UNIVERZITA PARDUBICE

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2025 Radek Hoffmann

Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Systém řízení klimatických podmínek pěstování hub

Bakalářská práce

2025 Radek Hoffmann

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Radek Hoffmann**
Osobní číslo:	**I21149**
Studijní program:	**B0688A140009 Informační technologie**
Téma práce:	**Systém řízení klimatických podmínek pěstování hub**
Zadávající katedra:	**Katedra informačních technologií**

Zásady pro vypracování

Cílem bakalářské práce je navrhnutí systému řízení klimatických podmínek pěstování hub. Systém bude tvořen řídícím prvkem pěstírny, serverou částí a mobilní aplikací. Řídící prvek pěstírny reguluje prostředí pěstírny na základě naměřených údajů a odesílá tyto údaje na server, kde jsou ukládány. Mobilní aplikace bude zobrazovat stav pěstírny poskytovaný serverem a umožňovat nastavení požadovaných vlastností prostředí pěstírny. V rámci práce bude navrhnutí vhodného technického řešení pro jednotlivé části systému a implemnentace alespoň některé části systému.

Rozsah pracovní zprávy: **30**
Rozsah grafických prací:
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

OSE, S. – KUNDU, A. – MUKHERJEE, M. – BENERJEE, M. A comparative study: Java vs Kotlin programming in android application development. International Journal of advanced research in computer science, Volume 9, No. 3, ISSN 0976-5697
SIMS, Gary. Android authority [online]. Dostupné z: https://www.androidauthority.com/develop-android-apps-languages-learn-391008/
HARKIRAN, Kaur. Top Programming Languages for Android App Development Dostupné z: https://www.geeksforgeeks.org/top-programming-languages-for-android-app-development/
Joseph Ingeno, Software Architect's Handbook, Birmingham Packt Publishing Ltd., ISBN 978-1-78862--406-0
Eric J. Braude, Michael E. Bernstein, Software Engeneering Modern Approaches, Boston University, Metropolitan College, ISBN 978-1-4786-3230-6

Vedoucí bakalářské práce: **Ing. Martin Pozdílek, Ph.D.**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. prosince 2023**
Termín odevzdání bakalářské práce: **10. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Jan Panuš, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 28. února 2024

Prohlašuji:

Práci s názvem Systém řízení klimatických podmínek pěstování hub jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 19. 8. 2025

Radek Hoffmann

PODĚKOVÁNÍ

Tímto bych rád poděkoval především vedoucímu této bakalářské práce Ing. Martinu Pozdílkovi, Ph.D., za vstřícný přístup, poskytnuté rady a pomoc při realizaci této práce. Dále bych rád poděkoval své rodině a přátelům za podporu v průběhu studia.

ANOTACE

Cílem této bakalářské práce je navrhnutí systému pro řízení klimatických podmínek pěstování hub. Systém je tvořen třemi částmi: řídící jednotkou, serverovou částí a mobilní aplikací. Řídící jednotka snímá stav prostředí, zjištěné hodnoty odesílá serverové části a pomocí připojených zařízení udržuje nastavené klimatické podmínky. Serverová část sestává z databáze a REST API sloužícímu pro komunikaci s řídící jednotkou a mobilní aplikací. Mobilní aplikace zobrazuje stav ovzduší pěstírny a umožňuje nastavení jeho udržovaného stavu.

KLÍČOVÁ SLOVA

Řízení klima prostředí, ESP32, ESP-IDF, REST API, mobilní aplikace, Jetpack Compose, vzdálená správa, Axum, TimescaleDB

TITLE

Mushroom cultivation climate control system

ANNOTATION

The aim of this bachelor’s thesis is to design a climate control system for mushroom cultivation. The system consists of three parts: a control unit, a server component, and a mobile application. The control unit monitors the environmental conditions, sends the measured values to the server component, and maintains the desired climate conditions using connected devices. The server component consists of a database and a REST API used for communication with the control unit and the mobile application. The mobile application displays the conditions inside the cultivation room and allows the user to configure the desired maintained state.

