ChatGPT

Roadmapa MVP regulatora TCM na Raspberry Pi (offline)

Wprowadzenie

Telecom Climate Manager (TCM) to system do zarządzania klimatem szafy telekomunikacyjnej – monitoruje warunki (temperaturę, wilgotność, otwarcie drzwi, itp.) i steruje urządzeniami (wentylatory, grzałki, alarmy) dla optymalnego działania. Celem jest zaprojektowanie **MVP aplikacji TCM** działającej na Raspberry Pi **bez dostępu do Internetu**, z lokalnym panelem webowym i API zgodnym z dokumentem *„TCM 2.0 – Web Panel, API"*. Poniższa roadmapa opisuje architekturę systemu, technologię, strukturę projektu oraz kolejne kroki wdrożenia – uwzględniając bezpieczeństwo i wszystkie wymogi.

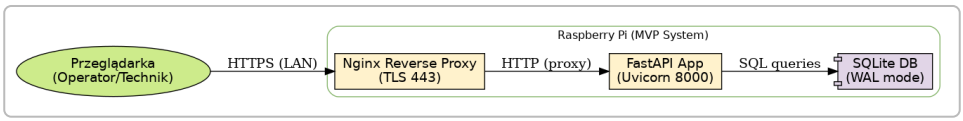
Założenia systemu i wymagania

- **Środowisko uruchomieniowe:** Aplikacja zostanie wdrożona na Raspberry Pi, uruchamiana w kontenerach Dockera (Docker Compose) bez dostępu do Internetu po wdrożeniu. Dostęp do panelu tylko w sieci lokalnej (LAN).
- **Reverse proxy z TLS:** Przed aplikacją działa serwer Nginx w osobnym kontenerze jako reverse proxy, terminujący TLS (HTTPS). Połączenia z przeglądarki idą przez HTTPS (certyfikat wystawiony offline) do Nginx, który **lokalnie** przekazuje ruch do aplikacji FastAPI po HTTP. Serwer nie wystawia usługi publicznie poza LAN (np. brak przekierowania portów na routerze).
- **Backend i frontend:** Aplikacja napisana w Pythonie z użyciem **FastAPI** (do obsługi logiki i API) oraz **Jinja2** (renderowanie szablonów HTML dla interfejsu web). FastAPI obsługuje zarówno żądania API (zwracając JSON), jak i żądania od przeglądarki (zwracając strony HTML renderowane).
- **Frontend UI bez zależności online:** Do stylowania interfejsu użyty zostanie **Tailwind CSS**, ale **bez zewnętrznych zależności w czasie działania** – CSS będzie skompilowany podczas procesu budowania i dołączony jako plik statyczny. Oznacza to, że *nie będzie* żadnych odwołań do CDN czy zewnętrznych skryptów w trakcie pracy aplikacji. Tailwind zostanie zainstalowany i wygenerowany offline (np. przy użyciu standalone CLI lub w procesie build) [1] – po instalacji pakietów z Internetem, gotowe pliki CSS/JS będą dołączone lokalnie.
- **Baza danych lokalna:** Użyta zostanie lekka baza **SQLite** (plik na dysku Raspberry Pi) z włączonym trybem **WAL** (Write-Ahead Logging) dla lepszej wydajności i równoległości dostępu. WAL umożliwia większą współbieżność – odczyty nie blokują zapisów i vice versa [2]. Baza będzie przechowywać konfigurację urządzenia (np. progi, ustawienia sieci), logi zdarzeń, dane do panelu oraz konta użytkowników (hasła zahashowane).
- **Konteneryzacja:** System zostanie podzielony na **dwa kontenery** (usługi) w ramach Docker Compose:
- `app` – kontener z aplikacją FastAPI (Python) oraz wbudowanym serwerem (Uvicorn). Ten kontener zawiera cały backend, logikę biznesową, pliki statyczne i bazę danych.
- `proxy` – kontener z Nginx, działający jako reverse proxy z TLS. Nginx nasłuchuje na porcie 443 (HTTPS) i przekazuje ruch do `app` (np. na port 8000) wewnątrz sieci Docker. Certyfikaty TLS zostaną wygenerowane **poza Pi** (np. na komputerze deweloperskim) i wgrane do obrazu Nginx lub zamontowane jako wolumin. Komunikacja z przeglądarką jest szyfrowana, ale **ograniczona do sieci lokalnej**.

- **Interfejs użytkownika zgodny z dokumentem „TCM 2.0 – Web Panel, API”**:
- Ekran logowania z wyborem roli (dwa poziomy uprawnień: **operator** i **technik** [3]) oraz wyświetleniem podstawowych informacji: ID urządzenia, szybki podgląd ostatnich logów i ustawienia sieciowe (IP, itp.).
- **Dashboard operatora:** widok podsumowujący aktualny stan – sekcja *Sensor Data* (np. temperatury baterii i szafy, wilgotność), *Inputs* (stan czujników drzwi, zalania) oraz *Outputs* (stan wyjść przekaźnikowych/tranzystorowych) [4] [5] . Paski nawigacyjne (góra/dół) zawierają ID urządzenia, skrót do logów i info sieciowe (tak jak na ekranie logowania) [6] .
- **Panel technika:** zawiera wszystko co dashboard operatora, plus dostęp do zakładki *Ustawienia* (przycisk dostępny tylko dla technika) [7] . W ustawieniach technik może konfigurować m.in. ID urządzenia, **ustawienia sieciowe** (IP, maska, brama), **progi temperatur** (dla załączania grzałki, klimatyzacji, awaryjnych wentylatorów) oraz **histerezę** przełączania [8] [9] . Te elementy muszą być uwzględnione w UI.
- **Logi zdarzeń:** dostępne dla obu ról przynajmniej w trybie podglądu (operator) i pełnego wglądu (technik). Każde zdarzenie (np. zmiana stanu wejścia/wyjścia, alarm) jest zapisywane z czasem, typem i opisem [10] . Panel logów pozwoli przeglądać te wpisy (oraz opcja eksportu do pliku `.logs` dla serwisu) [11] .
- Interfejs powinien aktualizować się na bieżąco – **dashboard w czasie rzeczywistym** pokazuje zmiany (np. natychmiastowa reakcja na otwarcie drzwi: wyłączenie klimatyzacji/grzania, włączenie oświetlenia, alarmu itp. zgodnie z logiką) [12] . W MVP można to zrealizować poprzez cykliczne odświeżanie danych AJAX-em (polling co kilka sekund) lub wykorzystując mechanizm WebSocket/SSE FastAPI do push notyfikacji.
- **API REST:** system udostępnia kilka zasobów API (typu **HTTP GET**) do integracji zewnętrznej (np. system SCADA) [13] . Wymagane endpointy (zgodne z dokumentacją) to:
- `GET /api/state` – zwraca bieżące stany urządzenia (JSON zawierający m.in. stany wejść, wyjść, czujników, powód alarmu, znacznik czasu itp.) [14] . *(Np. struktura zbliżona do przykładu JSON w dokumentacji, z polami* `inputs` *,* `outputs` *,* `sensors` *,* `alarm_reason` *,* `ts` *etc.)*
- `GET /api/strike/<id>/trigger` – wyzwolenie elektrozaczepu drzwi o podanym ID (np. otwarcie zamka elektromagnetycznego drzwi nr X na określony czas) [15] . Zwraca potwierdzenie (np. które wyjście zostało aktywowane i na jak długo).
- `GET /api/logs` – zwraca logi systemowe (zdarzenia, alarmy, zmiany stanów) w formacie JSON – lista wpisów z polami czas, typ, źródło, wiadomość [16] [17] .
- `GET /api/service/mapping` – zwraca mapowanie sygnałów/wejść-wyjść oraz konfigurację (np. przypisanie wyjść do konkretnych portów sterownika – według dokumentacji, np. `{"alarm": ["K1"], "ac": ["K2"], ...}` [18]).
- `GET /api/network` – zwraca aktualne ustawienia sieciowe urządzenia (ID, adres IP, maska, brama) [19] [20] .

