# SmartGarden – Documentație Tehnică de Proiect

## 1. Prezentare generală a proiectului și scopul aplicației

SmartGarden este o aplicație **IoT (Internet of Things)** destinată iubitorilor de plante care doresc monitorizarea și îngrijirea automată a grădinii sau plantelor de interior. Proiectul constă dintr-un **dispozitiv hardware ESP32** echipat cu senzori (umiditate sol, temperatură, lumină, nivel apă, calitate aer) și o pompă de irigare, un **server web (API REST)** pentru gestionarea datelor și logicii, și o **aplicație mobilă (realizată în .NET MAUI)** prin care utilizatorul interacționează cu sistemul. Scopul aplicației este de a automatiza udarea plantelor și de a oferi utilizatorului informații în timp real despre starea fiecărei plante, precum și notificări și statistici relevante.

În practică, SmartGarden permite utilizatorilor să **adauge plante în aplicație**, să atașeze un dispozitiv fiecărei plante, să **monitorizeze parametrii de mediu** (umiditatea solului, temperatura aerului, luminozitatea, nivelul apei în rezervor etc.) și să primească **alerte** (de exemplu, dacă rezervorul de apă e gol sau dacă planta are nevoie de apă). Totodată, sistemul poate realiza **udarea automată** a plantei conform unor praguri prestabilite, folosind pompa controlată de ESP32. Utilizatorul poate de asemenea porni manual udarea din aplicație, poate calibra senzorii pentru precizie sporită și poate vizualiza **statistici** zilnice/săptămânale despre îngrijirea plantelor. În ansamblu, proiectul are ca scop să **uşureze îngrijirea plantelor** și să prevină uscarea sau supra-udarea acestora prin automatizare și monitorizare inteligentă.

## 2. Arhitectura soluției pe proiecte

Proiectul este organizat pe un design multi-layer, cu patru componente majore, fiecare în propriul proiect (assembly) .NET, plus firmware-ul dispozitivului IoT. Arhitectura urmărește separarea responsabilităților în **layer**-e distincte, pentru a asigura o dezvoltare și mentenanță facilă:

[[1]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Structur%C4%83%20proiect%3A)[[2]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04)

* **SmartGarden.Core** – conține logica de domeniu și definițiile contractelor (modele de date, DTO-uri, interfețe pentru servicii, reguli de validare de bază). Acest proiect **nu are dependențe** către EF Core sau ASP.NET, fiind un layer pur de business/domain logic[[3]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Toate%20entit%C4%83%C8%9Bile%20%2B%20DTO,interfe%C8%9Be%20compilabile%2C%20cu%20XML%20summaries).
* **SmartGarden.Data** – stratul de acces la date, bazat pe Entity Framework Core. Include configurarea modelului entităților (DbContext, mapări Fluent API), migrații și seeding de date inițiale. Expune metode de inițializare a datelor și se ocupă de conectarea la baza de date (ex. SQLite în dev, SQL Server în producție).
* **SmartGarden.API** – serviciul web **ASP.NET Core Web API** care orchestrează toate cerințele. Expune endpoint-urile REST către aplicația mobilă și dispozitivul IoT, implementează serviciile (business logic) folosind modelele din Core și accesul la DB din Data. Se ocupă de autentificare JWT, autorizare, validarea intrărilor, trimiterea de răspunsuri JSON și integrarea cu serviciul de notificări. Conține controller-ele API, serviciile concrete și configurările cross-cutting (CORS, JWT, mapping etc).
* **SmartGarden.App** – aplicația mobilă cross-platform (realizată în .NET MAUI) care oferă interfața utilizatorului. Urmează un model MVVM, conține pagini (UI XAML) și ViewModel-uri pentru fiecare ecran, servicii pentru comunicarea cu API-ul și stocare locală, precum și integrarea cu sistemul de notificări al dispozitivului mobil. Această componentă folosește API-ul REST pentru a primi și trimite date despre plante.
* **Firmware ESP32 (SmartGarden IoT)** – codul sursă C/C++ rulând pe microcontrollerul ESP32, responsabil de colectarea datelor de la senzori și comunicarea cu serverul prin Wi-Fi. Firmware-ul citește periodic senzorii și trimite datele catre API, primește comenzi de udare, și gestionează modul de calibrare al senzorilor în colaborare cu aplicația mobilă.

Arhitectura de ansamblu este prezentată în diagrama de mai jos, evidențiind structura proiectelor și legăturile dintre ele:

*Figura 1: Structura proiectului SmartGarden (soluție .NET cu 4 proiecte și un modul IoT separat).*

Fiecare din aceste componente este detaliată în secțiunile următoare, explicând ce conține și ce responsabilități are în cadrul sistemului.

## 3. Descriere detaliată a fiecărui layer (proiect)

În continuare este prezentat în detaliu conținutul fiecărui layer al aplicației SmartGarden – de la modelul de date și accesul la DB, până la API-ul web și aplicația mobilă – inclusiv principalele clase și fragmente de cod ilustrative. Acest capitol acoperă modelele entităților, DTO-urile transferate prin API, configurările EF Core, serviciile și controller-ele API, precum și structura aplicației MAUI.

### 3.1 Layer-ul de date – SmartGarden.Data (Entity Framework Core)

Acest proiect definește baza de date și modul de mapare a obiectelor de domeniu la tabele, folosind **Entity Framework Core**. Conține clasa de context SmartGardenDbContext și configurările fluent (clase ce implementează IEntityTypeConfiguration<T> pentru fiecare entitate). De asemenea, include migrațiile EF Core și eventual scripturi de populare a datelor inițiale (seeding minimal pentru tabelele de referință). Prin separarea acestui layer, se facilitează schimbarea providerului de date (ex. trecerea de la SQLite la SQL Server în producție) fără a afecta logica de business.

**Modelul bazei de date:** Baza de date este relațională și conține mai multe tabele interconectate. Lista de tabele și relațiile finale dintre ele sunt următoarele[[4]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Tabele%20%26%20rela%C8%9Bii%20)[[5]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite):

* Users – tabelul utilizatorilor aplicației (cheia primară UserId). Fiecare utilizator are asociat **în relație one-to-one** un rând în tabela UserSettings și **one-to-many** cu plantele din Plants[[6]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Users%20,UserSettings%2C%201%E2%80%93N%20Plants).
* UserSettings – stochează preferințe/configurații la nivel de utilizator (cheia primară este totodată cheie externă către Users, indicând relația 1-1)[[6]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Users%20,UserSettings%2C%201%E2%80%93N%20Plants). Conține de exemplu setarea dacă udarea automată este activă, pragul de umiditate pentru udare automată etc.
* Plants – reprezintă plantele adăugate de utilizatori (cheie primară PlantId). Fiecare plantă aparține unui utilizator (UserId ca foreign key), are specificată o specie (SpeciesId) și un tip de sol (SoilTypeId). Relațiile: o plantă aparține unui singur utilizator, unei singure specii și unui singur tip de sol (relații many-to-one către Users, Species, SoilTypes), iar o plantă are **multe** citiri de senzori (SensorReadings) și **multe** log-uri de udare (WateringLogs) asociate (relații one-to-many)[[7]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=interval).
* Species – lista de specii de plante (cheie primară SpeciesId), conținând denumirea comună, denumirea științifică și **praguri implicite** recomandate pentru acea specie (interval optim de umiditate sol, temperatură, lumină, etc.)[[8]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite). Aceste valori implicite pot fi folosite la crearea unei plante sau la notificări.
* SoilTypes – tipurile de sol (cheie primară SoilTypeId), conținând denumiri/descripții și recomandări de udare asociate acelui tip de sol[[9]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite). De exemplu, un sol nisipos vs. unul argilos ar putea avea nevoi diferite de udare (durată de udare, pauze).
* SensorReadings – citirile senzorilor colectate periodic (cheie primară ReadingId). Fiecare citire aparține unei plante (PlantId foreign key) – deci relație many-to-one către Plants[[10]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorReadings%20,Plants). Înregistrarea conține valorile măsurate la un moment dat: umiditate sol, temperatură aer, umiditate aer, nivel de lumină, calitatea aerului, nivelul apei etc., plus timestamp-ul (CreatedAt).
* WateringLogs – log-urile de udare (cheie primară WateringId), indicând fiecare eveniment când a fost pornită pompa pentru o plantă[[11]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorReadings%20,Plants). Fiecare log are referință la planta aferentă (PlantId), durata udării în secunde, modul (Manual sau Auto), timestamp etc. Relație many-to-one către Plants.
* (Opțional) PlantNotes – nu este obligatoriu, dar se poate extinde cu un tabel de notițe/sfaturi legate de plante (dacă se dorește stocarea de notițe ale utilizatorului pentru fiecare plantă).

În diagrama de date de mai jos se poate observa structura entităților și legăturile (chei străine) dintre ele conform descrierii de mai sus:

*Figura 2: Diagrama entităților bazei de date SmartGarden, cu tabelele și relațiile (PK – cheie primară, FK – cheie externă).*

Notă: Pentru performanță, se vor defini și indexuri pe coloane utilizate frecvent la filtrare: de exemplu un index compus pe (PlantId, CreatedAt DESC) în SensorReadings (pentru a optimiza obținerea ultimei citiri sau a istoricului cronologic), un index pe Plants(UserId) (listarea plantelor per utilizator) și pe WateringLogs(PlantId, CreatedAt)[[12]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Indexuri%20cheie%3A%20SensorReadings,PlantId%2C%20Timestamp).

**Clasa DbContext și configurarea EF Core:** Clasa principală de context, SmartGardenDbContext, definește DbSet<T> pentru fiecare entitate: Users, UserSettings, Plants, Species, SoilTypes, SensorReadings, WateringLogs. În metoda OnModelCreating, contextul încarcă toate configurările specifice entităților folosind Fluent API. Aceste configurări sunt implementate în clase separate (exemplu: UserConfiguration, PlantConfiguration etc.), pentru a menține codul organizat[[13]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SmartGardenDbContext%20cu%20toate%20DbSet%3C%3E)[[14]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20).

Fiecare configurație setează explicit cheile primare, relațiile și eventuale constrângeri suplimentare cum ar fi lungimi maxime, valori implicite și indexuri[[13]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SmartGardenDbContext%20cu%20toate%20DbSet%3C%3E). De exemplu, relația one-to-one dintre User și UserSetting se mapează astfel în UserConfiguration:

// Extras din UserConfiguration.cs (Data Layer)  
builder.HasOne(u => u.UserSetting)  
 .WithOne(s => s.User)  
 .HasForeignKey<UserSetting>(s => s.UserId);

În UserSettingConfiguration se pot seta valori implicite pentru anumite coloane, cum ar fi pragul de umiditate și intervalul de citire, dacă nu sunt specificate de utilizator:

builder.HasKey(s => s.UserId);  
builder.Property(s => s.SoilMoistThreshold).HasDefaultValue(30.0); // prag default 30%  
builder.Property(s => s.DataReadIntervalMin).HasDefaultValue(15); // interval default 15 min

*(Exemplul de mai sus arată că, implicit, dacă utilizatorul nu și-a configurat altfel, udarea automată va porni la scăderea umidității sub 30%, iar dispozitivul va citi senzorii la fiecare 15 minute.)*

Configurările pentru celelalte entități stabilesc relațiile many-to-one. De exemplu, PlantConfiguration va specifica relația cu citirile și log-urile de udare (one-to-many), iar SensorReadingConfiguration doar definește cheie primară compusă sau index (dacă e cazul). Multe mapează convențiile implicite (ex. EF recunoaște automat PlantId ca FK către Plants, deci configurări suplimentare pot fi minime).

**Migrații și seeding:** Se pornește de la o migrație inițială (Init) ce creează toate tabelele necesare[[15]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Migrations%20%28Init%29%20%2B%20scripturi). Ulterior, se pot adăuga migrații (ex. pentru modificări la scheme sau date). Pentru popularea datelor de referință, se prevede **seeding minim** cu 6-10 specii și 4-6 tipuri de sol, precum și un utilizator demo cu 1-2 plante asociate[[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants). Aceste date pot fi inserate fie direct în migrația inițială (folosind InsertData în Up()) sau prin metoda HasData în configurări. De exemplu, se pot adăuga în SpeciesConfiguration câteva instanțe predefinite:

builder.HasData(  
 new Species { SpeciesId = 1, CommonName = "Ficus lyrata", ScientificName = "Ficus lyrata",   
 DefaultSoilMoistMin = 30, DefaultSoilMoistMax = 60, ... },  
 // ... alte specii  
);

*(în codul final, fiecare Species are și valorile implicite pentru intervalele ideale de umiditate, temperatură, lumină, etc.)*

Similar pentru SoilType, se pot predefini tipuri ca “Universal (pământ flori)”, “Cactuși (sol nisipos)”, “Argilos”, cu recomandări de udare (ex: durată de udare, pauză). De asemenea, un **utilizator demo** (ex. email demo@smartgarden.com cu o parolă cunoscută) și o plantă demo se pot crea pentru a facilita testarea rapidă a aplicației[[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants).

**Integrare și DI:** Proiectul Data expune o metodă de extindere pentru a înregistra contextul în containter-ul de IoC al aplicației: de exemplu services.AddDataLayer(connectionString, useSqlite) care intern apelează options.UseSqlite sau UseSqlServer după caz și adaugă contextul ca serviciu (scop Scoped)[[17]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Data%20seeding%20,demo). Astfel, în startup-ul API (în Program.cs), se va apela această extensie pentru a configura baza de date.

În rezumat, layer-ul SmartGarden.Data definește **structura persistentă a datelor** și modul de mapare EF Core, izolând astfel accesul la DB de restul aplicației. Orice operațiune CRUD pe entități va trece prin acest layer (direct prin context sau prin repository/servicii care îl folosesc).

### 3.2 Layer-ul de core (domeniu) – SmartGarden.Core

SmartGarden.Core reprezintă inima logicii de domeniu a aplicației. Acest proiect conține **modelele de date (entitățile)**, **DTO-urile** folosite pentru transfer de date între client și server, precum și **interfețele serviciilor** ce definesc contractele logice. Scopul este să avem într-un singur loc toate definițiile esențiale, fără a depinde de vreun framework specific – de aceea acest assembly nu referențiază EF Core sau ASP.NET, putând fi ușor de testat separat[[3]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Toate%20entit%C4%83%C8%9Bile%20%2B%20DTO,interfe%C8%9Be%20compilabile%2C%20cu%20XML%20summaries).

**Modele de entități (Domain Models):** Acestea sunt clasele C# care reflectă tabelele din baza de date și definesc proprietățile și relațiile la nivel de cod. Exemple de modele incluse în Core.Models[[18]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Ce%20livrez%3A):

* User – cu proprietăți UserId, Email, PasswordHash, Name, CreatedAt, plus navigațională UserSetting (one-to-one) și colecții Plants (one-to-many).
* UserSetting – proprietăți precum UserId (cheie primară și externă), AutoWateringEnabled (bool – dacă udarea automată e activă), SoilMoistThreshold (pragul de umiditate al solului sub care se udă automat), DataReadIntervalMin (intervalul de citire a senzorilor, în minute) etc.
* Plant – proprietăți PlantId, UserId (FK către User), SpeciesId (FK către Species), SoilTypeId (FK către SoilType), RoomName (numele camerei unde se află planta, dacă e indoor), IsOutdoor (dacă planta e în exterior), DateAcquired (data adăugării achiziției plantei)【22†】. De asemenea, conține navigații: referințe către User, Species, SoilType și colecții SensorReadings și WateringLogs asociate【22†】.
* Species – cu SpeciesId, CommonName, ScientificName și câmpuri de *praguri implicite*: ex. DefaultSoilMoistMin, DefaultSoilMoistMax (procente de umiditate), DefaultTempMin/Max (°C), DefaultLightMin/Max (nivel lumină, eventual în lux sau într-o scară), DefaultHumidityMin/Max (umiditate aer). Aceste valori pot fi folosite la setarea inițială a plantelor și la notificări.
* SoilType – cu SoilTypeId, Name, Description, plus parametri de recomandare udare: de exemplu RecWaterDueSec (durata recomandată de udare în secunde) și PauseBetweenWaterMin (pauza recomandată între două udări consecutive, în minute), utile pentru a configura algoritmul de udare automată optim pentru acel sol.
* SensorReading – proprietăți: ReadingId, PlantId (FK), valorile citite: de ex. SoilMoisture (poate stoca raw ADC sau un %), AirTemp (°C), AirHumidity (%), LightLevel (ex. lux), AirQuality (un indice sau ppm, dacă se folosește senzor MQ135 pentru calitatea aerului), WaterLevel (de ex. cm rămași sau procent din rezervor), și CreatedAt (timpul citirii).
* WateringLog – proprietăți: WateringId, PlantId (FK), DurationSec (durata udării), Mode (enum – manual sau auto), CreatedAt (timpul pornirii udării).

Aceste clase sunt simple **POCO**-uri, conținând doar proprietăți auto-implementate și eventual colecții inițializate. Logica legată de ele (ex. validări de consistență) poate fi adăugată sub formă de metode. De exemplu, se pot defini metode de validare precum Plant.Validate() care să verifice că valorile pragurilor sunt corecte (min < max etc.) sau că datele nu sunt în viitor, etc. – pentru a garanta *invariantele de domeniu*[[19]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Invariante%20minime%20%28ex,intervale%20corecte).

**Enumerări utile:** În Core pot exista definiri de enum-uri pentru lizibilitate, de exemplu: Placement (cu valorile posibile *Indoor* sau *Outdoor*) pentru a indica dacă planta e în interior sau exterior (poate fi derivat și din IsOutdoor bool) și WateringMode cu valorile *Manual* și *Auto* pentru logurile de udare[[20]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Models%3A%20User%2C%20UserSettings%2C%20Plant%2C%20Species%2C,op%C8%9Bional). Astfel de enum-uri sunt folosite atât în entități (WateringLog.Mode), cât și în interpretarea cererilor API.

**DTO-uri (Data Transfer Objects):** Pentru comunicarea prin API, Core definește o serie de DTO-uri sub forma unor record structuri (sau clase simple) ce reprezintă fie **cereri** primite, fie **răspunsuri** trimise clientului[[21]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DTOs%20). Aceste DTO-uri sunt separate de entitățile de bază, pentru a controla exact ce date expunem și primim prin API (de exemplu, nu vom trimite niciodată PasswordHash către client, ci doar date necesare). DTO-urile includ:

* **Autentificare:** LoginRequest (cu proprietăți Email și Password transmise la login sau înregistrare) și LoginResponse (conține de obicei token-ul JWT generat și eventual ID-ul utilizatorului autentificat)[[22]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auth%3A%20LoginRequest%2C%20LoginResponse).
* **Plants:** PlantCreateRequest și PlantUpdateRequest – structuri ce reprezintă datele necesare la adăugarea sau actualizarea unei plante (ex. nume/roomName, specie, tip sol, dată achiziție, flag indoor/outdoor, eventual praguri personalizate dacă se permite)[[22]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auth%3A%20LoginRequest%2C%20LoginResponse). PlantResponse – DTO folosit la trimiterea datelor de plantă către client, conținând detaliile plantei (nume, specie, sol, ultimele valori măsurate eventual etc.).
* **Sensors:** SensorReadingRequest – formatul datelor trimise de dispozitivul ESP32 către API la o citire senzori. Conține valorile numeric: ex. temperatureC, humidityAir, soilMoisture, lightLux, airQuality, waterLevel[[23]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Loop%20logic%3A)[[24]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). Acestea corespund valorilor măsurate de senzori; unele pot fi trimise ca raw (de exemplu soilMoisture: 612 – valoare analogică brută) sau ca procente dacă dispozitivul a aplicat calibrarea[[25]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). De asemenea, SensorReadingDto (sau Response) – folosit când clientul mobile cere ultima citire sau istoricul; conține valorile relevante plus timestamp-ul. HistoryQuery – un DTO care conține parametri de interogare pentru istoric (de ex. interval de timp, granulare pe oră/zi).
* **Watering:** WaterCommandRequest – folosit la trimiterea unei comenzi de udare manuală către dispozitiv. Conține de ex. un câmp command care poate fi "ON" sau "OFF" și opțional durationSec pentru a specifica câte secunde să pornească pompa[[26]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Sensors%3A%20SensorReadingRequest%2C%20SensorReadingDto%2C%20HistoryQuery). Răspunsul la o comandă de udare poate fi un mesaj simplu ({ message: "WATER ON" } ca confirmare)[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). Separat, WateringLogDto ar fi folosit pentru a transmite istoricul de udări către client (dacă se afișează).
* **Statistici:** DailyStatsDto și WeeklyStatsDto – conțin agregări pentru o zi sau o săptămână (de ex. număr total de udări, media valorilor de umiditate, top plante udate etc., conform cerințelor)[[28]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Sensors%3A%20SensorReadingRequest%2C%20SensorReadingDto%2C%20HistoryQuery).

**Interfețele serviciilor:** Core definește contractele (interfețe) pe care layer-ul de business/API trebuie să le implementeze[[29]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interfaces%20,ISensorService%2C%20IWateringService%2C%20IStatsService). Aceste interfețe reprezintă funcționalitățile esențiale ale sistemului:

* IAuthService – metode pentru înregistrare utilizator nou și autentificare (validare user/parolă și generare token JWT)[[30]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token).
* IPlantService – operațiile de gestionare a plantelor: creare, citire (toate sau una), actualizare, ștergere, plus eventual operații adiționale precum setarea stării de udare automată (on/off) pentru o plantă[[31]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20CRUD%20%2B%20ownership%20checks,user%20isolation).
* ISensorService – metode pentru salvarea unei citiri de senzori (când vine de la dispozitiv), obținerea ultimei citiri pentru o plantă, obținerea istoricului de citiri într-un interval (cu agregare pe ore sau zile)[[31]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20CRUD%20%2B%20ownership%20checks,user%20isolation).
* IWateringService – metode pentru a trimite comenzi de udare manuală către dispozitiv și pentru a gestiona udarea automată. De exemplu, TriggerManualWaterAsync(plantId, duration) pentru udare manuală (care va loga evenimentul și va marca comanda undeva pentru a o prelua dispozitivul) și SetAutoModeAsync(plantId, enabled) pentru a activa dezactiva modul automat pentru o plantă[[32]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday).
* IStatsService – metode pentru a calcula statistici zilnice și săptămânale (de ex. câte udări a avut o plantă, care sunt plantele cele mai udate, câte notificări au fost etc.), folosite de controlerul de statistici[[33]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday).
* (Opțional) IAlertService – ar putea exista pentru gestionarea alertelor/notificărilor, deși blueprint-ul sugerează că alertele pot fi generate de un background service direct în API (vezi secțiunea Alerts)[[34]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsService%3A%20agreg%C4%83ri%20,plante%2C%20sub%20prag%20azi%2C%20etc).

Interfețele definesc semnăturile metodelor, iar implementările lor vor exista într-un alt layer (în cazul nostru, în SmartGarden.API, la folderul Services). Această abordare permite **injectarea dependențelor** (Dependency Injection) și testarea ușoară a logicii prin mock-uri, asigurând totodată o separare clară a contractului față de implementare.

**Reguli de business și validare:** Layer-ul Core poate conține și logica de validare a datelor de intrare și a constrângerilor de business, în special prin metode în modele sau servicii. De exemplu, la crearea unei plante noi, PlantService.CreatePlantAsync ar verifica prin invariantele definite că valorile implicite ale pragurilor (min/max umiditate) sunt consistente, sau că data adăugării nu este în viitor. Unele validări pot fi făcute și la nivel de DTO (ex. folosind [DataAnnotations] pe proprietăți, cum ar fi [Required] pe email, etc.), restul fiind asigurate în servicii sau controlere cu cod imperativ.

În concluzie, SmartGarden.Core cuprinde **definițiile** fundamentale ale aplicației (fără implementarea efectivă a persistenței sau a web API-ului), oferind astfel o sursă unică de adevăr pentru structura datelor și contractele logice. Un avantaj al acestui design este că putem utiliza aceste definiții și pe client (ex. aplicația MAUI folosește aceleași DTO-uri la deserializare, evitând duplicarea claselor de mesaje).

### 3.3 Layer-ul API web – SmartGarden.API (ASP.NET Core)

Proiectul SmartGarden.API este cel care **leagă totul laolaltă**, expunând funcționalitatea prin HTTP REST și implementând efectiv serviciile definite în Core folosind EF Core (Data) pentru stocare. Este o aplicație ASP.NET Core Web API (target .NET 8) configurată cu autentificare JWT, documentație Swagger și alte middleware-uri necesare. Structura principală a acestui proiect cuprinde:

* **Controllers** – un set de controlere API care mappează direct cererile HTTP la acțiuni (metode) ce apelează serviciile. Fiecare controller corespunde unei grupe logice de endpoints (ex: AuthController pentru autentificare, PlantController pentru gestionarea plantelor, etc.).
* **Services (implementări)** – clase care implementează interfețele din Core (IPlantService, ISensorService, etc.) și conțin logica de business efectivă, folosind contextul de date sau alte servicii. Aceste implementări sunt înregistrate în DI container și injectate în controlere.
* **Models/DTOs** – de regulă, proiectul API poate conține modele pentru request/response. În cazul SmartGarden, s-au definit DTO-urile în Core, deci aici vor fi reutilizate. Pot exista totuși clase specifice API, de ex. modele pentru autentificare sau configurări interne.
* **Configuration & Program** – codul de inițializare al aplicației (ex. Program.cs) unde se configurează serviciile (AddDbContext, AddAuthentication/JWT, AddCors, AddControllers, etc.), se adaugă mapping (ex. AutoMapper dacă e folosit) și se definesc setările (citite din appsettings.json).

Să detaliem principalele componente:

**Autentificare și autorizare (JWT):** API-ul folosește JSON Web Tokens pentru autentificare. La login sau register, se generează un token semnat cu o cheie secretă definită în configurare (ex. "Jwt": { "Key": "...", "ExpiresMinutes": 120 } în appsettings)[[35]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). Acest token este trimis clientului și trebuie furnizat ulterior în header-ul Authorization la fiecare cerere protejată. În Startup/Program, se configurează autenticarea JWT (schemă "Bearer") indicând cheia secretă și parametrii (algoritm HMAC SHA256 implicit, timp de expirare etc.). Toate rutele ce necesită utilizator logat au [Authorize] aplicat la nivel de controller sau acțiune. În mod particular, **AuthController** permite acces anonim (pentru login/înregistrare), iar **Plants**, **Watering**, **Stats** etc. necesită token valid[[36]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantsController%20)[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). Endpoint-urile folosite de dispozitivul IoT (ex. POST sensor readings) pot fi lăsate fie deschise cu alt mecanism de chei (vezi discuția despre Device Keys mai jos la Configurări), fie protejate (dar ar implica autentificarea device-ului). S-a optat în proiect ca aceste endpoint-uri specifice ESP32 să nu necesite JWT de user, ci o cheie de dispozitiv alternativă.