KEYWORDS

Climate control, ESP32, ESP-IDF, REST API, mobile application, Jetpack Compose, remote control, Axum, TimescaleDB

OBSAH

SEZNAM ILUSTRACÍ A TABULEK

SEZNAM ZDROJOVÝCH KÓDŮ

SEZNAM ZKRATEK A ZNAČEK

ACK	Acknowledgement
API	Application Programming Interface
ARM	Advanced RISC Machines
CA	Certificate authority
ESP-IDF	Espressif IoT Development Framework
GND	Ground
GPIO	General-Purpose Input/Output
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
I^2C	Inter-Integrated Circuit
IDE	Integrated Development Environment
IoT	Internet of Things
JSON	JavaScript Object Notation
JWT	JSON Web Tokens
LED	Light-Emitting Diode
MAC	Media Access Control Address
MCU	MicroController Unit
MVVM	Model View ViewModel
NDIR	Nondispersive Infrared sensor
NTP	Network Time Protocol
OLED	Organic Light-Emitting Diode
PWM	Pulse/Width Modulation
REST	Representational State Transfer
RTC	Real/Time Clock
RTOS	Real-Time Operating System
SBC	Single-board Computer
SCL	Serial Clock Line
SDA	Serial Data Line
SMD	Surface-Mount Technology
SNTP	Simple Network Time Protocol
SPI	Serial Peripheral Interface
SPIFFS	Serial Peripheral Interface Flash File System
SQL	Structured Query Language
TLS	Transport Layer Security
UART	Universal Asynchronous Receiver-Transmitter
URL	Uniform Resource Locator

USB	Universal Serial Bus
UTC	Coordinated Universal Time
XML	Extensible Markup Language

ÚVOD

Cílem práce je navržení a implementace systému sloužícímu k řízení klimatický podmínek pro pěstování hub. Takového systému je třeba, jelikož pěstování hub probíhá v prostředí, ve kterém je pro ideální růst třeba regulovat veličiny jako je vlhkost vzduchu, úroveň CO_2, nebo množství umělého osvětlení.

Proto se práce zabývá navržením jednotky snímající okolní hodnoty, podle kterých následně spouští zařízení jako je ventilace, zvlhčovač vzduchu nebo topení. Snímané hodnoty neslouží pouze k přímému řízení prostředí, ale jsou také odesílány na server, kde jsou vloženy do databáze. Ze serveru si řídicí jednotka také bere aktuální žádoucí stav pěstírny, podle kterého se řídí.

Poslední částí systému je mobilní aplikace, která umožňuje zobrazení stavu prostředí pěstírny v průběhu času a nastavení cílových hodnot, jaké pěstírna udržuje. Pro komunikaci mezi aplikací a řídicí jednotkou slouží na serveru běžící REST API.

Proč řídit klima pěstírny hub

S rozvojem počítačem řízených systémů a jejich zvyšující se dostupností se nabízí jejich využití k automatizaci dříve manuálních procesů. Mezi tyto procesy patří i pěstování plodin, kde je záhodno udržovat určité podmínky vhodné k růstu. Pro tuto práci jsem jako cíl řízení vybral pěstování jedlých a léčivých hub, konkrétně takových druhů, které umožňují kultivaci v uměle vytvořeném prostředí v měřítku uplatnitelném pro komerční účely. Řídit veličiny prostředí se dá v mnoha oblastech, ať již jde o jiné odvětví zemědělství, nebo například kvalitu ovzduší v obývaných prostorech. Proto se naskýtá otázka, proč zrovna pěstování hub. Mezi hlavní důvody patří uzavřenost pěstíren a důležitost správných růstových podmínek. Pokud se podíváme na komerční pěstování hub, zjistíme, že většinou jde o uzavřené prostory, s regulovaných přísunem vzduchu, umělým osvětlením, zvlhčovacím systémem a řízením teploty. Všechny tyto parametry je potřeba udržovat v optimálních hodnotách odpovídajících aktuální fázi růstu a pěstovanému druhu hub. Díky těmto vlastnostem pěstování můžeme u automatizovaného systému pro monitoring a řízení sledovat jeho vliv a účinnost a zároveň máme více veličin pro řízení, než naříklad u automatizovaného zavlažování venkovních záhonů, nebo ventilace budov.

