nodes tell the coordinator what they can do, enabling heterogeneous device support
2. **Channel Synchronization**: The `wifi_channel` field ensures node switches to coordinator's channel
3. **Power Configuration**: `rx_window_ms` and `rx_period_ms` let us tune the power/latency tradeoff per-node

3.2 Coordinator Manager Pattern

Why This Code Is Important:

The Coordinator class demonstrates how we applied the Manager pattern to keep embedded code maintainable. Each subsystem is isolated, has a clear responsibility, and can be tested independently.

File: `coordinator/src/core/Coordinator.cpp`

```cpp
// Constructor - Initialize all manager pointers to nullptr
// This ensures we can safely check if managers were created
Coordinator::Coordinator()
    : espNow(nullptr)
    , mqtt(nullptr)
    , mmWave(nullptr)
    , nodes(nullptr)
    , zones(nullptr)
    , buttons(nullptr)
    , thermal(nullptr)
    , wifi(nullptr)
    , ambientLight(nullptr)
    , manualLedMode(false)
    , manualR(0), manualG(0), manualB(0)
    , manualLedTimeoutMs(0) {}

// Destructor - Clean up in reverse order of initialization
// This is critical for embedded systems where destructors
// may need to release hardware resources
Coordinator::~Coordinator() {
    if (thermal) { delete thermal; thermal = nullptr; }
    if (buttons) { delete buttons; buttons = nullptr; }
    if (zones) { delete zones; zones = nullptr; }
    if (nodes) { delete nodes; nodes = nullptr; }
    if (mmWave) { delete mmWave; mmWave = nullptr; }
    if (ambientLight) { delete ambientLight; ambientLight = nullptr; }
    if (mqtt) { delete mqtt; mqtt = nullptr; }
    if (wifi) { delete wifi; wifi = nullptr; }
    if (espNow) { delete espNow; espNow = nullptr; }
}

bool Coordinator::begin() {
    Logger::info("Smart Tile Coordinator starting...");

    // Create all manager instances
    espNow = new EspNow();
    mqtt = new Mqtt();
    mmWave = new MmWave();
    nodes = new NodeRegistry();
    zones = new ZoneControl();
    buttons = new ButtonControl();
    thermal = new ThermalControl();
    wifi = new WifiManager();
    ambientLight = new AmbientLightSensor();

    // Initialize ESP-NOW FIRST (before WiFi connects)
    // This is critical because WiFi connection changes the channel,
    // and we need ESP-NOW registered before that happens
    bool espNowOk = espNow->begin();
    if (!espNowOk) {
        Logger::error("Failed to initialize ESP-NOW");
```

```
        return false;
    }

    // Link EspNow to WiFi so channels sync on connection
    wifi->setEspNow(espNow);

    // Initialize WiFi - may block for provisioning
    bool wifiReady = wifi->begin();

    // Register callback for incoming node messages
    // This lambda captures 'this' to call instance methods
    espNow->setMessageCallback([this](const String& nodeId,
                                      const uint8_t* data,
                                      size_t len) {
        if (data && len > 0) {
            this->handleNodeMessage(nodeId, data, len);
        }
    });

    // Visual feedback callback for send errors
    espNow->setSendErrorCallback([this](const String& nodeId) {
        statusLed.pulse(180, 0, 0, 200); // Red flash
        Logger::warn("ESP-NOW send failed to node %s", nodeId.c_str());
    });

    // Pairing callback - handles JOIN_REQUEST from nodes
    espNow->setPairingCallback([this](const uint8_t* mac,
                                      const uint8_t* data,
                                      size_t len) {
        // ... pairing logic
    });

    return true;
}
```

What Makes This Interesting:

1. **Dependency Ordering**: ESP-NOW must initialize before WiFi to avoid channel conflicts
2. **Callback Injection**: Lambdas with captured `this` connect managers without tight coupling
3. **Boot Status Recording**: Each subsystem reports its status for diagnostics
4. **Graceful Degradation**: WiFi failure doesn't prevent local operation

3.3 MQTT Handler Pipeline

Why This Code Is Important:

This code shows how the backend processes incoming MQTT messages, persists them to MongoDB, and broadcasts to WebSocket clients—all in a clean, composable pipeline.

