부산대학교
PUSAN NATIONAL UNIVERSITY

대규모 사물인터넷 단말 관리를 위한 클라우드 기반 **OTA** 기술 개발

부산대학교 정보컴퓨터공학부
2025학년도 전기 졸업과제 중간보고서

지도교수 김 태 운
팀 명 커피는생필품
팀 원 201924479 박준우
202255665 송승우
202255670 장민준

목 차

1. 자문의견서 반영 사항

1-1. 보안 강화 방안

산업 환경에서 발생하는 보안 위협은 실제 물리적 피해로 이어질 수 있다. 따라서 임베디드 시스템을 구현할 때 보안 요소를 충분히 반영하는 것은 매우 중요하다. 자문을 통해 기존 계획에 보안 대응 메커니즘이 부족하다는 지적을 받았고, 이에 따라 다음과 같은 보안 요소를 추가하였다.

본 프로젝트에서는 Pre-signed URL이 MQTT 메시지를 통해 IoT 기기로 전달되는 구조를 사용한다. 그러나, MQTT 메시지가 위조되거나 탈취될 경우 Pre-signed URL이 노출될 수 있다는 보안적 취약점이 존재한다. 이를 보완하기 위해 다음과 같은 보안 강화 방안을 적용하였다.

1. **Mutual TLS(mTLS)** 기반 보안 세션 적용

    - **TLS** 적용**:** MQTT 통신 구간 전체에 TLS(Transport Layer Security)를 적용하여 데이터 전송 시 암호화된 보안 세션을 보장한다.

    - **Mutual TLS(mTLS)** 도입**:** 기존의 단방향 인증(TLS)에서 더 나아가, 클라우드와 IoT 기기 양쪽 모두에서 상호 인증을 수행하는 mTLS를 도입하였다. 이를 통해 클라우드 서버와 IoT 단말이 서로의 신원을 검증할 수 있다.

    - **Private CA** 운영**:** 프로젝트 전용 Private Certificate Authority(CA)를 구성하여, IoT 기기와 클라우드 시스템 각각에 인증서를 발급 및 배포하였다. 모든 MQTT 통신은 이 인증서를 기반으로 한 mTLS 환경에서만 허용된다.

    + mTLS 관련 자세한 디자인은 “3. 설계 상세화 및 변경 내역” 에서 설명한다.

2. 안전한 펌웨어 다운로드 로직 강화

    - 펌웨어 무결성 검증: OTA 배포 시 펌웨어 파일의 해시 체크섬을 함께 전달하고, 기기에서 다운로드 후 무결성 검증을 수행할 수 있도록 구현할 예정이다.

    - 롤백 메커니즘: 업데이트 실패 또는 무결성 검증 실패 시, 이전 정상 버전으로 자동 복구하는 롤백 기능을 설계에 추가하였다.

    + 3-5. IoT 기기에서 자세히 설명한다.

1-2. 네트워크 불안정성에 대한 견고성 향상

착수보고서에서 오프라인 디바이스의 자동 재배포만 언급되어 있었는데, 부분 다운로드 복구, 진행률 추적, 대역폭 적응형 다운로드 등의 고급 기능들이 필요하다는 피드백이 있었다.

이러한 네트워크 불안정성에 대비하기 위해 다음과 같은 고급 기능과 설계 개선을 반영하였다. 기본적으로 네트워크가 불안정한 것은 클라우드가 아니라 IoT 기기이므로, 기기가 주도적으로 다운로드하고 서버는 기기에 필요한 것이 있을 경우 지원해주는 방향으로 접근하였다**.**

1. 기기 주도적 다운로드 및 상태 보고

- 기기 주도 점진적 다운로드: IoT 기기는 펌웨어 파일을 직접 다운로드하며, 전체 파일을 한 번에 받지 않고, 단말이 원하는 크기로 청크 단위로 분할하여 **HTTP Range** 요청을 통해 순차적으로 다운로드한다.

- 진행상황 실시간 보고: 각 기기는 다운로드 진행 상황과 완료 상태를 MQTT 메시지로 클라우드 서버에 주기적으로 전송한다. 이를 통해 서버는 각 기기의 다운로드 상태를 실시간으로 추적할 수 있다.

2. 서버의 능동적 지원 및 업데이트 재시도

- 진행상황 기반 지원: 서버는 기기가 보고한 다운로드 진행상황과 결과를 바탕으로, 만료되었거나 유효하지 않은 Signed URL을 새로 발급해주어 기기가 다운로드를 계속할 수 있도록 지원한다.

- 업데이트 재시도 관리: 다운로드 실패, 무결성 검증 오류 등 문제가 발생한 경우, 서버는 이를 감지하고 업데이트 재시도를 자동으로 스케줄링하거나, 필요한 안내 메시지를 기기에 전달하여 원활한 복구가 이루어지도록 돕는다.

1-3. 연구 및 실험 설계의 체계화

이 프로젝트의 핵심 목표는, 클라우드 기반 OTA 기술을 활용하여 대규모 단말 환경에서도 안정적이고 효율적인 펌웨어 배포와 기기 관리를 실현하는 것이다.

객관적 성능 분석 지표와 이를 검증하기 위한 실험은 다음과 같이 설계한다.

1. 성능 평가 지표

- 배포 성공률: 전체 OTA 배포 시도 중 정상적으로 완료된 건수의 비율을 측정하여, 시스템의 신뢰성과 견고성을 평가한다.

- 평균 다운로드 시간: 각 IoT 기기가 펌웨어를 다운로드하는 데 소요된 평균 시간을 측정하여, 네트워크 환경 및 시스템 효율성을 분석한다.

- 진행률 보고 및 복구 성공률: 네트워크 단절 등 장애 발생 시, 부분 다운로드 복구 및 전체 롤백 성공률을 별도로 집계한다.

2. 실험 계획 및 시나리오

- 가상 **IoT** 기기 기반 대규모 테스트: 실제 하드웨어를 대규모로 구축하는 대신, Python 스크립트를 활용하여 다수의 가상 IoT 기기를 멀티스레드로 생성하고, 각 기기가 독립적으로 펌웨어 다운로드 및 상태 보고를 수행하도록 시뮬레이션 할 예정이다.

- 네트워크 환경 제어: Python의 네트워크 시뮬레이션 라이브러리(예: tcconfig) 등을 활용하여, 각 스레드 내에서 임의의 지연, 패킷 드롭, 대역폭 제한을 적용한다.

1-4. 구현 범위의 현실적 조정

본 프로젝트는 클라우드 기반 OTA(Over-the-Air) 서비스 개발을 목표로 하며, 졸업과제 기간 내에 핵심 기능에 집중하여 실질적 성과를 달성하고자 다음과 같이 구현 범위를 조정하였다.

1. 최우선 구현 기능
    - 펌웨어 업로드
        - 프론트엔드에서 관리자가 펌웨어 파일을 업로드 할 수 있어야 한다.

    - 펌웨어 업로드 및 배포
        - 프론트엔드에서 관리자가 “배포”를 클릭하면 기기가 OTA 업데이트를 수행할 수 있어야 한다.
        - IoT 기기는 OTA 요청을 받아, 클라우드에서 제공하는 펌웨어를 직접 다운로드하고 적용할 수 있어야 한다.

    - 안정적인 펌웨어 다운로드
        - 네트워크가 불안정하거나 다운로드 도중 연결이 끊겨도 진행상황을 추적하고 완료할 수 있어야 한다.
        - 펌웨어에 문제가 있는 경우 롤백할 수 있어야 한다.

2. 우선순위 하향 기능
    - 기기의 KPI(성능 지표) 수집, 고도화된 모니터링, 역할 기반 접근 제어(웹 페이지의 관리자 및 사용자), 복잡한 배포 스케줄링 기능

1-5 상용 플랫폼과 비교

임베디드 OTA 분야의 대표적인 상용 솔루션 중 하나인 **Mender**[1]는 본 프로젝트와 유사하게 대규모 OTA 업데이트 기능을 제공한다. 그러나 Mender는 Raspberry Pi와 같은 임베디드 리눅스 기기를 기반으로 Docker 컨테이너 교체를 중심으로 작동하는 구조이기 때문에, **MCU** 기반의 소형 **IoT** 기기에는 적용이 어렵다.

MCU 환경에서 사용할 수 있는 OTA 솔루션으로는 AWS의 FreeRTOS OTA, TI(Texas Instruments)의 OTA 기술 등 다양한 시도가 존재하지만, 이들 기술은 주로 저수준 API 제공에 초점을 두고 있으며, 펌웨어 등록부터 배포까지 일관된 서비스 흐름을 갖춘 솔루션은 드물다.

물론 MCU 환경은 하드웨어 API에 강하게 종속되기 때문에, 모든 기기에 범용적으로 적용 가능한 OTA 솔루션을 개발하는 것은 현실적으로 어렵다. 그럼에도 불구하고, 펌웨어 등록 → 송신 → 배포까지의 기초적인 **OTA** 파이프라인을 통합적으로 제공하고, 각 MCU의 특성에 맞춰 펌웨어를 맞춤 조정해 적용할 수 있는 프레임워크를 개발하는 일은 기술적 / 산업적 가치 모두에서 의미 있는 작업이라 할 수 있다.