Wszystkie powyższe endpointy będą zaimplementowane w FastAPI. **Uwaga dot. autoryzacji:** Ponieważ dostęp do systemu jest tylko lokalny, można założyć, że API będzie używane wewnętrznie lub przez zaufany system (np. SCADA w tej samej sieci). Dla bezpieczeństwa jednak warto rozważyć wymaganie tokena API lub przynajmniej uwierzytelnienia (np. podstawowego lub wykorzystanie sesji użytkownika) dla operacji wrażliwych jak *strike trigger*. W MVP można uprościć przyjmując, że sieć lokalna jest zaufana, a interfejs API jest *read-only* poza operacją `/api/strike/...` – tę ostatnią można ograniczyć do roli technika (np. wywoływana tylko z panelu po zalogowaniu). - **Brak zdalnych aktualizacji:** System nie przewiduje mechanizmu auto-update ani aktualizacji przez Internet. **Aktualizacja oprogramowania odbywa się offline** – np. przez podłączenie się do urządzenia fizycznie i wgranie nowej wersji (lub wymianę karty SD z nowym image). Oznacza to, że po instalacji aplikacja działa w niezmienionej wersji tak długo, aż obsługa techniczna nie przeprowadzi manualnie aktualizacji. W projekcie nie trzeba więc implementować updatera OTA, a konfiguracja urządzenia jest stała między aktualizacjami.

Architektura systemu



Architektura systemu TCM (MVP): komponenty i przepływ danych. Na powyższym schemacie przedstawiono kluczowe elementy rozwiązania i sposób ich komunikacji. **Raspberry Pi** hostuje dwa kontenery Docker: Nginx oraz aplikację FastAPI. Nginx nasłuchuje na porcie 443/TCP i obsługuje szyfrowane połączenia HTTPS od klientów w LAN (np. przeglądarka operatora lub technika). Ruch TLS jest terminowany na Nginx (przy użyciu certyfikatu wygenerowanego offline), po czym Nginx przekazuje żądania HTTP do aplikacji FastAPI (kontener `app`) przez wewnętrzną sieć Docker. FastAPI nasłuchuje na porcie wewnętrznym (np. 8000) i obsługuje zarówno żądania interfejsu web (HTML) jak i wywołania API REST.

Wewnątrz kontenera aplikacji znajduje się logika biznesowa TCM, która komunikuje się z **bazą SQLite** (plik bazy zamontowany jako wolumin, tryb WAL aktywny). Baza danych przechowuje m.in. **stany czujników/wejść** (ostatnio odczytane), **stany wyjść**, historię **logów zdarzeń**, ustawienia (progi temperatur, konfigurację sieci, mapping portów) oraz dane o **użytkownikach i ich rolach**. FastAPI poprzez odpowiednie moduły będzie również komunikować się z warstwą sprzętową – odczyt czujników (np. przez interfejs GPIO, I2C itp.) oraz sterowanie wyjściami (przekaźniki, diody, elektrozaczepy). W MVP, jeśli brak fizycznego sprzętu, można zaimplementować warstwę abstrakcji sprzętu, która zwraca symulowane dane (np. losowe temperatury w bezpiecznym zakresie, fikcyjne stany czujników) i loguje akcje sterowania zamiast faktycznie przełączać sprzęt.

Przepływ danych: Gdy użytkownik w przeglądarce wpisuje adres urządzenia (np. https://tcm.local lub https://192.168.0.100), następuje nawiązanie szyfrowanego połączenia do Nginx na Raspberry Pi. Po uwierzytelnieniu (logowanie) dalsze żądania HTTP trafiają do FastAPI (np. żądanie renderu dashboardu lub wywołania API AJAX). FastAPI pobiera potrzebne dane (np. z bazy lub bezpośrednio z czujników) i zwraca wynik – w przypadku strony HTML następuje wyrenderowanie szablonu Jinja2 z wstawionymi danymi; w przypadku API – zwracany jest JSON z danymi. Nginx przekazuje odpowiedź do przeglądarki użytkownika. Jeśli użytkownik np. zmienia ustawienie (próg temperatury) w panelu technika, aplikacja zapisuje nową wartość do bazy i (opcjonalnie) od razu uwzględnia ją w logice kontroli klimatu.

W architekturze nie ma żadnych elementów komunikujących się z chmurą czy internetem – cały system działa **off-line** w obrębie lokalnej infrastruktury. Dzięki temu nawet w przypadku braku łączności z siecią zewnętrzną panel TCM i automatyka działają nieprzerwanie. Zewnętrzne systemy (np. SCADA) mogą integrować się poprzez wymienione API REST, o ile mają dostęp do sieci lokalnej urządzenia.