Další využití

Jak již bylo zmíněno v předchozí části, pěstování hub není jediné odvětví, které využije automatizovaného řízení prostředí. Základ systému by se dal popsat jako systém zpětné vazby, kde řídicí zařízení snímá stav rozhodujících parametrů pomocí jednoho, či více senzorů a na základě zjištěných hodnot ovládá připojená zařízení manipulující jejich stav (zvlhčovač, klimatizace, ventilace). Stejný princip se tedy dá využít i v dalších odvětvích, zde uvedu pár příkladů:

- Řízení zavlažování rostlin na základě vlhkosti půdy [22]
- Řízení kvality ovzduší uvnitř budovy. Ventilace na zakladě hladiny CO_2, vytápění/ klimatizace na základě zjištěné teploty. [23]
- Monitoring a řízení napájecí sítě

1 Kultivace hub a podmínky na prostředí

Existuje mnoho druhů hub a díky tomu můžeme narazit i na různé podmínky potřebné k jejich růstu. Proto se v této práci zaměřuji na pěstování hub rostoucích z organických zbytků, jako je sláma, piliny, kompost nebo kávová sedlina. Mezi nejčastěji pěstované zástupce této skupiny patří žampiony, hlíva, nebo shiitake. Tyto druhy vybíráme kvůli možnosti jednoduché kultivace na pytlích, či pevných nádobách se substrátem, umožňující pěstování většího množství plodnic pro komerční účely.

1.1 Fáze růstu

Pokud chceme pěstovat jedlé houby, musíme vzít v potaz, že jejich růst má tři fáze. Každá z těchto fází má jiné nároky na okolní prostředí, tudíž je potřeba prostředí pěstírny přizpůsobit aktuální fázi růstu. [1, 24]

1.1.1 Fáze kolonizační

Fáze, ve které je naočkovaný pytel substrátu prorůstán podhoubím. Délka této fáze záleží na pěstovaném druhu, ale i na poměru množství použité sadby na množství substrátu.

- Vlhkost substrátu: 60-75%, nižší vlhkost snižuje rychlost růstu podhoubí. Vysychání substrátu lze zabránit držením vysoké vlhkosti vzduchu.
- Větrání: Podhoubí snáší vysokou koncentraci CO_2, až 20% pro některé druhy hlívy. Ventilace za účelem snížení CO_2 má i nechtěný efekt v podobě snižování vlhkosti, změn teploty a zvýšeného rizika kontaminace.
- Teplota: V této fázi je potřeba vyšších teplot, ty by však neměly přesahovat 35˚C (obvykle 20-25˚C).
- Světlo: Tma nebo nepřímé světlo v této fázi nemají efekt. Přímé, silné světlo by vedlo k poškození podhoubí.

1.1.2 Fáze nasazování zárodků plodnic

Fáze nasazování primordií (zárodků plodnic) je nejdůležitější částí růstu. Pro spuštění růstu plodnic je třeba změnit hodnoty okolního prostředí.

- Vlhkost: 95-100%, po vytvoření zárodků snížení na 90-95%.
- Větrání: Množství CO_2 by se mělo držet pod hranicí 1000 ppm, nejlépe pod 500 ppm.
- Teplota: Většina odrůd potřebuje pro vytvoření plodnic pokles okolní teploty.
- Světlo: Zde je již potřeba nepřímé světlo o teplotě nejčastěji 6500K.

1.1.3 Fáze růstu plodnic

- Vlhkost: Je třeba snížit oproti fázi nasazování primordií za účelem řízeného odpařování z plodnic. Několikrát denně je vhodno rosit. Pro zvýšení trvanilosti lze několik hodin před sklizní snížit okolní vlhkost.
- Větrání: Pro udržení nízké hladiny CO_2.
- Teplota: Stejná nebo vyšší než v předchozí fázi. Vyšší teplota má za následek rychlejší růst, nižší teplota však produkuje kvalitnější plody.
- Světlo: Nedostatek světla způsobuje protažení třeně a znetvoření klouboku. U některých druhů má množství světla vliv na zbarvení.