File: `IOT-Backend-main/internal/mqtt/handlers.go`

```
// Handler combines all dependencies needed for MQTT processing
type Handler struct {
```

```go
    logger      *zap.Logger           // Structured logging
    repo        repository.Repository // Database access (interface!)
    broadcaster *http.WSBroadcaster   // WebSocket push
}

// handleCoordTelemetry processes coordinator sensor data
func (h *Handler) handleCoordTelemetry(client mqtt.Client, msg mqtt.Message) {
    h.logger.Info("Received coordinator telemetry",
        zap.String("topic", msg.Topic()))

    // Step 1: Parse JSON payload into struct
    var telemetry struct {
        Ts                int64   `json:"ts"`
        LightLux          float32 `json:"light_lux"`
        TempC             float32 `json:"temp_c"`
        MmwavePresence    bool    `json:"mmwave_presence"`
        MmwaveConfidence float32 `json:"mmwave_confidence"`
        WifiRssi          int     `json:"wifi_rssi"`
    }

    if err := json.Unmarshal(msg.Payload(), &telemetry); err != nil {
        h.logger.Error("Failed to parse telemetry", zap.Error(err))
        return
    }

    // Step 2: Extract IDs from topic path
    // Topic format: site/{siteId}/coord/{coordId}/telemetry
    parts := strings.Split(msg.Topic(), "/")
    siteId := parts[1]   // e.g., "site001"
    coordId := parts[3]  // e.g., "coord001"

    // Step 3: Build domain object
    coordinator := types.Coordinator{
        Id:       coordId,
        SiteId:   siteId,
        LightLux: telemetry.LightLux,
        TempC:    telemetry.TempC,
        WifiRssi: telemetry.WifiRssi,
        LastSeen: time.Unix(telemetry.Ts, 0),
    }

    // Step 4: Persist to database (upsert = insert or update)
    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    if err := h.repo.UpsertCoordinator(ctx, &coordinator); err != nil {
        h.logger.Error("Failed to save telemetry", zap.Error(err))
        return
    }

    // Step 5: Broadcast to WebSocket clients for real-time dashboard
    if h.broadcaster != nil {
```

```
        h.broadcaster.BroadcastCoordinatorTelemetry(
            coordId, siteId,
            telemetry.Ts,
            telemetry.LightLux,
            telemetry.TempC,
            telemetry.WifiRssi,
            telemetry.MmwavePresence,
            telemetry.MmwaveConfidence,
        )
    }
}
```

What Makes This Interesting:

1. **Clean Pipeline**: Parse → Extract → Build → Persist → Broadcast
2. **Interface Injection**: `repository.Repository` is an interface, enabling testing with mocks
3. **Structured Logging**: `zap.Logger` with fields for easy debugging
4. **Context Timeout**: Prevents database operations from hanging indefinitely

3.4 WebSocket Real-time Service

Why This Code Is Important:

This Angular service demonstrates modern reactive programming with Signals and RxJS, enabling real-time updates to the dashboard without polling.