Struktura projektu i komponenty aplikacji

Aby spełnić powyższe założenia, projekt zostanie zorganizowany w czytelną strukturę plików i modułów. Poniżej proponowany układ katalogów i plików MVP:

```
tcm-project/
├── app/
│   ├── main.py             # Główny plik aplikacji FastAPI (uruchomienie
uvicorn, definicja app)
│   ├── auth.py             # Moduł autentykacji (zarządzanie sesją/loginem,
hashowanie haseł)
```

```
│   ├── api/                # Pakiet z trasami API
│   │   ├── __init__.py
│   │   ├── state.py        # Endpoint /api/state
│   │   ├── strike.py       # Endpoint /api/strike/<id>/trigger
│   │   ├── logs.py         # Endpoint /api/logs
│   │   ├── service.py      # Endpoint /api/service/mapping
│   │   └── network.py      # Endpoint /api/network
│   ├── web/                # Pakiet z obsługą stron web (frontend)
│   │   ├── __init__.py
│   │   ├── routes.py       # Trasy dla stron HTML (login, dashboard,
settings, etc.)
│   │   └── dependencies.py  # Wspólne zależności (np. funkcja
get_current_user)
│   ├── core/               # Logika rdzeniowa aplikacji
│   │   ├── hardware.py     # Abstrakcja dostępu do czujników/wyjść (GPIO);
symulacja w MVP
│   │   ├── controller.py   # Logika sterowania klimatem (np. decyzje on/off
na podstawie czujników)
│   │   └── config.py       # Konfiguracja aplikacji (stałe, odczyt .env)
│   ├── models/             # Definicje modeli danych
│   │   ├── __init__.py
│   │   ├── database.py     # Inicjalizacja bazy SQLite, funkcje dostępu
│   │   ├── user.py         # Model użytkownika (id, login, hash hasła,
rola)
│   │   ├── log_entry.py    # Model wpisu logu zdarzeń
│   │   ├── thresholds.py   # Model progów i histerezy
│   │   └── ...             # (ew. inne, np. mapping portów, konfiguracja
sieci)
│   ├── templates/          # Szablony Jinja2 dla interfejsu web
│   │   ├── base.html       # Szablon bazowy (zawiera np. wspólny układ,
nagłówek, osadzenie Tailwind CSS)
│   │   ├── login.html      # Strona logowania
│   │   ├── dashboard_op.html   # Dashboard operatora
│   │   ├── dashboard_tech.html # Panel technika (dashboard + przycisk
ustawień)
│   │   ├── settings.html       # Strona ustawień technika (formularze zmiany
progów, sieci itp.)
│   │   └── logs.html           # Strona podglądu logów zdarzeń
│   ├── static/
│   │   ├── css/
│   │   │   └── tailwind.css  # Wygenerowany plik CSS zawierający potrzebne
style Tailwind
│   │   ├── js/
│   │   │   └── app.js        # Ewentualne skrypty JS (np. do odświeżania
danych, obsługi interakcji)
│   │   └── img/              # (opcjonalnie) pliki graficzne, ikony
│   └── Dockerfile          # Dockerfile dla obrazu aplikacji (Python +
skompilowany Tailwind)
├── nginx/
│   ├── nginx.conf          # Plik konfiguracyjny Nginx (globalny, może
```

```
zawierać include)
│   ├── site.conf            # Konfiguracja serwera (listen 443 ssl;
proxy_pass do app:8000; certyfikaty)
│   └── Dockerfile           # (opcjonalnie) Dockerfile dla obrazu Nginx,
jeśli nie korzystamy z gotowego
├── docker-compose.yml       # Definicja usług: app + proxy, sieci,
woluminów, zmiennych
├── requirements.txt         # Wymagane pakiety Python (FastAPI, uvicorn,
Jinja2, pydantic, itp.)
├── tailwind.config.js       # Konfiguracja Tailwind (np. paths do szablonów
do purge)
├── input.css                # Główny plik CSS Tailwind (importy base/
components/utilities lub directives)
└── README.md                # Dokumentacja projektu, instrukcje uruchomienia
```

Opis komponentów:

- **Moduły FastAPI (backend):** `main.py` tworzy instancję FastAPI, dołącza routery API i stron web, konfiguruje statyczne pliki (np. `app.mount("/static", StaticFiles(directory="app/static"), name="static")`) oraz uruchamia serwer Uvicorn. Rozdzielone podpakiety `api` i `web` zawierają odpowiednio endpointy REST i obsługę stron HTML. Dzięki podziałowi, kod jest czytelniejszy i łatwiej zapewnić, że np. endpointy API zwracają JSON, a widoki web renderują szablony.
- **Szablony Jinja2:** W katalogu `templates/` znajdują się pliki HTML budujące interfejs użytkownika. Wspólny `base.html` może zawierać nagłówek, wczytanie statycznego CSS (`<link rel="stylesheet" href="{{ url_for('static', path='css/tailwind.css') }}">`), menu itp. Poszczególne strony (login, dashboard, settings, logs) dziedziczą z base.html i wstawiają odpowiednie treści. **Tailwind CSS** będzie używany poprzez klasy CSS w elementach HTML (np. `<div class="text-xl font-bold">` itp.), zgodnie z utility-first podejściem. W celu **działania offline**, plik `tailwind.css` zawiera wszystkie potrzebne style – zostanie on wygenerowany na etapie budowania i dołączony lokalnie. Unikamy w ten sposób potrzeby dostępu do CDN z frameworkiem CSS. *(Do wygenerowania tailwind.css można użyć np. komendy* `tailwindcss -i input.css -o app/static/css/tailwind.css --minify`*, po zainstalowaniu TailwindCLI. Internet potrzebny jest tylko podczas instalacji Tailwind, nie przy użyciu już zainstalowanego narzędzia* [1] *.)*
- **Warstwa danych (baza SQLite):** W `models/database.py` aplikacja inicjalizuje połączenie z SQLite (plik np. `data/tcm.db` na host lub `/app/data/tcm.db` w kontenerze). **Tryb WAL** zostanie włączony przy starcie (np. poprzez wykonanie `PRAGMA journal_mode=WAL;` po ustanowieniu połączenia). Dzięki temu wielu użytkowników/procesów może jednocześnie czytać, gdy inny zapisuje, co jest ważne np. gdy panel stale odczytuje stany a jednocześnie logi są zapisywane [2] . SQLite jest idealny dla takiego embedded zastosowania – jest lekki i działa lokalnie w pliku. Należy pamiętać o otwarciu DB z opcją **check_same_thread=False** (jeśli używamy wielu wątków w FastAPI) lub użyć mechanizmu pooling w SQLAlchemy, by uniknąć błędów dostępu z różnych wątków. Struktura bazy obejmie:
- Tabela **users** (id, username, password_hash, role), aby przechowywać użytkowników systemu (co najmniej dwa domyślne konta: operator, technik). Hasła będą zahashowane (np. funkcją bcrypt) – nigdy w plain-text.
- Tabela **logs** (timestamp, type, source, message) do zapisywania zdarzeń. Każdy wpis logu reprezentuje zmianę lub alarm, zgodnie z opisem w dokumencie (np. *OUTPUT_ON*, *ALARM* itp. z

polami dodatkowymi) [21]. Logi mogą być przechowywane w SQLite, a dodatkowo udostępnimy funkcję eksportu do pliku tekstowego (po prostu wyciągnięcie wszystkich wpisów do pliku `.logs`).