**Controller-ele și Endpoint-urile REST:** Fiecărui set de funcționalități îi corespunde un controller. Mai jos este lista completă a endpoint-urilor și acțiunilor, conform specificației[[37]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthController%20)[[38]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK):

* **AuthController** (/api/auth):
* POST /register – primește un LoginRequest (conținând email și parolă) și creează un cont nou. Răspuns: ID-ul utilizatorului nou creat (int)[[39]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthController%20). Va valida unicitatea email-ului și complexitatea parolei în serviciul de auth.
* POST /login – primește LoginRequest (email și parolă), validează credențialele și returnează un LoginResponse ce conține tokenul JWT și eventual detalii utilizator (de ex. userId)[[40]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Fregister%20%E2%86%92%20LoginRequest%20%E2%86%92,int%20userId). Token-ul are o durată (ex. 2 ore) conform configurării.
* **PlantController** (/api/plants, autorizat [Authorize]):
* GET / – întoarce lista tuturor plantelor utilizatorului curent, ca List<PlantResponse>[[41]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantsController%20). Intern, folosește userId din token pentru a filtra plantele doar ale acelui user.
* GET /{plantId} – întoarce detaliile unei singure plante (DTO PlantResponse), dacă aparține utilizatorului curent[[42]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F%20%E2%86%92%20List). Dacă plantId nu există sau nu aparține user-ului, returnează 404 sau 403.
* POST / – adaugă o nouă plantă. Primește PlantCreateRequest cu datele plantei și returnează ID-ul noii plante create[[43]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F). Serverul va seta proprietăți implicite (ex. preia pragurile default din specie).
* PUT /{plantId} – actualizează o plantă existentă. Primește PlantUpdateRequest și returnează 204 No Content la succes[[44]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId). Se folosește pentru modificarea numelui, camerei, eventual a pragurilor personalizate sau actualizarea speciei/solului.
* DELETE /{plantId} – șterge planta specificată. Răspunde cu 204 No Content dacă a reușit[[45]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId). Va șterge cascada citirile și logurile asociate (configurarea DB poate face cascade sau se ocupă serviciul).
* PATCH /{plantId}/auto?enabled=true|false – activează sau dezactivează modul de udare automată pentru planta respectivă[[46]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DELETE%20%2F). Această acțiune actualizează câmpul AutoWateringEnabled din UserSettings sau alt mecanism de marcare (posibil direct la nivelul plantei dacă ar fi fost per plantă, însă conform modelului actual modul auto este la nivel utilizator global, deci s-ar putea aplica la toate plantele; totuși, putem interpreta acest endpoint ca setând un flag per plantă într-o implementare extinsă). Răspuns: 204 No Content.
* **SensorController** (/api/plants/{plantId}/sensor): gestionează datele de la senzori pentru o anumită plantă. Aceste endpoint-uri pot fi accesate de dispozitivul hardware (ESP32) **fără JWT de utilizator**, presupunând că securitatea se face prin cheia dispozitivului.
* POST /readings – folosit de dispozitivul ESP32 pentru a trimite o nouă citire de senzori. Body-ul cererii este un SensorReadingRequest JSON cu valorile măsurate[[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F). Răspuns: {status:"OK"} la succes[[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F). Serverul va salva citirea în DB și poate declanșa verificări (ex. dacă modul auto e pornit și pragurile sunt depășite, poate marca o udare necesară).
* GET /latest – întoarce ultimul SensorReadingDto disponibil pentru planta respectivă[[48]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK). Folosit de aplicația mobilă pentru a afișa valorile curente pe ecranul de detaliu plantă. Realizează o interogare în DB după cea mai recentă citire (eventual folosind indexul pe timestamp).
* GET /history?from={ISO}&to={ISO}&bucket={hour|day} – întoarce lista de citiri agregate pe ore sau zile în intervalul specificat[[49]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2Flatest%20%E2%86%92%20SensorReadingDto). Parametrii query from și to definesc perioada, iar bucket poate fi "hour" sau "day" pentru a specifica nivelul de agregare (ex. medii pe oră sau pe zi). Răspunsul este o listă de obiecte SensorReadingDto sau un DTO dedicat cu valorile medii/minime/maxime pe intervale. Această funcție permite afișarea graficelor istorice în aplicație (ex. grafic pe ultimele 24h sau pe ultima săptămână).
* **WateringController** (/api/plants/{plantId}/water, [Authorize] pentru comenzi manuale din aplicație):
* POST / – trimite o comandă de udare manuală. Body-ul conține un JSON de forma {"command": "ON", "durationSec": 5} sau {"command": "OFF"}[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). În mod tipic doar comanda "ON" cu o durată este relevantă (pump off se va face automat după durată). Răspunsul este un mesaj simplu, ex: { "message": "WATER ON" } care confirmă că serverul a înregistrat comanda[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). La primirea acestei comenzi, serverul va loga intenția (creează un WateringLog de tip manual) și va semnala către dispozitiv să pornească pompa (vezi secțiunea de firmware despre cum se transmite comanda).
* GET /status – permite dispozitivului să interogheze dacă există o comandă de udare în așteptare. Răspunsul poate conține un obiect { isOn: true/false, lastRunUtc: ISO8601 }[[50]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20body%3A%20%7B,WATER%20ON) indicând dacă pompa ar trebui pornită. Ideea este că dispozitivul va face polling pe acest endpoint și, dacă primește isOn:true împreună cu eventual timestamp sau durată, va executa udarea. După execuție, serverul poate reseta flag-ul. (Notă: Această abordare asigură că, chiar dacă dispozitivul nu a primit un push, va verifica periodic dacă trebuie să ude).
* **CalibrationController** (/api/plants/{plantId}/calibration): acest controller ajută la modul de calibrare a senzorilor.
* GET / – permite aplicației sau dispozitivului să afle dacă modul calibrare este activ. Răspuns: { "mode": "ON" | "OFF" }[[51]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). Când user-ul inițiază procesul de calibrare din app, serverul setează undeva un status ON (de ex. într-o variabilă sau DB).
* POST / – setează modul de calibrare. Body: { "mode": "ON" | "OFF" }[[52]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). Apelat de aplicația mobilă: la începutul calibrării trimite mode=ON, la final trimite mode=OFF. Răspuns: 204 No Content. Serverul poate reține starea modului (de ex. într-o variabilă static sau într-un câmp IsCalibrating al plantei). Dispozitivul ESP32 va face polling pe acest endpoint și dacă vede mode":"ON", își reduce intervalul de citire la 1s (intră în modul calibrare)[[23]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Loop%20logic%3A)[[53]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Calibration%3A%20dac%C4%83%20GET%20%2Fplants%2F,%E2%86%92%20readInterval%20%3D%201000%20ms).
* **StatsController** (/api/stats, [Authorize]): oferă statistici globale la nivel de utilizator sau aplicație.
* GET /daily?date={ISO} – întoarce statisticile zilei specificate (sau curente dacă nu se dă parametru). Răspunsul este un DailyStatsDto cu diverse informații agregate[[54]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20), de exemplu: număr de udări manuale/automate efectuate în acea zi, lista plantelor care au avut sub prag umiditatea (pentru eventuale alerte), etc.
* GET /weekly?weekStart={ISO} – statistici pentru săptămâna care începe la data specificată (sau săptămâna curentă implicit). Returnează un WeeklyStatsDto[[55]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20), de exemplu cu date cumulate pe 7 zile: total udări, top 3 plante cu cele mai multe udări, zilele când s-a udat cel mai mult, etc.

În cazul oricărei erori (de validare, de resurse inexistente, etc.), API-ul folosește un format unificat de răspuns de eroare, de exemplu:

{   
 "error": {   
 "code": "ValidationFailed" | "NotFound" | "Unauthorized",   
 "message": "Descriere eroare..."   
 }   
}

acompaniat de codul HTTP corespunzător (400, 404, 401, etc.)[[56]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=R%C4%83spuns%20eroare%20). Astfel, clientul poate interpreta ușor tipul erorii.

**Implementarea serviciilor (logică de business):** În spatele acestor endpoint-uri, se află serviciile implementate (în folderul *Services* din API). Aceste clase, cum ar fi AuthService, PlantService, SensorService, etc., sunt **injectate** în controlere prin constructor. De exemplu, PlantController primește un IPlantService prin constructor și îl stochează (\_plantService), apoi metodele controller-ului apelează metodele serviciului. Un exemplu simplu de metodă de controller:

[HttpGet]  
public async Task<ActionResult<IEnumerable<PlantResponse>>> GetAll()  
{  
 var plants = await \_plantService.GetAllAsync(User.Identity.Name);  
 return Ok(plants);  
}

*(Pseudo-codul de mai sus ilustrează obținerea tuturor plantelor pentru utilizatorul curent; User.Identity.Name ar putea reprezenta email-ul sau ID-ul utilizatorului extras din token).*

Serviciile implementează interfețele din Core. De exemplu, PlantService : IPlantService va folosi SmartGardenDbContext (injectat la rândul său) pentru a accesa tabelele. Ea va conține logica de **verificare a drepturilor** (ex: să se asigure că un user nu poate modifica planta altui user – de obicei verificând proprietatea UserId a plantei), precum și logica efectivă CRUD. Similar, SensorService la apelul de salvare a unei citiri va adăuga un obiect SensorReading în DbContext și îl va salva, iar la get latest va interoga contextul cu ordonare descrescătoare pe timestamp și cu AsNoTracking pentru performanță[[57]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday). WateringService implementează acțiunea de **udare manuală**: când primește comanda, va înregistra în baza de date un log de udare (poate cu un flag că e pending) și va marca undeva comanda pentru device. Alternativ, ar putea pune comanda într-o structură în memorie (o coadă) ce va fi citită la următorul polling al device-ului[[32]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday). De asemenea, dacă se cheamă SetAutoModeAsync, ar putea actualiza AutoWateringEnabled (global sau per plantă). StatsService va face calcule agregate, interogând DB cu grupări (posibil folosind LINQ sau SQL brut) pentru a genera DTO-urile de răspuns[[58]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringService%3A%20manual%20trigger%20,set%20auto%20flag).

În plus, se poate implementa un serviciu de tip background (HostedService) pentru alerte/monitorizare continuă. De exemplu, un AlertService ar putea rula periodic în fundal și verifica ultimele citiri: dacă găsește plante sub prag, poate crea evenimente de alertă (ce vor fi apoi preluate de NotificationService din aplicația mobilă). Blueprint-ul menționează că alertarea e opțională, dar că un astfel de job ar putea scrie entități de tip AlertEvents când pragurile sunt depășite[[34]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsService%3A%20agreg%C4%83ri%20,plante%2C%20sub%20prag%20azi%2C%20etc). În versiunea de bază, nu e implementat, s-a preferat ca aplicația mobilă să decidă notificarea (vezi secțiunea Notificări).

**Configurare globală și middlewares:** În Program.cs (sau Startup.cs) se realizează configurările necesare: - Se încarcă configurațiile (appsettings.json și appsettings.Development.json). De exemplu, un fișier de configurare de dev arată astfel[[59]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Config%20%28appsettings)[[60]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=true%20):

{  
 "ConnectionStrings": { "Default": "Data Source=smartgarden.db" },  
 "Jwt": { "Key": "dev-secret-change-me", "ExpiresMinutes": 120 },  
 "Irrigation": { "DefaultOnSeconds": 5, "DefaultPauseMinutes": 2, "DayOnly": true },  
 "Cors": { "Origins": [ "http://localhost", "http://10.0.2.2", "\*" ] }  
}

Aici, ConnectionStrings:Default specifică un SQLite file local pentru dezvoltare. Secțiunea Jwt conține cheia secretă (trebuie schimbată în producție) și expirarea token-ului. Irrigation definește parametri default pentru udare: durata unei porniri de pompă (5 secunde), pauza minimă între două porniri automate (2 minute), și DayOnly:true indicând că udarea automată se face doar pe timp de zi (evitând udarea noaptea, ca exemplu de opțiune)[[61]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,). Secțiunea Cors listează origin-urile permise pentru cereri cross-origin: de ex. http://localhost (pentru debug local web), http://10.0.2.2 (adresa emulatorului Android către host), și "\*" (orice, pentru teste)[[62]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120).

* **JWT**: se adaugă AddAuthentication().AddJwtBearer(...) cu parametrii din config (cheie, issuer etc.). Se setează și opțiunea ca la validarea token-ului să se extragă poate userId/email în claims.
* **CORS**: se configurează AddCors permițând origin-urile definite (și metodele GET, POST, etc.), pentru a permite aplicației mobile să apeleze API-ul. De asemenea, dispozitivul ESP32 nu are restrictii de CORS (nu fiind browser), dar e bine de permis toate origin-urile la nevoie.
* **Swagger**: în dezvoltare, se activează generarea de documentație Swagger/OpenAPI, eventual cu exemple predefinite pentru ușurință la testare (ex. un exemplu de payload la /sensor/readings).
* **Global Error Handling**: se poate introduce un middleware pentru a intercepta excepțiile și a le transforma în răspunsul JSON de eroare unificat (ca structura menționată mai sus). Astfel, serviciile pot lansa excepții custom (ex. NotFoundException, ValidationException) care să fie convertite la codul HTTP potrivit.

**Izolare multi-user:** Un aspect de securitate important implementat în API este că fiecare utilizator autenticat poate accesa doar resursele sale (plantele sale, citirile lor, log-urile lor). PlantService și celelalte servicii aplică *ownership check*-uri – practic, la fiecare operație, se verifică că Plant.UserId == currentUserId altfel se aruncă excepție sau se returnează 404[[63]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token). Acest lucru este menționat în definirea serviciilor (ex. "user isolation" la PlantService)[[64]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token).

**Testare integrată:** În cadrul API se recomandă scrierea unor teste de integrare (ex. folosind un DB in-memory sau SQLite în modul de test) pentru fluxurile principale: autentificare, operațiile de plante, trimiterea de date de senzori și comanda de udare[[65]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=JWT%20pe%20rutele%20protejate%2C%20CORS,OK%2C%20Swagger%20cu%20exemple). Aceste teste ar porni efectiv serverul în memorie și ar simula apeluri HTTP la endpoint-uri, verificând că se obțin răspunsurile așteptate și că starea în DB este corect actualizată.

În concluzie, SmartGarden.API este stratul care implementează **orchestrarea aplicației**: primește cereri de la client (mobil sau device), validează și procesează conform logicii de business, interacționează cu baza de date prin Data layer, și returnează răspunsuri HTTP standard. El gestionează și aspectele cross-cutting (autentificare/autorizare, CORS, logging). Practic, acest layer conține toată logica aplicației în afară de interfața utilizator și de citirea fizică a senzorilor.

### 3.4 Aplicația mobilă – SmartGarden.App (UI .NET MAUI)

Aplicația MAUI SmartGarden este **interfața cu utilizatorul**, permițând accesul la funcționalitățile sistemului într-un mod prietenos și portabil (Android, iOS, Windows). A fost dezvoltată folosind .NET MAUI cu modelul **MVVM (Model-View-ViewModel)**, facilitând separarea logicii de prezentare de UI. Structura proiectului este organizată pe foldere logice, după cum urmează[[66]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Structur%C4%83%20proiect%3A)[[67]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Views%2F%20%20%20%20,Dashboard%2C%20PlantDetail%2C%20AddPlant%2C%20Calibrate%2C%20Settings):

* **Models/** – conține modele folosite în UI, de obicei reflectând DTO-urile de la API. De exemplu, clase (sau record-uri) folosite pentru binding în UI: PlantModel, SensorReadingModel etc. Adesea, se pot reutiliza direct DTO-urile din Core (dacă sunt partajate ca library) sau se definesc echivalente. Exemple: un model de Plant pentru UI care combină PlantResponse de la server cu câmpuri derivate (ex. un șir format "Specie (Tip sol)").
* **Services/** – servicii specifice aplicației, pentru apeluri HTTP, stocare locală și notificări:
* ApiService – un serviciu generic pentru efectuarea de cereri HTTP (GET/POST/PUT/DELETE) către API. Implementează includerea automată a header-ului Authorization Bearer cu token-ul JWT salvat (de obicei stocat în Preferences["token"] pe device)[[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token). Poate folosi HttpClient sub capotă.
* AuthService – gestionează operațiile de login și înregistrare. Apelează ApiService (ex. POST /auth/login) cu credențialele, apoi salvează token-ul JWT primit și eventual userId în stocarea locală (Preferences) pentru sesiuni ulterioare[[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token).
* PlantService – oferă metode pentru acțiuni legate de plante: GetPlantsAsync() (GET /plants lista), CreatePlantAsync() (POST /plants), UpdatePlantAsync(), DeletePlantAsync(), ToggleAutoAsync() (apelează PATCH /auto pentru a comuta udarea automată) și WaterNowAsync() (apelează POST /water pentru a porni pompa)[[69]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20GetPlantsAsync%2C%20Create%2FUpdate%2FDelete%2C%20ToggleAutoAsync%2C%20WaterNowAsync). Intern, folosește ApiService să facă cererile și prelucrează răspunsurile.
* SensorService – metode pentru date senzori: GetLatestAsync(plantId) – apelează GET /sensor/latest, GetHistoryAsync(plantId, from, to, bucket) – apelează GET /sensor/history și returnează datele istorice pentru grafic[[70]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WaterNowAsync).
* NotificationService – un serviciu care se ocupă de programarea notificărilor locale pe dispozitiv. Folosește pachetul Plugin.Maui.Notifications pentru a programa notificări push locale[[71]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync). Metoda principală ar fi ScheduleAlert(title, message) care creează o notificare cu titlu și mesaj la momentul curent sau ulterior.
* StorageService (opțional) – un serviciu de stocare locală (SQLite) pentru caching offline. Poate salva lista de plante și ultimele N citiri pentru fiecare plantă, astfel încât aplicația să afișeze date chiar și fără conexiune sau să evite apelurile repetate la API[[72]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync). Acesta nu este esențial, dar util pentru extindere.
* **ViewModels/** – conține clasa ViewModel pentru fiecare ecran/pagină din aplicație:
* LoginViewModel – expune proprietăți Email, Password și comenzi pentru Login și NavigateToRegister. Folosește AuthService pentru login. Implementează validări (ex. asigură că email și parola nu sunt goale) și, la autentificare reușită, navighează la Dashboard[[73]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=LoginPage%20%2B%20LoginViewModel).
* DashboardViewModel – gestionează ecranul principal care listează plantele utilizatorului. Conține o listă observabilă de plante (ObservableCollection<PlantModel>), o comandă LoadCommand (apelată la inițializare și pull-to-refresh) care folosește PlantService.GetPlantsAsync pentru a încărca lista[[74]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DashboardPage%20%2B%20DashboardViewModel). De asemenea, expune acțiuni pentru interacțiuni rapide: de exemplu, butonul “Water now” de lângă fiecare plantă apelează o comandă ce invocă PlantService.WaterNowAsync(plantId)[[75]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=List%C4%83%20PlantResponse%20%2B%20Pull,la%2030%E2%80%9360s). Dashboardul poate permite filtrarea plantelor după cameră sau interior/exterior (conform mockup-ului, tab-uri All/Indoor/Outdoor și pe camere).
* PlantDetailViewModel – pentru ecranul de detalii a unei plante. Conține datele ultimei citiri (LatestReading), eventual pragurile și informațiile speciei. La inițializare, apelează SensorService.GetLatestAsync(plantId) pentru a obține ultima citire și SensorService.GetHistoryAsync(plantId, interval) pentru a obține datele istorice apoi populând un grafic[[76]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantDetailPage%20%2B%20PlantDetailViewModel). Utilizează biblioteca **Microcharts** pentru a genera graficul (de ex. serie de valori de umiditate pe ultimele 7 zile)[[76]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantDetailPage%20%2B%20PlantDetailViewModel). De asemenea, are un toggle pentru udare automată – proprietatea IsAutoEnabled legată la un Switch în UI, care la schimbare execută comanda ToggleAutoCommand ce cheamă PlantService.ToggleAutoAsync(plantId, newValue)[[77]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Latest%20%2B%20grafic%20History%20,cu%20Microcharts).
* AddPlantViewModel – gestionează formularul de adăugare a unei noi plante. Conține proprietăți legate de câmpurile formularului: nume plantei, specie selectată, tip de sol selectat (probabil populate prin listare din Species și SoilTypes preluate de la API), cameră (text sau select din opțiuni predefinite: living, dormitor etc.), opțiunea indoor/outdoor, data achiziției[[78]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AddPlantPage%20%2B%20AddPlantViewModel). Expune comanda AddPlantCommand care validează input-urile și apelează PlantService.CreatePlantAsync cu un PlantCreateRequest construit din aceste date. După creare, navighează înapoi la Dashboard.
* CalibrateViewModel – corespunde ecranului de calibrare senzori. Gestionează un **wizard în mai mulți pași**: calibrararea luminii, a senzorului de umiditate sol și a senzorului de nivel apă, în această ordine (conform mockup). Are starea internă pentru pasul curent și rezultate parțiale. La inițierea calibrării, apelează CalibrationController – trimite POST mode=ON pentru a notifica serverul și device-ul că se intră în modul calibrare[[79]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibratePage%20%2B%20CalibrateViewModel). Device-ul va începe să trimită citiri foarte frecvent (aprox. la 1s)[[23]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Loop%20logic%3A). Aplicația, în timpul calibrării, **polling**-uieste endpoint-ul /calibration sau /sensor/latest la fiecare 1 sec pentru a obține valori în timp real[[80]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibratePage%20%2B%20CalibrateViewModel). Practic, pe ecranul de calibrare se vor afișa instrucțiuni: ex. "Pune senzorul la întuneric și apasă Calibrate" – când utilizatorul apasă, se ia ultima citire de lumină ca valoare "Dark". Apoi instructează "Pune senzorul la lumină medie..." etc. Similar pentru sol uscat/umed și rezervor gol/plin. La fiecare astfel de sub-pas, aplicația poate folosi ultima citire primită de la device pentru a stabili valoarea de referință și eventual trimite la server sau direct la device (posibil calibrările pot fi stocate pe device). După finalizarea tuturor pașilor, se trimite POST mode=OFF la server pentru a ieși din modul calibrare[[81]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auto%3A%20activezi%20Auto%20%E2%86%92%20la,min%2C%20apare%20WateringLog%20auto). Device-ul revine la intervalul normal de citire de 15 min. CalibrateViewModel se ocupă deci de coordonarea acestui proces și de afișarea progresului (indicatori Completed/Calibrate pentru fiecare nivel, exact ca în mockup).
* SettingsViewModel – pentru ecranul de setări, cu opțiuni precum tema (Dark/Light mode), notificări (activare/dezactivare notificări locale), și logout. Logout ar șterge token-ul JWT din stocare și navighează la ecranul de login[[82]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SettingsPage). Poate conține și opțiuni legate de contul utilizatorului (nume, email) sau altele.
* **Views/** – paginile XAML corespondente ViewModel-urilor de mai sus: LoginPage.xaml, DashboardPage.xaml, PlantDetailPage.xaml, AddPlantPage.xaml, CalibratePage.xaml, SettingsPage.xaml[[83]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ViewModels%2F%20%20%20%20,DashboardVM%2C%20PlantDetailVM%2C%20AddPlantVM%2C%20CalibrateVM%2C%20SettingsVM). Fiecare are în resources definit ViewModel-ul ca BindingContext (folosind perhaps CommunityToolkit markup). De exemplu:

<!-- Exemplu de definire ViewModel în XAML (pseudo) -->  
<ContentPage ...  
 xmlns:vm="clr-namespace:SmartGarden.App.ViewModels" >  
 <ContentPage.BindingContext>  
 <vm:PlantDetailViewModel />  
 </ContentPage.BindingContext>  
 <!-- UI elements bind to properties of PlantDetailViewModel -->  
</ContentPage>

*DashboardPage* afișează o listă (CollectionView) de plante, cu Template ce include numele plantei, specie, eventual o pictogramă indoor/outdoor, procentul de umiditate a solului și un buton "Water Now" (Udă acum) care leagă comanda din VM[[74]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DashboardPage%20%2B%20DashboardViewModel)[[75]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=List%C4%83%20PlantResponse%20%2B%20Pull,la%2030%E2%80%9360s). *PlantDetailPage* afișează valorile curente ale senzorilor (ex. grafic cu evoluția lor, slider sau progress bars pentru ultima valoare) și un toggle pentru auto-irrigation[[76]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantDetailPage%20%2B%20PlantDetailViewModel). *CalibratePage* implementează interfața wizard cu pașii succesivi de calibrare, butoane Next/Calibrate, animații (ex. countdown de 5 secunde) și feedback "Completed" pe măsură ce se realizează fiecare punct din calibrare (conform mockup-ul din PDF). *AddPlantPage* conține un formular cu picker pentru specie (populat cu lista de Species din DB), picker pentru tip de sol, date picker pentru data achiziției, radio sau segmente control pentru indoor/outdoor, și un field text pentru numele plantei și camera. *SettingsPage* oferă toggle pentru dark mode (poate direct legat la Application.Current.UserAppTheme), un toggle sau picker pentru notificări (sau trimiterea la setările sistemului) și buton de logout.

* **Utils/** – utilitare comune. De exemplu, clase helper ca Constants (cu URL-ul API de bază, chei pentru Preferences etc.), Converters (IValueConverters pentru binding – de ex. un converter care dintr-un bool IsOutdoor returnează string "Outdoor"/"Indoor", sau care formatează un datetime la format prietenos), Validator (dacă există reguli generale de validare, ex. email regex). De asemenea, pot exista extinderi pentru servicii.
* **App.xaml și AppShell.xaml** – definesc resursele globale (de ex. dicționare de stiluri, culori) și structura de navigație Shell. S-a folosit .NET MAUI Shell pentru navigare, definind rutele pentru pagini:
* Dacă utilizatorul este neautentificat (nu are token sau expiră), se navighează către pagina de Login (//Login route)[[84]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Naviga%C8%9Bie%20).
* După login cu succes, se navighează către //Dashboard (pagina principală)[[84]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Naviga%C8%9Bie%20). Shell-ul poate defini tab-uri sau flyout meniu; în acest proiect se poate ca Dashboard să fie implicit, iar Settings să fie accesibil dintr-un tab sau meniul profil.
* AddPlantPage, PlantDetailPage, CalibratePage sunt invocate via navigare din Dashboard (ex. un buton + pentru add, un tap pe o plantă pentru detalii, un buton de calibrare în pagina de detalii care navighează la wizard).
* **MauiProgram.cs** – configurarea aplicației MAUI. Aici se initializează servicii și se înregistrează în container-ul DI: de exemplu services.AddSingleton<ApiService>(), services.AddSingleton<AuthService>(), etc. (Singleton sau Transient depinde, dar majoritatea pot fi Singletons în contextul aplicației mobile). Se configurează platform-specific settings (ex. cerințe de permisii). După care se build-uiește app-ul.

**Comunicarea cu backend-ul:** Aplicația folosește HttpClient (prin ApiService) pentru a comunica cu API-ul. De fiecare dată când face o cerere, include token-ul JWT dacă există. Spre exemplu, ApiService poate arăta astfel (pseudo-cod simplificat):

public class ApiService {  
 private readonly HttpClient \_http = new HttpClient() { BaseAddress = new Uri(Constants.ApiBaseUrl) };  
  
 public async Task<T> GetAsync<T>(string uri) {  
 AttachAuthHeader();  
 var response = await \_http.GetAsync(uri);  
 response.EnsureSuccessStatusCode();  
 return JsonSerializer.Deserialize<T>(await response.Content.ReadAsStringAsync());  
 }  
 // similar PostAsync, etc.  
 private void AttachAuthHeader() {  
 var token = Preferences.Get("token", null);  
 \_http.DefaultRequestHeaders.Authorization = token != null ?   
 new AuthenticationHeaderValue("Bearer", token) : null;  
 }  
}

Astfel, serviciile specifice (AuthService, PlantService etc.) vor utiliza aceste metode pentru a apela endpoint-urile. De exemplu, AuthService.LoginAsync(email,pwd) va face var resp = await \_api.PostAsync<LoginResponse>("/auth/login", new LoginRequest(email,pwd)); și dacă resp.token există, îl va salva în Preferences["token"] și Preferences["userId"] pentru folosire ulterioară.

**Notificări locale:** Aplicația implementează notificări locale pentru a alerta utilizatorul când apar situații critice – de exemplu, **umiditatea solului sub pragul minim**, **temperatura peste pragul maxim optim** pentru specie, sau **rezervorul de apă aproape gol**. Conform cerințelor, aceste alerte se realizează ca notificări locale programate[[85]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Login%20func%C8%9Bional%2C%20liste%20plante%2C%20udare,manual%C4%83%2C%20toggle%20auto). Cel mai probabil, la fiecare actualizare de date (de exemplu, când primește un nou SensorReading prin refresh manual sau push), aplicația verifică valorile: dacă SoilMoisture < DefaultSoilMoistMin al speciei sau sub un prag absolut, atunci va chema NotificationService.ScheduleAlert("Udat necesar", $"Planta {plant.Name} are solul uscat!"). Similar, dacă temperatura depășește maxima recomandată, sau dacă ultimul WaterLevel indică rezervor gol, se programează notificări. Aceste notificări sunt locale pe dispozitiv (nu push de la server, deci user-ul le vede imediat). Pe Android, se definește un canal de notificări la startup[[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A), iar pe Windows se asigură capacitatea în manifest.