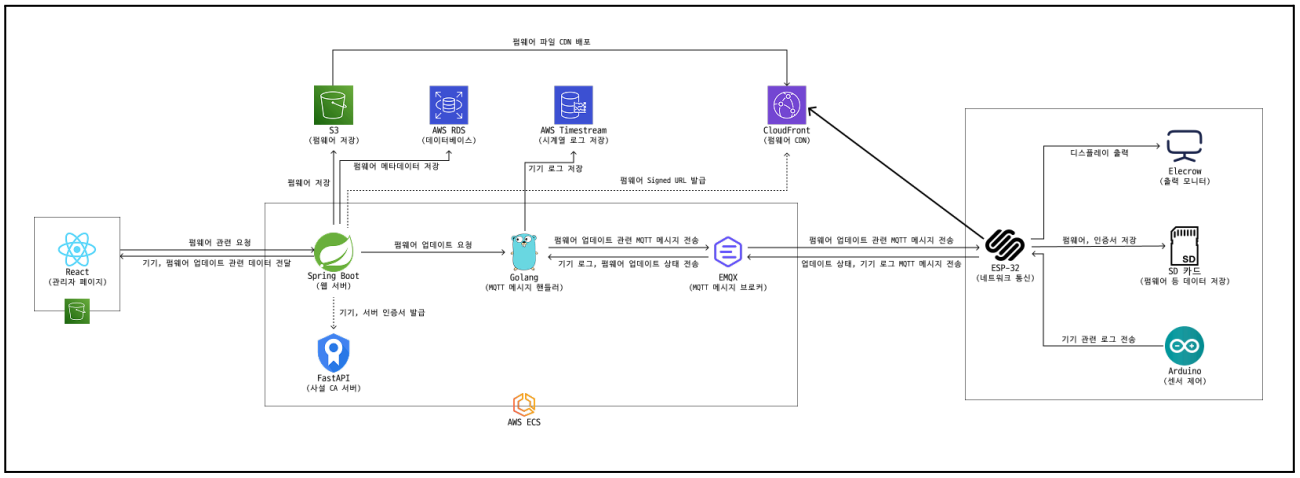
2. 요구조건 및 제약 사항 분석에 대한 수정사항

아래는 착수보고서 작성 당시에 작성되었던 요구사항을 기준으로, 변경된 부분에 대해 설명한다.

기능	설명	비고
펌웨어 업로드	관리자가 웹 클라이언트를 통해 새로운 펌웨어 파일을 업로드 할 수 있어야 한다.	유지
펌웨어 OTA 배포	업로드된 펌웨어를 특정 디바이스 또는 그룹에 원격으로 배포할 수 있어야 한다.	유지
~~배포 스케줄 설정~~	~~통합된 컨텐츠를 특정 기기 또는 디바이스 집단에 지정된 조건에 맞춰 배포할 수 있어야 한다.~~	우선순위 낮춤
디바이스별 펌웨어 조회	각 디바이스에 적용된 펌웨어 이력을 조회할 수 있어야 한다.	유지
~~디바이스 KPI 모니터링~~	~~디바이스의 온도, 네트워크 상태, 작동 시간 등의 상태 정보를 확인할 수 있어야 한다.~~	우선순위 낮춤
디바이스 목록 확인	디바이스 ID, 위치 등의 목록 정보를 확인할 수 있어야 한다.	유지
로그 수집	디바이스에서 발생한 로그나 오류 정보를 서버로 수집할 수 있어야 한다.	유지
파일 메타데이터 관리	업로드한 펌웨어에 대한 버전, 작성자, 등록일의 정보를 기록할 수 있어야 한다.	유지
MQTT 메시지 송수신	MQTT를 통해 디바이스에 다운로드 링크를 전달할 수 있어야 한다.	유지
업데이트 실패 대응	업데이트 실패 시 롤백하거나 재시도를 할 수 있어야 한다.	유지
CDN 연동	펌웨어를 CDN을 통해 빠르게 전달하고, 디바이스는 링크를 통해 다운로드할 수 있어야 한다.	유지
기기-클라우드 보안 통신	디바이스와 클라우드 사이에 주고 받는 메시지는 외부로 노출되거나 조작되어서는 안된다.	추가

3. 설계 상세화 및 구현 내용

3-1. 주요 아키텍처 설계 방향 및 변경 배경



초기에는 AWS Lambda를 활용한 서버리스 백엔드와 AWS IoT Core 기반의 MQTT 브로커를 사용하여 시스템을 구축할 계획이었다. 그러나 이 방식은 **AWS** 서비스에 대한 의존도가 높아지고, 서버리스 환경 특성상 기능 확장과 커스터마이즈에 한계가 있으며, 장기적으로 다른 클라우드 서비스나 온프레미스 환경으로의 이관이 어렵다는 문제가 있었다.

이에 따라, 시스템의 유연성과 이식성을 높이기 위해 아키텍처를 다음과 같이 재설계하였다.

- Lambda → **Spring Boot**
  - 백엔드 서버는 서버리스(Lambda) 대신 Spring 프레임워크 기반의 전통적인 서버로 전환하여, 복잡한 비즈니스 로직 구현과 다양한 확장 기능을 유연하게 지원할 수 있도록 했다.

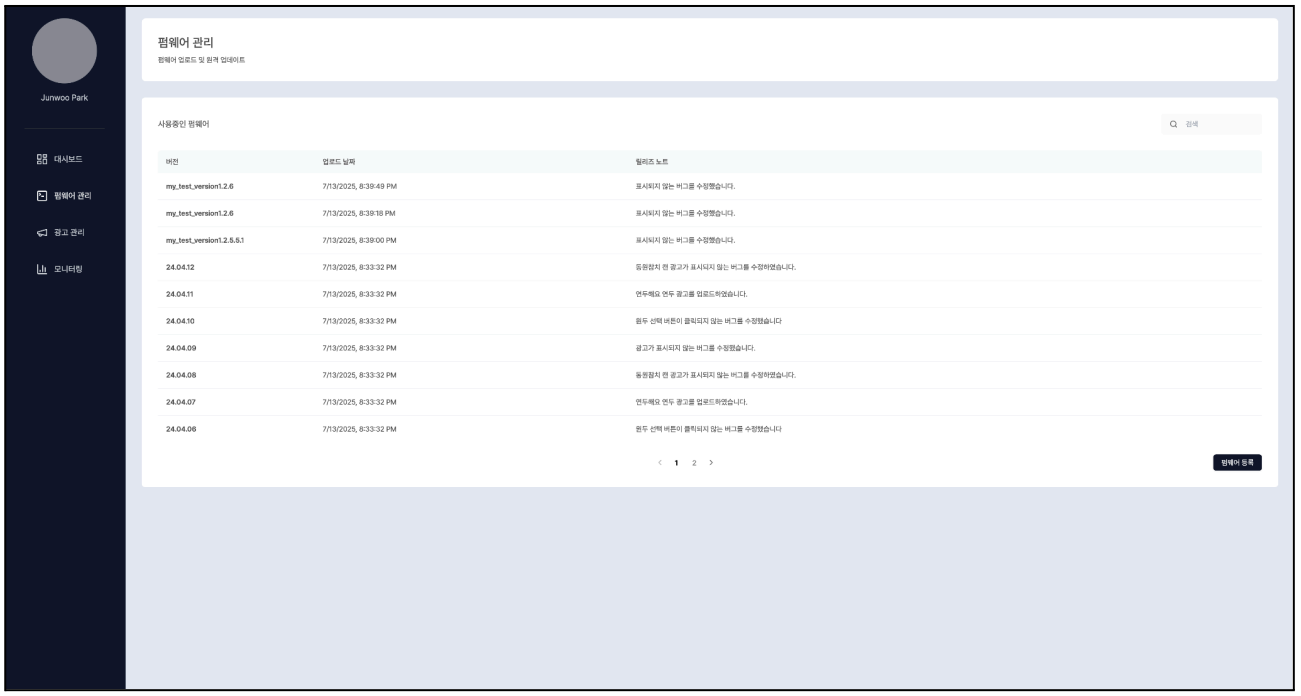
- AWS IoT Core → **EMQX**
  - MQTT 브로커는 AWS IoT Core 대신 오픈소스 MQTT 브로커인 EMQX를 도입하여, 클라우드 환경뿐 아니라 온프레미스 환경에서도 동일한 아키텍처로 운영이 가능하도록 했다.

- N/A → **Golang MQTT** 핸들러 서비스
  - MQTT 메시지 처리를 담당하는 핸들러 서비스는 Golang으로 별도 구현하여, 경량화와 고성능 처리가 가능하도록 설계했다.

이러한 변경을 통해 AWS에 대한 종속성을 줄이고, 필요 시 타 클라우드 서비스 또는 온프레미스 환경으로의 손쉬운 마이그레이션과 아키텍처의 유연성을 확보하였다. 각 서비스에 대한 자세한 설명은 아래에서 설명한다.