- Tabela **settings** (rożne kolumny lub JSON) przechowująca konfigurację: progi temperatur (np. `temp_threshold_heat`, `temp_threshold_cool`, `temp_threshold_fan`), histereza, przypisanie wyjść, itp. Ewentualnie osobne tabele: **thresholds**, **mapping**, **network**. Z API `/api/service/mapping` i `/api/network` wynika, że mapping i ustawienia sieci mogą być stałe lub konfigurowalne – w MVP najprościej przechowywać je także w bazie, aby można je zmieniać z panelu ustawień.
- Tabela **state** (lub brak tabeli, a odczyt bezpośredni): aktualny stan wejść/wyjść/czujników. W rzeczywistym systemie stany czujników będą odczytywane z hardware na bieżąco, więc można je pozyskiwać dynamicznie przy wywołaniu `/api/state` zamiast trzymać w bazie. Jednak dla spójności można utrzymywać w pamięci lub bazie ostatni odczyt. MVP może symulować te stany – np. zmienne globalne aktualizowane co pewien czas.
- **Logika domenowa:** W folderze `core/` umieszczona jest logika działania regulatora:
- `hardware.py` – kapsułkuje dostęp do sprzętu. Dzięki temu, gdy przejdziemy z symulacji do prawdziwego sprzętu, wystarczy podmienić implementacje metod odczytu/zapisu GPIO. W trybie MVP/symulacyjnym metody mogą zwracać z góry ustalone lub losowe wartości, a np. zmiana stanu wejścia "otwarcie drzwi" można zasymulować przyciskiem w UI lub togglowaniem parametru.
- `controller.py` – zawiera reguły sterowania klimatem. Może uruchamiać w tle zadanie (background task) monitorujące czujniki i włączające/wyłączające wyjścia według ustawionych progów. Np. jeśli `temp_cab` > próg klimatyzacji, włączy klimatyzator (AC), a gdy < próg grzania – włączy grzałkę, itd. Uwzględnia również logikę reakcji na drzwi otwarte: natychmiast wyłącza AC/grzanie, włącza światło i alarm [12]. Wszystkie te akcje loguje w tabeli logów. Ten moduł jest sercem automatyki – w MVP może być uproszczony, ale powinien pokazywać koncepcję.
- `auth.py` – obsługa logowania i sesji. Można wykorzystać mechanizmy FastAPI (OAuth2PasswordBearer + JWT) lub prostsze rozwiązanie: utrzymanie zalogowanego użytkownika w **sesji cookies** (np. przy pomocy `starlette.middleware.sessions`). MVP może zastosować tradycyjne logowanie formularzem: użytkownik podaje login/hasło na stronie logowania, aplikacja weryfikuje hash hasła z bazy i zapisuje w sesji informacje o zalogowanym użytkowniku (np. `request.session["user_role"] = "technician"`). Każda strona/endpoint chroniony sprawdza obecność tej sesji i rolę.
- **Statyczne pliki i front-end:** Katalog `static/` zawiera skompilowany plik **Tailwind CSS** oraz ewentualne skrypty JavaScript. W MVP możemy zastosować odświeżanie danych na dashboardzie poprzez JavaScript (polling co parę sekund do `/api/state` i aktualizacja DOM, lub użycie Fetch API do wywołania akcji np. otwarcia elektrozaczepu bez przeładowania strony). Wszystko to jednak działa *lokalnie*, więc obciążenie sieci jest znikome. JavaScript będzie również offline (własny kod lub ewentualnie biblioteki, ale wgrane lokalnie – np. można dołączyć plik `.js` z minimalnym kodem).
- **Nginx (reverse proxy):** Konfiguracja Nginx (`nginx/site.conf`) będzie zawierać m.in.:
- Blok `server` na port 443 z włączonym SSL (`ssl_certificate` i `ssl_certificate_key` wskazujące na wgrany certyfikat *.pem* i klucz *.key*). Certyfikat może być self-signed lub pochodzić z wewnętrznego CA – ważne, by klient (przeglądarka) go akceptował. **Certyfikaty będą przygotowane poza Raspberry Pi** (np. wygenerowane wcześniej na PC za pomocą OpenSSL), ponieważ Pi nie ma dostępu do Internetu ani do usług typu Let's Encrypt. Certyfikat może być wygenerowany na nazwę hosta w sieci lokalnej (np. `tcm.local` lub adres IP, choć certyfikat dla IP wymaga ustawienia w SAN). Klucze i certyfikat zostaną umieszczone w obrazie Nginx lub zamontowane przez Docker Compose z hosta (np. volumin `./certs/:/etc/nginx/certs:ro`).

- Konfiguracja proxy_pass dla ruchu HTTP do aplikacji: `location / { proxy_pass http://app:8000; }`. Ważne ustawienia to m.in. `proxy_set_header Host $host; proxy_set_header X-Real-IP $remote_addr;` i ewentualnie ograniczenia dostępu. Ponieważ dostęp ma być tylko z LAN, Nginx może np. filtrować adresy IP (jeśli wiadomo, że sieć to np. 192.168.0.0/24, można dopuścić tylko te zakresy).
- (Opcjonalnie) przekierowanie HTTP port 80 -> 443. Można włączyć drugi server blok na port 80, który każdą próbę połączenia przekieruje (301) na adres https:// (zapewnia, że nawet wpisanie przez użytkownika `http://` trafi na bezpieczne HTTPS). Ponieważ jednak urządzenie nie jest publiczne, użytkownicy mogą zostać poinstruowani, by używać od razu HTTPS. Warto jednak to dodać dla wygody.
- Nginx może także obsłużyć cache statycznych plików (nagłówki Cache-Control) lub kompresję gzip, choć przy sieci lokalnej i ograniczonej liczbie użytkowników nie jest to krytyczne.
- **Docker Compose:** Plik `docker-compose.yml` definiuje oba kontenery i ich relacje:
- Usługa **app**: zbudowana z Dockerfile w kontekście projektu. Bazuje np. na obrazie `python:3.11-slim`. W Dockerfile instalowane są zależności z `requirements.txt` (FastAPI, uvicorn, itp.). Następnie kopiowany jest kod aplikacji. Proces budowania uwzględnia **kompilację Tailwind** – można to zrobić np. w multi-stage build: etap 1 z obrazem Node, instalacja `tailwindcss` (npm lub pobranie binarki CLI), wygenerowanie `tailwind.css` z plików projektu, potem etap finalny (Python) kopiuje gotowy CSS do static. Dzięki temu finalny obraz jest mniejszy i nie zawiera Node ani npm. Kontener app będzie miał punkt wejścia uruchamiający Uvicorn: np. `CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]`.
    - Do kontenera app można **montować woluminy**: przede wszystkim wolumin na bazę danych, aby dane przetrwały restarty/kolejne wersje. Np. w Compose: `volumes: ["tcm-data:/app/data"]` oraz definicja `volumes: tcm-data:` (anonimowy named volume) lub montaż lokalny `./data/:/app/data/`. To sprawi, że plik `tcm.db` (i pliki WAL/SHM) pozostaną zachowane nawet gdy zbudujemy nowy obraz aplikacji. Dodatkowo można w ten sam sposób montować np. plik konfiguracyjny zewnętrzny czy inne zasoby, ale w MVP nie jest to konieczne.
    - **Restart policy:** ustawimy np. `restart: unless-stopped` aby kontenery uruchamiały się przy starcie systemu i próbowały automatycznie wznawiać działanie po ewentualnej awarii. W ten sposób aplikacja **działa cały czas po instalacji**, bez potrzeby manualnego włączania przy każdym restarcie Pi.
- Usługa **proxy**: może korzystać z oficjalnego obrazu `nginx:1.25-alpine` lub podobnego. Konfigurację podajemy przez montowanie: np. `volumes: ["./nginx/site.conf:/etc/nginx/conf.d/default.conf:ro"]` (na Alpine image domyślny host config to /etc/nginx/conf.d/default.conf). Certyfikaty TLS również montujemy: `./nginx/certs/:/etc/nginx/certs:ro`. `depends_on: ["app"]` aby Nginx startował po uruchomieniu aplikacji (chociaż warto i tak w Nginx mieć mechanizm retry – można ustawić `proxy_connect_timeout` i kilka retry w configu). Porty: `ports: ["443:443", "80:80"]` aby wystawić je na hoście (Raspberry Pi). Jeśli nie chcemy otwierać portu 80, można go pominąć.
- Sieć: Docker Compose domyślnie umie usługi w jednej sieci bridge. Możemy jawnie nazwać sieć (np. `app-network`) i przypisać obie usługi do niej [22] [23]. Wtedy Nginx może się komunikować z app poprzez nazwę hosta `app`. Ważne, by w configu Nginx użyć dokładnie tej nazwy kontenera jako hosta docelowego (docker zapewnia DNS).
- **Zasoby sprzętowe Raspberry Pi:** Ponieważ aplikacja ma docelowo działać 24/7 na RPi, należy upewnić się, że Pi ma wystarczające zasoby (procesor, RAM) na utrzymanie dwóch kontenerów. Nginx i FastAPI są dość lekkie. SQLite operuje na pliku – intensywne logowanie może generować operacje dyskowe, ale tryb WAL minimalizuje narzut fsync. Dobrze jest umieścić bazę na karcie SD (standardowo) – trzeba jednak monitorować, czy intensywne logi nie zużyją karty (ew. rozważyć przeniesienie bazy na tmpfs jeśli dane nie muszą przetrwać utraty zasilania, ale tutaj

logi są istotne, więc raczej zapis persistent). W konfiguracji SQLite można ewentualnie zmniejszyć częstotliwość checkpointów WAL, by nie nadwyrężać I/O (domyślnie checkpoint co 1000 stron ~ 4MB) [24].