**Experiența utilizator și UX:** Aplicația pune accent pe simplitate: - După login, utilizatorul vede o listă cu plantele sale și starea lor sumară (procent umiditate, eventual icon de avertizare dacă ceva nu e în regulă – ex. un semn de exclamare lângă “Water tank” dacă e 10% plin, conform mockup "Please fill the water tank!")[[87][88]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Ficus%20lyrata). De aici poate adăuga plante noi sau selecta una pentru detalii. - În ecranul de detalii, utilizatorul poate vizualiza valorile curente și istoricul. Are posibilitatea de a activa/dezactiva udarea automată pentru acea plantă cu un simplu comutator și poate iniția udare imediată cu un buton. - Procesul de **adăugare plantă** e asistat cu pași (conform mockup-ului de început – conectează dispozitivul la WiFi, calibrează senzori, apoi adaugă detalii plantă)[[89]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=%28make%20sure%20it%E2%80%99s%202)[[90]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Hello%2C%20). În aplicație, aceste etape pot fi fie integrate într-un singur flux (ex. wizard la prima utilizare) fie separate: se presupune că utilizatorul mai întâi configurează hardware-ul (conexiune WiFi prin altă metodă, apoi calibrarea prin aplicație), apoi completează formularul plantei. - **Calibrarea senzorilor** este concepută pentru a fi cât mai clară: utilizatorul este ghidat cu instrucțiuni text pas cu pas (vezi PDF: ex. “Put the sensor in Wet Soil, Wait for 5 seconds, then press Calibrate”)[[91]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Soil%20Sensor)[[92]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Put%20the%20sensor%20in). Aplicația se ocupă să culeagă valorile în fundal și să afișeze "Completed" când un pas s-a înregistrat. Dacă ceva nu merge (ex. device nu răspunde), utilizatorul poate anula calibrarea.

**Permisiuni și platforme suportate:** Proiectul țintește Android (net8.0-android) și Windows (net8.0-windows10.0.19041.0) în mod implicit, fiind posibilă extensia pe iOS[[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A). Sunt setate permisiuni: - Android: permisiunea de Internet (în AndroidManifest, necesară pentru a putea face cereri HTTP) și configurarea canalului de notificări (Plugin.Maui.Notifications solicită definirea acestuia)[[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A). - iOS: ar necesita permisiuni de rețea și notificări, similar, deși iOS e opțional. - Windows: in Package.appxmanifest se asigură că are capabilitatea InternetClient activată[[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A).

**Design adaptiv:** Se asigură că UI este responsiv, folosind layout-uri adaptabile la diferite dimensiuni de ecran (telefon, tabletă). De asemenea, se respectă tema setată (light/dark) și se oferă user-ului opțiunea de a schimba (dark mode toggle in Settings). Se gestionează stările de încărcare și erori – de exemplu, în ViewModel-uri se poate avea un bool IsBusy legat la un ActivityIndicator în UI, iar eventualele erori la apeluri API afișate prin mesaje de alertă (DisplayAlert) sau prin label-uri vizibile când apare eroarea.

**Flux offline (opțional):** StorageService menționat ar putea salva date în SQLite local, însă implementarea de bază poate fi fără caching. În absența conexiunii la internet, aplicația ar putea încă afișa ultimele date sincronizate și notifica user-ul că este offline.

În concluzie, aplicația SmartGarden MAUI oferă **toate funcționalitățile front-end** necesare: autentificare, listare și management al plantelor, vizualizarea datelor de la senzori în timp real și grafic, controlul udării manuale/automate, calibrarea inițială a dispozitivului, notificări privind starea plantelor, totul într-o interfață modernă ușor de folosit.

## 4. Fluxuri funcționale complete ale sistemului

În această secțiune vom descrie cum funcționează cap-coadă principalele fluxuri în SmartGarden, integrând interacțiunile dintre aplicația mobilă, server (API) și dispozitivul IoT. Aceste fluxuri acoperă calibrările de senzori, udarea automată, transmiterea datelor de la senzori și mecanismul de notificări – toate fiind elemente cheie ale aplicației.

### 4.1 Fluxul de calibrare a senzorilor

**Scop:** Calibrarea permite dispozitivului să obțină valorile de referință necesare pentru a interpreta corect citirile brute ale senzorilor de lumină, umiditate sol și nivel de apă. Acest proces se desfășoară de obicei la instalarea inițială a dispozitivului pentru o plantă.

**Pași detaliați:**

1. **Inițierea calibrării (aplicație mobilă):** Utilizatorul, după ce a conectat dispozitivul ESP32 la rețeaua Wi-Fi (probabil printr-un alt proces sau implicit), deschide aplicația și selectează opțiunea de calibrare. În UI, se prezintă un “wizard” care explică pașii (de ex. „Calibrează senzorii de lumină” ca prim pas)[[93]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors). Când utilizatorul apasă “Start calibration”, aplicația face o cerere POST /api/plants/{id}/calibration cu body { "mode": "ON" }[[94]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). Serverul setează modul de calibrare la ON pentru acea plantă (eventual stochează într-o variabilă static sau DB această stare).
2. **Dispozitivul intră în modul calibrare:** Firmware-ul ESP32 are în bucla principală o verificare: la fiecare iterație, face un GET /api/plants/{id}/calibration. Dacă răspunsul conține "mode":"ON", atunci dispozitivul reduce readInterval la 1000ms (1 secundă)[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min)[[53]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Calibration%3A%20dac%C4%83%20GET%20%2Fplants%2F,%E2%86%92%20readInterval%20%3D%201000%20ms). Altfel, intervalul rămâne cel normal (15 minute). În momentul în care serverul a setat modul ON, dispozitivul va detecta în maxim 1 secundă și va începe **să citească senzorii foarte frecvent** (aprox. o dată pe secundă) și să trimită date.
3. **Colectarea datelor în timp real:** Cu modul calibrare activ, ESP32 trimite periodic (la ~1s) cereri POST /api/plants/{id}/sensor/readings conținând valorile senzorilor în timp real[[96]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Send%20readings%3A%20POST%20%2Fplants%2F,payload)[[97]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,calibrare%20e%20aplicat%C4%83%20la%20device). Aceste valori brute sunt recepționate de server și stocate fie complet (dacă se decide să se rețină citirile calibrării) fie temporar. Practic, serverul poate stoca în memorie ultimele citiri pentru a le expune aplicației mobile.
4. **Ghidarea utilizatorului prin pașii de calibrare:** Aplicația mobilă, odată calibrerea pornită, trece prin diferite etape pe ecran:
5. **Calibrare luminozitate:** Va afișa instrucțiunea “Calibrating the Light Sensor” și subpașii: Întuneric, Lumină medie, Lumină puternică indirectă, Lumină directă, etc. (conform necesarului senzorului de lumină)[[98]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors)[[99]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Bright%20Direct%20Calibrate). Pentru primul subpas (întuneric complet), se cere utilizatorului să acopere senzorul sau să-l pună în beznă și să apese “Calibrate” când e gata. Când utilizatorul apasă, aplicația ia valoarea de lumină curentă de la ultima citire primită de la device (prin eventual un GET /sensor/latest pe care aplicația îl poate chema în fiecare secundă pentru update UI). Acea valoare este înregistrată ca nivel “Dark”. Aplicația trimite eventual la server sau reține local. Subpasul este marcat “Completed” în UI[[100]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Dark%20Completed).
6. Se trece la următorul subpas: utilizatorul este instruit “Pune senzorul în lumină medie/indirectă” (ex. într-o cameră luminată moderat) și să apese calibrate. Se procedează similar: aplicația preia valoarea actuală (de ~1 secundă) și o marchează ca “Medium Indirect”. Apoi “Bright Indirect” – lumina puternică dar nu direct în senzor (umbră luminoasă), apoi “Bright Direct” – expus direct la soare sau lumină foarte puternică. Pentru fiecare, aplicația are un countdown de ~5 secunde pentru a stabiliza senzorul[[101]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Bright%20Direct%20Calibrate), apoi permite calibrate.
7. Senzorul de lumină BH1750, dacă e folosit, măsoară în lux. Aceste patru-cinci valori calibrate (dark, low, medium, bright indirect, bright direct) pot fi folosite apoi de aplicație sau device pentru a categorisi condițiile de iluminare (de ex. device-ul poate converti un lux măsurat într-una din etichetele acestea pe baza acestor praguri).
8. **Calibrare umiditate sol:** După finalizarea luminii, se trece la senzorul de umiditate a solului. UI afișează “Calibrating the Soil Sensor” cu doi pași: “Dry Soil” și “Wet Soil”[[102][91]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Soil%20Sensor). Utilizatorul este instruit să scoată senzorul din pământ și să îl lase în aer (uscat) pentru calibrul de sol uscat. Aplicația poate aștepta câteva secunde pentru stabilizare și apoi user apasă calibrate – se ia valoarea curentă de umiditate (practic tensiunea analogică maximă în aer, considerată 0% umiditate)[[103]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Dry%20Soil%20Completed). Apoi pentru calibrul “Wet Soil”, utilizatorul introduce senzorul în pământ foarte umed (sau apă) – așteaptă 5 secunde[[104]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Wet%20Soil%20Calibrate) și apasă calibrate. Valoarea citită (curentă) devine referința pentru 100% umiditate. Astfel se obțin cele două capete (uscat/umed) pentru senzorul de umiditate, folosite ulterior la calculul procentual al umidității solului.
9. **Calibrare nivel apă rezervor:** Următorul pas (dacă există un senzor de nivel apă, ex. ultrasonic HC-SR04) – UI afișează “Calibrating the Water Sensor”. Pași: “Min” (rezervor gol) și “Max” (rezervor plin)[[105][106]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Water%20SensorCalibrating%20the,Water%20Sensor). Pentru calibrul minim, utilizatorul golește recipientul de apă sau scoate senzorul (dacă e într-o anumită poziție) – app așteaptă 5 sec și înregistrează valoarea distanței maxime (senzorul ultrasonic va citi o distanță mare când e gol). Pentru calibrul maxim, utilizatorul umple rezervorul la nivel optim – se ia valoarea distanței minime (senzorul vede apa aproape)[[107]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Min%20Completed)[[108]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Max%20Calibrate). Astfel se pot converti ulterior citirile in % nivel apă.
10. (Calibrarea altor senzori, ex. aer calitate, nu e necesară în mod obișnuit, deci se sare.)
11. **Confirmarea calibrării și salvarea datelor:** După parcurgerea tuturor subpașilor, aplicația afișează un ecran final “Setup Complete – Now let’s add a plant!”[[109]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Setup%20Complete%20Now%2C%20let%E2%80%99s%20add,a%20plant), indicând că procesul de calibrare s-a terminat. Se face acum apelul POST /calibration { "mode": "OFF" } la API[[81]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auto%3A%20activezi%20Auto%20%E2%86%92%20la,min%2C%20apare%20WateringLog%20auto). Serverul marchează calibration=OFF pentru plantă. Device-ul detectează la următorul GET /calibration că modul a revenit la OFF și astfel setează intervalul de citire înapoi la normal (15 minute)[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min)[[53]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Calibration%3A%20dac%C4%83%20GET%20%2Fplants%2F,%E2%86%92%20readInterval%20%3D%201000%20ms).

Valorile calibrate obținute trebuie stocate. Aici sunt două abordări: - **În dispozitiv:** Cel mai eficient este ca ESP32 să își stocheze în memoria nevolatilă (NVS) valorile calibrate (de ex. lux\_dark, lux\_bright, soil\_dry\_adc, soil\_wet\_adc, dist\_empty, dist\_full). Acestea pot fi transmise de aplicație către device. Cum? Fie în timpul calibrării, serverul la fiecare calibrate ar putea trimite un răspuns special la device cu acele valori – complicat. Mai simplu: la final, aplicația ar putea face un *alt endpoint* de calibrare unde să trimită valorile calibrate, iar la următoarea citire device-ul să le preia. De exemplu, POST /api/plants/{id}/calibration/values cu payload valorile, iar device-ul periodic în modul calibrat să ceară și acele valori. - **Pe server:** Alternativ, se pot salva în DB (ex. în entitatea Plant sau separat) și la fiecare citire serverul să convertească valorile raw în procente folosind aceste referințe. Dar asta înseamnă trafic raw plus logică server, pe când device-ul ar putea direct să trimită procente dacă știe calibrările. Blueprint-ul nu a intrat în aceste detalii, însă dat fiind contextul, cel mai probabil calibrările se folosesc direct pe device.

Indiferent de implementare, la final device-ul cunoaște valorile de referință și de acum înainte va putea raporta, de exemplu, *umiditatea solului în procente* (calculată din citirea analogică între calibru uscat=0% și umed=100%), lumina poate fi clasificată sau trimisă ca lux brut (și interpretată în UI după praguri calibrate), nivelul apei trimis ca % sau cm rămași.

1. **Utilizarea calibrării:** După calibrare, utilizatorul va trece la adăugarea efectivă a plantei în aplicație (dacă era în onboarding) sau doar încheie procesul. Dispozitivul acum funcționează normal, trimițând la fiecare 15 minute date cu acuratețe calibrată. Astfel, de exemplu, când solul e foarte uscat, device-ul va ști că valoarea analogică corespunde unui procent mic și va putea lua decizia de udare automată corect.

Resumând, fluxul de calibrare implică o **coordonare strânsă între aplicație și dispozitiv** prin intermediul API-ului: aplicația instruiește serverul să pornească modul calibrat, device-ul reacționează accelerând citirile, utilizatorul parcurge pașii și aplicația preia valorile de la device pentru a le considera referințe, apoi oprește modul calibrat. Aceasta asigură că **fiecare senzor este adaptat la condițiile concrete** (de mediu și hardware) pentru planta respectivă, ceea ce va face ca datele ulterioare să fie mult mai relevante.

### 4.2 Fluxul de udare automată a plantei

**Scop:** Udarea automată permite sistemului să ude planta fără intervenția utilizatorului, atunci când condițiile indică necesitatea – în special când umiditatea solului scade sub un prag minim. Acest flux implică decizia automată și acțiunea de pornire a pompei de către dispozitiv, la momentul potrivit.

**Condiții prealabile:** Utilizatorul a activat modul *Auto-irrigation* pentru planta respectivă fie la nivel global, fie per plantă (în funcție de implementare). În modelul dat, există UserSettings.AutoWateringEnabled global pentru user[[110]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=UserSettings%20,irrigation%2C%20thresholds%2C%20read%20interval), deci presupunem că dacă acest flag e true, toate plantele utilizatorului se udă automat conform pragurilor. (O extindere firească e să fie configurabil pe fiecare plantă; dacă s-ar dori asta, am putea interpreta că în UI se memorează opțiunea pe plantă și se salvează tot în UserSettings ca prag – o abordare mai simplă consideră același prag pentru toate plantele unui user).

**Pași flux:**

1. **Activarea modului automat:** În aplicația mobilă, utilizatorul accesează detaliile plantei și comută switch-ul “Automatic watering” la ON[[77]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Latest%20%2B%20grafic%20History%20,cu%20Microcharts). Aceasta declanșează un apel la API: PATCH /api/plants/{id}/auto?enabled=true[[46]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DELETE%20%2F). Serverul procesează: setează AutoWateringEnabled=true în UserSettings sau stochează altfel preferința. De asemenea, poate reține și pragul de umiditate folosit. Pragul poate proveni din UserSettings.SoilMoistThreshold (ex. 30%)【27†】 sau din valorile implicite ale speciei plantei (ex. dacă specia are DefaultSoilMoistMin=30%, acela e pragul). Să presupunem că pragul e 30%. Serverul răspunde 204, iar în aplicație switch-ul rămâne On.
2. **Monitorizarea umidității solului:** Dispozitivul ESP32 continuă să trimită citiri periodice la fiecare 15 minute (sau intervalul specificat) cu soilMoisture (deja calibrat, să zicem trimite procent). Aceste citiri ajung la API prin POST /sensor/readings. Cum se decide udarea automată? Sunt două posibile locus-uri de decizie:
3. **Pe server (logică centralizată):** Când serverul primește o nouă citire de la device, verifică dacă AutoWateringEnabled este true și dacă valoarea soilMoisture este sub prag (ex. < 30%). Dacă da, trebuie să declanșeze o udare. Dar serverul nu poate porni direct pompa, trebuie să comunice cu device-ul. Două moduri: a) Răspunsul la POST /sensor/readings ar putea conține un semnal (ex. un câmp extra "waterCommand": {"on": true, "duration":5})[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate). Blueprint-ul a menționat această variantă însă a notat că o soluție mai simplă e prin polling separat[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate)[[112]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20poll%20la%20GET%20%2Fplants%2F). b) Să seteze un flag ce va fi prins la următorul poll de status al device-ului.
4. **Pe dispozitiv (logică distribuită):** Alternativ, device-ul însuși poate decide local. Având pragul de umiditate și starea auto (trebuie însă să le cunoască), ar putea verifica la fiecare citire: *dacă AutoWatering e activ și moisture% < threshold, pornește pompa.* Aceasta are avantajul reacției imediate, fără comunicare suplimentară. Dar necesită ca device-ul să știe pragul și starea auto. Ar putea primi aceste informații prin API la nevoie: de exemplu, la fiecare GET /water/status sau chiar la POST /readings serverul i le poate trimite. Sau, mai simplu, device-ul poate descărca la pornire configurațiile (ex. un GET /plants/{id} care conține pragul și autoEnabled). Vom considera totuși fluxul condus de server (pentru că acolo e sursa de adevăr a datelor).

În implementarea noastră, vom merge pe varianta logicii centrale (server), deoarece modul în care blueprint-ul definea WateringService și Alerts sugerează că serverul are un rol în a detecta pragurile.

1. **Declanșarea udării automate (server):** Așadar, serverul primește o citire sub prag. În SensorService.SaveReading, după ce salvează datele, adaugă:

* if(userSettings.AutoWateringEnabled && reading.SoilMoisture < userSettings.SoilMoistThreshold) {  
   \_wateringService.QueueAutoWater(reading.PlantId);  
  }
* QueueAutoWater poate face două lucruri: creează un WateringLog cu Mode=Auto și timestamp acum (indicând că ar trebui udat), și setează undeva o comandă pentru device. De exemplu, WateringLog se poate salva imediat cu DurationSec = DefaultOnSeconds (din config, ex. 5 sec)[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120) și poate marca că e "pending execution". Concomitent, WateringService ar putea seta un flag in-memory (ex. intr-un dictionar static pendingWater[plantId] = 5 sec).

1. **Dispozitivul află de comanda de udare:** După ce a trimis citirea, modul de comunicare ales intră în joc:
2. Dacă s-ar folosi răspuns imediat: serverul ar răspunde la POST /readings nu doar cu status:"OK", ci cu ceva precum {"status":"OK","water":"ON","duration":5}. Firmware-ul ar trebui să parseze răspunsul și dacă vede water=ON, să pornească pompa imediat pentru 5 secunde.
3. Dacă se folosește polling: Dispozitivul, la scurt timp după ce a trimis citirea (sau chiar imediat după), face un GET /plants/{id}/water/status. Serverul, deoarece a detectat sub prag și a pregătit logul, va răspunde acum cu { "isOn": true, "lastRunUtc": "<timestamp>" } sau eventual direct cu durata de pornire. De exemplu, ar putea răspunde { "isOn": true, "duration": 5 }. (Blueprint-ul arată un exemplu simplu cu isOn boolean și lastRun, posibil gândit mai mult pentru manual, dar putem extinde pentru auto)[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate)[[112]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20poll%20la%20GET%20%2Fplants%2F).
4. Indiferent de metodă, dispozitivul primește semnalul că trebuie să ude.
5. **Execuția udării pe dispozitiv:** La primirea comenzii (fie via răspuns direct, fie via poll), firmware-ul activează pompa. De exemplu:

* digitalWrite(PUMP\_PIN, HIGH);  
  delay(durationSec \* 1000);  
  digitalWrite(PUMP\_PIN, LOW);
* Astfel, pompa funcționează pentru durata specificată (default 5 secunde, se poate ajusta în config sau de către user eventual)[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120). După udare, dispozitivul poate imediat să trimită o citire nouă (umiditatea solului va crește, deci ar fi util să actualizeze). Dacă se respectă strict intervalul de 15 min, poate nu imediat; dar ar fi logic după udare să facă o citire specială post-udare, sau cel puțin la următorul interval vom vedea efectul.

1. **Logarea și prevenirea udărilor repetate:** Pe server, deja avem un WateringLog salvat pentru acest eveniment (momentul declanșării). Dacă dispozitivul ar trimite iar citiri imediate ce încă sunt sub prag (poate solul nu s-a umezit complet), riscăm să declanșăm iar udare. Pentru a evita supra-udarea, se folosesc parametrii precum DefaultPauseMinutes (ex. 2 minute pauză)[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120) și eventual DayOnly. WateringService/QueueAutoWater poate verifica când a fost ultima udare automată a acelei plante și dacă e prea recent (ex. <2 min), să nu pornească din nou. De asemenea, DayOnly:true înseamnă că dacă e noapte (poate definit ca orele 22-6), să nu ude chiar dacă pragul e sub (plantele oricum nu se udă de obicei noaptea). Aceste condiții sunt aplicate înainte de trimiterea comenzii.
2. **Notificarea utilizatorului (opțional):** Dacă se dorește, se poate trimite o notificare către utilizator (prin aplicație) atunci când are loc o udare automată – de exemplu: "Planta X a fost udată automat (5s) deoarece solul era uscat." Acest lucru se poate realiza în două moduri:
3. Serverul creează un *AlertEvent* de tip "AutoWaterExecuted" cu detalii, și aplicația la refresh vede asta și programează o notificare locală.
4. Mai simplu: Aplicația la următoarea sincronizare va vedea un nou log de udare (în istoricul plantei) marcat auto – și poate notifica atunci. Sau chiar imediat ce primește citiri (dacă ar fi conexiune persistentă, dar nu e).
5. Alternativ, dacă s-ar folosi push notifications de la server (nu menționat explicit), serverul ar putea trimite direct o notificare push. Dar în blueprint se menționează doar notificări locale generate de app.
6. **Rezultat:** Planta a fost udată automat fără intervenție. În baza de date, avem un WateringLog (Mode=Auto) ce apare eventual în statisticile zilnice/săptămânale (ex. contorizat la auto waterings)[[114]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringService%3A%20manual%20trigger%20,set%20auto%20flag). La următoarea citire, umiditatea solului ar trebui să crească peste prag, astfel fluxul auto se oprește până când din nou scade.

**Observație:** Unde se stochează pragul de umiditate folosit? Având UserSettings.SoilMoistThreshold, se consideră acela pentru toate plantele utilizatorului. Ar avea sens ca specii diferite să aibă nevoi diferite, dar pentru simplitate studentul a ales un prag global sau l-a lăsat fix (30% default)[[115]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Praguri%20default%20udare%20%28ex.%20SoilMoistureMin%3D30,Max%3D60). În practică, s-ar putea îmbunătăți per plantă. Dacă s-ar dori, Plant ar putea avea câmpuri MinMoistureThreshold și MaxMoistureThreshold personalizabile (inițializate cu valorile din Species default). Blueprint-ul a menționat “praguri default udare ex. 30-60%” ca decizii de produs[[116]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Decizii%20de%20produs%3A), deci credem că 30% e folosit ca prag minim.

**Edge cases:** - Dacă rezervorul de apă e gol, device-ul tot ar încerca să ude, dar pompa poate merge în gol. Ideal, device-ul ar trebui să știe nivelul apei și să nu pornească pompa dacă nivelul e foarte scăzut (putem implementa: dacă waterLevelCm indică rezervor gol, ignoră comanda). - Fail-safe: blueprint-ul menționează un *hard limit* de 60s pentru pompă să evite inundarea[[117]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20prime%C8%99te%20flag%20%C3%AEn%20r%C4%83spunsul,durationSec%20apoi%20LOW%3B%20log%20local) – device-ul nu va menține pompa pornită peste acest interval nici dacă primește altfel (ex. dacă dintr-o eroare ar rămâne ON). - De asemenea, după udarea auto, solul poate avea nevoie de câteva secunde/minute să se îmbibe; un algoritm mai complex ar putea porni pompa în pulsuri: X secunde on, așteaptă Y minute, măsoară din nou, și repetă dacă încă sub prag. Parametrii DefaultPauseMinutes (2 min) sugerează tocmai asta[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120). Dar cred că implementarea minimală e o singură udare scurtă și atat, urmând ca la următorul ciclu de 15 min să reevalueze.

Fluxul de udare automată evidențiază cu adevărat valoarea sistemului SmartGarden: **monitorizare continuă și reacție automată**. Practic, utilizatorul, după ce a activat modul auto, se poate baza că planta sa va fi udată la nevoie, în limita configurărilor. El poate vedea în aplicație când s-a udat (prin loguri și statistici) și primește alerte dacă ceva nu a mers (de ex. rezervor gol – discutat la notificări). Aceasta reduce riscul ca planta să se usuce din neatenție.

### 4.3 Fluxul de transmitere a datelor de la senzori (telemetrie plantă)

**Scop:** Colectarea datelor de mediu de la fiecare plantă și trimiterea lor către server pentru stocare și afișare către utilizator. Acesta este un flux periodic, de telemetrie, ce funcționează continuu pe durata de viață a sistemului.

**Pași flux:**

1. **Citirea periodică a senzorilor (dispozitiv):** În modul normal (nu calibrare), codul firmware-ului execută o buclă în care, la fiecare readInterval milisecunde (valoare default 900.000ms, adică 15 minute)[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min), pornește secvența de citire:
2. Colectează valorile brute de la toți senzorii conectați:
   * Senzor umiditate sol (ex. un divizor rezistiv cu electrod, citit pe pin analogic) – returnează un integer între 0-4095 (pe 12 biți ADC).
   * Senzor temperatură/umiditate aer (SHT20 sau DHT22 etc., citit pe magistrala I2C sau one-wire) – returnează temperatură în °C și umiditate relativă %.
   * Senzor de lumină (BH1750 pe I2C) – returnează lumina ambientală în lux.
   * Senzor calitate aer (MQ-135 analogic) – returnează un analog value corelat cu concentrația de gaz (CO2 echivalent sau altceva).
   * Senzor nivel apă (HC-SR04 ultrasonic) – se măsoară timpul de ecou pentru a calcula distanța până la suprafața apei, de unde se deduce nivelul de umplere (ex. dacă distanța e lungă, apa e jos = rezervor gol).
3. Aplică conversiile necesare: dacă a fost calibrat:
   * Umiditatea solului: transformă valoarea ADC în procent folosind valorile calibrate dry=0% și wet=100%. De ex:
   * int raw = analogRead(SOIL\_PIN);  
     int percent = map(raw, soilDryCal, soilWetCal, 0, 100);  
     percent = constrain(percent, 0, 100);
   * Lumina: BH1750 deja dă lux. Poate nu e nevoie de conversie, dar s-ar putea clasifica lumina: de ex. dacă lux < calib\_dark+delta => "Dark", etc. Pentru simplitate trimitem lux brut.
   * Nivel apă: folosind calibrările, putem calcula procent de umplere. Ex:
   * float dist = readUltrasonic();  
     int waterPercent = map(dist, waterMaxDistCal, waterMinDistCal, 0, 100);  
     waterPercent = constrain(waterPercent, 0, 100);
   * sau putem trimite direct distanța (cm) și lăsa UI-ul să interpreteze ("10 cm de la senzor" – dacă calibrăm știe că la 10 cm e aproape plin).
   * Calitate aer: MQ-135 necesită calibrări complexe, dar se poate trimite valoarea ADC brută ca un indice.
4. **Compunerea payload-ului:** Dispozitivul formează un obiect JSON cu aceste valori. Exemplu de payload trimis:

* {  
   "temperatureC": 23.15,  
   "humidityAir": 41.2,  
   "soilMoisture": 46, // procent, presupunând calibrare aplicată  
   "lightLux": 78, // lumina in lux  
   "airQuality": 410, // valoare analogica (ex.)  
   "waterLevelCm": 12.7 // nivelul apei in cm sau %, cum s-a decis  
  }
* Acest format corespunde cu SensorReadingRequest de pe server[[96]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Send%20readings%3A%20POST%20%2Fplants%2F,payload)[[97]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,calibrare%20e%20aplicat%C4%83%20la%20device) (în blueprint exemplul avea soilMoisture=612, ceea ce sugerează un raw necalibrat, dar ideal ar trimite calibrat în procente). Notăm că blueprint-ul nu include explicit humidityAir sau waterLevel în exemplul JSON, dar le-a menționat textual (SHT2x pentru umiditate aer, HC-SR04 pentru apă)[[118]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04)[[24]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=), deci le considerăm parte din payload.