File: `IOT-Frontend-main/.../services/websocket.service.ts`

```
@Injectable({ providedIn: 'root' })
export class WebSocketService {
    private readonly env = inject(EnvironmentService);
    private ws: WebSocket | null = null;
    private reconnectAttempts = 0;

    // Angular Signals for connection state (reactive primitives)
    public readonly connected = signal<boolean>(false);
    public readonly connecting = signal<boolean>(false);
    public readonly connectionError = signal<string | null>(null);

    // RxJS Subjects for message streams (push-based)
    private readonly telemetrySubject = new Subject<NodeTelemetry | CoordinatorTelemetry>();
    private readonly presenceSubject = new Subject<PresenceEvent>();

    // Public observables for components to subscribe to
    public readonly telemetry$ = this.telemetrySubject.asObservable();
    public readonly presence$ = this.presenceSubject.asObservable();

    /**
     * Establish WebSocket connection with auto-reconnect
     */
    connect(): void {
        // Prevent duplicate connections
        if (this.ws?.readyState === WebSocket.OPEN) {
```

```
            console.warn('[WebSocket] Already connected');
            return;
        }

        this.connecting.set(true);
        this.connectionError.set(null);

        try {
            this.ws = new WebSocket(this.env.wsUrl);

            this.ws.onopen = () => {
                this.connected.set(true);
                this.connecting.set(false);
                this.reconnectAttempts = 0;
                console.log('[WebSocket] Connected');
            };

            this.ws.onmessage = (event) => {
                this.handleMessage(JSON.parse(event.data));
            };

            this.ws.onclose = () => {
                this.connected.set(false);
                this.scheduleReconnect();
            };

            this.ws.onerror = (error) => {
                this.connectionError.set('Connection error');
                console.error('[WebSocket] Error:', error);
            };
        } catch (error) {
            this.handleConnectionError(error);
        }
    }

    /**
     * Route incoming messages to appropriate subjects
     */
    private handleMessage(msg: WSMessage): void {
        switch (msg.type) {
            case 'telemetry':
                // Push to telemetry stream - all subscribers receive update
                this.telemetrySubject.next(msg.payload);
                break;
            case 'presence':
                this.presenceSubject.next(msg.payload);
                break;
            case 'status':
                // Handle device status updates
                break;
        }
    }
```

```
    /**
     * Exponential backoff reconnection
     */
    private scheduleReconnect(): void {
        const delay = Math.min(1000 * Math.pow(2, this.reconnectAttempts), 30000);
        this.reconnectAttempts++;

        setTimeout(() => {
            if (!this.connected()) {
                this.connect();
            }
        }, delay);
    }
}
```

What Makes This Interesting:

1. **Signals + RxJS Hybrid**: Connection state uses Signals (synchronous), message streams use RxJS (asynchronous)
2. **Auto-reconnect with Backoff**: Handles network interruptions gracefully
3. **Type-safe Messages**: TypeScript interfaces ensure message structure is validated
4. **Push-based Updates**: Components don't poll—they subscribe and react

---

4. Development Environment

4.1 Frameworks and Platforms

Component	Framework/Platform	Version
Coordinator	PlatformIO + Arduino	ESP-IDF 5.x
Node	PlatformIO + Arduino	ESP-IDF 5.x
Backend	Go with Uber fx	Go 1.21+
Frontend	Angular	17+
Database	MongoDB	7.0
Broker	Eclipse Mosquitto	2.0
Containerization	Docker Compose	3.8

4.2 Programming Languages

Layer	Language	Rationale
Embedded	C++ (Arduino)	Hardware access, existing libraries, team familiarity
Backend	Go	Performance, simple concurrency, single binary deployment
Frontend	TypeScript	Type safety, Angular requirement, better tooling

Configuration	YAML/JSON	Human readable, widely supported

4.3 Build and Launch Instructions

Coordinator Firmware

```
# Navigate to coordinator directory
cd coordinator

# Build and upload to ESP32-S3
pio run -e esp32-s3-devkitc-1 -t upload

# Monitor serial output
pio run -e esp32-s3-devkitc-1 -t monitor

# Combined build + upload + monitor
pio run -e esp32-s3-devkitc-1 -t upload -t monitor
```

Node Firmware

```
# Navigate to node directory
cd node

# Build for ESP32-C3
pio run -e esp32-c3-mini-1

# Upload and monitor
pio run -e esp32-c3-mini-1 -t upload -t monitor

# Debug build (verbose logging)
pio run -e esp32-c3-mini-1-debug -t upload -t monitor
```

Backend Server

```
# Navigate to backend directory
cd IOT-Backend-main/IOT-Backend-main

# Install dependencies
go mod download

# Run the server
go run cmd/iot/main.go

# Or build and run binary
go build -o iot-backend cmd/iot/main.go
./iot-backend
```

Frontend Application

```bash
# Navigate to frontend directory
cd IOT-Frontend-main/IOT-Frontend-main

# Install dependencies
npm install

# Start development server
npm start
# or
ng serve

# Build for production
npm run build
```

Full Stack with Docker Compose

```bash
# From project root
docker-compose up -d

# This starts:
# - MongoDB on port 27017
# - Mosquitto MQTT broker on port 1883
# - Backend on port 8080
# - Frontend on port 80
```

---

5. Results and Analysis

5.1 What Succeeded

✅ ESP-NOW Local Mesh Communication

Result: Achieved <10ms latency for local communication between nodes and coordinator.