- **Wymagania dot. bezpieczeństwa:** Aplikacja jest dostępna **tylko lokalnie**, co znacząco zmniejsza wektor ataku. Niemniej jednak, wrażliwe operacje (np. dostęp do ustawień technika) są chronione hasłem i rolą użytkownika [25]. Komunikacja sieciowa jest szyfrowana TLS, uniemożliwiając podsłuchanie haseł czy danych przez osoby trzecie w sieci. Certyfikat TLS powinien być unikalny dla urządzenia i przechowywany bezpiecznie na Pi (w katalogu Nginx tylko root ma dostęp). Hasła użytkowników w bazie są hashowane, co zabezpiecza je w razie wglądu w plik bazy. Dodatkowo, ponieważ urządzenie nie ma dostępu do Internetu, ryzyko zainfekowania z zewnątrz jest niższe – ważne jest jednak, aby przy instalacji wgrać wszystkie aktualne łatki systemu operacyjnego RPi i biblioteki, gdyż później nie będzie automatycznych aktualizacji.
- Dostęp do samego Raspberry Pi (SSH czy fizyczny) powinien być ograniczony do personelu technicznego. Panel TCM stanowi jedyną potrzebną powierzchnię dostępu dla użytkowników końcowych (operator/technik) – nie powinni oni potrzebować logować się bezpośrednio do systemu operacyjnego. W razie potrzeby można zablokować np. porty SSH dla sieci, zostawiając tylko lokalną konsolę.
- **Certyfikaty poza Pi:** Generowanie certyfikatu TLS wymaga źródła zaufania. Dla MVP najprostszym podejściem jest utworzenie **self-signed cert** (np. ważny kilka lat, z Subject Alternative Name dla nazwy hosta urządzenia). Jeśli urządzeń TCM będzie więcej i klient posiada wewnętrzne PKI, można wygenerować certyfikat z wewnętrznego CA – wtedy każdy klient musi ufać temu CA. Proces generowania: na komputerze (offline lub online) użyć OpenSSL do wygenerowania klucza prywatnego i certyfikatu X.509. Następnie te pliki umieścić na Pi (np. kopia przez pendrive lub scp w sieci lokalnej).
- **Ograniczenie do LAN:** Upewniamy się, że Raspberry Pi nie jest dostępne spoza sieci lokalnej – np. brak routowalnego adresu publicznego, brak konfiguracji wpięcia do internetu. Można też ustawić firewall (UFW lub iptables) na Pi, dopuszczający tylko ruch z określonej podsieci lokalnej na port 443.
- **Dostęp do API:** Jak wcześniej wspomniano, jeśli istnieje obawa, że ktoś w sieci lokalnej mógłby nieautoryzowanie korzystać z API, można wprowadzić token API. Np. klucz API przechowywany w config (wygenerowany offline) wymagany jako parametr lub header przy wywołaniu (SCADA wtedy musiałaby go znać). W MVP jednak, skupiamy się na podstawowej ochronie poprzez fizyczne odizolowanie i uwierzytelnianie użytkowników panelu.

Plan wdrożenia krok po kroku

Poniżej przedstawiono kolejne etapy realizacji MVP – od przygotowania środowiska, poprzez implementację, po uruchomienie systemu na docelowym urządzeniu:

1. **Przygotowanie środowiska developerskiego:** Rozpocznij od utworzenia szkieletu projektu na komputerze deweloperskim. Zainstaluj wymagane narzędzia:
2. Python 3.10+ oraz biblioteki FastAPI, Uvicorn, Jinja2, SQLAlchemy (lub inna warstwa bazy, jeśli użyta), pydantic, itp.
3. Node.js oraz Tailwind CSS CLI (opcjonalnie, jeśli nie użyjesz standalone binarki). Możesz też pobrać standalone Tailwind CLI dla swojej platformy [26].
4. Docker i Docker Compose na potrzeby testów konteneryzacji (na dev-machine lub bezpośrednio na Pi, jeśli rozwijasz na nim).
5. (Opcjonalnie) narzędzia do symulacji hardware, jeśli chcesz testować logikę kontrolera (np. można pisać testy jednostkowe symulujące różne scenariusze czujników).

6. **Implementacja funkcjonalności backendu:** Stwórz aplikację FastAPI:

7. Zdefiniuj modele danych i utwórz schemat bazy SQLite. Możesz użyć **SQLAlchemy ORM** do zdefiniowania modeli (User, LogEntry, Setting itp.) i generować schemat automatycznie, lub ręcznie przygotować skrypt SQL do utworzenia tabel przy starcie. Upewnij się, że przy pierwszym uruchomieniu (gdy brak pliku DB) aplikacja utworzy bazę i tabele.
8. Zaimplementuj logikę logowania w FastAPI. Najprościej zrealizować to poprzez *formularz HTML + sesje*:
   - Endpoint `POST /login`: pobiera dane z formularza, weryfikuje użytkownika (np. funkcja w `auth.py` sprawdza w DB użytkownika i `bcrypt.verify` hasło). Jeśli ok, ustawia `session` lub wystawia token JWT w ciasteczku.
   - Endpoint `GET /logout`: usuwa sesję/ciasteczko.
   - Dodaj *dependency* do chronionych tras (dashboard, settings, API) sprawdzający obecność ważnej sesji/JWT. Jeśli brak – redirect do `/login`.
9. Zaimplementuj endpointy **API** zgodnie ze specyfikacją:
   - `/api/state`: Zbiera aktualne dane: stany wejść/wyjść/czujników. W MVP może to pobierać dane z modułu `hardware.py` (który zwraca symulowane albo ostatnio zaktualizowane wartości). Zwraca słownik/obiekt JSON zgodny ze specyfikacją (np. {"inputs":{...},"outputs":{...},"sensors":{...},"alarm_reason": ..., "ts": ...}).
   - `/api/strike/{id}/trigger`: Wywołuje funkcję kontrolera, która np. ustawia wyjście `strike_id` na aktywne na określony czas (np. 10s jak w przykładzie) [27]. W praktyce może to dodać zadanie opóźnione aby po czasie wyłączyć wyjście. Zwraca JSON z potwierdzeniem (`ok: true/false`, ewentualnie info o aktywnym zamku).
   - `/api/logs`: Odczytuje z bazy listę ostatnich N logów (lub wszystkie) i zwraca w formacie listy obiektów. Można dodać paginację lub ograniczenie, by nie zwracać nadmiernie dużego JSON.
   - `/api/service/mapping`: Zwraca aktualne mapowanie sygnałów, np. strukturę który port odpowiada której funkcji. To może być stała konfiguracja (jeśli hardware ma przypisane na sztywno, np. alarm->K1, AC->K2 itd. jak w dokumencie [28]) albo edytowalna – w MVP można zwrócić stały słownik zgodny z dokumentacją.
   - `/api/network`: Zwraca ustawienia sieci – ID urządzenia (np. *TCM-01*), adres IP, maska, brama [20]. Te informacje można odczytać z konfiguracji systemu (np. komenda ifconfig) lub przechowywać w bazie jeżeli mają być edytowalne z panelu. MVP może trzymać je w tabeli settings (i umożliwić zmianę IP tylko “na papierze”, bo faktyczna zmiana IP RPi wymaga uprawnień systemowych – poza zakresem aplikacji). ID urządzenia można przechować w bazie i wyświetlać na UI (np. w nagłówku) [29].
10. Dodaj obsługę **kontrolera klimatu** (opcjonalnie w MVP): możesz zaimplementować *background task* w FastAPI uruchamiający się przy starcie (używając `startup event`). Ten task co kilka sekund/minut sprawdza czujniki i podejmuje akcje (grzać/chłodzić/wietrzyć) według ustawień. Każda akcja generuje wpis log (użyj modelu LogEntry do dodania wpisu z timestamp i opisem). Uwzględnij też logikę drzwi: jeśli czujnik drzwi otwarty -> od razu akcje jak w dokumencie (wyłączenia i alarm) [30].