1. **Trimiterea datelor la server (dispozitiv → API):** ESP32, având conexiune WiFi activă, inițiază o cerere HTTP POST către URL-ul configurat, de forma http://<server\_ip>:<port>/api/plants/{id}/sensor/readings (unde {id} este ID-ul plantei asociate cu acest device, cunoscut eventual din configurare). În header, poate include o autentificare a dispozitivului – de ex. un ?deviceKey=XYZ în query sau un header X-Device-Key dacă s-a implementat. Dacă nu, serverul poate identifica device-ul doar prin plantId și poate accepta de la oricine (mai puțin sigur). Vom detalia la configurări despre deviceKey. Device-ul transmite JSON-ul ca body, și așteaptă răspuns. Această operațiune folosește librăria HTTPClient în Arduino, deci codul ar semăna cu:

* HTTPClient http;  
  http.begin(apiBase + "/plants/" + plantId + "/sensor/readings");  
  http.addHeader("Content-Type", "application/json");  
  int httpCode = http.POST(payloadJson);  
  if(httpCode == 200 || httpCode == 201) {  
   String resp = http.getString(); // poate contine {"status":"OK"}  
  }  
  http.end();
* După trimitere, modul normal e ca serverul să răspundă cu 200 OK și un corp {"status":"OK"}[[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F), confirmând primirea. Device-ul poate pur și simplu să verifice codul de status și să continue.

1. **Stocarea și procesarea pe server:** La primirea unei cereri POST /sensor/readings, API-ul verifică dacă plantId-ul este valid și (dacă securizat) dacă deviceKey-ul corespunde. Apoi mappează corpul JSON la un DTO SensorReadingRequest și creează un obiect SensorReading cu acele valori plus timestamp curent. Acest obiect este adăugat în baza de date (DbContext). Fiind o operație frecventă, se optimizează folosind Add fără tracking sau chiar un insert direct posibil (AsNoTracking nu e relevant la insert, dar se pot folosi optimizări de tip batch dacă vin multe deodată; blueprint-ul menționează batch insert ca idee de performanță la citiri)[[119]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Observability%3A%20logging%20cereri%2Ferori%3B%20GET%20%2Fhealth,200%20OK). După salvare, serverul returnează răspunsul JSON status OK.

Tot în acest moment, serverul poate avea logică suplimentară: - Să actualizeze un cache cu ultima citire pe plantă (pentru a răspunde rapid la GET latest fără interogare DB). - Să verifice praguri pentru declanșare auto-irrigation (flux deja discutat la 4.2). În exemplul nostru, dacă modul auto e activ și citește moisture mic, va trece la fluxul de udare auto. - Să verifice praguri pentru alerte: de exemplu dacă temperatura este mult peste DefaultTempMax al speciei plantei, serverul ar putea crea un event de alertă (sau direct trimite notificare push, deși nu implementăm push). - Să calculeze valori derivate, dacă e cazul (ex. indice de disconfort, etc., nu neapărat aici).

1. **Accesarea datelor de către utilizator:** Odată ce datele sunt stocate:
2. **Ultima citire (Live data):** Când utilizatorul deschide aplicația sau trage refresh, aplicația apelează GET /plants/{id}/sensor/latest[[120]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK). Serverul interoghează DB (sau cache-ul) pentru cea mai recentă citire a plantei și întoarce un DTO cu valorile. Aplicația afișează aceste valori pe ecranul de detalii al plantei, sub formă numerică și eventual grafică (ex. 46% Soil Moisture, 24°C Temper., 78 lux Light, etc., cum arată și mockup-ul)[[121]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=10%20Water%20tank)[[122]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=tank%21). Valorile sunt actualizate la fiecare refresh sau la un interval (ex. DashboardViewModel poate avea un timer de 30-60s să facă refresh automat)[[74]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DashboardPage%20%2B%20DashboardViewModel).
3. **Istoric și grafic:** Când utilizatorul cere istoricul (ex. schimbă graficul pe 7 zile), aplicația apelează GET /plants/{id}/sensor/history?from=...&to=...&bucket=day[[49]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2Flatest%20%E2%86%92%20SensorReadingDto). Serverul grupează citirile din DB pe zi (sau oră) și calculează medii sau extrage valorile relevante. Răspunsul conține o listă de valori (sau perechi date-valoare) pentru intervalul cerut. Aplicația afișează graficul corespunzător cu Microcharts. De exemplu, pentru umiditate sol, se poate arăta evoluția zilnică pe ultima săptămână – dacă se culeg la 15 min, ar fi ~96 puncte pe zi, dar la bucket=day s-ar arăta medie zilnică sau maxima/minima zilnică.
4. **Notificări threshold:** Aplicația poate local verifica și notifica: ex. dacă ultima citire indică “Water tank: 10%” cum e în UI[[123][124]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Please%20fill%20the%20water), aplicația poate declanșa o notificare locală: "Rezervorul de apă e aproape gol, te rugăm să îl umpli."[[87]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Ficus%20lyrata). Similar, dacă soilMoisture este sub species.DefaultSoilMoistMin, poate notifica "Planta X are solul uscat.". Această logică e locală, deci imediat după ce primește data prin GET latest, ViewModel-ul poate face:

* if(latest.SoilMoisture < plant.DefaultSoilMoistMin)  
   \_notificationService.ScheduleAlert("Plant dry", $"{plant.Name} needs watering soon (soil moisture {latest.SoilMoisture}%).");
* Sigur, dacă modul auto e deja pornit, poate nu notifica la fiecare scădere sub prag (ca să nu sperie user-ul când oricum se udă automat). Dar poate notifica doar dacă auto e off, semnalând user-ului că trebuie să ude manual sau să pornească auto.

1. **Vizualizare și interpretare:** Utilizatorul poate vedea în dashboard un rezumat pentru fiecare plantă. De exemplu, în mockup la o plantă "Ficus lyrata" scrie:
2. Water tank 10 (presupunem %),
3. Light 78 (presupunem lux sau %?),
4. Temper. 24°C,
5. Soil Moisture 46%.[[121]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=10%20Water%20tank)[[122]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=tank%21) S-ar părea că lumina e dată ca % și nu lux (78%?), poate au calibrat lumina relativ. Greu de spus, dar posibil UI-ul transformă lumina în % din maxim calibrat. Oricum, aceste date vin direct din ultima citire. Dashboard-ul fie primește ultimele citiri pentru toate plantele la un refresh (posibil endpoint-ul GET /plants include și ultimul reading în PlantResponse), fie face câte un call la /latest pe fiecare plantă. O metodă eficientă e ca /plants (lista) să includă și valorile curente, pentru a evita n apeluri. Depinde implementare; blueprint-ul nu precizează, dar se putea.
6. **Error handling:** Dacă device-ul nu reușește să trimită date (ex. WiFi down), are un mecanism de retry la următorul ciclu[[125]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Fail). Nu stochează offline datele probabil (memorie limitată), deci se pot pierde citiri. Dacă serverul e inaccesibil temporar, device-ul le va omite și reîncerca la următorul interval.

Acest flux practic constituie **pulsul constant al sistemului**. Datele colectate permit nu doar udarea automată (flux precedent) ci și informarea utilizatorului și generarea de insight-uri: - **Statistici zilnice/săptămânale:** Bazate pe aceste citiri, StatsService poate determina de exemplu “câte ore a fost planta sub pragul de lumină” sau “cât a crescut temperatura față de ieri”, etc. Blueprint-ul enumeră potențiale agregări la StatsService (cele mai udate plante, sub prag azi etc.)[[58]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringService%3A%20manual%20trigger%20,set%20auto%20flag) – deci se pot extrage câte citiri sub prag umiditate au fost într-o zi (indicând stress plantă). - **Detecție alerte:** Un serviciu background ar putea parcurge citirile noi și genera alerte dacă persistă condiții proaste (de exemplu 3 citiri consecutive cu lumină foarte scăzută la o plantă care ar trebui să aibă lumină multă -> notificare de reamplasare planta).

Din perspectiva utilizatorului, acest flux se vede ca actualizări periodice ale datelor din aplicație și eventual notificări. Pentru dezvoltator, e important că este un flux heavy (multe cereri în timp, pentru fiecare device) și trebuie optimizat: indexuri pe DB, opțional mecanisme de batch, limitarea retenției datelor (discutată la config). Blueprint-ul sugerează arhivarea/ștergerea citirilor mai vechi de 12 luni ca un job periodic, pentru ca baza de date să nu crească la infinit[[126]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Observability%3A%20logging%20cereri%2Ferori%3B%20GET%20%2Fhealth,200%20OK).

### 4.4 Fluxul de notificări către utilizator

**Scop:** Alertarea utilizatorului privind starea plantelor sau acțiuni necesare, prin intermediul notificărilor (locale pe dispozitivul mobil). Astfel de notificări îmbunătățesc experiența, asigurând că utilizatorul nu ratează situații critice (ex: rezervor gol, plantă neudată).

**Tipuri de notificări:** Conform cerințelor, aplicația va genera notificări locale pentru: - **Sol uscat sub prag minim** – dacă o plantă nu este în modul auto și umiditatea scade sub limita recomandată, user-ul e avertizat să ude planta. - **Temperatură prea mare** (sau prea mică) – dacă temperatura ambientală iese din intervalul optim al speciei, poate fi notificat (ex: dacă planta e într-un mediu prea rece iarna). - **Rezervor de apă scăzut** – când nivelul apei în rezervorul dispozitivului scade sub un anumit procent (ex. 10-20%), se notifică user-ul să realimenteze.

Aceste notificări sunt **locale** (nu push de la server), ceea ce înseamnă că logica declanșării lor este implementată în aplicația mobilă.

**Flux generare notificare:**

1. **Detecția condiției de alertă:** Cel mai probabil, aplicația detectează condițiile în momentul primirii datelor de la server. Când user-ul deschide aplicația sau refresh-uiește datele, primește SensorReadingDto cu valorile curente. Imediat, fie în ViewModel-ul de plantă, fie într-un serviciu dedicat, se verifică:
2. if (AutoWateringEnabled == false && soilMoisture < DefaultSoilMoistMin) – dacă modul auto nu e pornit și solul e sub prag, atunci planta e în pericol de uscare => creează notificare "Planta X are solul foarte uscat! Udați-o sau activați udarea automată."
3. if (temperature > DefaultTempMax || temperature < DefaultTempMin) – notificare "Temperatura pentru X a ieșit din intervalul optim (Y°C)."
4. if (waterLevelPercent < 15) – notificare "Rezervorul de apă al dispozitivului pentru X este aproape gol."
5. Alte posibile alerte: lumina scăzută pentru plante iubitoare de lumină ("Planta X nu primește suficientă lumină."), sau umiditate aer prea mică (dacă relevant). Aceste comparări se fac fie față de valori implicite din Species, fie față de config globale. În blueprint, speciile au valori default min/max deci logic e să folosească alea[[8]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite).
6. **Programarea notificării locale:** Odată condiția detectată, aplicația folosește NotificationService.ScheduleAlert(title, message) pentru a seta o notificare. Această funcție intern utilizează API-ul platformei:
7. Pe Android, creează un NotificationRequest (prin Plugin.Maui.Notifications) cu Title, Text și opțional id. Apoi Notifications.Default.Schedule(request). Dacă se dorește notificare imediată, se pune request.Schedule = DateTime.Now (practic imediat).
8. Notificarea apare în tray-ul telefonului cu sunet/vibrație standard. Dacă user-ul o apasă, poate fi configurată să deschidă aplicația (Plugin.Maui.Notifications permite acest context linking).
9. **Persistență minimă:** Aplicația ar putea salva într-o listă locală ce alerte a trimis, pentru a nu repeta excesiv (ex. să nu notifice la fiecare refresh că e solul uscat, ideal unul și bun până se rezolvă). Simplu, se poate pune o condiție să nu repete notificarea până la următoarea udare sau pentru un interval. Dar aceste detalii pot fi lăsate și la latitudinea implementării.
10. **Notificări atunci când aplicația nu e activă:** Un avantaj al notificărilor locale este că pot fi programate și în background, însă aplicația trebuie totuși să ruleze codul de detecție la un moment dat. În scenariul curent, dacă utilizatorul nu deschide aplicația deloc, ar putea să nu primească alerte. O soluție ar fi ca aplicația să planifice actualizări periodice (ex. un background task la fiecare X ore care să facă un call la API și să verifice). MAUI nu are out-of-the-box background tasks crossplatform ușoare; s-ar putea implementa pe Android un Worker care să pornească la interval (dar e mai complex). Posibil ca pentru proiectul curent s-a considerat suficient că user-ul va deschide regulat aplicația sau că device-ul va trimite, iar app-ul interceptând la refresh.

O alternativă ar fi fost implementarea alertelor la server și trimiterea de push notifications (via Firebase APNs etc.), dar asta complică mult. Blueprint-ul a inclus concept de alert background service (hosted) pe server, dar nu neapărat finalizat[[34]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsService%3A%20agreg%C4%83ri%20,plante%2C%20sub%20prag%20azi%2C%20etc).

1. **Exemplu practic:** Să zicem că modul auto e dezactivat pentru o anumită plantă. Solul se usucă sub 30%. Device-ul trimite citiri: 28%, 25% etc. Utilizatorul nu a deschis aplicația azi. Data viitoare când o deschide (sau dacă implementăm un refresh periodic in background), imediat ce se primesc datele cu moisture 25%, aplicația lansează o notificare "Planta Ficus lyrata este uscată – sol la 25%. Udă planta sau activează udare automată." Notificarea apare chiar dacă utilizatorul minimizează rapid aplicația, pentru a atrage atenția. Dacă user-ul apasă pe notificare, aplicația se deschide (poate direct la detaliile plantei). La fel, dacă rezervorul scade la 10%, la următoarea actualizare, notificare "Rezervorul de apă e aproape gol. Reumpleți-l."
2. **Notificări la udare automată:** Putem notifica și astfel: "Planta X a fost udată automat 5s la ora 14:30." – ca user-ul să știe că sistemul a acționat. Asta se poate face fie de server (când adaugă WateringLog auto, trimite un event), fie tot local: când user-ul deschide data următoare, vede logul nou și poate notifica retrospectiv (mai puțin util). Probabil nu s-a implementat la nivelul proiectului inițial, dar e o idee.
3. **Testare notificări:** La testare (Definition of Done App) se specifică: "Notificări locale la praguri (soil < min, temp > max, water tank low) funcționale"[[127]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20). Deci clar acestea trebuie să apară. Se va verifica de ex. scăzând artificial pragurile sau punând device-ul să trimită valori extreme și văzând dacă aplicația notifică.
4. **Dezactivare notificări:** În Settings, user-ul poate opta să dezactiveze notificările (ex. un toggle). Dacă e off, NotificationService.ScheduleAlert poate să nu facă nimic. Android oricum permite user-ului să blocheze notificările din setări OS, dar e bine să aibă opțiune in-app.

În ansamblu, fluxul notificărilor este destul de drept înainte: **detectează condiția –> creează notificare locală**. Cea mai mare provocare e să te asiguri că detectarea se face la timp. Într-o versiune viitoare, s-ar putea dori trimiterea datelor de la device direct la server și notificarea push imediată – dar asta necesită infrastructură push. Cu soluția actuală, se acoperă cazurile obișnuite când user-ul verifică aplicația zilnic sau cam așa; dacă uită complet de app, modul auto tot va uda planta (dacă e pornit), iar dacă nu, planta s-ar putea usca fără ca notificarea să-l ajungă la timp. Acesta ar fi un punct de îmbunătățit.

## 5. API-ul REST – specificații complete

Această secțiune consolidează specificațiile API-ului RESTful oferit de SmartGarden, listând toate endpoint-urile disponibile, metodele HTTP utilizate, formatele de input/output (payload și răspuns), precum și detalii de validare și cerințe de autorizare pentru fiecare. Scopul este să servească drept referință pentru dezvoltatori atunci când extind API-ul sau dezvoltă clienți care îl consumă.

**Notă generală:** Baza URL-ului este https://<server>/api/. Toate datele se trimit și primesc în format JSON. Timpurile (timestamps) sunt exprimate ISO 8601 (UTC). Endpoint-urile marcate cu *🔒 necesită autentificare JWT* (header Authorization: Bearer <token>), altele sunt publice sau folosesc alt mecanism (cheie device). Erorile se întorc folosind structura uniformă a erorilor menționată anterior.

Lista endpoint-urilor este organizată pe controlere:

* **AuthController** (/auth)
* POST /auth/register – *Înregistrare utilizator.*  
  **Request:** Obiect JSON de tip LoginRequest cu câmpuri email și password. Exemplu: {"email": "john@example.com", "password": "Secret123!"}. Parola trebuie să respecte eventual politica (minim 6 caractere, etc.).  
  **Răspuns:** 201 Created dacă succes (sau 200 OK). Body: ID-ul numeric al utilizatorului nou creat (ex: {"userId": 42}). Alternativ, s-ar putea returna direct un token JWT și userId ca la login, dar conform spec se returnează doar un ID[[39]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthController%20).  
  **Erori posibile:** 400 ValidationFailed (dacă email-ul e invalid sau parola prea slabă), 409 Conflict (dacă email-ul există deja).
* POST /auth/login – *Autentificare utilizator.*  
  **Request:** LoginRequest cu email și password ale unui cont existent.  
  **Răspuns:** 200 OK. Body: LoginResponse conținând token-ul JWT de acces și ID-ul utilizatorului autentificat, ex: {"token": "<JWT...>", "userId": 42}[[40]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Fregister%20%E2%86%92%20LoginRequest%20%E2%86%92,int%20userId). Token-ul trebuie stocat de client pentru a autoriza cererile ulterioare.  
  **Erori:** 401 Unauthorized (dacă credențialele sunt greșite), 400 ValidationFailed (dacă lipsesc email/parola).
* **PlantController** (/plants) 🔒 *[Necesită JWT]* – gestionează plantele utilizatorului logat.
* GET /plants – *Obține lista de plante.*  
  **Request:** Fără body. Doar header Authorization necesar.  
  **Răspuns:** 200 OK. Body: un array JSON de obiecte PlantResponse pentru fiecare plantă a utilizatorului[[41]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantsController%20). Un PlantResponse conține tipic: plantId, nickname/room (dacă există), species (poate ca sub-obiect cu nume comun și științific), soilType, isOutdoor, dateAcquired, și posibil ultima citire cunoscută (umiditate actuală, etc.) pentru afișare rapidă pe dashboard.  
  **Erori:** Niciuna specifică în afară de cele generale (ex. token expirat -> 401).
* GET /plants/{plantId} – *Detalii despre o plantă.*  
  **Request:** Parametrul de path plantId (int).  
  **Răspuns:** 200 OK. Body: un obiect PlantResponse complet pentru planta cerută[[42]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F%20%E2%86%92%20List). Include toate detaliile ca mai sus, posibil și pragurile implicite ale speciei, plus eventual date calibrări (dacă se doresc expuse, deși probabil nu).  
  **Erori:** 404 NotFound dacă plantId nu există sau nu aparține user-ului autentic.
* POST /plants – *Adaugă o nouă plantă.*  
  **Request:** Body JSON de tip PlantCreateRequest cu detaliile plantei: ex:
* { "nickname": "Ficus living",   
   "speciesId": 3,  
   "soilTypeId": 2,  
   "roomName": "Living Room",  
   "isOutdoor": false,  
   "dateAcquired": "2025-08-14"  
  }
* nickname ar fi un nume dat plantei (poate optional, dacă nu trimite se poate folosi numele speciei), speciesId și soilTypeId sunt selectate din listele predefinite, roomName e text liber sau dintr-o listă (nu e strict validat decât lungimea probabil), isOutdoor bool, dateAcquired data (format ISO date). Aplicația asigură că aceste câmpuri sunt completate.  
  **Răspuns:** 201 Created. Body: ID-ul noii plante: {"plantId": <nou\_id>}[[43]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F). De asemenea, serverul inițiază eventual setări implicite – creează o intrare de calibrări default dacă e cazul sau preia pragurile din specie pentru notificări.  
  **Erori:** 400 ValidationFailed (dacă lipsesc câmpuri obligatorii sau id-uri necunoscute), 409 Conflict (dacă există deja o plantă cu același nume? Depinde dacă se consideră duplicat problematic – probabil nu, deci 409 nu prea apare aici).
* PUT /plants/{plantId} – *Actualizează o plantă.*  
  **Request:** Body JSON de tip PlantUpdateRequest cu câmpurile ce pot fi modificate. Similar cu create, poate include aceleași proprietăți în caz că pot fi schimbate: exemplu, utilizatorul poate edita doar nickname, camera, tip sol (dacă transplantă) – specia rar se schimbă, dar poate a greșit inițial. Trimite întregul obiect sau doar diferențe (dar fiind PUT, serverul așteaptă tot obiectul probabil).  
  **Răspuns:** 204 No Content la succes (planta a fost modificată)[[44]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId). Fără body.  
  **Erori:** 400 ValidationFailed (dacă datele sunt invalide), 404 NotFound (dacă plantId invalid sau nu aparține user).
* DELETE /plants/{plantId} – *Șterge o plantă.*  
  **Request:** fără body, doar Authorization și plantId în URL.  
  **Răspuns:** 204 No Content la succes (planta și toate datele asociate au fost șterse)[[45]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId).  
  **Erori:** 404 NotFound (dacă nu există sau nu aparține user).
* PATCH /plants/{plantId}/auto?enabled=true|false – *Comută udarea automată pentru plantă.*  
  **Request:** Parametrul query enabled trebuie să fie "true" sau "false". Alternativ, ar fi putut fi și body JSON, dar blueprint-ul indică param query[[46]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DELETE%20%2F). Acțiunea va activa/dezactiva modul auto-irigare pentru planta respectivă. Dacă implementarea e la nivel global user, serverul va seta UserSettings.AutoWateringEnabled la valoarea respectivă (și eventual SoilMoistThreshold default dacă vine de la specie). Dacă ar fi pe plantă, ar marca planta intern.  
  **Răspuns:** 204 No Content la succes[[46]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DELETE%20%2F).  
  **Erori:** 404 NotFound (dacă plantId invalid), sau 400 dacă param enabled lipsește sau nu e boolean valid.
* **SensorController** (/plants/{plantId}/sensor) – gestionează datele senzorilor. Observație: Auth nu e menționat explicit la acest controller; pentru securitate, GET-urile pot necesita token ca user (ca să nu poți vedea senzorii altei persoane), dar POST-ul de la device nu are user. Se poate configura [AllowAnonymous] pentru POST și [Authorize] pentru GET-uri. În plus, POST va verifica o cheie de dispozitiv.
* POST /plants/{plantId}/sensor/readings – *Înregistrează o citire de senzori.* (apelat de dispozitiv, nu de aplicație)  
  **Request:** Body JSON SensorReadingRequest cu valorile măsurate[[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F). Ex:
* {  
   "temperatureC": 24.1,  
   "humidityAir": 40.5,  
   "soilMoisture": 22,  
   "lightLux": 300,  
   "airQuality": 415,  
   "waterLevelCm": 5.2  
  }
* Valorile trimise trebuie să fie numere (float sau int). plantId din URL trebuie să corespundă device-ului care trimite – dispozitivul va avea această informație configurată. Pentru securizare, se recomandă includerea unui parametru de autentificare a dispozitivului, de exemplu:
  + fie un header X-Device-Key: <unic> (cheie generată la înrolarea device-ului),
  + fie un query param ?deviceKey=.... Serverul, la primirea cererii, validează deviceKey (ex. că corespunde plantei respective, comparând cu o listă de chei valide stocate). Dacă nu se pot implementa chei, atunci acest endpoint ar trebui măcar restricționat CORS (dar asta nu protejează în fața abuzului). **Răspuns:** 200 OK (sau 201 Created, fie, dar nefiind o resursă pe care o accesezi direct ulterior, 200 e ok). Body: un obiect cu status de confirmare, ex. {"status": "OK"}[[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F). **Erori:** 401 Unauthorized (dacă deviceKey invalid – practic ar fi [Unauthorized] alt mod), 400 BadRequest (dacă lipsește vreo valoare necesară sau format greșit), 404 dacă plantId nu e găsit (ex. device-ul are config greșit).
* GET /plants/{plantId}/sensor/latest – *Obține ultima citire de senzori.*  
  **Request:** doar Auth JWT (utilizator) și plantId. User-ul poate cere ultima citire pentru planta sa (ex. pentru afișare live).  
  **Răspuns:** 200 OK. Body: un obiect SensorReadingDto conținând valorile ultimei citiri disponibile[[120]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK). Ex:
* {  
   "timestamp": "2025-08-14T15:30:00Z",  
   "temperatureC": 24.1,  
   "humidityAir": 40.5,  
   "soilMoisture": 22,  
   "lightLux": 300,  
   "airQuality": 415,  
   "waterLevelPercent": 20  
  }
* Notă: se poate decide să trimitem waterLevel direct în procente aici, dacă preferăm (serverul poate calcula din cm calibrat). Important e că UI-ul primește informația necesară.  
  **Erori:** 404 NotFound (dacă planta nu există sau nu are citiri).
* GET /plants/{plantId}/sensor/history?from={ISO}&to={ISO}&bucket={hour|day} – *Obține istoricul citirilor.*  
  **Request:** Parametri query obligatorii:
  + from și to – intervalul de timp dorit (timp de început și sfârșit, format ISO 8601, preferabil cu fusul orar sau Z). Dacă nu se dau, serverul ar putea folosi implicit ultima săptămână/lună.
  + bucket – rezoluția agregării: "hour" pentru valori orare, "day" pentru zilnice. Acest parametru ajută serverul să reducă volumul datelor (în loc să trimită fiecare citire la 15 min, agregă). **Răspuns:** 200 OK. Body: un array de SensorReadingDto *sau* un DTO special pentru istoric. Blueprint-ul menționează List<SensorReadingDto> pentru acest GET[[49]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2Flatest%20%E2%86%92%20SensorReadingDto). Dacă se trimite List, fiecare element poate fi media/suma pe perioada de bucket (depinde cum definim). O abordare: dacă bucket=hour, list conține 24*nr\_zile valori, fiecare cu timestamp la începutul orei și valorile medii pe ora respectivă. Dacă bucket=day, 7 valori (pt o săptămână) cu timestamp la începutul fiecărei zile. S-ar putea trimite min/max pe perioada respectivă în loc de medie, dar probabil medie e suficient. Exemplu de răspuns simplificat (bucket=day, 7 zile):*
  + [  
     { "date": "2025-08-08", "soilMoisture": 55, "temperatureC": 22.3, "lightLux": 1500, ... },  
     { "date": "2025-08-09", "soilMoisture": 48, "temperatureC": 23.1, "lightLux": 1200, ... },  
     ...  
     { "date": "2025-08-14", "soilMoisture": 30, "temperatureC": 24.8, "lightLux": 300, ... }  
    ]
  + Aplicația va folosi aceste date pentru a desena graficul corespunzător.  
    *Erori:*\* 400 dacă parametrii lipsesc sau sunt incorecți (ex. format dată greșit, bucket invalid), 404 dacă plantId invalid/neautorizat.
* **WateringController** (/plants/{plantId}/water) 🔒 – gestionează comenzi de udare (de obicei manuale) și starea pompei.
* POST /plants/{plantId}/water – *Pornește/oprește pompa (cerere manuală).*  
  **Request:** Body JSON ce conține comanda. Format definit de WaterCommandRequest: ex. {"command": "ON", "durationSec": 5} pentru a porni pompa 5 secunde[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). command poate fi "ON" sau "OFF". În practică, utilizatorul va folosi doar "ON", iar OFF e implicit după durată. Dacă se trimite "OFF", ar putea semnala oprirea anticipată a pompei (rareori folosit). **Răspuns:** 200 OK. Body: un mesaj de confirmare, ex: {"message": "WATER ON"} dacă comanda a fost acceptată[[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F). Acest mesaj poate fi afișat în UI (ex. toast "Udat manual pornit"). **Acțiune server:** Creează un log de udare în DB (WateringLog cu Mode=Manual, Duration=5, timestamp now). Apoi fie transmite direct la device comanda, fie o pune în coada de așteptare. După cum am discutat, serverul va face ca GET /water/status să reflecte această comandă. **Erori:** 404 NotFound (plantId invalid), 400 Validation (dacă command nu e corect sau durata depășește limitele, ex >60), 409 Conflict (dacă deja există o comandă în curs, de exemplu dacă pompa e deja pornită).
* GET /plants/{plantId}/water/status – *Verifică starea pompei / comenzi.*  
  **Request:** Doar autorizare, plantId. Acest endpoint e folosit de firmware prin polling. **Răspuns:** 200 OK. Body: un obiect ce indică dacă e vreo comandă de executat. Blueprint-ul dă exemplu { "isOn": boolean, "lastRunUtc": ISOdatetime }[[50]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20body%3A%20%7B,WATER%20ON). Aici, isOn:true ar semnala dispozitivului să pornească pompa, iar lastRunUtc eventual indică momentul ultimei porniri (sau momentul la care ar trebui să pornească?). Formatul nu e super clar pentru comandă. Alternativ, se putea răspunde cu {"command":"ON","durationSec":5} direct, similar cum se trimite de client. Dar blueprint-ul pare să fi gândit GET status mai mult pentru ca device-ul să raporteze ce face, nu să primească (contrar textului care sugerează poll pentru command)[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate). Să presupunem totuși că se folosește pentru polling comandă: atunci implementarea ar fi: dacă în server există o comandă nepreluată pentru plantă (ex. creată la un manual trigger), atunci la primul GET al device-ului se răspunde cu isOn:true și eventual durationSec într-un câmp extins. Device-ul execută și apoi ar putea face un POST /water (sau tot GET status interpretat altfel) ca confirmare. Simplu: după ce serverul trimite true, poate considera comanda consumată. **Erori:** 404 dacă plantId invalid/neautorizat.
* **CalibrationController** (/plants/{plantId}/calibration) – modul calibrării senzori.
* GET /plants/{plantId}/calibration – *Interoghează modul calibrare.*  
  **Request:** plantId, autorizare: user-ul logat poate chema să vadă statusul (deși el îl știe, doar a setat), însă mai important, device-ul poate chema (care nu are JWT). S-ar putea permite [AllowAnonymous] dar cu deviceKey. **Răspuns:** 200 OK. Body: {"mode": "ON"} sau {"mode": "OFF"}[[94]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). Aplicația nu prea are de ce să folosească asta, că doar ea decide ON/OFF. Dar device-ul folosește: poll la fiecare secundă și vede când devine OFF ca să iasă din modul calibrat[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min)[[53]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Calibration%3A%20dac%C4%83%20GET%20%2Fplants%2F,%E2%86%92%20readInterval%20%3D%201000%20ms). **Erori:** 404 dacă plantId invalid.
* POST /plants/{plantId}/calibration – *Activează/dezactivează modul calibrare.*  
  **Request:** Body JSON cu { "mode": "ON" } sau "OFF"[[52]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). Apelat de aplicație când începe sau termină calibrarea. **Răspuns:** 204 No Content la succes (modificat intern starea calibrării)[[52]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F). **Acțiune:** Serverul poate memora starea calibrării, de exemplu într-un dicționar static Map<plantId, bool isCalibrating> (sau chiar într-un câmp în DB la Plant, dar calibrările sunt rare și scurte, deci poate nu merită persistent). **Erori:** 400 dacă valoarea invalidă, 404 dacă plantId invalid.
* **StatsController** (/stats) 🔒 – statistici globale (posibil la nivel user).
* GET /stats/daily?date={ISO date} – *Statistici zilnice.*  
  **Request:** param date (fără oră, eventual). Dacă lipsește, se consideră azi.  
  **Răspuns:** 200 OK. Body: DailyStatsDto[[54]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20). Conținutul exact al acestui DTO poate fi:
  + date: data respectivă,
  + plantsWateredManual: număr de udări manuale făcute de user în ziua aia,
  + plantsWateredAuto: număr de udări automate (sau total evenimente de udare automată),
  + alertsCount: câte alerte s-au declanșat (ex. câte situații sub prag au fost),
  + eventual top 1 plantă cu cele mai multe alerte sau udări (ex. "Ficus lyrata has been watered 2 times today"),
  + orice altceva relevant (de ex. media valorilor pe zi? Mai puțin probabil aici). **Erori:** 400 dacă format data greșit.
* GET /stats/weekly?weekStart={ISO date} – *Statistici săptămânale.*  
  **Request:** param weekStart = data de luni (sau oricare din săptămână + semnalăm că se consideră săptămâna din care face parte). Dacă lipsește, se ia săptămâna curentă (începând luni).  
  **Răspuns:** 200 OK. Body: WeeklyStatsDto[[55]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20). Poate conține:
  + weekStart și weekEnd,
  + totalWaterings: total udări în acea săptămână (manuale + auto),
  + autoWaterings: număr udări auto în săpt,
  + manualWaterings: număr udări manuale,
  + mostWateredPlant: numele plantei cu cele mai multe udări și câte,
  + daysBelowThreshold: număr de zile în care vreo plantă a avut valori sub prag (poate calitate indicator),
  + ... și alte statistici sumare. **Erori:** 400 param invalid, etc.