3-2. 프론트엔드

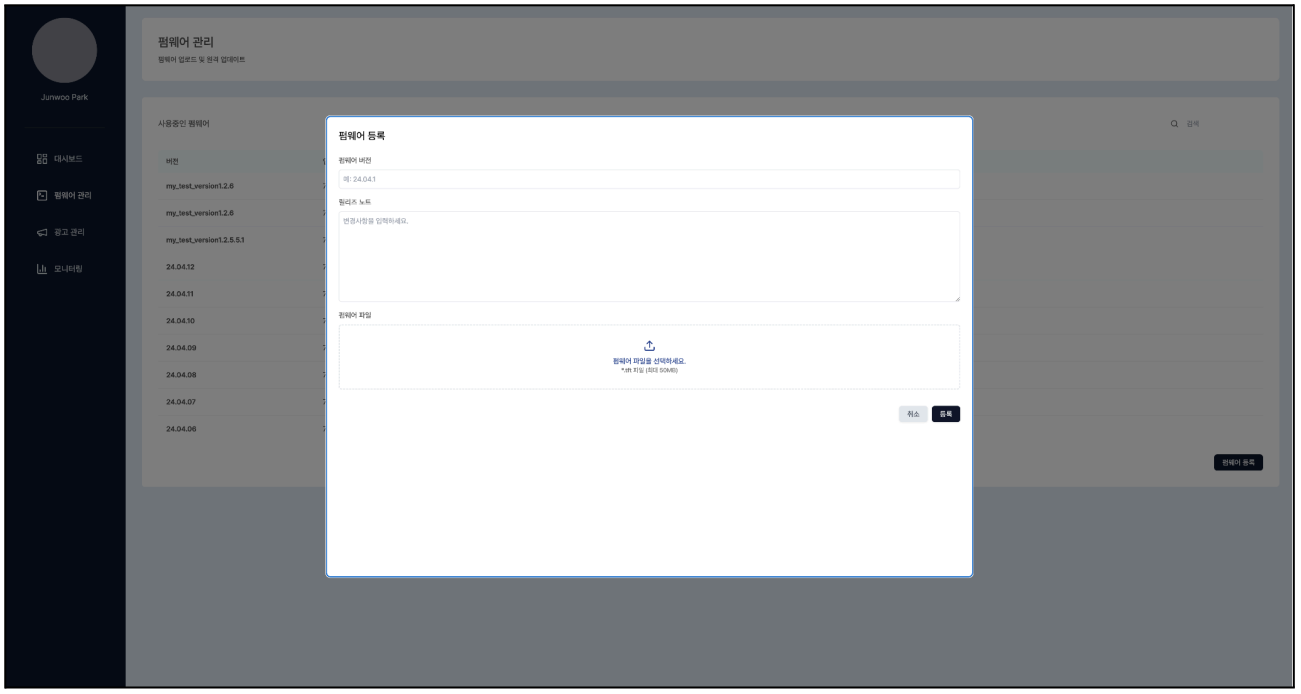
펌웨어 관리 페이지



펌웨어 관리 페이지는 관리자가 다양한 펌웨어 정보를 한눈에 파악하고, 효율적으로 관리할 수 있도록 다음과 같이 구성했다.

1. 표 형태의 펌웨어 목록 제공
    - 모든 펌웨어는 표 형태로 나열하여, 각 펌웨어의 주요 정보를 직관적으로 확인할 수 있도록 했다.
    - 표는 펌웨어 이름, 버전, 업로드 날짜, 릴리즈 노트 등의 정보를 포함하도록 하였다.
2. 키워드 검색 및 필터링 기능
    - 상단에 검색 입력창을 배치하여, 관리자가 펌웨어 이름이나 버전 정보를 입력하면 관련 항목만 신속하게 필터링하여 보여주도록 했다.
    - 검색 결과는 표 형태로 즉시 반영되어, 원하는 펌웨어를 빠르게 찾을 수 있도록 했다.

펌웨어 등록 페이지



펌웨어 등록 페이지는 관리자가 새로운 펌웨어를 효율적으로 등록할 수 있도록 다음과 같은 구조와 절차를 따른다.

입력 항목

- 버전 정보: 등록할 펌웨어의 버전(예: 1.0.3)
- 릴리즈 노트: 해당 버전의 변경사항 및 주요 내용을 입력
- 펌웨어 파일: 실제 펌웨어 바이너리 파일 업로드

저장 방식

- 버전 정보, 릴리즈 노트: RDS의 메타데이터로 저장
- 펌웨어 파일: S3 버킷에 저장

업로드 절차

펌웨어 등록 과정은 다음 3단계로 이루어진다.

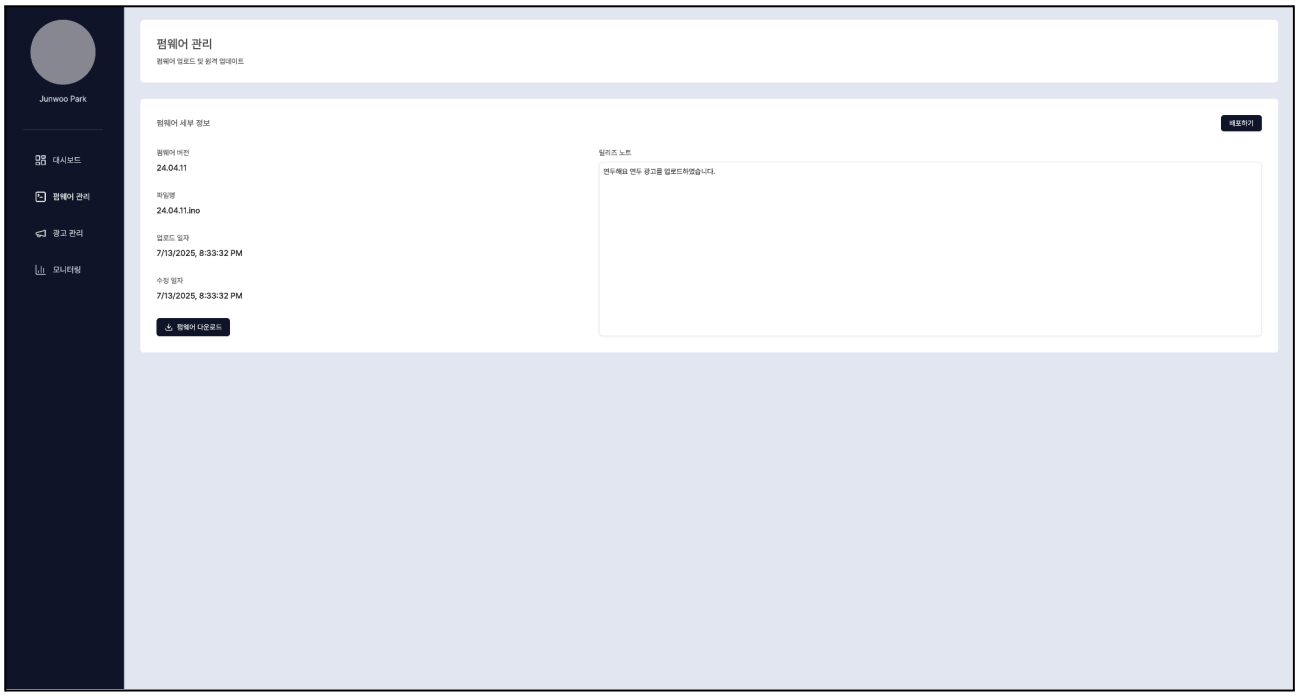
1. **Presigned URL** 발급 요청
   - 사용자가 입력한 버전, 파일 이름 정보를 서버에 전송하여 Presigned URL 발급을 요청한다.

2. 펌웨어 파일 업로드
   - 서버로부터 받은 Presigned URL을 이용해, 클라이언트에서 S3 버킷으로 펌웨어 파일을 직접 업로드한다.
   - S3에 10MB 이상의 대용량 파일을 안전하게 업로드하기 위해 Presigned URL 방식이 필수적이다.

3. 메타데이터 등록 요청
   - 파일 업로드가 성공적으로 완료된 경우에만, 버전 정보와 릴리즈 노트 등 메타데이터를 서버에 등록 요청한다.

이 절차를 통해 파일 업로드 실패 시 잘못된 메타데이터가 등록되는 것을 방지한다.

펌웨어 상세정보 페이지

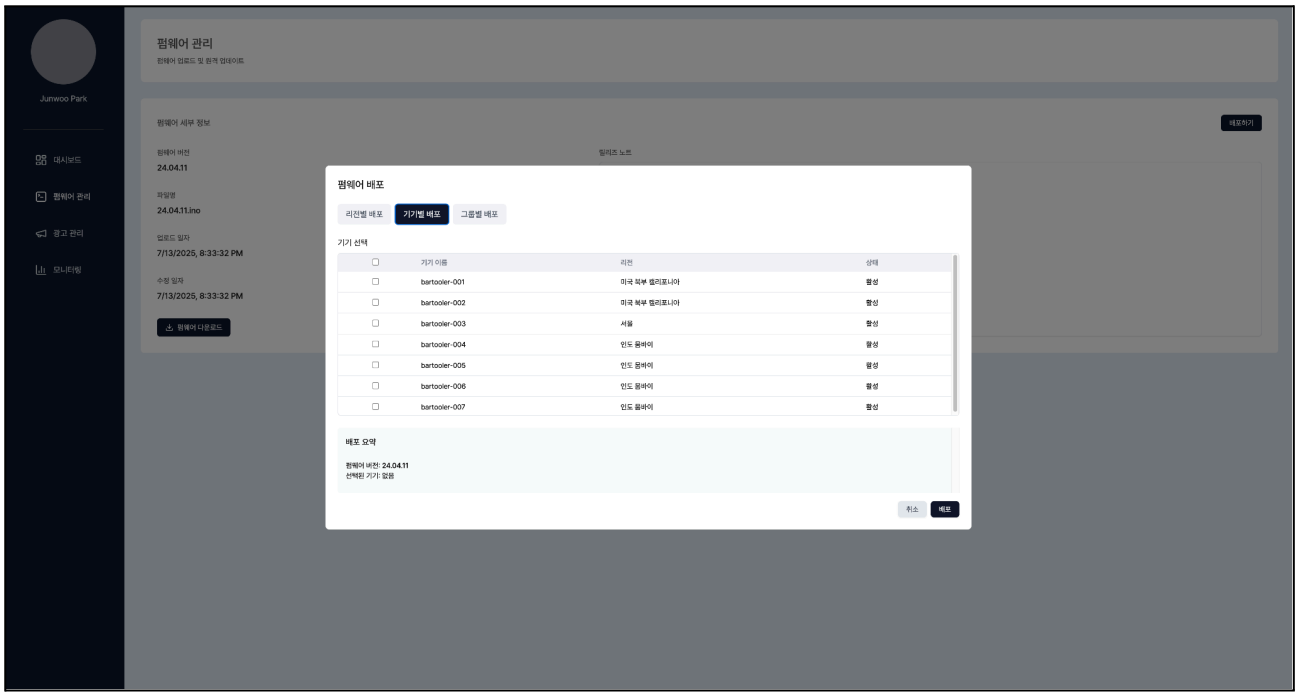


펌웨어 상세정보 페이지는 관리자가 펌웨어 관리 페이지에서 특정 펌웨어 항목을 클릭했을 때 접근할 수 있도록 설계했다. 이 페이지에서는 선택한 펌웨어의 모든 세부 정보를 한눈에 확인할 수 있으며, 직접 펌웨어 파일을 다운로드할 수 있는 기능도 제공한다.