11. Upewnij się, że przy zapisie do bazy (np. nowy log) oraz odczytach nie dochodzi do konfliktów – przy odpowiedniej konfiguracji WAL powinno być dobrze. Testuj również scenariusze równoczesnych odczytów (np. wywołaj szybko kilka razy /api/state gdy background task zapisuje log) – SQLite powinno obsłużyć to (jeden zapis na raz, reszta czeka krótko).

12. **Implementacja frontend (Jinja2 + Tailwind):** Utwórz szablony HTML zgodnie z potrzebami:

13. **Login (`login.html`):** prosty formularz (login, hasło) oraz elementy informacyjne: w rogu ID urządzenia, link/ikonka do logów, na dole ustawienia sieci (można wypisać IP, maskę – pobrane z obiektu przekazanego do template) [31]. Formularz POSTuje do `/login`.
14. **Dashboard operatora (`dashboard_op.html`):** zawiera sekcje: **Sensor Data** (tabela lub karty z aktualną temperaturą baterii, temperaturą szafy, wilgotnością – z odpowiednimi jednostkami, np. °C, %RH) [32]; **Inputs** (stany czujników: np. Door1: Closed/Open, Flood1: OK/FLOOD) [33]; **Outputs** (stan urządzeń: Alarm, AC, Light, Heater, Fan_48V, Fan_230V, etc – np. jako lista przełączników on/off z aktualnym stanem) [34]. Na górze i dole strony – paski informacyjne z ID, logami, siecią (tak jak w login) [6]. Można dodać też znacznik czasu ostatniej aktualizacji danych.
15. **Panel technika (`dashboard_tech.html` + `settings.html`):** Dashboard technika może rozszerzać szablon operatora – wyświetla te same dane sensorów, inputs, outputs [35]. Dodatkowo zawiera przycisk **Ustawienia** widoczny tylko dla roli technik (kontrolę można zrobić w szablonie: `{% if user.role == 'tech' %}...{% endif %}`). Po kliknięciu przenosi do strony `settings.html`.
    W *Ustawieniach* technika udostępniamy formularze do zmiany konfiguracji:
    - **ID urządzenia:** pole tekstowe do zmiany identyfikatora (aplikacja po zatwierdzeniu zapisze do bazy i użyje przy wyświetlaniu na UI i w API).
    - **Ustawienia sieciowe:** pola dla IP, maski, bramy. **Uwaga:** zmiany tych wartości mogą być trudne do zastosowania "w locie" przez aplikację (bo konfiguracja sieci OS), więc można w MVP pozwolić je edytować i zapisać w bazie, ale z komunikatem że zmiana wymaga restartu i rekonfiguracji manualnej systemu. W praktyce technik raczej ustawia IP łącząc się przez konsolę, ale skoro dokument przewiduje podgląd/zmianę, aplikacja może to przechowywać.
    - **Progi temperatur:** pola numeryczne do ustawienia thresholdów załączenia dla: grzałki, klimatyzacji, awaryjnej wentylacji [36]. Oraz **histereza załączenia** (różnica temperatur dla ponownego włączenia) [9]. Te wartości będą używane przez logikę w kontrolerze – po zmianie od razu można je zastosować (np. kontroler zawsze odczytuje aktualne progi z bazy lub z pamięci).
    - Przycisk "Zapisz" dla zapisania zmian (PATCH/POST do jakiegoś endpointu np. `/settings/update` – implementujemy w FastAPI odpowiednio).
    - Można też wyświetlić inne informacje diagnostyczne w ustawieniach, np. wersja oprogramowania, miejsce na dysku itp., choć nie jest to w wymaganiach.
16. **Logi (`logs.html`):** Tabela lub lista logów. Każdy wpis: data/godzina (sformatowana), typ, źródło, wiadomość [11]. Można dodać mechanizm filtrowania po typie czy przycisk "Eksportuj" – który po kliknięciu wygeneruje plik (FastAPI może udostępnić endpoint do pobrania pliku .logs, generując go na podstawie bazy). W minimalnej wersji, po prostu lista ostatnich np. 100 logów z opcją przewinięcia. Ten widok może być dostępny dla technika, a operator np. tylko podgląd przez ikonkę na górze (która np. otworzy okno modal z kilkoma ostatnimi logami).
17. **Stylowanie i responsywność:** Tailwind CSS ułatwia stworzenie estetycznego UI szybko. Warto skorzystać z gotowych komponentów (np. stylowane tabele, przyciski, formularze) – ponieważ wszystko jest lokalne, należy te style zawrzeć w skompilowanym CSS (uwzględnić klasy w HTML lub w configu Tailwind). Panel nie musi być skomplikowany graficznie – priorytetem jest czytelność danych w terenie (często na laptopie technika). **Responsywność:** Można przewidzieć, że technik może używać np. tabletu czy nawet telefonu, więc zastosowanie prostego RWD (flexbox/grid z Tailwind) jest wskazane.

18. **Dynamiczna aktualizacja:** Dla efektu "real-time dashboard" można dołączyć krótki skrypt JS, który co parę sekund wykona zapytanie do `/api/state` i zaktualizuje na stronie wartości czujników i wyjść. FastAPI może zwrócić tylko JSON, a w JS (np. fetch + `element.textContent = data.sensors.temp_cab + ' °C'` itd.) odświeżyć zawartość. To podejście (polling) jest łatwe do wdrożenia i przy sieci LAN opóźnienie będzie minimalne.