Pentru completitudine, putem menționa și endpoint-uri utilitare: - **Health Check**: blueprint-ul zice *GET /health simplu (200 OK)* ca check de stare[[128]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CORS%3A%20permite%20MAUI%20%2B%20ESP). Ar putea fi implementat un /api/health care întoarce 200 și eventual info (versiune). Doar pentru monitorizare.

* **Species & SoilTypes (pentru populare)**: Nu a fost specificat explicit, dar pentru a popula dropdown-urile la adăugare plantă, aplicația are nevoie de listele de specii și tipuri de sol. Două opțiuni:
* Fie sunt incluse direct în PlantController: de exemplu GET /plants/species -> listă species, GET /plants/soiltypes -> listă soiltypes.
* Fie se face un controller dedicat Admin sau Common: /common/species, /common/soiltypes accesibile fără auth sau cu auth. Ori, dacă se face seed fix, aplicația ar putea avea aceste valori hardcodate – dar mai bine prin API pentru flexibilitate.

Să presupunem că există: - GET /species – returnează lista de Species (id, nume comun, nume științific, plus pragurile implicite poate incluse). - GET /soiltypes – returnează lista de SoilType (id, nume, descriere). Ambele ar fi [Authorize] eventual sau public (nu e grav dacă oricine vede speciile).

Dacă există import CSV pentru acestea, ar putea exista endpoint-uri admin: - POST /species/import (doar admin) – trimite un fișier sau conținut CSV, serverul îl parsează. Totuși, dat fiind că importul e o acțiune de configurare, probabil se face offline de către developer, nu printr-un endpoint public.

**Validări și convenții la nivel de API:** - Format email validare simplă (conține @ etc.). Parola minim 6 caractere, eventual 1 literă mare, 1 cifră (depinde ce s-a cerut, blueprint-ul nu explicitează, deci lăsăm minim lungime). - PlantCreateRequest: necesită speciesId și soilTypeId valide (serverul va verifica existența acelor ID-uri, altfel 400). dateAcquired nu în viitor (dacă user pune data viitoare, se poate considera invalid). - PlantUpdateRequest: dacă schimbă specie/sol, asigurat ID-urile valide. RoomName și Nickname lungime maximă (ex. 50 caractere). - WaterCommandRequest: dacă command = "ON", necesită durationSec (int) să fie >0 și <= probabil o limită (60). Dacă e "OFF", nu are durată. Comanda poate fi case-insensitive dar se poate forța uppercase. - SensorReadingRequest: valori numeric convertibile. Serverul poate valida că valorile au sens (ex. temperature între -40 și 85°C cât suportă senzorul, moisture între 0-100 dacă e procent etc.), dar nu e neapărat strict – doar stochează. Dacă vine ceva absurd (ex. moisture 500%), serverul poate să-l taie la 100 sau să-l lase cum e dar interpretarea la UI va fi oricum nepotrivită. Minim, se asigură că JSON-ul e valid și conține toți senzorii așteptați (sau se pot face opțional anumiți senzori; ex. MQ-135 poate nu e montat la toți). - CalibrationController: acceptă doar "ON" sau "OFF" (case sensitive). Dacă vine altceva -> 400.

**Securitate API:** - JWT se folosește la aproape toate endpoint-urile cu date personale. Token-ul e generat la login și trebuie trimis la fiecare solicitare ulterioară (păstrat de client securizat). - Dispozitivul IoT nu are JWT; pentru el implementăm *Device Keys*. Soluția simplă: la asocierea unui device (când utilizatorul adaugă o plantă), serverul poate genera un deviceKey unic (GUID) și îl afișează utilizatorului (sau un QR code) pentru a-l introduce în firmware. Acest deviceKey se salvează în DB legat de planta respectivă (de exemplu un câmp DeviceKey în tabela Plants). Astfel, la primirea oricărei cereri de la device (POST readings, GET calibration, GET water/status), serverul validează că query param ?deviceKey=... coincide cu Plants.DeviceKey pentru plantId dat. Dacă nu, refuză. În documentație, trebuie menționat developerului să genereze și să utilizeze aceste chei. - Alternativ, se poate simplifica și considera că device-ul cunoaște user-ul și face login (dar e improbabil să punem JWT de user în device pentru că expiră și complică). - Având în vedere enumerarea "device keys" în cerințe, soluția cu DeviceKey per plantă este cea mai potrivită. Configurațiile se pot trece în appsettings (ex. un dictionary plant->key doar pentru test) sau se generează la runtime.

**CORS:** Este configurat să permită origini multiple: - Aplicația mobilă (Android emulator folosește 10.0.2.2 ca alias către localhost PC, de aceea e în listă)[[60]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=true%20). - Browser-ul de test (localhost) – de exemplu dacă testăm API-ul via Swagger UI sau un front web. - "\*" era menționat, ceea ce echivalează cu allow any origin (doar pt dev/test).

În producție, se poate restricționa la originile cunoscute (dacă ar exista un front-end web, altfel la mobil nu e relevant origin). Oricum, fiind un API, se pot accepta toate originiile dacă nu se intenționează acces din browser – mobilul nu are noțiunea de origin.

Acoperind cele de mai sus, am prezentat exhaustiv API-ul. Orice dezvoltator ce preia proiectul ar trebui cu aceste informații să poată înțelege ce endpoints există, ce fac și cum să le folosească/incoke.

## 6. Configurări necesare în proiect (JWT, CORS, string de conexiune, chei de dispozitiv)

Proiectul SmartGarden necesită o serie de configurări și setări pentru a funcționa corect atât în mediul de dezvoltare, cât și în producție. Acestea includ setările pentru autentificare JWT, politica CORS pentru accesul de pe client, conexiunea la baza de date, cheile folosite de dispozitivele IoT, precum și diferențele de configurare între medii (Development vs Production).

**Configurarea JWT (JSON Web Token):** Autentificarea în API se bazează pe JWT, deci este esențial să se configureze o cheie secretă puternică și parametrii token-ului: - În fișierul de configurare (appsettings.json), există secțiunea Jwt cu două chei: "Key" și "ExpiresMinutes". În mediul de dezvoltare (appsettings.Development.json) se folosește o valoare placeholder "dev-secret-change-me" pentru cheie[[35]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). **Această cheie trebuie schimbată într-o valoare complexă și secretă în producție**, ideal stocată în variabile de mediu sau în manager securizat (nu hardcodat). De exemplu, se poate genera un GUID sau o frază aleatorie lungă. - "ExpiresMinutes": 120 indică că token-urile emise expiră după 2 ore[[35]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). Se poate ajusta după cerințe (de ex. 60 min). În startup, la configurarea JWT, se vor folosi aceste valori:

services.AddAuthentication(opt => {  
 opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;  
 opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;  
})  
.AddJwtBearer(opt => {  
 opt.RequireHttpsMetadata = false;  
 opt.SaveToken = true;  
 opt.TokenValidationParameters = new TokenValidationParameters {  
 ValidateIssuer = false,  
 ValidateAudience = false,  
 ValidateLifetime = true,  
 IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"])),  
 ValidateIssuerSigningKey = true,  
 ClockSkew = TimeSpan.Zero // nu acceptăm toleranță la expirare  
 };  
});

Cheia secretă este convertită în bytes și folosită la validarea semnăturii. Deoarece nu avem issuer dedicat sau audiență custom, le lăsăm nevalidate (serverul și clientul fiind de încredere pe același token secret). În plus, se poate seta un issuer/audience dacă dorim (ex. issuer = "SmartGardenAPI"). - Generarea token-ului se face în AuthService (metoda de login). Folosind System.IdentityModel.Tokens.Jwt se creează un token cu claims (de ex. un claim cu userId sau email) semnat cu aceeași cheie. În LoginResponse se trimite string-ul token JWT către client.

**Configurarea CORS:** Pentru ca aplicația mobilă și eventual instrumentele de testare să poată accesa API-ul, se configurează CORS să permită anumite origini: - În dev, Cors:Origins include http://localhost (pentru a testa din browser sau emulator web) și http://10.0.2.2 (Android emulator – aceasta e adresa echivalentă localhost-ului pe PC)[[60]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=true%20). De asemenea, se vede "\*" în listă, ceea ce înseamnă efectiv orice origine este permisă (acest lucru este permis la dev, deși nu e cea mai bună practică – posibil folosit doar temporar). - În cod, se va avea ceva de genul:

services.AddCors(options => {  
 options.AddPolicy("AllowMAUI", builder => {  
 builder.WithOrigins(Configuration.GetSection("Cors:Origins").Get<string[]>())  
 .AllowAnyHeader()  
 .AllowAnyMethod();  
 });  
});

Aici, se extrag originile din configurație (lista menționată) și se permit toate header-ele și metodele. În producție, ar trebui restrâns la originile cunoscute (de exemplu, dacă s-ar avea un domeniu web anume sau aplicația mobilă – însă mobilul nu are origin concept la httpclient). - Aplicația mobilă folosind HttpClient nu e supusă politicii CORS (CORS se aplică doar la cereri din browser). Totuși, pentru testare și pentru eventuale front-end-uri web, e bine configurat. - Important: Să nu se uite apelul app.UseCors("AllowMAUI"); înainte de app.UseAuthentication() și app.UseAuthorization() în pipeline, astfel încât pre-flight-urile OPTIONS să fie permise.

**String-ul de conexiune la baza de date:** Aplicația folosește EF Core, deci necesită un connection string către baza de date. - În dev, configurarea indică folosirea SQLite local: "Default": "Data Source=smartgarden.db"[[129]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=). Aceasta va crea (dacă nu există) un fișier smartgarden.db în directorul de lucru al API-ului. SQLite e convenabil în dev pentru simplitate. - În producție, se va folosi alt DB (posibil SQL Server sau Azure SQL). Probabil se va suprascrie connection string-ul via configurație de producție sau variabilă de mediu. De exemplu, un connection string SQL Server ar arăta ca: "Default": "Server=tcp:myserver.database.windows.net,1433;Database=SmartGarden;User Id=...;Password=...;". Acesta nu se pune direct în cod sursă, ci în appsettings.Production.json (care nu e commitat) sau ca secret la deploy. - SmartGarden.Data conține un extension AddDataLayer(connectionString, useSqlite)[[130]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Data%20seeding%20,demo) care probabil face:

public static void AddDataLayer(this IServiceCollection services, string conn, bool useSqlite) {  
 if(useSqlite)  
 services.AddDbContext<SmartGardenDbContext>(opt => opt.UseSqlite(conn));  
 else  
 services.AddDbContext<SmartGardenDbContext>(opt => opt.UseSqlServer(conn));  
}

În Program.cs, pentru dev se va apela cu useSqlite=true. În Prod, se poate seta useSqlite=false și da string de SQL Server. - Un alt mod: detectare automată după configurație – de exemplu, dacă connection string conține ".db" folosește SQLite altfel SQL Server. Indiferent, developerul trebuie să știe să pună corect connection string-ul relevant.

**Chei de dispozitiv (Device Keys):** Acesta este un aspect de securitate important, altfel orice ar putea trimite date la API pentru oricare plantă dacă știe un ID. În lipsa autentificării clasice pentru device (nu vrem JWT pe device), soluția e o cheie unică per dispozitiv. - **Generare:** Când utilizatorul adaugă o plantă nouă (deci asociază practic un dispozitiv), se poate genera o cheie. Aceasta ar putea fi un GUID sau un token random de lungime considerabilă. De exemplu: DeviceKey = Guid.NewGuid().ToString("N") (32 hex). Această cheie se salvează în baza de date, în înregistrarea plantei, pentru viitoare validări. - **Comunicare către utilizator:** Cheia generată trebuie apoi introdusă în firmware-ul device-ului. În practică, se poate face în mai multe moduri: - Afișarea cheii în aplicație imediat după ce adaugi planta, cerând user-ului să o configureze pe device (de ex. printr-un UI unde user-ul conectează device-ul la USB și programează cu PlatformIO, nu ideal). - Generarea unui QR code sau string pe care device-ul (dacă ar avea ecran sau modul pairing) să-l primească – acest nivel e complex. - O variantă manuală: device-ul vine cu un cod prestabilit (scris pe el), user-ul îl introduce la creare plantă, serverul îl folosește. Dar blueprint-ul nu menționează nimic despre cum se împerechează device-ul. Probabil o simplificare: device-ul la nivel de cod are un deviceKey hardcodat pe care developerul îl pune direct înainte de flash, și îl introduce același în DB. Pentru ușurință, se poate și evita deviceKey complet în faza de demo (bazându-se că nimeni altcineva nu cunoaște IP-ul).

Totuși, cum ni se cere în doc, vom specifica unde se pun cheile. O opțiune: - În appsettings.Development.json se poate pune o secțiune:

"DeviceKeys": {  
 "Planta1": "ABC123...",  
 "Planta2": "DEF456..."  
}

Dar mai bine în DB la Plant. - **Validare la API:** La primirea unui POST /sensor/readings sau GET /water/status de la device, modul de validare: - Dacă s-a inclus deviceKey în query, de ex. /sensor/readings?deviceKey=ABC123, serverul va extrage plantId din ruta, caută planta în DB și verifică că plant.DeviceKey == "ABC123". Dacă nu, returnează 401. - Dacă e ok, acceptă datele. Această verificare ar trebui implementată în middleware sau direct în controller pentru endpoint-urile respective. - Un alt mod: se poate crea un schema de autentificare custom: ex. device trimite un header Authorization: Device ABC123, și serverul poate configura un AuthenticationHandler care verifică acest token în DB. Dar e overkill; mai simplu, în metodele SensorController se face manual validarea deviceKey. - Developerul care preia proiectul trebuie informat să **seteze deviceKey-ul în codul firmware** (backendUrl complet poate include direct ca param, sau în codul de POST să adauge ?deviceKey=) și să se asigure că același key e în baza de date pentru acea plantă. Într-un mediu real, pairing-ul ar fi făcut mai elegant, dar pentru proiect se poate manual.

**Alte configurări:** - **Parametrii de irigație default:** Secțiunea Irrigation din config conține DefaultOnSeconds (cât stă pompa pornită la o comandă auto) – 5 secunde[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120), DefaultPauseMinutes – 2 minute pauză între cicluri de udare auto (dacă mai trebuie)[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120), și DayOnly: true ca să nu ude noaptea[[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120). Acești parametri sunt folosiți de logica auto: - DefaultOnSeconds e folosit dacă nu se specifică altfel la comandă (ex. pentru auto se folosește mereu 5s). - DefaultPauseMinutes poate fi folosit în HostedService-ul de auto-irrigation sau în calculul WateringService când decide următoarea udare. - DayOnly = true înseamnă probabil că WateringService ar trebui să verifice ora actuală înainte de a porni auto – implementează ceva gen:

if(config.Irrigation.DayOnly) {  
 var hour = DateTime.Now.Hour;  
 if(hour < 6 || hour > 21) // intre 22:00 si 5:59 nu udam  
 return;   
}

Astfel, dacă solul e uscat la 2 noaptea, va amâna până dimineața. - **Notificări (plugin)**: Pe mobil, pentru Android, trebuie definint canalul de notificare. Plugin.Maui.Notifications oferă un default, dar e bine de configurat în MauiProgram:

NotificationManager.Current.Initialize();  
if(DeviceInfo.Platform == DevicePlatform.Android) {  
 NotificationManager.Current.Channel = new NotificationChannel(  
 "alerts", "Alerts", "Notificări SmartGarden", NotificationImportance.High);  
}

Asta trebuie făcut o dată. Pe iOS, notificările locale necesită permisiune de la utilizator; se poate solicita la pornire (dar blueprint-ul nu intră în detalii, se poate prezuma acceptate). - **Logging & Serilog**: Opțional, blueprint-ul menționează Serilog ca opțiune[[131]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Stack%20%26%20setup%3A). Dacă se dorește, developerul poate adăuga Serilog (config în appsettings pentru sink-uri, de ex. console, file). În dev, log-level-ul poate fi Debug, în prod Info. - **Environment-specific settings:** - *Development* (ASPNETCORE\_ENVIRONMENT=Development) – folosește SQLite, secret dev JWT, CORS permisiv, Logging detaliat, Swagger activ. - *Production* – folosește SQL Server, secret JWT real (ideal tot în config, dar alt fișier sau KeyVault), poate dezactiva Swagger (sau îl protejează cu autentificare), Logging mai sumar, CORS restrâns. - Aplicația MAUI ar trebui și ea să știe ce URL să folosească. Probabil în dev se folosește emulat local (http://10.0.2.2:5000), iar în release se va seta url-ul public al serverului. Developerul trebuie să actualizeze Constants.ApiBaseUrl în codul mobil la adresa corectă când depune în prod. Ar fi elegant să aibă și app mobil config per build (de ex. un flag DEBUG vs RELEASE). - **Import CSV (Species, SoilTypes):** Pentru a popula tabelele de referință, se poate folosi: - Fie *data seeding* direct (cum am descris la Data, cu HasData sau migration). - Fie un utilitar separat (ex. un script console EF Core or Excel parse). În documentația de față (secțiunea 8) vom da un ghid să folosească direct un script/linq pad cu contextul. Aici, menționăm totuși configurarea: dacă se alege citirea din CSV, developerul trebuie să pună fișierul CSV și eventual să scrie calea undeva. Poate în config se pune o cheie "ImportFolder" etc., dar e overkill. Probabil e un proces manual, deci nu adăugăm configurare specifică.

În concluzie, pentru ca un dezvoltator nou să pună în funcțiune proiectul, trebuie: - Să seteze corect string-ul de conexiune (ex. dacă folosește SQLite, asigurându-se că are acces write în folder, sau dacă schimbă la SQL Server, să aibă instalat și connection string-ul configurat). - Să genereze o nouă cheie JWT pentru producție și să o plaseze în config în siguranță. - Să definească deviceKey-urile pentru fiecare dispozitiv și să le sincronizeze între firmware și server. Ideal, creează o metodă prin care la POST /plants serverul generează un key și îl afișează (poate sub forma unui QR în viitor). - Să verifice setările CORS pentru a se potrivi cu mediul (dacă hostează API-ul pe un anumit domeniu, să permită originile necesare). - Să rețină eventual modificarea config-ului de notificații (ex. alt canal name etc., minor).

Documentarea configurărilor asigură că proiectul poate fi rulat fără probleme și poate fi adaptat la diferite medii și scenarii de deployment.

## 7. Codul firmware ESP32 – explicații și comunicare cu API-ul

Firmware-ul scris pentru microcontrolerul **ESP32** este cel care interfațează direct cu lumea fizică (senzorii și actuatorul pompei) și cu serverul SmartGarden. Vom trece în revistă structura logică a acestui cod, librăriile folosite, configurările, precum și punctele cheie unde comunică cu API-ul.

**Platformă și librării:** Firmware-ul este scris în C/C++ (PlatformIO sau Arduino framework). Se folosesc următoarele librării hardware și de rețea: - **WiFi** – pentru conectarea la rețeaua wireless (inclusă în ESP32 core). - **HTTPClient** – pentru a face cereri HTTP (POST/GET) către API. - **Wire (I2C)** – pentru senzorii conectați pe magistrala I2C, cum ar fi SHT2x (temperatură/umiditate) și BH1750 (luminozitate)[[118]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04). - **SHT2x** – o librărie specifică pentru senzorul de temperatură/umiditate SHT20 (sau SHT21) care oferă citirea directă a valorilor. - **BH1750** – librăria pentru senzorul de lumină BH1750, care dă valori de iluminare în lux. - Pentru alți senzori: - **Analog read** – citiri de la senzorul de umiditate sol (un pin analogic). - **MQ135** – senzor analogic de gaz (calitate aer), se citește tot cu analogRead (poate folosind un algoritm de conversie la ppm, dar la prima iterație se poate doar trimite valoarea brută). - **HC-SR04** (ultrasonic) – necesită declanșare pe un pin (TRIG) și măsurare timp pe altul (ECHO). Se poate folosi o librărie de ultrasonic sau implementare custom. Blueprint-ul menționează "pini conform sketch-ului tău (soil analog, pump, MQ135, HC-SR04)"[[118]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04), deci pinii sunt defineți clar în codul dat (ex: #define SOIL\_PIN 34, etc). - **GPIO** – pentru pompă (un pin digital care controlează un tranzistor/releu).

**Configurații în cod:** În partea de sus a codului firmware, se definesc: - **SSID/PASS** – credențialele WiFi la care se va conecta device-ul. În codul furnizat, acestea pot fi placeholder (ex: const char\* WIFI\_SSID = "YourWiFi"; const char\* WIFI\_PASS = "password";)[[132]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Config%3A). Developerul trebuie să le pună pe cele reale înainte de flash. - **backendUrl** – URL-ul de bază al API-ului[[133]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SSID%2FPASS%20%28placeholder%29). De exemplu: "http://192.168.0.100:5000/api". În dev, dacă API-ul e pe aceeași rețea cu device-ul, se folosește IP-ul local al PC-ului (ex. 192.168.x.x). În produs, ar fi un URL public (ex. https://smartgarden.myserver.com/api). E important să includă http:// sau https:// corect. (ESP32 poate face HTTPS dacă certificatul e gestionat, dar e mai complicat, deci în dev se folosește HTTP). - **Device identifiers** – dacă se folosesc chei de dispozitiv, aici s-ar defini: ex: const char\* deviceKey = "ABC123"; const int plantId = 42;. Astfel încât request-urile să includă ?deviceKey=ABC123 și să știe plantId pentru URL. Dacă nu, se poate hardcoda direct ruta completă (fără securitate). - **Pinout** – definirea pinii la care sunt conectați senzorii: ex:

#define SOIL\_PIN 34 // ADC for soil moisture  
#define PUMP\_PIN 26 // GPIO for pump control (transistor)  
#define MQ135\_PIN 35 // ADC for air quality sensor  
#define TRIG\_PIN 25 // for HC-SR04 trigger  
#define ECHO\_PIN 33 // for HC-SR04 echo

Trebuie să corespundă conexiunilor fizice efective ale modulului. - **Sensor constants** – eventual, calibrări implicite. Dacă s-a calibrat manual, valorile pot fi stocate în variabile NVS (non-volatile storage) sau macro temporare dacă calibrat offline. Exemplu:

int soilDryCal = 3000; // reading in air  
int soilWetCal = 1500; // reading in water  
float waterEmptyDist = 20.0; // cm when tank empty  
float waterFullDist = 5.0; // cm when tank full

Acestea s-ar actualiza după calibrări prin comenzi de la server – implementarea completă a stocării calibrării nu e detaliată, dar se poate face cu Preferences (NVS) în ESP.