1. 펌웨어 주요 정보 제공
    - 버전 정보, 펌웨어 파일명, 릴리즈 노트, 업로드 일자, 수정 일자 등의 주요 정보를 제공한다.

2. 펌웨어 다운로드 기능
    - “펌웨어 다운로드 버튼”을 클릭하면 해당 펌웨어 파일을 직접 다운로드 할 수 있도록 하였다.

펌웨어 배포 페이지



펌웨어 배포 페이지는 관리자가 펌웨어 상세보기 페이지에서 "배포하기" 버튼을 클릭했을 때 접근할 수 있도록 설계했다. 이 페이지를 통해 관리자는 선택한 펌웨어를 다양한 기준(리전, 그룹, 디바이스)으로 손쉽게 배포할 수 있다.

1. 배포 대상 선택 기능
    - 리전별 배포: 관리자는 특정 리전(예: 서울)에 속한 모든 기기를 선택하여 일괄적으로 펌웨어 업데이트를 요청할 수 있다.
    - 그룹별 배포: 기기가 속한 그룹(예: 스타벅스 명지점)을 기준으로, 해당 그룹 내 모든 기기를 대상으로 업데이트를 진행할 수 있다.
    - 디바이스별 배포: 특정 디바이스들을 개별적으로 선택하여, 선택된 기기들에 한정해 펌웨어를 배포할 수 있다.

각 배포 방식은 드롭다운, 체크박스, 검색 기능 등을 통해 직관적으로 선택할 수 있도록 UI를 구성했다.

2. 펌웨어 배포 요약
    - 타겟 펌웨어 버전: 현재 배포할 펌웨어의 버전 정보를 명확히 표시한다.
    - 선택된 리전: 배포 대상 리전(들)을 나열한다.
    - 선택된 그룹: 배포 대상 그룹(들)을 나열한다.
    - 선택된 기기: 개별적으로 선택된 디바이스 목록을 보여준다.

각 항목은 명확하게 구분된 영역에 표시되어, 관리자가 배포 전 최종적으로 선택 내역을 쉽게 검토할 수 있도록 했다.

3. 배포 요청

- 관리자는 배포 대상과 펌웨어 정보를 최종 확인한 후, "배포 요청" 버튼을 클릭하여 서버에 배포 요청을 전송할 수 있다.

3-3. 백엔드

Lambda → Spring Boot

기존 아키텍처에서는 AWS Lambda 기반의 Serverless 방식을 사용하였으나, 다음과 같은 이유로 **Spring Boot** 기반의 서버 구조로 전환하였다.

1. **AWS** 의존성에 따른 플랫폼 락인 문제
   - Lambda 중심의 구조는 AWS의 다양한 서비스(API Gateway, CloudWatch 등)에 강하게 종속되어 있어, 향후 타 클라우드 또는 온프레미스 환경으로의 이식이 어렵고, 플랫폼 락인(lock-in) 가능성이 존재했다.
2. 장애 대응의 유연성 부족
   - Lambda는 실행 환경에 대한 제어가 제한적이며, 디버깅이나 상태 추적이 어려워 장애 발생 시 신속한 원인 분석 및 대응이 어렵다.
3. 복잡한 로직 처리의 비효율성
   - Serverless 구조는 로직을 여러 Lambda 함수로 분리하여 구성해야 하며, 상태 전달을 위해 중간 저장소(S3, DB 등)를 거쳐야 하므로, 전체 흐름의 가독성과 유지보수성이 떨어지고, 트랜잭션 단위의 오류 복구가 어렵다.

이러한 문제점을 고려하여 **Spring Boot** 기반의 백엔드로 아키텍처를 변경하였으며, 이를 통해 장애 대응 능력 향상**,** 복잡한 로직의 일관된 처리**,** 유지보수 용이성**,** 플랫폼 독립성 확보 등의 이점을 얻을 수 있었다.

백엔드 시스템 설계 및 기술 선정 배경

본 프로젝트의 백엔드는 **Spring Boot 3.5.3**과 **Java17**을 기반으로 개발되었으며, 데이터베이스 연동에는 **JPA (Java Persistence API)** 를 사용하였다. 각 기술은 다음과 같은 이유로 선택하였다.

1. **Spring Boot**
   - 빠른 초기 개발과 배포가 가능하며, 내장 톰캣과 자동 설정으로 생산성이 높다.
   - 예외 처리, 로깅, 트랜잭션 관리 등이 통합되어 운영과 유지보수가 용이하며, Docker Kubernetes 등 다양한 환경에서 유연하게 확장할 수 있다.
2. **JPA (Java Persistence API)**
   - ORM 기반 데이터 처리 방식으로, 객체와 테이블 간 매핑을 자동화 한다.
   - SQL 작성 없이 CRUD가 가능하고, @Transactional로 트랜잭션 관리가 쉬우며, 페이징, 정렬 등 기능도 내장되어 있어 생산성과 유지보수성이 높다.

API 설계

본 시스템은 RESTful한 방식으로 API를 설계하였으며, 자원 중심의 URL 구조와 HTTP 메서드의 의미를 고려하여 각 기능을 정의하였다.

URL	**Method**	설명
/api/s3/presigned_upload	POST	S3 Presigned URL을 발급 받아 펌웨어 파일 업로드를 위한 URL을 요청
/api/firmwares/metadata	POST	업로드된 펌웨어의 메타데이터를 등록
/api/firmwares/metadata	GET	현재 업로드된 모든 펌웨어 목록을 조회

/api/firmware/${id}	GET	특정 펌웨어의 세부 정보를 조회
/api/s3/presigned_download	GET	해당 펌웨어를 다운로드할 수 있는 S3 Presigned URL을 요청
/api/regions	GET	모든 리전과 각 리전에 속한 디바이스 리스트를 조회
/api/divisions	GET	모든 그룹과 각 그룹에 속한 디바이스 리스트를 조회
/api/devices	GET	모든 디바이스 리스트를 조회

API 상세 동작 흐름

1. 펌웨어 업로드용 **Presigned URL** 발급
    - 엔드포인트: /api/s3/presigned_upload (POST)
    - 입력: version, fileName
    - 처리과정:
        - DB에서 중복된 펌웨어(version + fileName) 존재 여부 확인
        - UUID를 생성하여 version/UUID/fileName 형식의 고유 S3 경로 구성
        - 해당 경로에 대해 PUT 방식 Presigned URL 생성 및 클라이언트에 반환

2. 펌웨어 메타데이터 등록
    - 엔드포인트: /api/firmwares/metadata (POST)
    - 입력: version, fileName, releaseNote, s3Path
    - 처리과정:
        - S3의 해당 경로에서 파일을 스트리밍 방식으로 읽어옴
        - 파일 내용을 기반으로 SHA-256 해시값 계산
        - 해시값 및 입력 정보를 DB에 저장

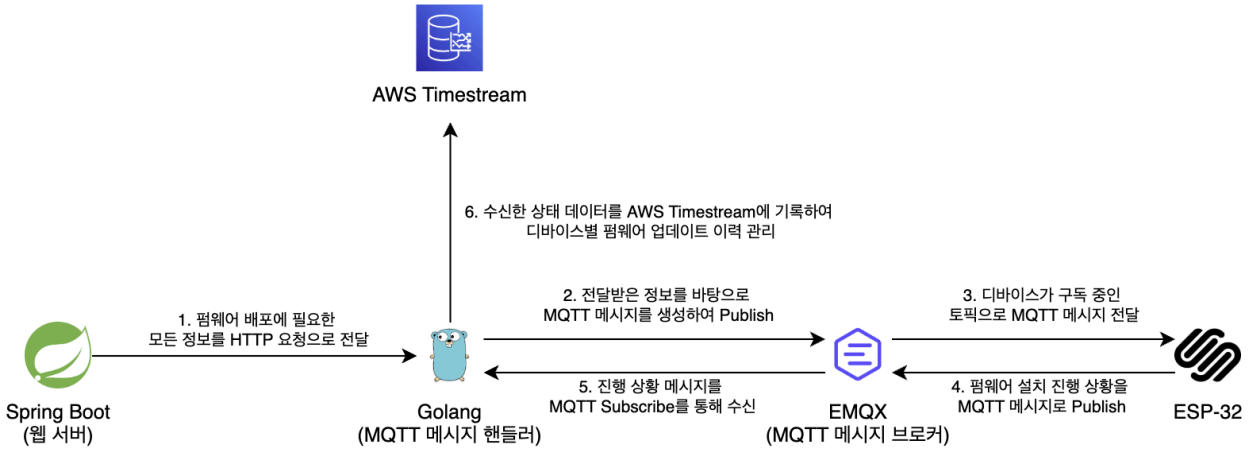
3. 펌웨어 목록 조회
    - 엔드포인트: /api/firmwares/metadata (GET)
    - 출력: 등록된 모든 펌웨어의 메타데이터 리스트

4. 펌웨어 상세 조회
    - 엔드포인트: /api/firmware/{id} (GET)
    - 입력: 펌웨어 ID
    - 출력: 해당 펌웨어의 상세 정보

5. 펌웨어 다운로드용 **Presigned URL** 발급
    - 엔드포인트: /api/s3/presigned_download (GET)
    - 입력: version, fileName
    - 처리 과정:
        - DB에서 해당 펌웨어의 s3Path 조회
        - 해당 경로에 대해 GET 방식 Presigned URL 생성 후 반환

6. 디바이스 및 리전**,** 그룹 정보 조회
    - /api/regions : 리전별 디바이스 리스트 반환
    - /api/divisions : 그룹별 디바이스 리스트 반환
    - /api/devices : 전체 디바이스 리스트 반환

Golang MQTT 핸들러 서비스



IoT 디바이스와의 실시간 통신 및 펌웨어 배포 처리를 전담하기 위해, EMQX 브로커와 연동되는 별도의 **MQTT** 핸들러 서비스를 **Golang** 기반으로 구성한다. 이 핸들러는 펌웨어 업데이트 관련 **MQTT** 메시지를 **Publish/Subscribe**하는 역할을 수행한다.