Alternatywnie można użyć WebSocket: FastAPI obsługuje WebSockety, więc w przyszłości możliwe jest wdrożenie kanału aktualizacji push (co byłoby czystsze). Jednak MVP nie wymaga koniecznie tej złożoności.

19. **Budowa i testy obrazu Docker dla aplikacji:** Gdy backend i frontend są gotowe i przetestowane w środowisku developerskim (np. uruchomione na PC z danymi testowymi), przygotuj **Dockerfile** dla kontenera aplikacji:

20. Bazuj na oficjalnym obrazie Pythona zgodnym z architekturą Raspberry Pi (jeśli Pi ma ARMv7 32-bit, np. `python:3.11-slim-bullseye` arm32v7, dla 64-bit OS Raspberry arm64 analogicznie). Możesz też budować obraz bezpośrednio na RPi by użył właściwej architektury.
21. W Dockerfile skopiuj pliki projektu i zainstaluj zależności. Ustaw `ENV PYTHONUNBUFFERED=1` aby logi były od razu widoczne.
22. Włącz etap budowy Tailwind: np.
    1. `FROM node:18-alpine AS build-front` – w nim kopiujesz `tailwind.config.js`, `input.css`, folder `templates` (bo Tailwind CLI będzie chciał przejrzeć klasy w HTML) i instalujesz `npm install tailwindcss@latest` (lub kopiujesz wcześniej pobrany binarny CLI). Następnie `npx tailwindcss -c tailwind.config.js -i input.css -o /tmp/tailwind.css --minify`.
    2. Potem `FROM python:3.11-slim` – instalujesz wymagania `RUN pip install -r requirements.txt`, kopiujesz cały kod źródłowy do `/app`, i kopiujesz wygenerowany plik `tailwind.css` z poprzedniego etapu do `app/static/css/tailwind.css`.
23. Ustaw domyślny `CMD` jak wspomniano (`uvicorn app.main:app --host 0.0.0.0 --port 8000`). Upewnij się, że port 8000 jest otwarty w kontenerze (`EXPOSE 8000`).
24. **Test lokalny:** Uruchom `docker build` i `docker run` na komputerze deweloperskim (można nawet z docker-compose) aby sprawdzić, czy kontener startuje i czy można się połączyć (np. http://localhost:8000/). Ponieważ TLS i Nginx w tej fazie mogą nie być skonfigurowane, testuj aplikację bezpośrednio. Sprawdź logowanie, wyświetlanie danych, wywołania API. Upewnij się, że pliki statyczne się serwują (FastAPI statyczne pliki będzie serwował, gdy idą przez port 8000; docelowo Nginx może je też serwować jeśli zmapujemy).

25. *Opcjonalnie:* Napisać kilka testów integracyjnych (np. używając `requests` do hitnięcia endpointów w uruchomionej aplikacji kontenerowej). Dla pewności, że nic nie brakuje w obrazie (np. migracje bazy, pliki statyczne).

26. **Konfiguracja Nginx i budowa/drug kontenera:** Przygotuj pliki konfiguracyjne Nginx:

27. W pliku `nginx.conf` ustaw globalne wartości, ale większość można w bloku server w `site.conf`. Przykład `site.conf`:

```
server {
    listen 443 ssl;
    server_name _;  # nasłuchuj na wszystkie hosty (w sieci lokalnej i
tak dostęp po IP lub hostname lokalnym)
    ssl_certificate /etc/nginx/certs/tcm.crt;
    ssl_certificate_key /etc/nginx/certs/tcm.key;
    # (ew. ssl_protocols, ciphers etc. - można użyć bezpiecznych
domyślnych)
    location / {
```

```
        proxy_pass http://app:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
    location /static/ {
        # przykład: przekazujemy również do app (lub lepiej: serwować
statyczne bezpośrednio? - w MVP można proxy)
        proxy_pass http://app:8000/static/;
    }
}
server {
    listen 80;
    server_name _;
    return 301 https://$host$request_uri;
}
```

Powyższa konfiguracja przekierowuje port 80 do 443 i proxyfikuje całą resztę do aplikacji. Jeśli chcemy, by Nginx sam serwował statyczne pliki, można zamiast proxy dla `/static` zrobić:

```
location /static/ {
    alias /usr/share/nginx/html/static/;  # jeśli statyki z kopii w
obrazie Nginx
}
```

To wymaga skopiowania folderu `app/static` do obrazu Nginx (np. w Dockerfile Nginx dodać COPY). **Prostsze rozwiązanie:** Proxy wszystko do FastAPI (który i tak szybko obsłuży statyki), przy małym obciążeniu to nie problem.

28. **Obraz Nginx:** Możesz użyć bezpośrednio `nginx:alpine` i jedynie montować konfigurację i certyfikaty przy `docker-compose up`. Alternatywnie, zbudować własny obraz (np. żeby wbudować config i certyfikaty w obraz). Budowa jest prosta:
    - Dockerfile bazujący na `nginx:alpine`, `COPY site.conf /etc/nginx/conf.d/default.conf`, `COPY certs/ /etc/nginx/certs/`, `COPY static/ /usr/share/nginx/html/static/` (opcjonalnie). Taki obraz będzie zawierał wszystko, ale wadą jest, że przy wymianie certyfikatu trzeba budować obraz od nowa. Montowanie woluminów daje łatwość podmiany cert bez ruszania obrazu.

29. Przetestuj Nginx lokalnie: możesz uruchomić docker-compose na swoim PC (jeśli masz cert self-signed, dodaj wyjątek w przeglądarce) i sprawdzić, czy po wejściu na https://localhost panel działa poprawnie przez proxy.

30. **Przygotowanie Raspberry Pi:** Zainstaluj system na Raspberry Pi (np. Raspberry Pi OS Lite 64-bit, jeśli planujesz 64-bitowe obrazy Dockera). Ustaw statyczny adres IP według wymagań (np. 192.168.0.100 jak w dokumentacji) – konfiguracja w `/etc/dhcpcd.conf` lub NetworkManager, w zależności od OS. Wyłącz zbędne usługi, zaktualizuj system do najnowszych pakietów (przed odłączeniem od Internetu). Zainstaluj **Docker Engine** i **docker-compose**:

31. Jeśli Pi nie ma internetu, musisz wcześniej ściągnąć pakiety .deb Dockera, lub użyć script (`get.docker.com`) kiedy jest podłączony. Alternatywnie, możesz przygotować obraz karty SD już z Dockerem.

32. Upewnij się, że użytkownik `pi` (lub inny) jest w grupie docker dla wygody.

33. (Jeśli to nowe urządzenie) Wgraj też certyfikat CA (jeśli używasz własnego CA) do zaufanych na laptopach, które będą łączyć się do panelu – inaczej każda przeglądarka będzie ostrzegać o certyfikacie self-signed. Można to pominąć, akceptując wyjątek bezpieczeństwa manualnie w przeglądarce za pierwszym razem.