**Funcționarea principală (bucla):** Firmware-ul urmează tiparul Arduino: o funcție setup() și o buclă loop() rulând continuu: - În setup(): - Se initializează serial (pentru debug). - Se configurează pinMode pentru pompă (OUTPUT, initial LOW), pentru TRIG (OUTPUT, LOW) și ECHO (INPUT). - Se inițializează comunicațiile I2C (Wire.begin() pentru SHT20 și BH1750). - Se initializează senzorii: ex. sht20.begin(); bh1750.begin(); dacă librăriile o cer. - Se conectează la WiFi: se apelează WiFi.begin(WIFI\_SSID, WIFI\_PASS) și se așteaptă WiFi.status() == WL\_CONNECTED. Se poate pune un timeout sau loop cu mici pauze. Odată conectat, se poate imprima IP-ul obținut. - În loop(): - Se verifică starea WiFi. Dacă WiFi.status() != WL\_CONNECTED, înseamnă că am căzut de pe rețea – în acest caz, se poate încerca reconectarea (poate se apelează WiFi.reconnect() sau se reia WiFi.begin). Până nu e conectat, nu se pot trimite date. - Se determină dacă e momentul pentru o nouă citire de senzori. Codul ar putea folosi variabila statică unsigned long lastRead = 0 și unsigned long readInterval = 900000; (15 min)[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min). În loop:

if(millis() - lastRead >= readInterval) {  
 lastRead = millis();  
 // ... do sensor reading and send logic ...  
}

Astfel la fiecare ~900000ms execută secvența. - **Calibrare vs normal:** Înainte de citire, se poate verifica modul calibrate. De exemplu, la începutul loop sau în condiția de citire:

// check calibration mode by polling server  
bool calMode = false;  
if(checkCalibrationFlagFromServer()) {  
 calMode = true;  
 readInterval = 1000;  
} else {  
 readInterval = 900000;  
}

Unde checkCalibrationFlagFromServer() face un GET la /plants/{id}/calibration și parsează răspunsul mode. Dacă primește "ON", întoarce true. Astfel, când modul calibrate e pornit de user, device-ul în maxim 15 min (sau imediat dacă punem totuși ca device-ul să verifice calibrate mai des) va comuta intervalul la 1 sec. Pentru a reacționa imediat la calibrate, se poate separa poll-ul calibrate de citirea la 15 min. De exemplu, se poate face:

static unsigned long lastCalCheck = 0;  
if(millis() - lastCalCheck > 5000) { // la fiecare 5 sec  
 lastCalCheck = millis();  
 if(checkCalibrationFlagFromServer()) { calMode = true; }  
 else { calMode = false; }  
}  
if(calMode) {  
 // in calibration, ignore normal schedule and set to frequent  
 readInterval = 1000;  
} else {  
 readInterval = 900000;  
}

Apoi restul codului de citire se poate executa fie la 1 sec dacă calMode adevărat, fie la 15 min dacă nu. Blueprint-ul arată direct: "dacă GET calibration -> ON, readInterval=1000ms"[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min). Deci similar logic. - **Citirea senzorilor:** Când se intră în secvența de citire: - Citire temperatură & umiditate aer: ex. float temp = sht20.readTemperature(); float hum = sht20.readHumidity(); - Citire luminozitate: uint16\_t lux = bh1750.readLightLevel(); - Citire umiditate sol analog: int soil = analogRead(SOIL\_PIN); apoi conversie la % dacă calibrat (cum am explicat la flux calibrate). - Citire calitate aer: int air = analogRead(MQ135\_PIN); (poate se face medie din mai multe sample pentru acuratețe). - Citire nivel apă:

long duration;  
digitalWrite(TRIG\_PIN, LOW);  
delayMicroseconds(2);  
digitalWrite(TRIG\_PIN, HIGH);  
delayMicroseconds(10);  
digitalWrite(TRIG\_PIN, LOW);  
duration = pulseIn(ECHO\_PIN, HIGH, 30000); // așteaptă max 30ms  
float distanceCm = duration \* 0.0343 / 2;

Rezultă distanceCm. Apoi procent dacă calibrat:

int waterPerc = map(distanceCm, waterEmptyDist, waterFullDist, 0, 100);  
waterPerc = constrain(waterPerc, 0, 100);

- Se formează payload JSON (folosind e.g. String payload = "{...}" sau ArduinoJson library). De exemplu, folosind ArduinoJson:

StaticJsonDocument<200> doc;  
doc["temperatureC"] = temp;  
doc["humidityAir"] = hum;  
doc["soilMoisture"] = calibratedSoilPercent;  
doc["lightLux"] = lux;  
doc["airQuality"] = air;  
doc["waterLevelCm"] = distanceCm;  
String json;  
serializeJson(doc, json);

- **Trimitere la server:** Folosind HTTPClient http;:

String url = String(apiBase) + "/plants/" + plantId + "/sensor/readings";  
url += "?deviceKey=" + deviceKey; // dacă folosim cheie  
http.begin(url);  
http.addHeader("Content-Type", "application/json");  
int respCode = http.POST(json);  
String respBody = http.getString(); // ar trebui "status":"OK"  
http.end();

Se poate verifica respCode == 200 pentru succes. Dacă nu, se poate retine că nu a reușit. - Dacă calibrate mode este ON, se poate omite temporizarea: practic, se va repeta la fiecare secundă oricum. În calibrate mode, se poate trimite date mult mai des (1 req/s). Serverul poate gestiona asta pentru scurt timp. Ar fi bine totuși device-ul să aibă un mic delay la calibrate (ex. de 1s fix). - **Primirea comenzilor de udare:** După ce trimite citirea sau la fiecare ciclu, dispozitivul trebuie să verifice dacă serverul are vreo comandă de pornire a pompei: - Dacă s-a implementat piggyback, atunci imediat după http.POST(json), se poate parsa respBody pentru un eventual flag. Cum blueprint-ul zice că s-a ales varianta cu polling separat[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate)[[112]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20poll%20la%20GET%20%2Fplants%2F), vom considera: - Se poate face imediat un GET:

HTTPClient http2;  
String url2 = String(apiBase) + "/plants/" + plantId + "/water/status";  
url2 += "?deviceKey=" + deviceKey;  
http2.begin(url2);  
int code = http2.GET();  
if(code == 200) {  
 String body = http2.getString();  
 // parse body e.g. {"isOn": true, "durationSec": 5}  
 StaticJsonDocument<100> doc2;  
 deserializeJson(doc2, body);  
 bool isOn = doc2["isOn"];  
 int dur = doc2["durationSec"] | 0;  
 if(isOn) {  
 // start pump  
 digitalWrite(PUMP\_PIN, HIGH);  
 delay(dur \* 1000);  
 digitalWrite(PUMP\_PIN, LOW);  
 }  
}  
http2.end();

Astfel, dacă serverul a semnalat o comandă, se execută. Dacă durationSec nu e dat în JSON, se poate folosi un implicit (ex. config default). - Trebuie gândit să nu rămână blocat pompa dacă vine Off. Probabil Off se întâmplă oricum după delay. - De asemenea, dacă modul calibrate e ON, e posibil utilizatorul face calibrări iar device-ul tot întreabă water status – nu e problemă, serverul va spune isOn false dacă nimic. - **Fail-safe la udare:** - Se implementează un *hard limit* pentru durată: chiar dacă serverul cere 120 sec, codul poate impune dur = min(dur, 60) pentru a preveni inundare[[125]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Fail). - De asemenea, se poate pune o condiție de securitate: dacă waterLevelPerc (calculat mai sus) este ~0, să nu pornească pompa deloc, chiar dacă comandă vine (să protejeze pompa să nu meargă pe uscat). - **Capătul loop:** - După trimitere date și eventual udare, se iese din if-ul de interval. - Loop-ul apoi reia, face delay mici implicit oricum (nu e nevoie de delay() global, doar se lasă să itereze repede). - Dacă modul calibrate era ON, readInterval e 1000 deci la ~1 sec va reintra. Practic, device-ul în calibrate va bombarda API-ul cu ~1 cerere POST/s plus GET calibrate/s plus GET water/status/s – ar trebui să reziste pentru câteva minute cât calibrăm. - Când user-ul termină calibrate (POST mode=OFF), serverul răspunde la calibrate GET cu OFF, deci device-ul va reseta readInterval la 15 min. - Ar fi drăguț să se reset lastRead astfel încât imediat după calibrate, următoarea citire normală să nu aștepte fix 15 min, dar nu e crucial.

**Interacțiunile cu API rezumate:** - **La pornire:** Device-ul ar putea face un ping la API sau înregistra prezența – nu e obligatoriu. Ar putea descărca config praguri, etc., dar nu cred că s-a implementat. Minim, ar putea face un GET /calibration pentru a asigura modul normal (va primi OFF de obicei). - **Periodic la 15 min:** - POST /plants/{id}/sensor/readings cu payload JSON (datele senzorilor)[[96]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Send%20readings%3A%20POST%20%2Fplants%2F,payload)[[97]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,calibrare%20e%20aplicat%C4%83%20la%20device). - Răspuns așteptat: {"status":"OK"}. - Imediat după: GET /plants/{id}/water/status pentru comenzi de udare. - Răspuns așteptat: ex. {"isOn": false} majoritatea timpului, sau {"isOn": true, "durationSec":5} când user-ul a dat "Water now" sau auto a declanșat. - Mai periodic (ex. la fiecare 5s): GET /plants/{id}/calibration pentru a verifica modul calibrate. - **În modul calibrate (când mode ON):** - Intervalul dintre POST readings devine ~1s. Practic device-ul va spama citiri continue. Aplicația oricum ia ultima eventual. - GET calibration tot la 1s (sau 5s cum am zis) – dar oricum calibrate e ON tot timpul cât calibrăm, deci nu trebuie check super frecvent până la final, să zicem 1s e ok, atunci va detecta OFF rapid. - GET water/status poate continua la 15min sau tot 1s? În calibrate probabil nu vine nicio comandă, deci se poate menține la 15min pentru a nu genera overhead. Ar fi complicat să modul calibrate să întrerupă modul auto – cred că se poate ignora auto în calibrate (oricum calibrăm, deci auto nu e esențial). - Odată ce calibrate devine OFF, device-ul revine la modul normal.

**Tratarea erorilor și robustețe:** - Dacă WiFi se deconectează, codul reasigură reconectarea înainte de a trimite. Poate implementa un backoff dacă rețeaua nu e disponibilă, dar și un watchdog hardware eventual (reboot dacă mult timp offline). - Dacă o cerere HTTP eșuează (ex. server down), http.POST va returna <0 code sau altceva. Codul ar trebui să gestioneze: de ex:

if(respCode != 200) {  
 Serial.println("Failed to send data, code: " + String(respCode));  
 // poate reîncercăm la următorul interval fără să așteptăm tot 15 min?   
 // Posibil puteti pune readInterval = 60000 pentru a reîncerca mai repede.  
}

- Folosirea memoriei: ar fi bine să se folosească serializeJson pentru payload ca mai sus, ca să nu construim string de mână (evităm erori de format). - **Profil runtime:** Citirea și transmiterea datelor la 15 min nu consumă mult. În modul calibrate, 1 request/s timp de 1-2 minute e ~120 request-uri, ceea ce serverul ar trebui să ducă ușor. ESP32 la conexiuni repetate ar trebui să reziste, HTTPClient e destul de robust dacă se face http.end() după fiecare.

**Rezumat comunicare:** - Dispozitivul *trimite date* la server periodic prin HTTP **POST**[[96]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Send%20readings%3A%20POST%20%2Fplants%2F,payload). - Dispozitivul *primește comenzi* de la server prin HTTP **GET** (polling status)[[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate). - Dispozitivul *sincronizează modul calibrate* tot prin HTTP **GET** polling[[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min). Nu există conexiune persistentă (ex. WebSocket) în această implementare; totul e pe cereri HTTP periodice.

**Ulterior update firmware:** Dacă se descoperă bug-uri sau se vrea extindere, dispozitivul se poate reprograma OTA (Over-the-Air) dacă se adaugă suport, dar cel mai probabil se actualizează manual la nevoie. Documentația ar trebui să includă notițe: - "Asigură-te că actualizezi backendUrl în firmware cu adresa actuală a serverului (ex. dacă se schimbă IP-ul) înainte de a flasha device-ul." - "Setează credențialele WiFi corecte în firmware." - "După calibrări, consideră salvarea valorilor calibrate în NVS pentru persistenta între reset-uri."

Blueprint-ul definea *Definition of Done (Firmware)*: - "Trimite payload corect, răspunde la calibrare, execută WATER ON/OFF."[[134]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20) Asta sumarizează: device-ul trebuie să trimită datele în formatul așteptat, să intre/iasă din modul calibrate la comanda serverului și să pornească/oprească pompa la comenzi. Conform fluxurilor detaliate, aceste cerințe sunt îndeplinite.

Un exemplu de pseudocod integrat pentru loop poate arăta așa (combining tot ce am spus):

void loop() {  
 static unsigned long lastRead = 0;  
 // maintain WiFi connection  
 if(WiFi.status() != WL\_CONNECTED) {  
 reconnectWiFi(); // function that tries WiFi.begin if not connected  
 }  
 // poll calibration mode frequently  
 bool calMode = false;  
 if(millis() - lastCalCheck > 5000) {  
 lastCalCheck = millis();  
 calMode = getCalibrationMode(); // GET /calibration, returns true if "ON"  
 if(calMode) {  
 Serial.println("Calibration mode ON, switching to fast interval");  
 }  
 }  
 // choose interval based on mode  
 unsigned long interval = calMode ? 1000 : 900000;  
 if(millis() - lastRead >= interval) {  
 lastRead = millis();  
 // read sensors  
 float temp = sht20.readTemperature();  
 float hum = sht20.readHumidity();  
 int soilRaw = analogRead(SOIL\_PIN);  
 int soilPerc = map(soilRaw, soilDryCal, soilWetCal, 0, 100);  
 soilPerc = constrain(soilPerc, 0, 100);  
 uint16\_t lux = bh1750.readLightLevel();  
 int airRaw = analogRead(MQ135\_PIN);  
 float dist = readDistanceCm(); // uses trigger/echo as above  
 int waterPerc = map(dist, waterEmptyDist, waterFullDist, 0, 100);  
 waterPerc = constrain(waterPerc, 0, 100);  
 // create JSON  
 StaticJsonDocument<256> doc;  
 doc["temperatureC"] = temp;  
 doc["humidityAir"] = hum;  
 doc["soilMoisture"] = calMode ? soilRaw : soilPerc;   
 // if calibrate, maybe send raw to calibrate properly? Or still perc.  
 doc["lightLux"] = lux;  
 doc["airQuality"] = airRaw;  
 doc["waterLevel"] = waterPerc;  
 String json;  
 serializeJson(doc, json);  
 // send HTTP POST  
 if(sendPostRequest("/plants/" + plantIdStr + "/sensor/readings", json)) {  
 // if success  
 if(!calMode) { // in calibrate, skip watering check to avoid conflicts  
 // get water status  
 String resp = sendGetRequest("/plants/" + plantIdStr + "/water/status");  
 if(resp.length() > 0) {  
 StaticJsonDocument<128> doc2;  
 deserializeJson(doc2, resp);  
 bool isOn = doc2["isOn"] | false;  
 int dur = doc2["durationSec"] | configDefaultOnSec;  
 if(isOn) {  
 if(waterPerc < 5) {   
 Serial.println("Water low, abort pump to protect");  
 } else {  
 Serial.println("Watering ON for " + String(dur) + " sec");  
 if(dur > 60) dur = 60;  
 digitalWrite(PUMP\_PIN, HIGH);  
 delay(dur \* 1000);  
 digitalWrite(PUMP\_PIN, LOW);  
 Serial.println("Watering OFF");  
 }  
 }  
 }  
 }  
 }  
 // else if sendPost fails, perhaps retry sooner by not waiting full interval  
 }  
}

*(Codul de mai sus este doar ilustrativ, nu complet, dar arată structura.)*

Cu acest comportament, codul ESP32 va colabora cu API-ul pentru a atinge toate cerințele: - **Culege și trimite periodic datele senzorilor** (verificat prin flux 4.3). - **Se comută la modul de citire rapidă în timpul calibrării** și trimite continuu valori (flux 4.1). - **Primește și execută comenzi de udare** fie manuale fie automate (flux 4.2), protejând pompa și notând local executarea (ex. prin serial log).

Astfel, noul dezvoltator poate înțelege codul firmware fără dificultăți: este un loop care face citiri și comunică prin HTTP. Dacă ar fi disponibil codul sursă complet, secțiunile sale ar urma acest pattern. Pentru debugging, se pot urmări mesajele pe Serial Monitor, iar pentru testare, se poate simula serverul răspunzând la GET calibration/water.

## 8. Ghid pentru importul CSV/Excel în tabelele Species și SoilTypes (operând upsert)

Tabelele **Species** și **SoilTypes** din baza de date trebuie populate inițial cu un set de date (speciile de plante suportate de aplicație, respectiv tipurile de sol posibile). Adăugarea acestor date manual prin cod poate fi anevoioasă și rigidă, de aceea e util să avem o modalitate de **import din fișiere** (CSV sau Excel) care să permită actualizarea ușoară a listei de specii/soluri. Mai jos este prezentat un ghid pas cu pas pentru realizarea importului cu upsert (update sau insert după caz):

**Pregătirea fișierelor de intrare:** - Decideți formatul: CSV (Comma-Separated Values) este mai simplu de procesat programatic, Excel (.xlsx) este mai ușor de editat manual. Vom exemplifica pe CSV (pentru Excel se poate folosi o librărie gen EPPlus, similar logicii). - Creați fișierele species.csv și soiltypes.csv care să conțină capetele de coloană și datele. - Pentru species.csv, coloanele pot fi: SpeciesId, CommonName, ScientificName, DefaultSoilMoistMin, DefaultSoilMoistMax, DefaultTempMin, DefaultTempMax, DefaultLightMin, DefaultLightMax, DefaultHumidityMin, DefaultHumidityMax. Includeți valorile în procente pentru umiditate/lumină/humiditate aer și în °C pentru temperaturi. Exemplu de rând:

1, Ficus lyrata, Ficus lyrata, 30, 60, 18, 27, 1000, 10000, 30, 70

(Aici am presupus lumina în lux min 1000 lux, max 10000 lux pentru ficus, etc.) SpeciesId poate fi precompletat sau lăsați gol (dacă lăsați gol, importul va trebui să decidă ID-uri noi). - Pentru soiltypes.csv, coloane: SoilTypeId, Name, Description, RecWaterDueSec, PauseBetweenWaterMin. Exemplu:

1, Universal Potting Soil, General purpose potting mix for indoor plants, 5, 2  
2, Cactus Soil, Fast-draining sandy soil for succulents, 3, 5

Aici RecWaterDueSec e durata recomandată de udare (ex. 5 sec pentru sol universal), PauseBetweenWaterMin e pauza recomandată (ex. cactușii preferă pauze mai mari). - Asigurați-vă că textul este curat (fără virgule ne-escape-uite care pot strica formatul CSV; dacă folosesc virgule în descriere, puneți întreg textul între ghilimele). - Alternativ, dacă folosiți Excel, aveți similar coloane și le exportați ca CSV sau citiți direct cu o librărie.

**Metoda de import:** - Puteți scrie un script de import sub forma unui **Tool Console** sau chiar ca un endpoint API separat (doar pentru administratori). O variantă simplă este să folosiți LINQPad sau un mic program C# care folosește SmartGarden.Data. - Pseudocod pentru import species:

using var context = new SmartGardenDbContext(...); // configurat pe același DB  
var lines = File.ReadAllLines("species.csv");  
foreach(var line in lines.Skip(1)) { // skip header  
 var fields = ParseCsvLine(line);  
 int id = int.Parse(fields[0]);  
 string common = fields[1], scientific = fields[2];  
 double moistMin = double.Parse(fields[3]), moistMax = double.Parse(fields[4]);  
 double tempMin = double.Parse(fields[5]), tempMax = double.Parse(fields[6]);  
 double lightMin = double.Parse(fields[7]), lightMax = double.Parse(fields[8]);  
 double humidMin = double.Parse(fields[9]), humidMax = double.Parse(fields[10]);  
 var species = context.Species.Find(id);  
 if(species == null) {  
 // Insert new species  
 species = new Species { SpeciesId = id };  
 context.Species.Add(species);  
 }  
 // Update fields (upsert logic)  
 species.CommonName = common;  
 species.ScientificName = scientific;  
 species.DefaultSoilMoistMin = moistMin;  
 species.DefaultSoilMoistMax = moistMax;  
 species.DefaultTempMin = tempMin;  
 species.DefaultTempMax = tempMax;  
 species.DefaultLightMin = lightMin;  
 species.DefaultLightMax = lightMax;  
 species.DefaultHumidityMin = humidMin;  
 species.DefaultHumidityMax = humidMax;  
}  
context.SaveChanges();

Această logică: - Citește fiecare linie CSV, separă câmpurile (puteți folosi string.Split(',') simplu dacă nu sunt virgule în text, altfel un parser robust). - Verifică dacă există deja o specie cu acel ID (folosind Find sau SingleOrDefault după PK). Dacă da, face update la toate coloanele (astfel reflectă eventuale modificări în fișier – de exemplu, dacă ați corectat un nume sau prag). - Dacă nu există, creează un nou obiect Species cu acel ID și-l adaugă în context. - După ce parcurge toate, salvează schimbările. EF Core va face automat INSERT pentru noile intrări și UPDATE pentru cele existente modificate. (Notă: dacă vreți ca EF să asigne ID automat, lăsați SpeciesId generat de DB și nu-l setați manual, în schimb căutați după nume să decideți upsert – dar e mai simplu să controlați ID-urile în acest caz). - Procedați similar pentru SoilTypes:

var lines = File.ReadAllLines("soiltypes.csv");  
foreach(var line in lines.Skip(1)) {  
 var fields = ParseCsvLine(line);  
 int id = int.Parse(fields[0]);  
 string name = fields[1], desc = fields[2];  
 int recSec = int.Parse(fields[3]), pauseMin = int.Parse(fields[4]);  
 var soil = context.SoilTypes.Find(id);  
 if(soil == null) {  
 soil = new SoilType { SoilTypeId = id };  
 context.SoilTypes.Add(soil);  
 }  
 soil.Name = name;  
 soil.Description = desc;  
 soil.RecWaterDueSec = recSec;  
 soil.PauseBetweenWaterMin = pauseMin;  
}  
context.SaveChanges();

- Atenție la conversii: folosiți CultureInfo.InvariantCulture dacă parse de double din string ce conține punct vs virgulă zecimală. - Dacă fișierul CSV nu conține ID (adică doriți ca import-ul să aloce ID-uri consecutive noi la specii noi), atunci: - Pentru upsert, se poate decide un identificator unic alternativ, de exemplu ScientificName ar putea fi considerat unic. În loc să Find(id), ați face:

var species = context.Species  
 .FirstOrDefault(s => s.ScientificName == scientific);

Dacă găsit, update, altfel creați nou (dar atunci alocarea ID se face automat de DB la SaveChanges). - În acest caz însă, dacă se modifică scientific name în fișier, scriptul ar crea dublură. Mai robust e să includă ID sau un alt cod intern stabil. - După rularea scriptului, verificați în DB că tabelele au datele corecte (ex. specia 1 Ficus lyrata etc.). EF core va menține integritatea la inserție (de ex. nu există restricții speciale în aceste tabele). - Rulați importul de câte ori e nevoie: cum e upsert, dacă rulați din nou același fișier, nu se dublează intrări, ci doar se actualizează (nu se creează noi deoarece ID-urile există). Dacă eliminați o intrare din CSV, scriptul nu o va șterge din DB – deci pentru sincronizare deplină, ar trebui eventual să comparăm listele și să ștergem pe cele care sunt în DB dar nu mai sunt în fișier (doar dacă se dorește, altfel nu). - Alternativ, se poate realiza importul și direct în Excel folosind scripturi sau import extern, dar programatic prin EF asigură integritatea (convertim corect tipurile, etc.).

**Integrare în fluxul de dezvoltare:** - La initializarea proiectului (prima migrare), se pot importa direct datele folosind **data seeding**. Dar EF seeding e static și nu ușor de actualizat. Soluția CSV permite echipei non-tehnice (ex. horticultori) să editeze listele într-un Excel, iar dev-ul doar re-rulează importul. - Puteți include scriptul de import ca parte a unei rutine de deploy sau ca utilitar separat (ex. un proiect consolă "SmartGarden.Tools" care să ruleze importul). - Documentați sursa datelor: de exemplu, "Species data compiled from NASA houseplants DB" etc., dacă relevant.

**Upsert vs reset:** Upsert, cum am implementat, menține ID-urile existente și actualizează restul. Dacă se preferă ștergerea completă și re-inserarea (purge & insert din CSV), se poate face:

context.Species.RemoveRange(context.Species);  
context.SaveChanges();  
// then re-add all from CSV with new IDs from file or database (ensuring PK no conflict).

Dar asta poate cauza probleme dacă Plant referă Species (FK). Cum la noi Plant are speciesId, dacă am șterge specii și le reinsertăm posibil cu alte ID-uri, ar rupe referințele. Deci upsert e potrivit: nu alterează ID existente, deci integritatea referențială se păstrează.

**Import SoilTypes**: Similar, Plant referă soilTypeId, deci nu vrem să regenerăm aleator ID-urile tipurilor de sol. Mențineți aceleași ID-uri în fișier/upsert.

**Testare import:** - După import, încercați în aplicație: la ecranul Add Plant, lista de specii ar trebui să apară (fie vine de la API, fie integrată – asuma că se va face un GET species). Verificați că toate valorile (common & scientific name) sunt vizibile și că la selectarea speciei se preiau corect pragurile implicite în formular (dacă se face asta). - De asemenea, testând fluxul de auto-watering, asigurați-vă că pragurile default min max de umiditate sol pentru specie sunt rezonabile, altfel ajustați CSV și reimportați.

Acest ghid ar trebui să permită oricui să actualizeze baza de cunoștințe a aplicației (specii suportate și tipuri de sol) fără a scrie migrații manuale, ci doar editând fișiere CSV/Excel ușor de întreținut. Astfel, adăugarea unui nou tip de plantă sau modificarea unui prag se face rapid, sincronizând fișierul și rulând scriptul de import.

## 9. Structura aplicației .NET MAUI – componentele și legătura cu backend-ul

Aplicația mobilă SmartGarden este construită pe .NET MAUI folosind arhitectura MVVM, ceea ce înseamnă că logica de prezentare este separată de codul UI, facilitând testarea și întreținerea. În continuare detaliem structura internă a aplicației – componentele cheie (ViewModels, Views, Services) – și modul în care acestea interacționează între ele și cu backend-ul.

**Arhitectură MVVM și toolkit-ul CommunityToolkit.Mvvm:** Proiectul utilizează .NET MAUI CommunityToolkit MVVM, care oferă facilități precum atribute [ObservableProperty] și [RelayCommand] pentru a genera automat codul de notificare și comenzi (înlocuind INotifyPropertyChanged boilerplate). Astfel, ViewModel-urile pot fi declarate concis, iar legăturile (bindings) cu interfața XAML sunt declarative.