펌웨어 배포 요청은 **Spring Boot** 서버에서 처리하며, 이때 대상 디바이스 정보**,** 펌웨어 버전**, Presigned URL, SHA-256** 해시값 등 모든 필요한 데이터를 생성하여 **Golang** 핸들러로 전달한다. **Golang** 핸들러는 메시지 전달에만 집중하며, 전달받은 정보를 바탕으로 해당 디바이스에 **MQTT** 메시지를 **publish**하여, **"**지정된 **URL**에서 펌웨어를 다운로드하라**"**는 명령을 전송한다. 이처럼 **Presigned URL** 생성 및 정책 관리는 **Spring Boot,** 실제 메시지 전송은 **Golang** 핸들러가 담당하는 구조로 역할을 명확히 분리하였다.

Golang은 높은 동시성 처리 능력과 효율적인 메모리 사용**,** 고성능 네트워크 처리에 최적화된 언어로, 수백~수천 개의 디바이스로부터 동시 요청이 들어오는 환경에서도 안정적인 처리가 가능하다. 핸들러는 각 디바이스에 대한 **MQTT** 메시지 전송을 고루틴을 활용해 병렬적으로 처리한다. 각 메시지 전송 작업은 **go publishToDevice(info)** 형태의 고루틴으로 실행되며, 채널 또는 **WaitGroup**등을 활용하여 전체 처리 흐름과 예외 처리를 제어한다. 이러한 구조를 통해 메인 스레드의 부하를 최소화하면서**,** 대량의 디바이스에 대한 신속하고 안정적인 메시지 전송을 지원한다.

또한, 디바이스로부터 수신한 펌웨어 설치 상태 및 로그 메시지는 **Subscribe**된 토픽을 통해 수신하고, 이를 분석하여 **AWS Timestream**에 저장함으로써 기기별 업데이트 이력 관리 및 모니터링을 수행한다.

펌웨어 배포 방식 및 파일 처리 전략

펌웨어 파일은 **AWS S3**에 원본 파일 그대로 업로드 및 보관한다. 이후 IoT 디바이스에서는 HTTP Range 요청을 통해 필요한 구간만 청크 단위로 다운로드할 수 있도록 설계하였다.
이러한 구조는 다음과 같은 장점을 가진다.

- 디바이스 네트워크 상황에 맞춘 유연한 다운로드
- 다운로드 중단 시 재시도 로직 구현이 용이함
- 서버에서의 불필요한 처리 부하 최소화

보안 검증**:** 해시섬**(Hash Checksum)** 제공

펌웨어의 무결성 검증을 위해, S3에 업로드된 펌웨어 파일에 대해 **SHA-256** 해시섬 **(Hash Checksum)** 을 계산하고, 해당 값을 메타데이터와 함께 저장한다.

펌웨어 배포 요청 시, Spring Boot 서버는 다음 정보를 **HTTP** 요청으로 **Golang** 핸들러에 전달한다.

- 다운로드용 Presigned URL (일회성, 시간 제한 포함)

- SHA-256 해시값
- 버전 및 파일명 등 메타데이터

디바이스는 다운로드 후 해시를 재계산하고, 전달받은 해시 값과 비교하여 무결성을 검증한다. 이를 통해 파일 위·변조 방지**,** 보안성 강화**, OTA** 배포의 신뢰성 확보가 가능하다.

3-4. 클라우드**/**인프라

AWS IoT Core → EMQX

클라우드 인프라 설계에서 AWS IoT Core 대신 EMQX를 도입한 주요 배경과 이유는 다음과 같다.

1. 서비스 기능적 한계 극복
    - **QoS 2** 미지원: AWS IoT Core는 MQTT 프로토콜의 QoS(Quality of Service) 2를 지원하지 않는다. QoS 2는 메시지의 중복 없는 정확한 1회 전달(Exactly once delivery)을 보장하는 등, 산업 환경에서 신뢰성이 중요한 메시지 전송에 필수적인 기능이다. EMQX는 MQTT 표준에 따라 QoS 2를 완전하게 지원하여, 더 높은 신뢰성과 데이터 무결성을 확보할 수 있다.

    - 메시지 순서 보장 미지원**:** AWS IoT Core는 메시지 순서(Message Ordering)를 보장하지 않는다. 대규모 IoT 환경에서는 기기 상태나 명령 메시지가 순서대로 처리되어야 하는 경우가 많으나, AWS IoT Core에서는 메시지 도착 순서가 보장되지 않아 애플리케이션 레벨에서 추가적인 처리 로직이 필요하다. EMQX는 메시지 순서 보장과 관련된 다양한 설정과 확장 기능을 제공한다.

2. 클라우드 종속성 최소화
    - **AWS** 서비스 의존도 감소: AWS IoT Core와 같은 매니지드 서비스는 편리하지만, 특정 클라우드 벤더에 종속되는 문제가 있다. 장기적으로 타 클라우드 또는 온프레미스 환경으로의 이전(마이그레이션)이 필요할 때 큰 제약이 될 수 있다. EMQX는 오픈소스 기반의 독립적인 MQTT 브로커로, 다양한 클라우드 환경과 온프레미스 모두에서 동일한 아키텍처로 운영이 가능하다.

    - 유연성 및 확장성 확보**:** EMQX는 커스터마이즈와 확장성이 뛰어나며, 자체적인 플러그인 개발, 다양한 인증·보안 옵션, 고급 메시지 라우팅 등 복잡한 요구사항도 유연하게 대응할 수 있다.

MQTT 토픽 설계

펌웨어 관련

토픽 **ID**	방향	데이터
firmware/download/request	클라우드 → 기기	펌웨어 다운로드 요청 명령
firmware/download/cancel	클라우드 → 기기	펌웨어 다운로드 취소 요청
firmware/download/request/ack	기기 → 클라우드	펌웨어 다운로드 요청 수신 확인
firmware/download/progress	기기 → 클라우드	펌웨어 다운로드 진행 상태
firmware/download/result	기기 → 클라우드	펌웨어 다운로드 최종 결과
firmware/download/cancel/ack	기기 → 클라우드	펌웨어 다운로드 취소 수신 확인

기기 상태 관련

토픽 **ID**	방향	데이터
status/system	기기 → 클라우드	기기 시스템 리소스 상태
status/network	기기 → 클라우드	기기 네트워크 연결 상태
status/health	기기 → 클라우드	기기 전반적 상태 및 진단 정보

기기 데이터 관련

토픽 **ID**	방향	데이터
data/sensors	기기 → 클라우드	Arduino 센서로부터 수집한 데이터
data/metrics	기기 → 클라우드	기기 성능 지표 및 KPI 데이터

기기 컨트롤 관련

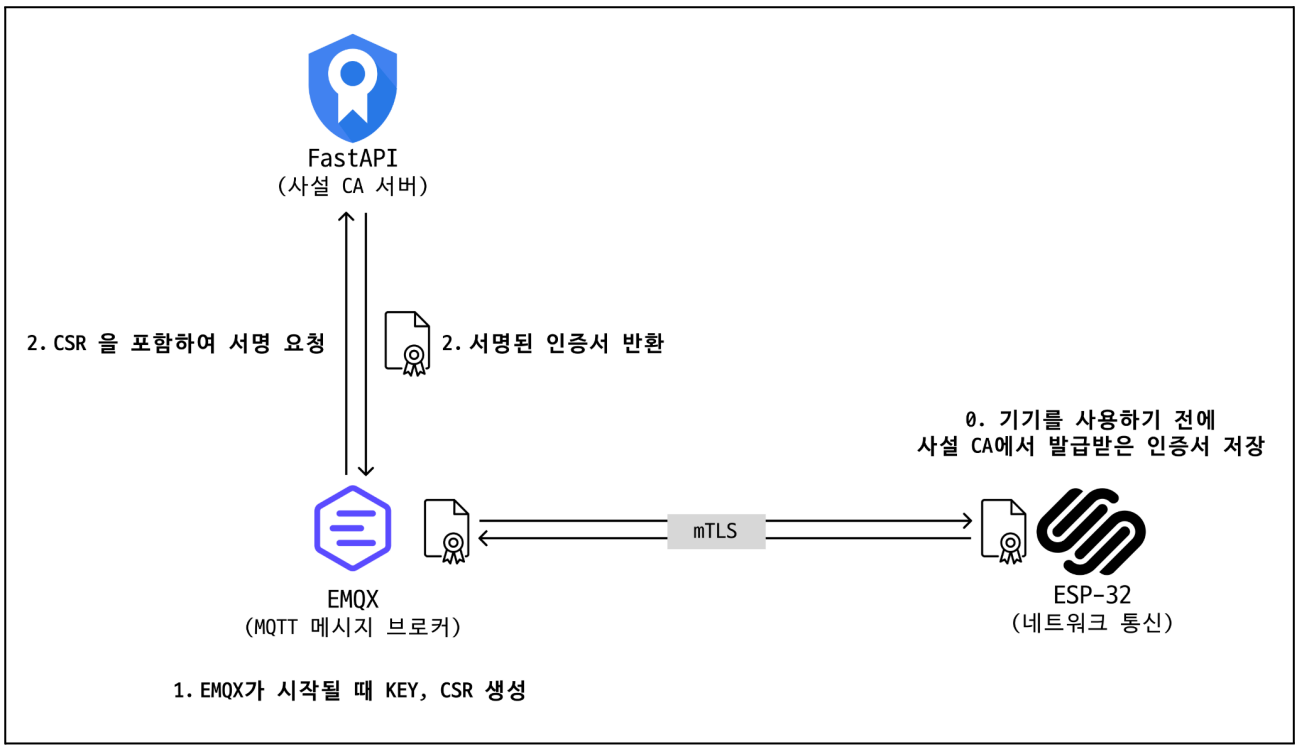
토픽 **ID**	방향	데이터
control/command	클라우드 → 기기	기기 제어 명령 (펌웨어 다운로드 외 일반 명령)
control/response	기기 → 클라우드	제어 명령에 대한 응답