Evidence:

- Measured round-trip time for `set_light` command: 8-12ms average
- Nodes successfully pair with coordinator using our JSON protocol
- Multiple nodes (tested with 4) can communicate simultaneously without collision

Why It Worked:

- ESP-NOW's connectionless design eliminates handshake overhead
- JSON payload small enough (<250 bytes) to fit in single frame
- Channel synchronization after pairing prevents lost messages

✅ MQTT Cloud Bridge

Result: Reliable telemetry delivery from embedded devices to cloud backend.

Evidence:

- Coordinator publishes telemetry every 5 seconds successfully
- Backend receives and stores 100% of messages (QoS 1)

- Auto-reconnect handles network interruptions gracefully

Why It Worked:

- QoS 1 ensures at-least-once delivery
- Topic hierarchy enables selective subscriptions
- Paho MQTT client handles reconnection automatically

✅ Real-time Dashboard Updates

Result: Dashboard shows live sensor data with <200ms total latency.

Evidence:

- WebSocket connection maintains persistent channel
- Updates appear immediately when coordinator publishes
- Charts update smoothly without polling

Why It Worked:

- WebSocket push eliminates polling overhead
- Backend immediately broadcasts on MQTT receive
- Angular Signals efficiently update only changed DOM elements

✅ Manager Pattern for Embedded Code

Result: Coordinator firmware is maintainable and extensible.

Evidence:

- Successfully added MmWave sensor without modifying other managers
- Added WiFi provisioning as new manager with minimal changes
- Team members could work on different managers in parallel

Why It Worked:

- Single Responsibility principle isolates changes
- Clear interfaces between managers
- Callback injection decouples components

5.2 What Failed and Why

❌ Deep Sleep Power Management (Partial)

Problem: Node deep sleep implementation incomplete; battery life targets not achieved.

Root Cause:

- ESP-NOW requires WiFi radio to stay on for receiving messages
- Wake-on-ESP-NOW not properly implemented
- Timeout calculations were incorrect for our use case

Impact:

- Nodes drain battery faster than designed
- Had to keep nodes powered via USB during testing

Proposed Solution:

```
// Proper deep sleep implementation (TODO):
// 1. Use scheduled wake instead of continuous listen
// 2. Implement ESP-NOW light sleep (keeps radio on, lower power)
// 3. Use longer rx_period_ms with shorter rx_window_ms

// Example configuration for ~1 week battery life:
cfg.rx_window_ms = 10;    // Listen for 10ms
cfg.rx_period_ms = 1000;  // Every 1 second (not every 100ms)
// This gives 1% duty cycle vs current 20%
```

❌ OTA Firmware Updates

Problem: Over-the-air update mechanism not implemented.

Root Cause:

- Prioritized core functionality over OTA
- ESP32 OTA requires careful partition management
- Security concerns about unauthenticated updates

Impact:

- Firmware updates require physical access to devices
- Field deployment and maintenance is impractical

Proposed Solution:

```
// OTA implementation plan:
// 1. Reserve OTA partition in partition table (already done)
// 2. Implement ArduinoOTA for development
// 3. Add MQTT-triggered OTA for production:
//    - Coordinator receives OTA command with firmware URL
//    - Downloads firmware via HTTPS
//    - Verifies signature
//    - Flashes to OTA partition
//    - Reboots to new firmware
```

❌ Node-to-Node Direct Communication

Problem: Nodes cannot communicate directly; all traffic goes through coordinator.