1.2 Opakování procesu

Po úspěšné sklizni životnost podhoubí ještě nekončí a běžně je možno opakováním cyklu provést více sklizní. Závisle na druhu to bývá 2-5, většinou se neprovádí více jak 3 sklizně, kvůli klesajícímu výnosu. [25]

1.3 Konkrétní příklad

Výše zmíněné nároky na prostředí k pěstování jsou všeobecné a je potřeba chápat, že každý druh bude mít ideální parametry jiné. Proto je vhodné přizpůsobit prostředí i aktuálně přestovanému druhu.

1.3.1 Hlíva ústřičná

	Kolonizační	**Nasazování plodnic**	**Růst plodnic**
Vlhkost	85-95%	95-100%	85-90%
CO_2	5,000-20,000 ppm	<1,000 ppm	<1000 ppm
Teplota	24˚C	10-15.6˚C	10-21˚C
Světlo	Nemá vliv	1,000-1,500 lux	1,000-1,500 lux
Trvání	12-21 dnů	3-5 dnů	4-7 dnů

Tabulka 1: Ideální podmínky pro růst hlívy ústřičné - Zdroj: [1]

1.3.2 Shiitake

	Kolonizační	Nasazování plodnic	Růst plodnic
Vlhkost	95-100%	95-100%	60-80%
CO_2	<10,000 ppm	<1,000 ppm	<1,000 ppm
Teplota	21-27˚C	10-16˚C	16-18˚C
Světlo	50-100 lux	500-2,000 lux	500-2,000 lux
Trvání	35-70 dnů	5-7 dnů	5-8 dnů

Tabulka 2: Ideální podmínky pro růst shiitake - Zdroj: [1]

2 Existující systémy

Než začneme s návrhem a vytvářením našeho projektu, je potřeba se nejdříve podívat, co se již nabízí a zdali existující produkty nejsou schopny splnit naše požadavky. Analýzou konkurenčních produktů můžeme zhodnotit potřebnost toho našeho, ale také upravit a lépe definovat požadavky podle nedostatků na trhu.

Hledáme tedy systém schopný automaticky řídit klimatické podmínky u více komor pěstírny hub. Dalším požadavkem je možnost centralizovaného dálkového ovládání pomocí mobilní aplikace, nebo webového rozhraní.

Při průzkumu trhu jsem narazil na několik produktů mířených na domácí užití, pro pěstování jednoho bloku podhoubí. Příkladem takového systému je pěstební komora od Shrooly. V těchto případech jde o systémy s automatizovaným zvlhčováním a svícením, neumožňují však použití pro větší pěstební operace. Proto podobné produkty nesplňují naše požadavky a dále se jim věnovat nebudu. [2]

Obrázek 1: Shrooly zařízení pro pěstování hub - Zdroj: [2]

2.1 BoomRoom II

BoomRoom II od společnosti Northspore je pěstební stan umožňující regulaci vlhkosti a proudění vzduchu. Součástí produktu není pouze systém řízení vlhkosti a proudění vzduchu, ale i samotný zvlhčovač a ventilátor. Hlavním problémem je uzavřenost systému, který je limitován na kapacitu jednoho stanu a neumožňuje použití pro větší pěstírny. Všeobecně se jedná o produkt pro malé domácí pěstitele a nesplňuje mnoho mých požadavků. [3]

Obrázek 2: Pěstební stan BoomRoom II - Zdroj: [3]

Funkce:

- Řízení vlhkosti a proudění vzduchu

Nedostatky:

- Neovládá osvětlení a regulaci teploty
- Nastavení udržovaných hodnot pouze přímo na zařízení

Tento produkt nesplňuje mé požadavky, jelikož neřídí všechny potřebné veličiny, nelze rozšířit mimo dodaný pěstební stan a neumožňuje dálkové řízení.

2.2 Fancom Lumina

Řada Lumina od společnosti Fancom jsou klimatické počítače určené pro pěstování hub. V době psaní této práce je nejnovější nabízený model Lumina 762, proto se zaměřím na ten. Jelikož se jedná o komerční produkt cílený na velké farmy, detailní popis fungování u něj není veřejně dostupný. Můžeme však odvodit základní funkce a vlastnosti z článků představujících produkt a přiložené brožury. [4, 26]

Jedná se o počítač schopný řídit až 4 oddělené komory a regulovat v nich proudění vzduchu, vlhkost a teplotu. Nastavení cílových hodnot a kontrola aktuálního stavu probíhají skrze dotykovou obrazovku počítače. Řízení a monitorování je možno provádět i vzdáleně pomocí programu F-Central FarmManager.