인증서 관련

토픽 **ID**	방향	데이터
/cert/renew/request	기기 → 클라우드	인증서 갱신 요청 (자신의 CSR 전달)
/cert/renew/response	클라우드 → 기기	인증서 전달

mTLS 구현

본 프로젝트에서는 상용 인증서 서비스의 비용적 한계를 극복하고, IoT 환경에 적합한 인증서 발급 및 관리 체계를 위해 사설 **CA(**인증기관**)**를 직접 구축하여 **mTLS**를 구현했다. 구현 방식은 다음과 같다.



1. 사설 **CA(**인증기관**)** 구축 및 운영
   - 루트 **CA** 생성: 최초에 루트 CA의 개인키와 루트 인증서(ca.cert)를 생성한다.
   - 인증서 발급 **API** 제공: /api/issue_cert 엔드포인트를 통해 CSR(Certificate Signing Request)을 접수받아, 서명된 인증서와 루트 CA 인증서를 반환한다.
   - 저비용·유연성 확보: Let's Encrypt 등 공개 인증서 발급 시스템은 IoT 기기 환경에 적합하지 않고, AWS IoT Core, Azure IoT Hub 등 상용 서비스는 월 $400 이상의 비용 부담이 있으므로, 사설 CA를 직접 운영하여 비용과 유연성을 모두 확보했다.

2. 기기 인증서 발급 및 저장
   - 사전 인증서 발급: IoT 기기는 제품 출고 전 사설 CA로부터 인증서를 미리 발급받아 저장한다.
   - 기기별 고유 인증서: 각 기기는 고유한 인증서를 사용하여, 네트워크 연결 시 mTLS 인증 과정을 거친다.

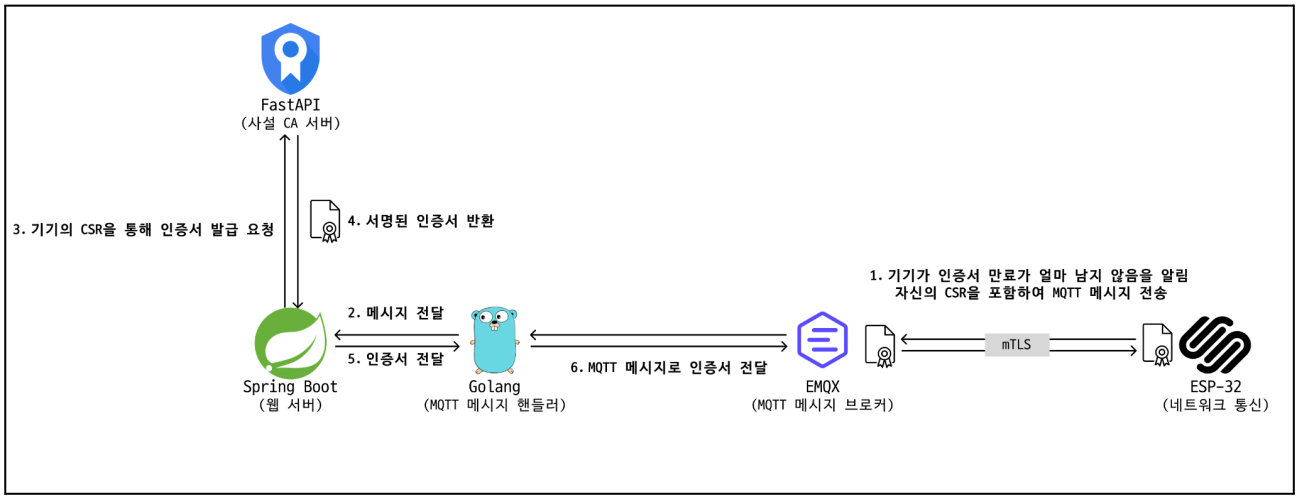
3. **EMQX MQTT** 브로커의 **mTLS** 활성화
   - 컨테이너 구동 시 인증서 자동 발급: EMQX 컨테이너가 시작될 때, 자체적으로 개인키와 CSR을 생성하여 사설 CA에 인증서 발급을 요청한다. 이 때, CSR의 SAN(Subject Alternative Name) 필드에 EMQX의 실제 IP 주소를 포함시켜 기기가 EMQX의 도메인을 신뢰할 수 있도록 하였다**.**
   - 인증서 저장 및 **mTLS** 활성화**:** 발급받은 인증서를 저장하고, MQTT의 8883 포트(보안 포트)에 대해 mTLS를 활성화하여, 기기와 브로커 간 상호 인증이 이루어지도록 한다

```
## EMQX SSL/TLS 리스너 설정 (emqx.conf)
listener.ssl.external = 8883
```

```
listener.ssl.external.keyfile = etc/certs/emqx.key        # 서버 개인키
listener.ssl.external.certfile = etc/certs/emqx.pem       # 서버 인증서(체인 포함)
listener.ssl.external.cacertfile = etc/certs/ca.pem       # 사설 CA 루트 인증서

listener.ssl.external.verify = verify_peer                # 클라이언트 인증서 검증 활성화 (mTLS)
listener.ssl.external.fail_if_no_peer_cert = true         # 클라이언트 인증서 없으면 연결 거부
```

기기의 인증서 갱신 방법



1. 인증서 만료 체크 및 갱신 요청
   a. 기기는 정기적으로 자신의 인증서 만료일을 확인한다.

   b. 인증서 만료 시점이 30일 이내로 남았을 경우, MQTT의 /cert/renew/request 토픽으로 "기기 ID"와 CSR(Certificate Signing Request, 인증서 서명 요청)을 안전하게 전송한다.

   - 이 과정에서 MQTT 통신 자체가 이미 mTLS로 보호되어 있으므로, 인증서 요청 메시지와 CSR이 외부에 노출되거나 변조될 위험 없이 안전하게 처리된다.

2. 인증서 갱신 처리 흐름
   a. 기기가 /cert/renew/request로 MQTT 메시지를 전송(자신의 ID, CSR 포함).

   b. Golang 기반 MQTT 메시지 핸들러가 해당 메시지를 수신하여 Spring Boot 서버에 전달.

   c. Spring Boot 서버는 받은 기기 ID와 CSR 정보를 바탕으로 자체 사설 CA(인증기관)에 새로운 인증서 발급을 요청.

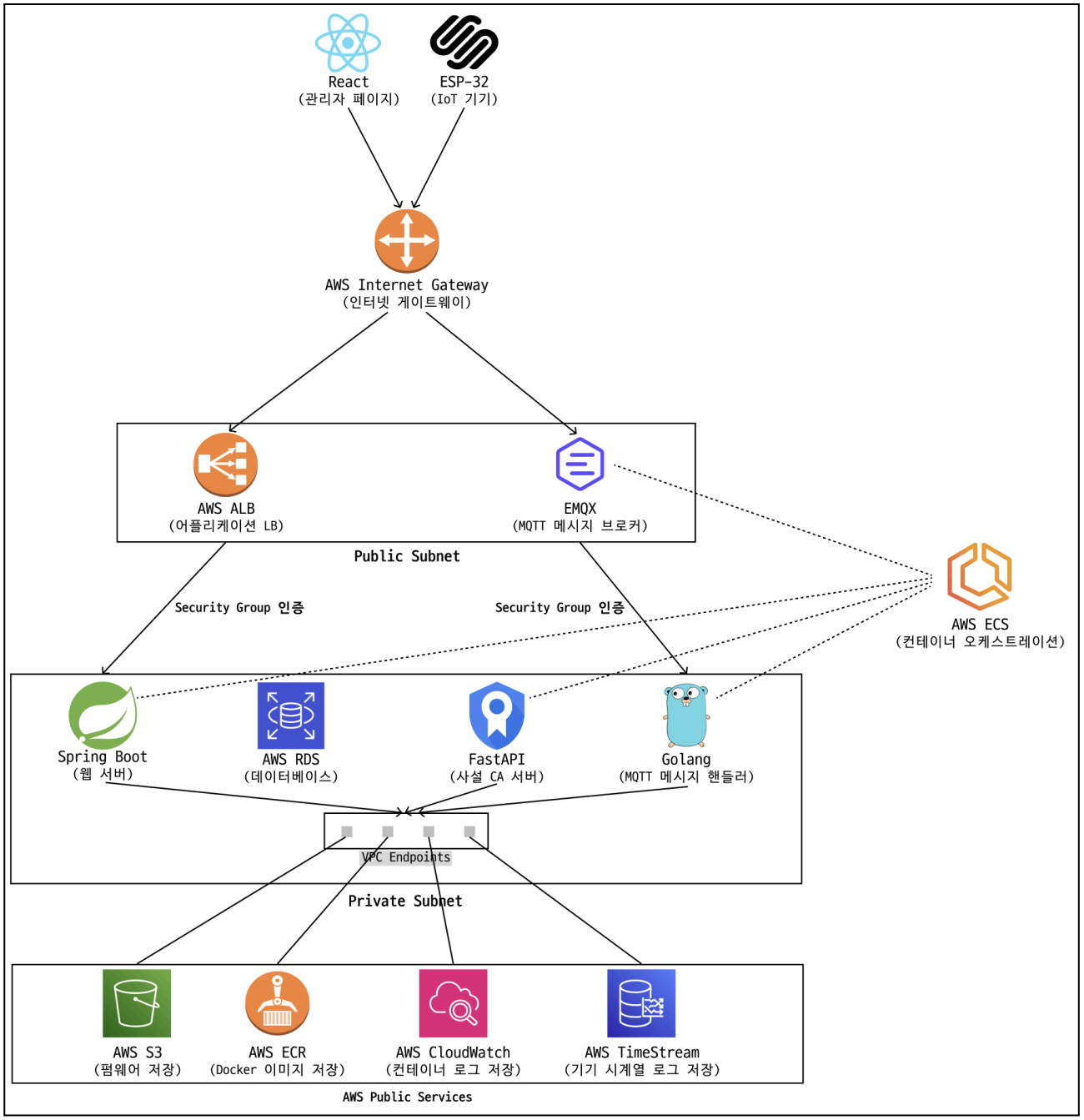
   d. 사설 CA가 서명된 인증서를 반환.