**Structura pe foldere:** (așa cum a fost prezentat la secțiunea 3.4 și confirmat de blueprint[[66]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Structur%C4%83%20proiect%3A)[[67]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Views%2F%20%20%20%20,Dashboard%2C%20PlantDetail%2C%20AddPlant%2C%20Calibrate%2C%20Settings))

* **Models/**: conține clasele care reprezintă datele manipulate în UI. De obicei, acestea corespund DTO-urilor primite de la API. În implementare, este posibil să se fi folosit direct DTO-urile din SmartGarden.Core (dacă s-a partajat codul) sau s-au definit modele noi cu proprietăți similare. De exemplu, PlantModel (sau chiar s-au folosit direct PlantResponse dacă e accesibil) poate avea proprietăți: Id, Name, SpeciesName, SoilTypeName, IsOutdoor, LatestSoilMoisture, etc. Modelele pot conține și logică ușoară, de ex. o proprietate calculată PlacementDisplay care returnează "Outdoor" sau "Indoor" în funcție de flag-ul IsOutdoor.
* Nu s-a menționat explicit, dar e posibil că LoginRequest, LoginResponse, PlantCreateRequest etc. sunt replicate în Models ca structuri pentru a fi trimise via ApiService. Totuși, cum Core e disponibil, s-ar putea reuse.
* Dacă s-a implementat caching offline, pot exista modele entitate pentru SQLite, dar blueprint-ul spune că storageService e opțional[[72]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync).
* **Services/**: servicii singleton care realizează sarcini specifice:
* ApiService – incapsulează accesul HTTP. Folosește HttpClient intern. Are metode generice sau specifice: e.g. SendAsync<T>(HttpMethod method, string uri, object? body) care adaugă token-ul și face deserializare în T. Sau separate GetAsync<T>(uri), PostAsync<TOut>(uri, object body). Conform blueprint, acest serviciu adaugă header-ul Bearer preluând token-ul din Preferences["token"][[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token). Probabil stochează HttpClient ca static or reuse (HttpClient is intended to be reused).
  + Token management: AuthService după login pune token în Preferences. ApiService la fiecare apel citește de acolo token-ul curent și îl atașează. Ar fi putut păstra și o variabilă statică internă pentru ușurință, dar oricum e acces rapid.
  + Erorile: dacă serverul întoarce 401, ApiService poate detecta și lansa un eveniment (ex. token expirat) care duce la relogare.
* AuthService – folosește ApiService pentru a chema /auth/login și /auth/register[[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token). Metoda LoginAsync(email, pass) trimite LoginRequest și primește LoginResponse. La succes, salvează token-ul și userId în Preferences (sau SecureStorage pentru mai multă securitate)[[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token). Poate de asemenea reține un flag de logged in. De asemenea, are RegisterAsync(username, email, pass) pentru înregistrare.
  + După login/reg, acest serviciu poate popula de ex. un câmp global cu user-ul curent (ex. Preferences["userId"]).
  + Asigură de asemenea log-out: șterge token-ul stocat.
* PlantService – include metodele de gestionare a plantelor, cum ar fi:
  + GetPlantsAsync() – GET /plants (returnează lista de PlantModel). Cheamă ApiService.Get<List<PlantResponse>>("/plants") și o convertește la listă de PlantModel sau direct ține PlantResponse dacă dat totdeauna.
  + CreatePlantAsync(PlantCreateRequest request) – POST /plants. La succes, poate întoarce plantId nou sau direct PlantModel adăugat.
  + UpdatePlantAsync(plantId, PlantUpdateRequest req) – PUT /plants/{id}.
  + DeletePlantAsync(plantId) – DELETE /plants/{id}.
  + ToggleAutoAsync(plantId, bool enabled) – PATCH /plants/{id}/auto?enabled=X[[135]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20GetPlantsAsync%2C%20Create%2FUpdate%2FDelete%2C%20ToggleAutoAsync%2C%20WaterNowAsync). Trimite nimic sau un mic body, nu primește altceva decât cod status. Poate actualiza local starea plantei (dacă avem in lista).
  + WaterNowAsync(plantId) – POST /plants/{id}/water cu {"command":"ON","durationSec":...}[[135]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20GetPlantsAsync%2C%20Create%2FUpdate%2FDelete%2C%20ToggleAutoAsync%2C%20WaterNowAsync). Durata se poate fie decide fix (ex. 5s default, server oricum poate suprascrie). Răspunsul (message) se poate afișa user-ului (ex. toaster "Watering started").
  + Toate aceste metode apelează ApiService și prind eventuale excepții (ex. net down) pentru a informa UI (ex. prin throw ulterior sau rezultat eșuat).
* SensorService – metode:
  + GetLatestAsync(plantId) – GET /sensor/latest[[70]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WaterNowAsync), returnează un SensorReadingModel (sau direct dtos) cu ultima citire.
  + GetHistoryAsync(plantId, from, to, bucket) – GET /sensor/history[[70]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WaterNowAsync), returnează listă de citiri (sau date agregate) pentru grafic. Poate prelucra datele în service: de ex. creează liste de ChartEntry pentru Microcharts:
  + var history = await \_api.GetAsync<List<SensorReadingDto>>(uri);  
    var entries = history.Select(h => new ChartEntry(h.SoilMoisture) {  
     Label = h.Date.ToString("dd MMM"),  
     ValueLabel = h.SoilMoisture + "%"  
    }).ToArray();  
    return new LineChart { Entries = entries, ... };
  + În acest mod, Service-ul întoarce direct un obiect Chart pregătit. Alternativ, ViewModel-ul poate primi listă de DTO și să formeze ChartEntries. Depinde de design; toolkit Microcharts e ușor de folosit direct din VM.
* NotificationService – folosește Plugin.Maui.Notifications pentru a programa notificări locale. Are metoda ScheduleAlert(title, message)[[71]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync) care creează o notificare imediată cu titlu și text dat.
  + Intern, poate folosi:
  + await Notifications.Default.Send(title, message);
  + sau Schedule cu un DateTime specificat (pentru amânare, dacă e cazul – dar aici sunt alerte imediate).
  + Mai poate avea o metodă CancelAll() dacă vrei să anulezi notificări programate (nu prea e cazul aici).
* StorageService (opțional) – dacă implementat, folosește SQLite local (prin SQLiteConnection sau EFCore SQLite local) pentru a stoca lista de plante și ultimele N citiri.
  + Ar fi utilizat dacă se dorește mod offline. De exemplu, la GetPlantsAsync s-ar putea face: dacă net e offline, iau lista din baza locală; dacă net e online, fac API call și salvez rezultatul în baza locală.
  + Similar, pot salva ultimele 200 citiri per plantă (conform blueprint)[[72]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync) pentru grafic offline.
  + Probabil nu a fost realizat complet dacă scrie "opțional".
* **ViewModels/**: fiecare ecran are un ViewModel care conține starea (proprietăți observable) și logica (comenzi) pentru acel ecran:
* LoginViewModel:
  + Proprietăți: Email, Password (folosesc [ObservableProperty] -> devin cu notificare automată).
  + Comenzi: LoginCommand (decorată cu [RelayCommand]), GoToRegisterCommand (dacă e ecran separat).
  + LoginCommand implementează: validează input (ex. Email nu e empty, Password lungime ok – altfel poate ridica erori care pot fi legate la UI, sau se folosește CommunityToolkit [ObservableProperty] + [NotifyCanExecuteChangedFor(nameof(LoginCommand))] pentru a (de)activa butonul login când nu complet). Apelă AuthService.LoginAsync; dacă reușit, navighează la Dashboard. Dacă eșuat (excepție sau token null), afișează eroare (ex. await Shell.Current.DisplayAlert("Login failed", "Email or password incorrect", "OK");).
  + Navigarea: se poate face prin Shell navigation: await Shell.Current.GoToAsync("//Dashboard"); (ruta Shell definită).
* DashboardViewModel:
  + Proprietăți: o listă observabilă de PlantModel, e.g. ObservableCollection<PlantModel> Plants. Alte stări: IsBusy (pentru loading indicator la refresh), eventual Filter (dacă implementează filtrare indoor/outdoor sau room).
  + Comenzi: LoadCommand (pull-to-refresh) și WaterNowCommand pe un plantă dată, AddPlantCommand, OpenPlantDetailCommand.
  + La crearea VM (în constructor sau OnAppearing al View), se poate chema LoadCommand. Aceasta setează IsBusy=true, apelează plants = await PlantService.GetPlantsAsync(), populând colecția. Apoi IsBusy=false.
  + Dacă e necesar, se poate filtra lista înainte de afișare (de ex. dacă exista conceptul de "All/Indoor/Outdoor" tab, se poate fie face separat colecții, fie un parametru in Plants model).
  + WaterNowCommand: primeste un PlantModel (dacă e legată la un buton în DataTemplate, parameter = plant). Apelează await PlantService.WaterNowAsync(plant.Id). Poate imediat marca ceva în UI (ex. un icon se aprinde). Dar cum pompa se va opri automat, e destul să notifice user-ul că s-a trimis comanda (ex. toast "Watering started"). PlantService.WaterNowAsync deja trimite comanda; eventual, se poate adăuga direct un log local sau se va reflecta în statistici mai târziu.
  + ToggleAutoCommand: dacă Dashboard are switch global (nu cred, mai degrabă per plant în PlantDetail), atunci ar fi aici. Dar blueprint zice toggle auto e in PlantDetail.
  + Navigare: AddPlantCommand -> Shell.Current.GoToAsync(nameof(AddPlantPage)). OpenPlantDetailCommand(plant) -> navigare la PlantDetail, cu parametru plantId (ex. Shell.Current.GoToAsync($"{nameof(PlantDetailPage)}?plantId={plant.Id}")). Trebuie configurat route in AppShell: <ShellContent Route="PlantDetailPage" ContentTemplate="{DataTemplate local:PlantDetailPage}" /> și query property in PlantDetailViewModel: [QueryProperty("PlantId", "plantId")].
* PlantDetailViewModel:
  + QueryProperty: PlantId (string or int) pentru a primi param din navigare.
  + Proprietăți:
  + Plant (tip PlantModel) – detaliile plantei curente.
  + LatestReading (SensorReadingModel) – ultima citire.
  + HistoryEntries – date pentru grafic (poate direct un Microcharts Chart property, sau listă de ChartEntry).
  + IsAutoEnabled (bool) – starea switch-ului "Auto irrigation".
  + Posibil SelectedHistoryRange (dacă user poate selecta 24h vs 7d grafic).
  + Comenzi:
  + ToggleAutoCommand – user toggles auto switch. Aceasta apelează PlantService.ToggleAutoAsync(plant.Id, IsAutoEnabled)[[77]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Latest%20%2B%20grafic%20History%20,cu%20Microcharts). Poate pune IsBusy un pic dacă vrea. Ar trebui și să reflecte imediat schimbarea – dar dacă serverul nu răspunde, poate reveni toggle-ul. Simplu, presupunem merge.
  + WaterNowCommand – la fel ca in Dashboard, dar aici aplicat pe această plantă. Eventual e același concept, poate reutiliza PlantService.WaterNowAsync.
  + LoadLatestCommand – pentru refresh manual al valorilor (sau se face automat via OnAppearing).
  + ChangeRangeCommand – dacă e buton/segmented control to switch daily/weekly chart.
  + Funcționare: Când PlantDetailVM se inițializează (sau când PlantId query property e set), ea poate:
  + Să ceară de la PlantService fie detaliile plantei (dacă /plants/{id} ar oferi mai multe decât are deja) – dar cum probabil am deja obiectul plant din listă, i-l pot pasa din navigare. Depinde implementare: dacă au navigat cu param, VM ar putea direct obține Plant din PlantService local list cache. Simpler, probabil au trimis obiectul prin BindingContext (shell route support passing complex object? nu cred direct, deci or query param).
  + Apel la SensorService.GetLatestAsync(PlantId) pentru a obține ultima citire. O atribuie la LatestReading. UI se actualizează (e property notificată).
  + Apel la SensorService.GetHistoryAsync(PlantId, week) pentru a obține istoric (ex. 7 zile default). Transformă datele în Entries pentru grafic. De exemplu:
  + var history = await \_sensorService.GetHistoryAsync(Plant.Id, DateTime.Now.AddDays(-6), DateTime.Now, "day");  
    HistoryEntries = new LineChart { Entries = historyEntries, ... };
  + Apoi UI XAML are ceva de gen: <microcharts:ChartView Chart="{Binding HistoryEntries}" />.
  + Populează IsAutoEnabled de undeva: poate din Plant.AutoWateringEnabled dacă ar fi per plant. Dar la noi e global userSettings; totuși UI e pe plant, deci cred ca in design s-a asumat modul auto global pt user = modul auto pt toate plantele, deci un singur switch in Settings global. Însă blueprint-ul definește PATCH /plants/{id}/auto, deci modul auto per plantă, deci PlantModel ar trebui să aibă un field AutoEnabled. Vom presupune că la GET /plants se trimite și dacă e auto pt fiecare (sau dedus din userSettings global). Cum ar fi, if global auto off, toate plantele auto off. Dacă global on, poate user vrea tot on, deci ar fi la nivel global oricum. E cam confuz, dar vom spune că PlantModel are IsAuto bool și e folosit pentru switch. ToggleAutoCommand face ceea ce trebuie la server.
  + Notificări: dacă LatestReading indică problemă, VM ar putea folosi NotificationService. Dar să trimiți notificare când user-ul deja uită la ecran e cam redundant. Notificările se lansează mai degrabă în background sau la Dashboard refresh. Aș face notificări la Dashboard refresh: dacă vine data și e proastă, se lansează totuși notificare, deși user-ul e deja în app... Poate e gândit în background.
* AddPlantViewModel:
  + Proprietăți: Name (nickname plantei), SelectedSpecies (Species model/DTO), SelectedSoilType, RoomName, IsOutdoor (bool), DateAcquired (DateTime).
  + Liste suport: SpeciesList și SoilTypeList – populate fie de la API (ex. la iniț VM face un call GET species și GET soiltypes) fie sunt cumva preîncărcate offline (seed).
  + Comenzi: SaveCommand (sau AddPlantCommand).
  + SaveCommand verifică input (ex. Name optional, species obligatoriu etc.), apoi construiește un PlantCreateRequest:
  + var req = new PlantCreateRequest {  
     Nickname = this.Name,  
     SpeciesId = SelectedSpecies.Id,  
     SoilTypeId = SelectedSoilType.Id,  
     RoomName = this.RoomName,  
     IsOutdoor = this.IsOutdoor,  
     DateAcquired = this.DateAcquired  
    };
  + Apelează await PlantService.CreatePlantAsync(req). Dacă succes, se întoarce la Dashboard (Shell.Current.GoToAsync("..") pentru a merge înapoi la pagina anterioară). Poate trebuie să declanșeze refresh pe Dashboard (dacă nu s-a legat automat navigarea).
  + Adesea se folosește MessagingCenter sau un event aggregator ca Prism's EventAggregator, dar aici se poate simplu: după navigare, Dashboard OnAppearing reface GetPlants sau PlantService intern a ținut o colecție și adaugă acolo noul plant (dacă PlantService.CreatePlantAsync returnează PlantModel creat).
  + Popularea SpeciesList și SoilTypeList:
  + La iniț VM (în constructor) poate chema un CommonService.GetSpeciesAsync() și GetSoilTypesAsync(). Probabil aceste metode sunt în PlantService (sau separate). Or, au preîncărcat în App.xam.cs la startup și stocat cumva.
  + Simplu: PlantService ar putea avea GetSpeciesAsync ce face GET /species (dacă implementat API-ul).
  + Or, dacă se seeded in app, ar putea avea o constantă list. Dar cum s-a vorbit de import CSV, ar fi logic să le ceară de la API (fie offline prepopulate).
  + Validări: poate folosi CommunityToolkit.Mvvm.Input [RelayCommand(CanExecute = nameof(CanSave))] etc., sau manual in Execute:
  + if(SelectedSpecies == null) { await DisplayAlert("Error", "Please select a species", "OK"); return; }
  + CancelCommand – anulează adăugare, navigând înapoi (Shell.Current.GoToAsync("..") sau pop).
* CalibrateViewModel:
  + Proprietăți: starea pașilor calibrării. De exemplu:
  + LightDarkCompleted, LightMediumCompleted, LightBrightIndirectCompleted, LightBrightDirectCompleted, LightLowCompleted (depinde cum au implementat, blueprint-ul menționează 5 stări de lumină)[[98]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors). Probabil, pentru simplitate, ar putea fi un enumerator Stage curent: Stage1=Dark, Stage2=Medium, Stage3=BrightInd, Stage4=BrightDir, Stage5=Low (ordinea exactă?), totuși enumerarea din UI: Dark, Medium Indirect, Bright Indirect, Bright Direct, Low Light (pare Low Light ultimul)[[98]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors). E puțin neclar.
  + similar, SoilDryCompleted, SoilWetCompleted.
  + WaterMinCompleted, WaterMaxCompleted.
  + Un Timer/Countdown property: când userul apasă calibrate, se poate afișa "Wait for 5 seconds". Ar putea fi contor 5-0 sec, actualizat de un Device.StartTimer.
  + Comenzi:
  + StartCalibrationCommand – când se intră pe pagina calibrate, user apasă un buton "Start" (dacă e prevăzut). Atunci:
    - Apelează CalibrationController: await \_api.PostAsync($"/plants/{plantId}/calibration", new { mode="ON" }) pentru a porni modul calibrate pe server[[81]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auto%3A%20activezi%20Auto%20%E2%86%92%20la,min%2C%20apare%20WateringLog%20auto).
    - Poate aștepta răspuns, dacă ok, atunci instruct UI să treacă la primul pas (Light Dark).
    - Pornește un mecanism de poll sensor readings rapid: blueprint-ul sugerează aplicația să facă polling la /calibration la 1s și device-ul trimite automat readings[[80]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibratePage%20%2B%20CalibrateViewModel). Dar cred că e confuz, mai logic e: device trimite readings la 1s, deci app face polling la /sensor/latest la 1s pentru a actualiza valorile.
    - Deci CalibrateVM poate avea un Timer (Device.StartTimer(TimeSpan.FromSeconds(1), () => { fetchLatest(); return true;}); ) care la fiecare sec:
    - var reading = await \_sensorService.GetLatestAsync(plantId);  
      CurrentLightValue = reading.LightLux;  
      CurrentSoilValue = reading.SoilMoistureRaw; // raw ADC value  
      CurrentWaterValue = reading.WaterLevelCm;
    - Astfel, pe UI pot arăta valoarea curentă a senzorului (sau doar folosit intern la calibrate).
  + CalibrateLightCommand – când userul apasă "Calibrate" pentru stadiul curent de lumină (ex. Dark):
    - Preia valoarea curentă a luminii (de la LatestReading sau stocată ca CurrentLightValue).
    - O stochează local într-un dicționar calibraji: e.g. calibration.LightDark = CurrentLightValue;
    - Marchează LightDarkCompleted = true (ce poate face UI să schimbe butonul in Completed).
    - Treci la următorul subpas: set CurrentLightStep = "Medium Indirect".
    - UI actualizează indicațiile (maybe done via a single property like CalibrationMessage).
  + Similar CalibrateSoilCommand, CalibrateWaterCommand pentru celelalte.
  + După ultimul calibrate (e.g. user did Water Max calibrate), execută FinishCalibrationCommand:
    - Apelează API: POST /plants/{id}/calibration {mode:"OFF"}[[81]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auto%3A%20activezi%20Auto%20%E2%86%92%20la,min%2C%20apare%20WateringLog%20auto).
    - Transmite cumva valorile calibrate la device sau server:
    - Poate se decide să trimitem la server prin alt endpoint (nu definit în blueprint) – poate nepotrivit că serverul oricum nu folosește calibrările pentru conversii.
    - Poate trimite direct device-ului: dificil fără canal. Ar fi putut piggyback off calibrate OFF cu data – dar blueprint nu menționează. Cel mai simplu: device-ul a calibrat intern deoarece i-am dat condițiile exacte (ex. soil dry calibrat când e în aer).
    - Probabil device-ul folosește prima valoare citită după ce user apasă calibrate ca referință – însă nu, blueprint zice altfel.
    - O altă soluție: după modul calibrate, aplicația face un POST /sensor/readings forțat cu un obiect special? Dar nu cred.
    - Poate se așteaptă ca developerul să implementeze ceva: de ex. la final calibrării, se creează un endpoint new PUT /plants/{id}/calibration/values cu tot calibrul. Device-ul la next boot poate cere (dar nu implementat).
    - E posibil să se amâne implementarea riguroasă a salvării calibrării.
    - O dată modul OFF trimis, se oprește Timer-ul de polling.
    - Indică UI "Setup Complete" și eventual oferă buton "Done" -> care duce la AddPlantPage (dacă calibrate a fost parte din onboarding).
    - Poate salva local calibrările (ex. in Preferences) dacă se dorește (de ex. pentru referință manuală).
  + Această parte e complexă și poate neimplementată complet în cod. Blueprint-ul dă direcții, dar implementarea calibrării e dificilă; un dev nou ar trebui să urmeze pașii de mai sus.
* **Views/** (XAML + code-behind):
* Ele se leagă de ViewModels fie prin Shell (ShellContent DataTemplate specified), fie explicit.
* Exemplu: LoginPage.xaml:
* <ContentPage ... Title="Login">  
   <ContentPage.BindingContext>  
   <viewModels:LoginViewModel />  
   </ContentPage.BindingContext>  
   <VerticalStackLayout Padding="30">  
   <Entry Text="{Binding Email}" Placeholder="Email" />  
   <Entry Text="{Binding Password}" Placeholder="Password" IsPassword="True" />  
   <Button Text="Login" Command="{Binding LoginCommand}" />  
   <Button Text="Create new account" Command="{Binding GoToRegisterCommand}" />  
   </VerticalStackLayout>  
  </ContentPage>
* Code-behind poate fi gol sau conține logică UI minoră (ex. event pentru Completed la Entry pentru trigger LoginCommand).
* DashboardPage.xaml: folosește CollectionView:
* <ContentPage ... Title="My plants">  
   <ContentPage.BindingContext>  
   <viewModels:DashboardViewModel />  
   </ContentPage.BindingContext>  
   <RefreshView Command="{Binding LoadCommand}" IsRefreshing="{Binding IsBusy}">  
   <CollectionView ItemsSource="{Binding Plants}">  
   <CollectionView.Header>  
   <!-- Filter controls: All / Indoor / Outdoor / Rooms tabs -->  
   </CollectionView.Header>  
   <CollectionView.ItemTemplate>  
   <DataTemplate x:DataType="models:PlantModel">  
   <Grid Padding="10">  
   <!-- Example item layout -->  
   <Label Text="{Binding CommonName}" FontAttributes="Bold"/>  
   <Label Text="{Binding RoomName}" />  
   <Label Text="{Binding SoilMoisturePercent, StringFormat='{0}%'}"/>  
   <Button Text="Water now" Command="{Binding Source={RelativeSource AncestorType=ContentPage}, Path=BindingContext.WaterNowCommand}" CommandParameter="{Binding .}" />  
   <!-- etc. icons for Outdoor/Indoor if needed -->  
   </Grid>  
   </DataTemplate>  
   </CollectionView.ItemTemplate>  
   </CollectionView>  
   </RefreshView>  
   <FloatingActionButton ... Command="{Binding AddPlantCommand}" />  
  </ContentPage>
* Observație: water now buton în template leagă comanda din ViewModel părinte folosind RelativeSource.
* PlantDetailPage.xaml:
  + Va afișa detalii:
  + <Label Text="{Binding Plant.CommonName}" FontSize="Large" />  
    <Label Text="{Binding LatestReading.SoilMoisture, StringFormat='Soil Moisture: {0}%'}" />  
    <Label Text="{Binding LatestReading.TemperatureC, StringFormat='{0} °C'}" />  
    <!-- etc for Light, Humidity, Water tank -->  
    <ChartView Chart="{Binding HistoryChart}" HeightRequest="200" />  
    <Switch IsToggled="{Binding IsAutoEnabled}" />  
    <Label Text="Automatic watering" />
  + plus un Button "Water now" similar care reiterează comanda (deși e nepotrivit dacă auto e On).
  + Calibrare: dacă există un buton "Calibrate sensors" în această pagină, la click navighează la CalibratePage (sau calibrate wizard).
* AddPlantPage.xaml:
  + Form inputs:
  + <Entry Placeholder="Plant Name (optional)" Text="{Binding Name}" />  
    <Picker Title="Plant Type" ItemsSource="{Binding SpeciesList}" ItemDisplayBinding="{Binding CommonName}" SelectedItem="{Binding SelectedSpecies}" />  
    <Picker Title="Soil Type" ItemsSource="{Binding SoilTypeList}" ItemDisplayBinding="{Binding Name}" SelectedItem="{Binding SelectedSoilType}" />  
    <Picker Title="Placement" ItemsSource="{Binding Placements}" SelectedItem="{Binding IsOutdoor}" /> <!-- maybe use a Switch or two Radio for indoor/outdoor -->  
    <Entry Placeholder="Room" Text="{Binding RoomName}" />  
    <DatePicker Date="{Binding DateAcquired}" MaximumDate="{x:Static system:DateTime.Now}" />  
    <Button Text="Add Plant" Command="{Binding SaveCommand}" IsEnabled="{Binding CanSave}" />  
    <Button Text="Cancel" Command="{Binding CancelCommand}" />
  + Real implementare poate varia, de ex. Indoor/Outdoor poate fi două RadioButton bind la IsOutdoor bool (true/false), not a list.
  + Observare: "Placements, Rooms" sunt mentionate în UI strings (All, Bedroom 1, etc.), posibil definitoriu. Poate RoomName se alege din cateva opțiuni predef (Dormitor, Living, etc.), blueprint enumeră taburi de camere[[136]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=All%20Bedroom%201%20Bedroom%202,Living%20Room%20Kitchen%20Bathroom). Dar implementarea minimă e un Entry liber. Dacă ar face predefinit, ar fi ItemsSource RoomsList = ["Living Room","Bedroom 1", ...]. Or blueprint: in PDF e tab selection separate, deci poate n-au implementat direct in Add, lăsând user să scrie.
* CalibratePage.xaml:
  + Realizează interfața wizard:
  + Poate fi un ContentPage care schimbă conținutul pentru fiecare categorie (Light, Soil, Water).
  + Ar putea fi un CarouselView cu 3 pages (Light, Soil, Water), dar blueprint-ul sugera ecrane separate per calibrate (sau stack with Next).
  + Ex:
  + <ContentPage.BindingContext>  
     <vm:CalibrateViewModel />  
    </ContentPage.BindingContext>  
    <Label Text="Calibrating the Light Sensors" FontSize="Medium" />  
    <StackLayout x:Name="LightCalibration" IsVisible="{Binding IsCalibratingLight}">  
     <Label Text="Dark" TextDecorations="{Binding DarkCompleted, Converter=BoolToStrike}" />  
     <Button Text="{Binding DarkCompleted, Converter={StaticResource StepButtonTextConverter}}" Command="{Binding CalibrateLightCommand}" CommandParameter="Dark" />  
     <!-- If completed, button text might be "Completed" disabled; if not, "Calibrate" -->  
     <!-- Similarly for Medium, etc. Could generate via DataTemplate for each substep but easier explicit -->  
    </StackLayout>  
    <Button Text="Next" IsVisible="{Binding LightCalComplete}" Command="{Binding NextStageCommand}" CommandParameter="Soil" />  
    <!-- Then similar stack for Soil sensors, etc -->
  + Aceasta e complicată de scris manual; poate s-a recurs la mai multe ContentPages separate (LightCalibrationPage, SoilCalibrationPage, etc.), navigând de la una la alta. Dar blueprint-ul indică Next pe același page cu different states.
  + Indiferent, logic: arată instructiuni, afișează eventual un temporizator 5 sec (text "Wait for: 5 seconds" decrementează)[[137]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Put%20the%20sensor%20in%20Bright,Indirect%20Light), apoi buton Calibrate devine activ. Poate folosi Device.StartTimer(1 sec) mentionat în VM.
  + UI update after calibrate: once user hits calibrate for Dark, you mark Completed. In UI you could strike-through or mark with a check icon that step.
  + Then Next to next stage (soil).
  + După ultimul stage (Water sensor), buttton "Finish".
  + Sigur e cea mai complexă pagină, e posibil implementată mai simplist decât designul (poate un text instructiv și un singur buton calibrate ce face intern toți pașii? Dar nu, UI multi-step clar).
  + Un dev nou ar trebui eventual să refineze calibrate page dacă nu e complet.
* **Utils/**:
* BoolToStrikeConverter (exemplu): pentru a adăuga TextDecorations = Strikethrough la text când un flag Completed e true (vizual reacție calibrate).
* ValueToColorConverter etc. dacă e nevoie (ex. pentru colorarea textului de moisture in red if < min).
* Constants: adrese, mesaje. Constants.ApiBaseUrl de ex.
* Validator: metode statice pentru vali (ex. email format).
* Extensii: e.g. DateTime ToShortDateStringCustom() etc.
* Eventual, enumerări pentru Rooms (dacă foloseau).
* **App.xaml/AppShell.xaml**:
* App.xaml: resurse globale, de exemplu:
  + Culori: definirea theming.
  + Styles: implicit Entry style etc.
  + MergedDictionaries for each theme possibly.
  + În SettingsPage, toggle theme se realizează prin Application.Current.UserAppTheme = AppTheme.Dark/Light.
* AppShell.xaml:
  + Definește structura de navigație.
  + O posibilă configurare:
  + <FlyoutItem Route="Main" FlyoutDisplayOptions="AsMultipleItems">  
     <Tab Title="Dashboard" Icon="plants.png">  
     <ShellContent ContentTemplate="{DataTemplate views:DashboardPage}" Route="Dashboard"/>  
     </Tab>  
     <Tab Title="Settings" Icon="settings.png">  
     <ShellContent ContentTemplate="{DataTemplate views:SettingsPage}" Route="Settings"/>  
     </Tab>  
    </FlyoutItem>  
    <ShellContent ContentTemplate="{DataTemplate views:LoginPage}" Route="Login"/>  
    <ShellContent ContentTemplate="{DataTemplate views:AddPlantPage}" Route="AddPlant"/>  
    <ShellContent ContentTemplate="{DataTemplate views:PlantDetailPage}" Route="PlantDetail"/>  
    <ShellContent ContentTemplate="{DataTemplate views:CalibratePage}" Route="Calibrate"/>
  + Astfel, Login nu e în meniu, e standalone route.
  + Shell navigation: în AppShell constructor se poate pune:
  + if (Preferences.ContainsKey("token")) {  
     // user logged in  
     GoToAsync("//Dashboard");  
    } else {  
     GoToAsync("//Login");  
    }
  + Dar de obicei asta se face înainte de launching main page. Alternative: la App.xaml.cs OnStart, se set MainPage = new AppShell(); and push login if needed.
  + Blueprint sugerează: "Dacă token valid → //Dashboard; altfel → //Login." (Shell nav)[[84]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Naviga%C8%9Bie%20). Asta se poate implementa cum am spus cu Preferences check la start.
* Navigation flows:
  + After login, //Dashboard opens. If user logs out, //Login navigates (shell resets stack).
* **Permissions & platform specifics:**
* Android: AndroidManifest.xml in Platforms/Android include <uses-permission android:name="android.permission.INTERNET" /> (by default MAUI might add it if HttpClient used? But likely manual)[[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A). Also for notifications: plugin might need a channel, but open-source plugin likely auto-adds needed meta-data. If not, you add Notification channel creation code in MainActivity or as shown.
* Windows: AppManifest - MAUI default includes InternetClient if accessing network. If not, developer add:
* <Capabilities>  
   <Capability Name="internetClient" />  
  </Capabilities>
* in Package.appxmanifest.
* iOS: would require ATS exceptions if calling non-HTTPS server (maybe dev uses http for test, then ATS must allow insecure for that domain).
* **Testing UI & App:**
* Developer ar trebui să testeze:
  + Login/reg flows (introduce email real format, short pass to see if vali error).
  + Add plant (with various combos, ensures goes to Dashboard and appears).
  + Toggle auto (ensuring state toggles).
  + Manual watering (press, see if any immediate feedback).
  + Pull-to-refresh (drag down on dashboard, see if it refreshes).
  + Graph display correctness.
  + Notifications: simulează condiții extreme sau fire events in code to see notifications while app in background. E.g. set an alert for dryness and do an actual local notif.
  + Shell nav: If token invalid, ensure it shows login (simulate by clearing Pref).
* UI responsive: test on different device sizes/orientation.
* Loading indicators: the RefreshView on plants, maybe an ActivityIndicator on login or add plant while waiting response.