Root Cause:

- Design decision to simplify architecture
- Mesh networking adds complexity (routing, loops)
- Time constraints prevented mesh implementation

Impact:

- Single point of failure (coordinator)
- Higher latency for node-to-node scenarios
- Limited range (nodes must reach coordinator)

Proposed Solution:

- For v2: Implement ESP-MESH or PainlessMesh library
- Allow nodes to relay messages to extend range
- Keep coordinator as gateway but enable local mesh shortcuts

⚠️ Security Limitations

Problem: No TLS encryption for MQTT; ESP-NOW uses basic encryption only.

Root Cause:

- ESP32 TLS has high memory overhead
- Certificate management complexity
- Development environment using unencrypted broker

Impact:

- Telemetry data transmitted in plaintext over WiFi
- MQTT credentials visible on network
- Not suitable for production deployment

Proposed Solution:

```yaml
# Production MQTT configuration:
mqtt:
  broker: "mqtts://broker.example.com:8883"  # TLS port
  ca_cert: "/certs/ca.crt"
  client_cert: "/certs/client.crt"
  client_key: "/certs/client.key"
```

5.3 Lessons Learned

Technical Lessons

1. **Start with ESP-NOW, add WiFi later**
   - WiFi channel changes affect ESP-NOW; initialize ESP-NOW first
   - Use `wifi->setEspNow(espNow)` pattern for channel sync
2. **JSON is fine for small payloads**
   - Considered binary protocols but JSON worked well under 250 bytes
   - Debugging JSON over Serial is much easier than binary
3. **Signals > State Management Libraries**
   - Angular Signals replaced complex NgRx setup
   - Simpler mental model, better performance
4. **fx Lifecycle Hooks are essential**
   - Go services need proper startup/shutdown ordering
   - fx handles this automatically

Process Lessons

1. **Integration testing early**
   - Should have tested Coordinator → Backend flow earlier

- Topic name mismatches caused hours of debugging

2. **Document message formats**
    - Created `mqtt_api.md` after several schema conflicts
    - Should have been first artifact

3. **Serial logging is invaluable**
    - Added structured logging to all embedded components
    - Made debugging disconnection issues possible

Appendix A: Configuration Files

platformio.ini (Coordinator)

```
[env:esp32-s3-devkitc-1]
platform = espressif32
board = esp32-s3-devkitc-1
framework = arduino
monitor_speed = 115200
lib_extra_dirs = ../shared
lib_deps =
    bblanchon/ArduinoJson@^6.21.0
    knolleary/PubSubClient@^2.8
    adafruit/Adafruit NeoPixel@^1.11.0
build_flags =
    -DARDUINO_USB_CDC_ON_BOOT=1
    -DCONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM=16
```

docker-compose.yml

```
version: '3.8'
services:
  mongodb:
    image: mongo:7.0
    ports:
      - "27017:27017"
    volumes:
      - mongodb_data:/data/db

  mosquitto:
    image: eclipse-mosquitto:2
    ports:
      - "1883:1883"
    volumes:
      - ./mosquitto.conf:/mosquitto/config/mosquitto.conf

  backend:
    build: ./IOT-Backend-main
    ports:
```

```yaml
      - "8080:8080"
    depends_on:
      - mongodb
      - mosquitto
    environment:
      - MONGO_URI=mongodb://mongodb:27017
      - MQTT_BROKER=tcp://mosquitto:1883

  frontend:
    build: ./IOT-Frontend-main
    ports:
      - "80:80"
    depends_on:
      - backend
```

Appendix B: Video Demonstration

For a complete demonstration of the system in operation, please refer to the accompanying video that shows:

1. **Coordinator boot sequence** - Serial output showing manager initialization
2. **Node pairing flow** - Visual feedback on LEDs, Serial logging
3. **Dashboard real-time updates** - WebSocket telemetry streaming
4. **Light control** - REST API command → MQTT → ESP-NOW → LED change
5. **Presence detection** - MmWave radar triggering automation

---

Document Version: 1.0
Last Updated: December 2024
Authors: Group 15 - IOT Smart Tile System Team