   e. Spring Boot가 MQTT 핸들러로 인증서를 전달.

   f. MQTT 메시지 핸들러는 해당 기기에 /cert/renew/response 토픽으로 새로운 인증서를 전달.

g. 기기는 이 응답 메시지를 수신하여, 내부 저장 인증서를 새로 갱신함.

클라우드 아키텍처 변경



본 프로젝트의 클라우드 아키텍처는 효율성, 보안, 확장성을 모두 고려하여 설계되었다. 외부 사용자(웹 클라이언트, IoT 기기), 내부 서비스(백엔드, 데이터베이스 등), AWS Public 서비스 간의 통신 구조와 흐름을 아래와 같이 정리한다.

1. 외부 클라이언트
   - React 웹 클라이언트와 ESP-32 IoT 기기는 모두 클라우드 외부(인터넷)에서 접속한다.

- 웹 클라이언트는 **HTTPS**로, IoT 기기는 **MQTT**로 클라우드 내부와 통신한다.

2. **VPC** 네트워크 구성
   - **VPC** 입구
     - 클라우드 네트워크(VPC) 입구에는 Internet Gateway(IGW)를 배치하여 외부와 내부가 연결된다.

   - **Public Subnet**
     - ALB: React 웹 클라이언트의 HTTP/HTTPS 요청을 받아 내부 Spring Boot 서비스로 전달.

     - EMQX: IoT 기기의 MQTT 메시지를 받아 내부 MQTT 메시지 핸들러로 전달.

   - **Private Subnet**
     - 외부 인터넷에서 직접 접근할 수 없으므로, 보안이 강화됨.

     - Spring Boot(백엔드), RDS(데이터베이스), 사설 CA, MQTT 메시지 핸들러 등 핵심 서비스가 위치.

3. 네트워크 트래픽 및 서비스 연동 흐름
   - 웹 클라이언트 → **ALB** → **Spring Boot**
     - 사용자가 웹에서 요청 시 ALB를 통해 전달받고, ALB는 트래픽을 내부 Spring Boot 백엔드로 라우팅한다.

   - **IoT** 기기 → **EMQX** → **MQTT** 메시지 핸들러
     - IoT 기기는 외부에서 MQTT 프로토콜로 EMQX에 접속, MQTT 메시지는 내부의 메시지 핸들러 서비스로 연동된다.

4. **AWS Public** 서비스와 **Private Subnet** 연동
   - Private Subnet은 기본적으로 외부 인터넷 접속(Outbound)이 차단되어 있음.

   - 인터넷 게이트웨이 대신 VPC Endpoint(Interface/ Gateway Endpoint)를 사용

   - S3, ECR 등 각 Public 서비스별 VPC Endpoint를 Private Subnet에 연결

   - 내부 서비스가 사설 IP 네트워크 안에서 Public AWS 서비스에 바로 안전하게 접근 가능

   VPC Endpoint를 사용함으로써, Private Subnet을 외부 네트워크(인터넷)과 격리하여 시스템의 보안성을 높였다.

5. 컨테이너화와 오케스트레이션
   - Spring Boot, EMQX, 사설 CA, MQTT 메시지 핸들러는 모두 Docker 컨테이너로 구현

   - AWS ECS가 각 서비스 컨테이너를 배포, 관리, 오토스케일링 등 오케스트레이션

3-5. IoT 기기

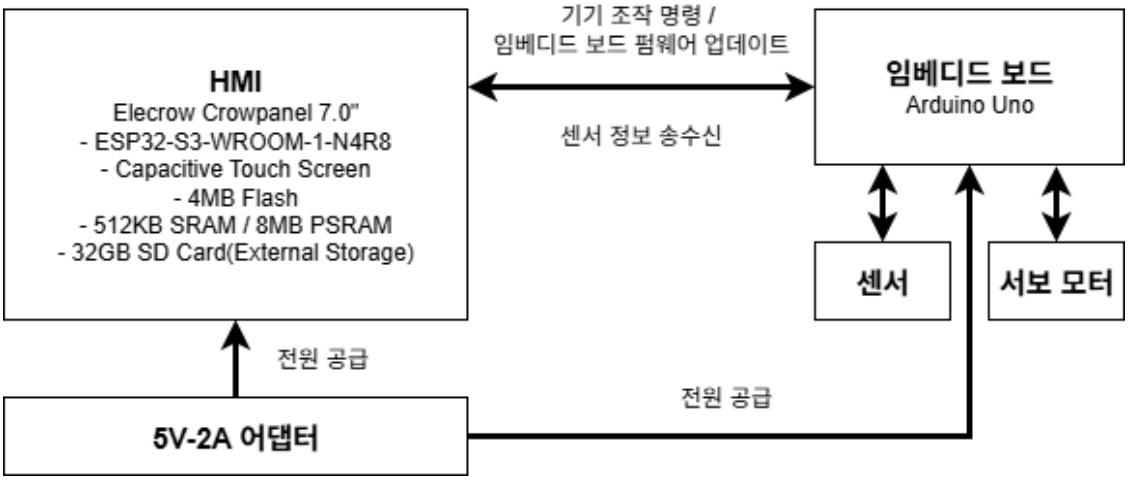
임베디드 소프트웨어 파트의 당초 계획은 시리얼 통신이 가능한 임베디드 보드에 IoT 기능을 추가하는 별도의 모듈을 두어 독립적으로 개발할 수 있는 하드웨어 모듈을 개발하는 것이였다. 그러나 HMI에서 동작하는 GUI 프로그램에 이미지나 영상처럼 대용량의 컨텐츠 파일이 포함되면서 구조적인 어려움이 발생했다. HMI에 OTA를 구현할 때, 시리얼 통신을 통한 대용량 파일 업로드 및 처리 속도가 느리고, 전송량을 늘리기 위해 보드레이트를 높히면 안정성이 크게 떨어졌다.

이 문제를 해결하기 위해, 대용량 파일 처리가 필요한 **HMI**에는 네트워크 모듈이 내장된 모델을 사용하는 방식으로 하드웨어 구조를 변경하였다. 이에 따라 GUI 프로그램이 포함되는 HMI의 펌웨어는 외부 모듈에서 시리얼 통신을 거치지 않고, HMI 자체가 직접 수신하는 구조로 변경하였다.

Arduino와 같은 소형 임베디드 보드는 펌웨어의 크기가 작아 기존 계획대로 시리얼 통신 방식으로 구현한다.

각종 제약사항을 반영하여 확정한 하드웨어 명세는 다음과 같다.

- **HMI**: Elecrow Crowpanel 7.0"[2]
- 기기 세부 조작을 위한 임베디드 보드: Arduino Uno Rev3[3]
- 기타 서보 모터, 온도 센서 및 전원 공급을 위한 **DC 5V** 어댑터



OTA를 수행할 HMI에는 IoT 분야에서 널리 사용되는 **ESP32** 계열 **MCU**를 적용하였다. 또한, 최근 산업계에서 활용도가 높은 **Arduino** 보드를 추가해 세부 기기 제어에 활용한다.

하드웨어 개발에는 ESP 보드 전용 개발 프레임워크인 ESP-IDF를 사용하며, GUI 프로그램 개발에는 임베디드 UI 개발의 사실상 표준으로 자리잡은[4] 오픈소스 그래픽 라이브러리인 LVGL을 사용한다.

저수준 하드웨어 API를 다뤄야 하는 특성상, 펌웨어는 특정 하드웨어 생태계에 종속될 수 있다. 이 점은 설계 초기부터 감안하여 개발을 진행한다.

IoT 기기 파트에서 자문의견서를 반영하여 수정된 세부 구현 사항은 다음과 같다.

아키텍처 및 설계 측면의 개선점

1. **TLS** 기반 보안 통신

펌웨어 수신에 사용하는 HTTP와, 다운로드 주소 수신 및 센서 데이터를 송신하는 데 사용하는 MQTT는 모두 **TLS** 상에서 작동한다. 이를 전제로, 산업 환경에서 사용될 기기는 출하 시 유효기간이 긴 루트 인증서**(CA)**를 미리 내장한다.