**Coding convenții & patterns:** - Observables: e.g. CommunityToolkit generates OnPropertyChanged calls for [ObservableProperty] fields. The VM likely use them extensively. - Async: Commands with [RelayCommand] can return Task for asynchronous operations, so UI automatically disables button while executing (Toolkit does this if method returns Task). - For example:

[RelayCommand]  
private async Task LoginAsync() {  
 if(string.IsNullOrWhiteSpace(Email) || string.IsNullOrWhiteSpace(Password)) {  
 await Shell.Current.DisplayAlert("Validation", "Email and password required", "OK");  
 return;  
 }  
 try {  
 await \_authService.LoginAsync(Email, Password);  
 // if success:  
 await Shell.Current.GoToAsync("//Dashboard");  
 } catch(Exception ex) {  
 await Shell.Current.DisplayAlert("Login Failed", ex.Message, "OK");  
 }  
}

This would be bound to the Login button. - Preferences vs SecureStorage: token might be in Preferences since blueprint mentions Preferences["token"][[138]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token), which is fine for a demo but in production maybe SecureStorage is better (since token is sensitive). - Microcharts: They likely used it by adding the nuget and in XAML referencing the control. Provided config used is minimal as above. It's straightforward (just ensure chart entries have color or it picks default).

**Summary of backend integration:** - The App calls the REST API through the ApiService and specific services. All endpoints enumerated in section 5 are used accordingly. - Auth: login/reg integrated. - Plants: retrieving and sending creation/update data. - Sensors: retrieving latest and history for display. - Watering: toggling auto and manual triggers implemented by calling endpoints. - Calibration: turning on/off mode via API, and retrieving sensor readings in real-time (a creative use of existing sensor endpoints). - Stats: possibly not implemented in UI explicitly (maybe a Stats page could be added showing daily/weekly, but blueprint doesn't mention a dedicated UI page except maybe "Tips & Tricks" text in PDF, but likely out-of-scope). - Notification: no server push, so app had to figure triggers, which we described.

Cu această structură bine definită, un dezvoltator nou poate naviga ușor codul aplicației MAUI și îi poate aduce modificări. De exemplu, dacă se dorește adăugarea unei noi funcționalități (precum un grafic nou sau editarea profilului utilizatorului), se știe unde anume să integreze: la nivel de servicii pentru API, un ViewModel nou sau extins, și apoi un View. Documentația de mai sus oferă o hartă clară a componentelor și relațiilor lor, accelerând curba de învățare.

## 10. Testare, date seed, medii de dezvoltare/producție și configurare

Pentru a asigura calitatea și fiabilitatea proiectului SmartGarden, este esențial să se efectueze testări adecvate, să se utilizeze date de test (seed data) pentru popularea inițială a sistemului, precum și să se aibă o strategie clară pentru configurarea diferitelor medii (Development vs Production). În această secțiune vom acoperi aceste aspecte.

**Testarea aplicației (plan de test):** Proiectul ar trebui testat la mai multe niveluri: - *Testare unitate (Unit Testing):* Se pot scrie teste pentru metodele din SmartGarden.Core (de exemplu, teste pentru validarea entităților sau a unor metode de calcul din servicii). Pentru aceasta, Core neavând dependențe externe, e ușor de testat logic izolată. - *Testare de integrare:* Având un API și o bază de date, este ideal să scriem teste de integrare care pornesc efectiv aplicația (sau cel puțin straturile API + Data) într-un mediu de test, folosind o bază de date de test (ex. SQLite in-memory) și să verifice scenarii end-to-end. Blueprint-ul subliniază necesitatea testelor de integrare pentru părțile importante: autentificare, operații pe plante, trimitere citiri senzori, udare[[65]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=JWT%20pe%20rutele%20protejate%2C%20CORS,OK%2C%20Swagger%20cu%20exemple). De exemplu: - Test de înregistrare + login: apelează /register, verifică răspuns userId, apoi /login cu aceleași credențiale, verifică primirea token și status 200. - Test CRUD plante: creează un plant via API, apoi solicită lista plantelor și asigură că apare, modifică planta și verifică actualizarea, șterge-o și verifică dispariția. - Test flux senzori: simulează trimiterea unei citiri (POST /sensor/readings) și apoi apelează /sensor/latest să vadă că se reflectă. - Test udare manuală: apelează /water (POST) pentru un plant, apoi face ca dispozitivul (simulate GET /water/status) să vadă flag-ul. - Test udare auto: setează modul auto on, inserează în DB manual o citire sub prag, rulează eventual un job/metodă ce ar trebui să insereze WateringLog, etc., sau testezi logică WateringService direct. - *Testare interfață (UI Testing):* .NET MAUI permite testare UI prin frameworks precum Appium, Xamarin.UITest sau TestCloud, dar asta e mai rar în proiecte mici. În absența, se testează manual pe device/emulator: - Scenariu complet: Înregistrează cont, adaugă plantă (fără device real, dar simulezi completare form), vezi planta în listă. - Simulează primire date: dacă ai logat valorile (fără device e greu, dar poți crea un plant și direct manipula DB să conțină SensorReading, apoi refresh UI). - Buton Water Now: asigură că nu produce erori (chiar dacă device-ul nu e conectat, serverul tot răspunde cu ok). - Comutator Auto: activează-l, serverul schimbă starea, dezactivează-l iar. - Notificări: Forțează condiții: de ex. setează manual în DB moisture la 10%, apoi Dashboard refresh, vezi dacă NotificationService se apelează (poți vedea notificarea pe emulator). - *Testare firmware:* Dacă ai hardware, testezi calibrările în mediu real, vezi datele venind la server (prin loguri). Fără hardware, poți rula codul de firmware pe un simulator ESP (dificil) sau analizezi logic în mod teoretic. In practica, se testează direct cu device-ul: - Pornire device, confirmă se conectează la WiFi (vezi pe serial output). - Configurare API IP corect, vezi pe server log cererile primite de la device (ex. POST readings la fiecare 15 min). - Trimiți din aplicație "Water now", urmărești serial să vezi "Watering ON" și pompa fizic să se pornească câteva secunde. - Activezi calibrate mode din app, verifici pe serial că intervalul devine 1s (valori multe trimise), și la final calibrate off revine la 15 min. - testezi fail-safe: scoți WiFi sau oprești serverul, device-ul ar trebui să nu crash-eze, ci să tot încerce reconexiune.

**Datele de test (Seed data):** Pentru a facilita testarea și dezvoltarea fără a porni de la zero, se recomandă inserarea unor date implicite: - **Specii & Tipuri sol:** Cum am discutat la secțiunea 8, se vor avea ~6-10 specii și ~4-6 tipuri de sol predefinite[[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants). De exemplu: - Specii: Ficus lyrata, Monstera deliciosa, Epipremnum aureum (Golden Pothos), Cactus (generic), etc., cu praguri adaptate. - SoilTypes: "Universal potting soil", "Cactus mix", "Clay soil", "Hydroponic", etc., cu recomandări de udare. Acestea fie se seed-uiesc direct în migrația inițială, fie se importă ulterior cu scriptul CSV. - **Utilizator demo:** Un cont demo e util ca să se poată face login rapid într-un mediu de test. Blueprint menționează un user demo[[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants). S-ar putea crea user: email demo@smartgarden.com cu parolă simplă (ex. Password123). Parola fiind stocată hash, se poate genera un hash fie manual fie programatic (ex. dacă folosim Identity, se poate scrie direct hash-ul). - De exemplu, se poate decide algoritm de hash intern (nu e clar, probabil un simplu SHA256 sau Identity's PasswordHasher). - Dacă nu implementat complet, se poate stoca parolă ne-hash (nesigur, deci prefer hashing). - **Plante de exemplu:** Se pot crea 1-2 plante asociate utilizatorului demo, pentru a avea ce vedea în UI de la început[[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants). De ex: - Plant: Id 1, UserId=demo, SpeciesId=1 (Ficus lyrata), SoilTypeId=1 (Universal), RoomName="Living Room", IsOutdoor=false, DateAcquired=6 luni în urmă. - Plant: Id 2, UserId=demo, SpeciesId=2 (Cactus), SoilTypeId=2 (Cactus mix), RoomName="Kitchen", IsOutdoor=true, DateAcquired=2 ani în urmă. - Asigură că există citiri asociate? Ar fi nice to have. Putem insera în seed câteva SensorReadings pentru id 1: ex. o citire de ieri, alaltăieri, ca graficul să nu fie gol. WateringLogs la fel, câteva intrări auto/manual random. - Aceste date ajută la popularea ecranelor imediat ce se rulează aplicația cu cont demo. - Inserarea acestor seed data se poate face: - Prin EF Core *HasData* în configurări (ex. SpeciesConfiguration.HasData(new Species { ... })). Dacă s-au configurat entitățile cu HasData, la aplicarea migrației inițiale se vor insera. - Prin *script SQL* la finalul migrației (AddData se putea face acolo). - Sau programatic: la Startup, după context.Database.Migrate(), se verifică dacă nu există specii -> atunci se apelează script import. - Depinde de preferință; HasData e ok pt static data, dar e fix (trebuie migratii la orice update). - Acel fișier CSV import se poate chiar apela la startup dev mode: ex. if(env.IsDevelopment()) ImportCSVSpecies();.

**Mediul de Dezvoltare vs Producție:** - În mediu de dezvoltare, se folosesc configurările din *appsettings.Development.json*, iar pentru producție *appsettings.json* (și eventual override prin var de mediu). - Diferențe majore: - DB: Dev cu SQLite local (fără cost, ușor resetabil). Prod cu SQL Server sau alt RDBMS robust pentru multi-user concurrency și volume mari. - SQLite dev simplifică setup (nu necesită server DB instalat). Are limitări concurrency (multiple connections mainly read). - Migrations se pot rula automat în dev (ex. context.Database.EnsureCreated or MIGRATE on startup). În Prod, se pot aplica migrările separat. - JWT Secret: cum menționat, în dev e un secret cunoscut (nu-l divulgi totuși pe public). În prod, generezi altul lung și securizat. - CORS: dev poate allow "\*", Prod trebuie restricționat. Dacă aplicația mobilă e distribuită, origin nu e relevant (mobil->no cors). Dar e bine oricum să nu permiti toată lumea, eventual definești origin la domain-ul API. - Logging: dev - nivel Debug, cu output consolă detaliat (inclusiv sensitive data poate in EF logs). Prod - nivel Info/Warn, stocare în fișiere sau monitor (Serilog to file/seq etc.), fără date sensibile. - Swagger: dev activ (contribuie la testare manuală). Prod eventual dezactivat sau protejat (să nu expui doc la oricine). - Device keys: în dev, se poate accepta deviceKey fix/test (chiar ignorat complet pentru ușurință). În Prod, trebuie generate unice și transmise confidențial. - Probabil, modul cum user obține deviceKey e neclar: posibil prelode device cu un known key label. - De ex., s-ar putea livra device-ul cu un sticker "Device Key: ABC123", user la add plant introduce. Acel UI lipsește; eventual a plan might be via calibrate wizard to connect wifi (maybe user enters wifi credentials into device? Possibly they considered adding "Connect your device to WiFi" in wizard). - blueprint UI first screen suggests: 1) Connect device to WiFi (2.4GHz), 2) Calibrate sensors, 3) Add plant details[[89]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=%28make%20sure%20it%E2%80%99s%202). Deci pare modul de configurare: user porneste device in AP mode, se conecteaza phone la device, transmite wifi cred, device se conecteaza la wifi (common IoT pattern). Aici device keys ar putea fi handshake. Dar implementarea completă lipseste (posibil prea complex). - Anyway, in dev se poate cod deviceKey = "DEVKEY123" at both ends for simplicity.

**Gestionarea configurației și secretelor:** - Folosiți *User Secrets* in dev (dacă e ASP.NET Core local) pentru chei sensibile (ex. JWT secret, DB connection if includes password). - În Prod (dacă host on Azure), se pot folosi *Azure App Configuration* or environment variables to override appsettings (ex. Azure Web App can have ConnectionStrings:Default = ...). - Documentați pentru dev care sunt pașii: 1. Clonați repo, rulați dotnet ef database update pentru a crea DB local (sau aplicația face auto). 2. Porniți API-ul (profile perhaps in VS or dotnet run). 3. Porniți MAUI app (fie din VS emulator, fie debug on device). Asigurați-vă că setarea Constants.ApiBaseUrl referă la ip-ul PC (ex. "http://<PC IP>:5000/api") sau dacă rulează emu Android on same PC, "http://10.0.2.2:5000/api". 4. Conectați un device hardware (dacă test IoT) sau rulați un stub (device no stub though, maybe not). 5. Folosiți cont demo: email demo@smartgarden.com, pass demo (if set as such). 6. Observați plante preîncărcate, etc. - Pregătiți scripturi de populare pentru repetare (ex. a developer may need to reset DB, run seeds; can incorporate that into test or dev startup).

**Environment specific code:** - În API Program.cs, se poate folosi:

if(env.IsDevelopment()) {  
 app.UseDeveloperExceptionPage();  
 app.UseSwagger(); app.UseSwaggerUI();  
 // maybe auto seeding logic  
} else {  
 app.UseExceptionHandler("/error"); // custom error endpoint or generic  
 // ensure HSTS, etc if needed  
}

- În MAUI, se pot folosi direct config static for dev vs prod if needed (rare; maybe for API base URL difference, you could do #if DEBUG use 10.0.2.2 else production url). - Filtrarea log-urilor: exclude EF Core sensitive data in Prod.

**Securitate la test:** - Folosiți date demo doar în dev. În Prod, asigurați-vă că eliminați contul demo sau îi schimbați parola/secret. - Device keys: nu lăsați un device key universal; fiecare device in Prod must have unique. - Rate limiting: dacă se dorește, în Prod se poate implementa limitare la endpoint-ul IoT să nu accept infinite posts (someone malicious flooding). Dar deoarece device keys protejează oarecum, e okay. - Clean up demo data: e.g. no demo plants for actual user accounts in Prod except if purposeful.

**Deploy în Prod:** - Baza de date: aplicați migrările pe serverul de DB (ex. run dotnet ef database update pe connection string prod sau include in pipeline). In Azure, se pot folosi Azure DevOps pipelines, etc. - Seeds: dacă ați folosit EF seeding, migrările includ data. Altfel, rulați script import species/soil etc o dată. - Monitorizare: se poate adăuga /health la API (blueprint mentiona un GET /health)[[128]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CORS%3A%20permite%20MAUI%20%2B%20ESP), implementați un endpoint simplu care returnează 200 OK (poate cu info minim: "Up since ..."). Or use AspNetCore HealthChecks. - Logging Prod: direcționați log-urile într-un fișier rotativ sau Azure AppInsights pentru detectare erori runtime. - Aplicația mobilă Prod: - Configurați Release build cu Constants.ApiBaseUrl = "https://myprodapi.com/api". - Verificați că are CompileAOT if required for iOS etc., că permisiunile la Release sunt in ordine. - Distribuiți (Google Play, etc) eventual.

În concluzie, prin testare riguroasă a tuturor componentelor (inclusiv fluxuri de capăt, cum ar fi scenariul complet de la adăugare plantă până la udare automată generată), putem identifica și rezolva probleme înainte de lansare. Datele de seed asigură un punct de plecare robust atât pentru testeri cât și pentru demonstrații. Separarea configurațiilor pe medii garantează că dezvoltarea se face rapid și în siguranță, iar producția rămâne securizată și optimizată.

[[1]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Structur%C4%83%20proiect%3A) [[2]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04) [[3]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Toate%20entit%C4%83%C8%9Bile%20%2B%20DTO,interfe%C8%9Be%20compilabile%2C%20cu%20XML%20summaries) [[4]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Tabele%20%26%20rela%C8%9Bii%20) [[5]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite) [[6]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Users%20,UserSettings%2C%201%E2%80%93N%20Plants) [[7]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=interval) [[8]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite) [[9]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Species%20,nume%20comun%2F%C8%99tiin%C8%9Bific%20%2B%20praguri%20implicite) [[10]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorReadings%20,Plants) [[11]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorReadings%20,Plants) [[12]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Indexuri%20cheie%3A%20SensorReadings,PlantId%2C%20Timestamp) [[13]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SmartGardenDbContext%20cu%20toate%20DbSet%3C%3E) [[14]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20) [[15]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Migrations%20%28Init%29%20%2B%20scripturi) [[16]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Seeding%20minim%3A%206%E2%80%9310%20Species%2C%204%E2%80%936,user%20demo%20%2B%201%E2%80%932%20plants) [[17]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Data%20seeding%20,demo) [[18]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Ce%20livrez%3A) [[19]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Invariante%20minime%20%28ex,intervale%20corecte) [[20]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Models%3A%20User%2C%20UserSettings%2C%20Plant%2C%20Species%2C,op%C8%9Bional) [[21]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DTOs%20) [[22]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auth%3A%20LoginRequest%2C%20LoginResponse) [[23]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Loop%20logic%3A) [[24]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=) [[25]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=) [[26]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Sensors%3A%20SensorReadingRequest%2C%20SensorReadingDto%2C%20HistoryQuery) [[27]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringController%20%28%2Fapi%2Fplants%2F) [[28]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Sensors%3A%20SensorReadingRequest%2C%20SensorReadingDto%2C%20HistoryQuery) [[29]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interfaces%20,ISensorService%2C%20IWateringService%2C%20IStatsService) [[30]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token) [[31]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20CRUD%20%2B%20ownership%20checks,user%20isolation) [[32]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday) [[33]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday) [[34]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsService%3A%20agreg%C4%83ri%20,plante%2C%20sub%20prag%20azi%2C%20etc) [[35]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=) [[36]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantsController%20) [[37]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthController%20) [[38]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK) [[39]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthController%20) [[40]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Fregister%20%E2%86%92%20LoginRequest%20%E2%86%92,int%20userId) [[41]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantsController%20) [[42]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F%20%E2%86%92%20List) [[43]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2F) [[44]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId) [[45]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20%E2%86%92%20PlantCreateRequest%20%E2%86%92,int%20newId) [[46]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DELETE%20%2F) [[47]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorController%20%28%2Fapi%2Fplants%2F) [[48]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK) [[49]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=GET%20%2Flatest%20%E2%86%92%20SensorReadingDto) [[50]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2F%20body%3A%20%7B,WATER%20ON) [[51]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F) [[52]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F) [[53]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Calibration%3A%20dac%C4%83%20GET%20%2Fplants%2F,%E2%86%92%20readInterval%20%3D%201000%20ms) [[54]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20) [[55]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=StatsController%20) [[56]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=R%C4%83spuns%20eroare%20) [[57]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20save%20reading%2C%20latest%2C%20history,grouping%20by%20hour%2Fday) [[58]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringService%3A%20manual%20trigger%20,set%20auto%20flag) [[59]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Config%20%28appsettings) [[60]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=true%20) [[61]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,) [[62]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120) [[63]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token) [[64]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AuthService%3A%20user%20create%2Flogin%20%2B%20token) [[65]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=JWT%20pe%20rutele%20protejate%2C%20CORS,OK%2C%20Swagger%20cu%20exemple) [[66]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Structur%C4%83%20proiect%3A) [[67]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Views%2F%20%20%20%20,Dashboard%2C%20PlantDetail%2C%20AddPlant%2C%20Calibrate%2C%20Settings) [[68]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token) [[69]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20GetPlantsAsync%2C%20Create%2FUpdate%2FDelete%2C%20ToggleAutoAsync%2C%20WaterNowAsync) [[70]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WaterNowAsync) [[71]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync) [[72]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SensorService%3A%20GetLatestAsync%2C%20GetHistoryAsync) [[73]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=LoginPage%20%2B%20LoginViewModel) [[74]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=DashboardPage%20%2B%20DashboardViewModel) [[75]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=List%C4%83%20PlantResponse%20%2B%20Pull,la%2030%E2%80%9360s) [[76]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantDetailPage%20%2B%20PlantDetailViewModel) [[77]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Latest%20%2B%20grafic%20History%20,cu%20Microcharts) [[78]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=AddPlantPage%20%2B%20AddPlantViewModel) [[79]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibratePage%20%2B%20CalibrateViewModel) [[80]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibratePage%20%2B%20CalibrateViewModel) [[81]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Auto%3A%20activezi%20Auto%20%E2%86%92%20la,min%2C%20apare%20WateringLog%20auto) [[82]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SettingsPage) [[83]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ViewModels%2F%20%20%20%20,DashboardVM%2C%20PlantDetailVM%2C%20AddPlantVM%2C%20CalibrateVM%2C%20SettingsVM) [[84]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Naviga%C8%9Bie%20) [[85]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Login%20func%C8%9Bional%2C%20liste%20plante%2C%20udare,manual%C4%83%2C%20toggle%20auto) [[86]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Permissions%20%26%20build%3A) [[94]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CalibrationController%20%28%2Fapi%2Fplants%2F) [[95]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Interval%20normal%3A%20readInterval%20%3D%20900000,15%20min) [[96]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Send%20readings%3A%20POST%20%2Fplants%2F,payload) [[97]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=,calibrare%20e%20aplicat%C4%83%20la%20device) [[110]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=UserSettings%20,irrigation%2C%20thresholds%2C%20read%20interval) [[111]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Water%20commands%3A%20aplica%C8%9Bia%20%E2%86%92%20API,ESP%20poate) [[112]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20poll%20la%20GET%20%2Fplants%2F) [[113]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=%22Jwt%22%3A%20%7B%20%22Key%22%3A%20%22dev,120) [[114]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=WateringService%3A%20manual%20trigger%20,set%20auto%20flag) [[115]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Praguri%20default%20udare%20%28ex.%20SoilMoistureMin%3D30,Max%3D60) [[116]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Decizii%20de%20produs%3A) [[117]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=fie%20prime%C8%99te%20flag%20%C3%AEn%20r%C4%83spunsul,durationSec%20apoi%20LOW%3B%20log%20local) [[118]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Libr%C4%83rii%20%26%20pini%3A%20WiFi%2C%20HTTPClient%2C,SR04) [[119]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Observability%3A%20logging%20cereri%2Ferori%3B%20GET%20%2Fhealth,200%20OK) [[120]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=POST%20%2Freadings%20,OK) [[125]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Fail) [[126]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Observability%3A%20logging%20cereri%2Ferori%3B%20GET%20%2Fhealth,200%20OK) [[127]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20) [[128]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=CORS%3A%20permite%20MAUI%20%2B%20ESP) [[129]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=) [[130]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Data%20seeding%20,demo) [[131]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Stack%20%26%20setup%3A) [[132]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Config%3A) [[133]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=SSID%2FPASS%20%28placeholder%29) [[134]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=Definition%20of%20Done%20) [[135]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=PlantService%3A%20GetPlantsAsync%2C%20Create%2FUpdate%2FDelete%2C%20ToggleAutoAsync%2C%20WaterNowAsync) [[138]](file://file-1BLtVkGFdcEX1XGSBCbYeq#:~:text=ApiService%3A%20GET%2FPOST%2FPUT%2FDELETE%20%2B%20header%20Bearer,token) SmartGarden- cerinte.docx

<file://file-1BLtVkGFdcEX1XGSBCbYeq>

[[87]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Ficus%20lyrata) [[88]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Ficus%20lyrata) [[89]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=%28make%20sure%20it%E2%80%99s%202) [[90]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Hello%2C%20) [[91]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Soil%20Sensor) [[92]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Put%20the%20sensor%20in) [[93]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors) [[98]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Light%20Sensors) [[99]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Bright%20Direct%20Calibrate) [[100]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Dark%20Completed) [[101]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Bright%20Direct%20Calibrate) [[102]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Soil%20Sensor) [[103]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Dry%20Soil%20Completed) [[104]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Wet%20Soil%20Calibrate) [[105]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Water%20SensorCalibrating%20the,Water%20Sensor) [[106]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Calibrating%20the%20Water%20SensorCalibrating%20the,Water%20Sensor) [[107]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Min%20Completed) [[108]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Max%20Calibrate) [[109]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Setup%20Complete%20Now%2C%20let%E2%80%99s%20add,a%20plant) [[121]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=10%20Water%20tank) [[122]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=tank%21) [[123]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Please%20fill%20the%20water) [[124]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Please%20fill%20the%20water) [[136]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=All%20Bedroom%201%20Bedroom%202,Living%20Room%20Kitchen%20Bathroom) [[137]](file://file-XV3qTf1ydyXJb2gnMp7Rgp#:~:text=Put%20the%20sensor%20in%20Bright,Indirect%20Light) SmartGarden.pdf

<file://file-XV3qTf1ydyXJb2gnMp7Rgp>