34. **Deployment aplikacji na Raspberry Pi:** Przenieś obrazy Docker lub kod na Raspberry Pi:

35. **Opcja A: Budowanie obrazów na Raspberry Pi** – jeśli Pi ma wystarczające zasoby i czas, możesz skopiować projekt (SCP, pendrive) i wykonać `docker-compose build` bezpośrednio na Pi. To pobierze bazowy obraz Pythona i Nginx (wymaga internetu na czas builda!). Jeśli Internet nie jest możliwy, musisz dostarczyć te obrazy offline:
36. **Opcja B: Transfer obrazów offline** – zbuduj obrazy na komputerze (na architekturę ARM!). Można to zrobić przy użyciu emulacji (Docker Buildx QEMU) lub mając drugi Pi. Alternatywnie, użyj multiarch na PC: np. `docker buildx build --platform linux/arm/v7 -t tcm_app_image .` i tak samo dla Nginx. Potem użyj `docker save tcm_app_image > app.tar` i `docker save tcm_nginx_image > nginx.tar`. Przenieś te pliki tar na Raspberry Pi (np. na USB lub przez scp w sieci lokalnej). Na Pi wykonaj `docker load < app.tar` i `docker load < nginx.tar` – obrazy pojawią się lokalnie.
37. Skopiuj pliki konfig (docker-compose.yml, certyfikaty if needed, etc.) na Pi. Zmodyfikuj docker-compose.yml jeśli np. używasz image z registry vs local build.
38. Uruchom aplikację: `docker-compose up -d`.
39. Monitoruj logi: `docker-compose logs -f app` i `docker-compose logs -f proxy`. Sprawdź czy:
    - Aplikacja wystartowała bez błędów (log FastAPI powinien pokazać, że nasłuchuje na 0.0.0.0:8000).
    - Nginx wystartował i nasłuchuje na 443. Możesz użyć `ss -tlnp` na Pi by upewnić się, że port 443 jest otwarty przez nginx.
    - W logach Nginx nie ma błędów typu *502 Bad Gateway* (to by oznaczało problem z połączeniem do app – sprawdź nazwę hosta i sieć w compose) [37].
40. Jeśli wszystko wygląda ok, spróbuj połączyć się z komputera klienckiego: np. na laptopie w tej samej sieci wpisz `https://192.168.0.100/` (lub odpowiedni IP/nazwa). Powinien pojawić się panel logowania. Jeśli certyfikat jest self-signed, przeglądarka wyświetli ostrzeżenie – zaakceptuj i kontynuuj. Zaloguj się testowym kontem (np. *technik/haslo* – upewnij się, że takie konto jest w bazie; można na szybko dodać je ręcznie do bazy przed uruchomieniem albo mieć migrację tworzącą domyślnych userów).
41. Sprawdź funkcjonalność panelu na żywo: zmiana roli (logowanie jako operator vs technik), przeglądanie dashboardu, czy wartości sensowne (z symulacji), czy logi się zapisują gdy np. symulujesz zdarzenie (można dodać przycisk testowy generujący event). Wywołaj też ręcznie API (np. `curl https://192.168.0.100/api/state -k` jeśli self-signed, by zobaczyć JSON).

42. Wprowadź ewentualne poprawki po testach integracyjnych. Często drobne problemy wychodzą na docelowym sprzęcie (np. inne nazwy interfejsów sieciowych, wydajność itp.).

43. **Konfiguracja auto-start i finalizacja:** Upewnij się, że kontenery startują przy reboot Pi. Można to osiągnąć przez:

44. Dodanie do crontaba (lub systemd) polecenia `docker-compose -f /path/to/docker-compose.yml up -d` przy starcie systemu. Jednak Docker na Raspberry Pi OS zwykle ma już mechanizm, że ostatnio uruchomione kontenery z `restart: unless-stopped` powinny się

same uruchomić po restarcie Docker daemon. Test: zrestartuj Raspberry Pi i sprawdź `docker ps` czy oba kontenery działają.

45. Sprawdź, czy po restarcie aplikacja zachowała dane (np. ustawiony wcześniej inny próg temperatury w ustawieniach – czy dalej jest w bazie; logi historyczne – czy nie zniknęły).

46. **Backup/Recovery:** Jako że nie ma zdalnych aktualizacji, warto opracować procedurę aktualizacji offline: np. technik otrzymuje nową wersję oprogramowania na pendrive albo jako obraz Docker. Wtedy:

    - Może zatrzymać kontenery stare (`docker-compose down`), załadować nowe obrazy (`docker load`), podmienić pliki compose jeśli się zmieniły, i `docker-compose up -d` nową wersję. Baza SQLite pozostanie (wolumin), chyba że struktura bazy się zmieni – wtedy migracja powinna zostać wykryta i wykonana przez aplikację (można użyć np. Alembic do migracji schematu między wersjami).
    - Alternatywnie aktualizacja może polegać na wymianie całej karty SD z nową wersją systemu i aplikacji – wtedy jednak logi i ustawienia warto wcześniej eksportować i zaimportować do nowego (to już kwestia organizacji poza samym kodem).

47. **Dokumentacja i przekazanie:** Przygotuj dokument dla użytkowników technicznych:

48. Instrukcję logowania dla operatora i technika (domyślne hasła, zalecenie zmiany jeśli to planowane w UI).
49. Opis dostępnych funkcji panelu (co gdzie się ustawia, co oznaczają poszczególne wskazania – częściowo jest to w *Opis Web Panel i API*).
50. Opis interfejsów API dla integratorów (wraz z przykładowymi odpowiedziami JSON – można tu zamieścić przykłady z dokumentu, np. strukturę /api/state [38] [34], by ułatwić integrację).
51. Ewentualnie procedurę bezpiecznego wyłączenia/wymiany urządzenia, ponieważ to sprzęt embedded.
52. Te informacje mogą trafić do pliku `README.md` w projekcie lub osobnego dokumentu.

Podsumowanie

Powyższa roadmapa przedstawiła kompleksowy plan stworzenia MVP aplikacji **TCM 2.0** na Raspberry Pi w środowisku odizolowanym od Internetu. Rozwiązanie zakłada wykorzystanie nowoczesnego stosu technologicznego (FastAPI, Docker, Tailwind) dostosowanego do pracy offline. Wdrożenie skupia się na zapewnieniu lokalnej dostępności (LAN), bezpieczeństwie komunikacji (TLS), rozdzieleniu ról użytkowników oraz zgodności z założeniami interfejsu i API z dokumentu TCM 2.0 [3] [39]. Dzięki modularnej architekturze (podział na kontenery i czytelny kod) system będzie łatwy w utrzymaniu i rozwoju. Wszystkie krytyczne założenia – od braku zależności zewnętrznych po trwałość danych i mechanizmy bezpieczeństwa – zostały uwzględnione, aby MVP mógł zostać z powodzeniem zainstalowany i używany w docelowej infrastrukturze telekomunikacyjnej.

---

[1] [26] javascript - How can I use tailwindcss without internet? - Stack Overflow
https://stackoverflow.com/questions/76303105/how-can-i-use-tailwindcss-without-internet

[2] [24] Write-Ahead Logging
https://sqlite.org/wal.html

[3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [25] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [38] [39] TCM 2.0 - Web Panel, API.pdf
file://file-UVPTfuhHqKjKbsbAuLjoRj

22 23 37 Deploying a FastAPI Application with Docker-Compose, Nginx, and Cloudflare | by Fortismanuel | Medium
https://medium.com/@fortismanuel/deploying-a-fastapi-application-with-docker-compose-nginx-and-cloudflare-05a8c2fa4397