TLS 인증의 신뢰성 확보를 위해, 통신 전 NTP(Network Time Protocol)를 이용하여 시스템의 시계(RTC, Real-Time Clock)를 서버와 동기화한다. 이후 루트 CA를 기반으로 중간 CA를 발급받아 TLS 인증에 사용한다.

2. 펌웨어 무결성 검증

S3 버킷에 업로드된 펌웨어는, 메타데이터로 해시 체크섬**(SHA-256)**을 함께 저장한다. 기기가 MQTT를 통해 펌웨어 주소를 수신할 때, 이 체크섬도 함께 전달된다.

기기는 수신한 펌웨어 바이너리를 ESP32-S3의 하드웨어 가속 기능을 활용하여 SHA-256으로 해싱한 뒤, 전달받은 체크섬과 비교하여 펌웨어의 무결성을 검증한다.

3. 부팅 실패 시 자동 롤백

새로 수신한 펌웨어가 정상 부팅하지 못할 경우 무선으로 복구가 불가능하다면 기기별 수동 조치가 필요하며, 이는 막대한 비용을 초래할 수 있다. 특히 이 문제는 외부 공격뿐 아니라 내부 실수로도 발생할 수 있기 때문에 이를 방지하기 위해, **Flash** 파티션 기반 자동 롤백 메커니즘을 적용한다.

ESP32 플랫폼은 부트로더 등 필수 요소를 ROM에, 실제 펌웨어는 Flash에 저장한다. Flash는 사용자 지정 파티셔닝이 가능하므로 2개의 OTA 파티션을 구성한다. OTA 업데이트 시, 현재 실행 중이 아닌 파티션에 새 펌웨어를 저장하고, 시스템은 이 펌웨어로 부팅하면서 상태를 **ESP_OTA_IMG_PENDING_VERIFY**로 설정한다.

프로그램은 그래픽 드라이버 및 네트워크 초기화를 진행하며 자체 진단을 수행한다. 진단이 성공하면 **esp_ota_mark_app_valid_cancel_rollback()**을 호출해 펌웨어를 유효한 것으로 마킹한다. 반대로 진단 실패나 오류 발생 시 **esp_ota_mark_app_invalid_rollback_and_reboot()**을 호출해 롤백을 트리거하고, 자동 재부팅을 수행한다.

이 방법은 ESP-IDF 공식 문서[5]에서도 신뢰성 있는 방법으로 권장된다.

또한 현재 구현 범위에는 포함되지 않지만, 암호화된 펌웨어 수신 및 복호화 처리도 향후 도입을 고려하고 있다. 프로젝트에 사용 중인 ESP32-S3는 하드웨어 가속을 이용한 AES 복호화를 빠르게 수행할 수 있기 때문에, 이 방식이 적용되면 펌웨어를 암호화된 상태로 수신한 뒤, 기기 내에서 복호화하는 방식으로 보안을 강화할 수 있다.

기술적 구현의 견고성 향상

산업 현장에서는 네트워크가 불안정한 경우가 많다. 따라서 이러한 환경에 대응하기 위해 네트워크 불안정성에 대비한 구현 방식을 보다 견고하게 개선하였다. 자문 내용을 바탕으로 추가한 대응 방식은 다음과 같다.

1. 부분 다운로드 복구

ESP32의 표준 OTA 함수 **esp_https_ota()**는 스트리밍 방식으로 동작하기 때문에 부분 다운로드 복구 기능을 지원하지 않는다. 따라서 부분 다운로드 복구를 구현하기 위해서는 펌웨어 업데이트 함수를 직접 구현해야 한다. 펌웨어는 Flash의 OTA 파티션에 바로 저장하지 않고, 먼저 **SD** 카드에 다운로드한 후 **Flash**로 옮기는 방식으로 처리한다. 이 때 **HTTP**의 **Range** 헤더를 이용하여 청크 단위로 데이터를 수신한다. 만약 펌웨어 다운로드 중 네트워크가 끊기거나 기기의 전원이 꺼질 경우, 이미 수신한 펌웨어의 크기를 기준으로 나머지 데이터를 이어서 받을 수 있도록 구현한다.

2. 대역폭 적응형 다운로드

청크 단위 다운로드 시, 버퍼를 SRAM 대신 용량이 더 큰 PSRAM에 할당하여 버퍼의 크기를 동적으로 조절할 수 있도록 구현한다. 청크 수신 속도를 측정한 뒤, 속도가 일정 기준 이하일 경우 버퍼 크기를 줄여 전송 실패 시 손실을 최소화하고, 속도가 충분히 빠를 경우 버퍼 크기를 늘려 효율적인 전송이 가능하도록 구성한다. 이 방식을 통해 네트워크 상황에 따라 자동으로 전송 단위를 조절하는 대역폭 적응형 다운로드를 구현할 수 있다.

4. 갱신된 과제 추진 계획

업무	5월				6월				7월				8월				9월			
	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
기획 및 설계																				
착수 보고서 작성																				
프론트엔드 개발																				
백엔드 개발																				
IoT 개발																				
클라우드 개발																				
중간보고서 작성																				
프론트엔드 추가 개발																				
백엔드 추가 개발																				
IoT 추가 개발																				

시스템 통합																			
테스트 및 디버깅																			
최종 테스트 및 배포																			
최종 보고서 작성																			
발표 준비																			

5. 구성원별 진척도 및 중간 결과

이름	파트	진행 상황
박준우	프론트엔드 / 클라우드	프론트엔드 개발의 주요 작업 항목은 다음과 같다. • 관리자 웹 페이지 디자인 • 펌웨어와 관련된 컴포넌트 구현 • 각 컴포넌트 정적/동적 데이터 상태 관리 • 백엔드 **API** 연동 펌웨어와 관련된 컴포넌트의 구현을 완료하였으며, 백엔드와의 연결을 완료하였다. 모니터링과 관련된 컴포넌트 작업이 남아있지만, “1.4 구현 범위의 현실적 조정” 에 맞추어 우선 펌웨어 배포와 관련된 기능들을 개발하고, 펌웨어 배포 기능이 완료되었을 때 모니터링 컴포넌트 작업을 시작할 예정이다. 클라우드 개발의 주요 작업 항목은 다음과 같다. • **AWS Terraform** 초기 세팅 • **mTLS** 로직 디자인 • **FastAPI**를 사용한 사설**CA** 구현 • **VPC** 구성 • **ECS** 구성 및 **EMQX,** 사설**CA** 배포 클라우드/인프라 개발은 일정에 맞추어 순조롭게 진행되고 있다. VPC와 ECS 구성이 거의 끝나감에 따라, **CI/CD**파이프라인을 구축하여 개발 및 배포에 도움이 될 수 있도록 할 예정이다.
송승우	백엔드	백엔드 개발의 주요 작업 항목은 다음과 같다. • 펌웨어 **Presigned URL** 발급 기능 개발 • 펌웨어 메타데이터 저장 기능 개발 • **S3** 파일 해시**(SHA-256)** 계산 및 저장 기능 개발 • 펌웨어 정보 조회 기능 개발 • **DB** 설계

		• **MQTT** 핸들러 개발 **(Golang)** 현재까지 Spring Boot 기반의 **REST API** 서버 개발을 완료하였으며, 펌웨어 Presigned URL 발급, 메타데이터 등록 및 조회, 리전·디비전·디바이스 리스트 조회 등 주요 기능을 정상적으로 구현하였다. 이제 다음 단계로, **Golang**을 사용한 **MQTT** 핸들러 개발을 진행할 예정이다. 이를 통해 디바이스와의 실시간 통신 및 펌웨어 업데이트 요청/응답 처리를 수행할 수 있도록 시스템을 확장할 계획이다.
장민준	하드웨어	임베디드 소프트웨어 개발의 주요 작업 항목은 다음과 같다. • 기초 개발환경 구성 • 하드웨어 드라이버 개발 • 자동 롤백 / 대역폭 적응형 **OTA** 업데이트 기능 개발 • **LVGL** 기반 시연용 **GUI** 프로그램 개발 • **Arduino** 등 소형 임베디드 보드 개발 현재까지 기초 개발환경과 하드웨어 드라이버 개발을 완료하였으며, 이후 단계로 **OTA** 기능 개발에 착수하여 현재 구현을 진행 중이다. 하드웨어 드라이버 개발에 당초 예상보다 오랜 시간이 걸렸지만, 현재까지 전체 일정에 큰 차질은 없다. 앞으로는 **GUI** 기반 시연 프로그램과 **Arduino** 보드 개발을 병행하며, 하드웨어 패키징 및 보안 요소 추가도 순차적으로 검토할 예정이다. 전체 개발은 일정에 맞춰 원활하게 진행되고 있으며, 기능 완성도뿐 아니라 실험 준비도 동시에 갖춰가는 방향으로 조율 중이다.

참고 자료

[1] Mender: Over-the-air software updates for IoT devices(https://mender.io/)
[2] CrowPanel 7.0" - Elecrow(https://www.elecrow.com/esp32-display-7-inch-hmi-display-rgb-tft-lcd-touch-screen-support-lvgl.html)
[3] Arduino Uno Rev3 - Arduino Store(https://store-usa.arduino.cc/products/arduino-uno-rev3)
[4] 베리실리콘, LVGL과 협력하여 웨어러블 및 기타 디바이스를 위한 고급 GPU 가속화 지원 - NewsWire(https://www.newswire.co.kr/newsRead.php?no=1001933)
[5] Over The Air Updates (OTA) - ESP-IDF Programming Guide(https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ota.html)

-끝-