Obrázek 3: Klimatický počítač Lumina 762 - Zdroj: [4]

Funkce:

- Řízení teploty, vlhkosti, proudění vzduchu a hladiny CO_2
- Řízení osvětlení podle časových intervalů
- Systém řízení vzduchu rozhoduje, zda použít vnější vzduch, nebo cirkulovat vnitřní. Tím může zamezit nepotřebným tepelných ztrátám.
- Až 6 nastavitelných klimatických profilů

Nevýhody:

- Jedinou výraznou nevýhodu vidím ve velikosti a složitosti systému, což jej udělá nedostupný pro malé a střední pěstitele.

Tento systém splňuje většinu mých požadavků a nabízí řadu funkcí navíc. Liší se v návrhu jednotlivých částí systému, avšak hlavní problém vidím v měřítku pěstování, na které míří. Výrobce neuvádí ceník, ani složitost integrace do prostor pro pěstování a ostatní součásti systému mimo řídicího počítače. Troufám si proto říct, že pokud chceme automatizovat menší, či střední pěstírnu, toto zařízení nebude nejlepší volbou.

2.3 Mycodo

Mycodo je monitorovací a řídicí systém postavený na platformě Raspberry Pi. Jedná se o open-source projekt původně určený k automatizaci pěstování hub, ale díky zájmu veřejnosti se postupně rozrostl a slouží k automatizaci i v jiných oblastech, jako jsou hydroponické systémy, nebo umělá netopýří jeskyně. [5, 27, 28]

Systém je založen na bázi vstupů a výstupů. Jako vstupy umí využít načtená data z připojených senzorů, stavy GPIO pinů i analogové signály. Na výstupech umožňuje nastavení GPIO pinů, generování PWM signálů pro plynulé řízení motorů, ale i spouštění vlastních skriptů.

Pro nastavování systému a vizualizaci snímaných hodnot slouží přizpůsobitelný ovládací panel dostupný pomocí webového prohlížeče. Ten umožňuje zobrazení stavových ukazatelů a vykreslení grafů historie snímaných hodnot.

Funkce:

- Řízení teploty, vlhkosti, proudění vzduchu a hladiny CO_2
- PWM signály pro plynulé řízení ventilátorů a osvětlení
- Uživatelem vytvořené funkce
- Posílání email notifikací
- Přenos obrazu z kamery
- Vzdálený přístup pomocí webového řídicího panelu
- Možnost přizpůsobení a rozšíření
- Založen na Raspberry Pi

Nevýhody:

- Pro technicky nezdatného uživatele složitější zprovoznění
- Každá pěstební komora vyžaduje vlastní řídicí jednotku s databází a virtuálním ovládacím panelem

Všeobecně mohu říci, že systém Mycodo splňuje a v mnohém předčí mé představy a požadavky na automatické řízení klimatu pro pěstování hub. Open-source charakteristika

softwaru umožňuje širokou škálu přizpůsobení jakékoli situaci a potřebě, což vede k nasazení i v oblastech, kde by jiné produkty použít nešly. Hlavní rozdíl mezi Mycodo a mnou zamýšleným systémem je použitý hardware a architektura celého systému. Mycodo využívá jako platformu počítač Raspberry Pi, na kterém je možné spustit jak řízení pěstírny, tak databázi pro záznam hodnot a přístupový panel pro ovládání. Toto řešení umožní provoz celého systému na jediném zařízení, avšak neumožňuje řízení více pěstíren z jedné aplikace. V případě nasazení více pěstíren potřebuje každá vlastní počítač Raspberry Pi, což by bylo finančně náročnější, než řešení pomocí mikrokontrolerů a společného serveru, jak je tomu v mojí verzi systému.

Obrázek 4: Ovládací panel Mycodo - Zdroj: [5]

3 Požadavky

3.1 Funkční požadavky

- Řídicí jednotka snímá hodnoty prostředí pěstírny.
  - Měří teplotu, vlhkost vzduchu a hladinu CO_2.
- Regulace klimatických podmínek na základě naměřených hodnot a nastavených cílových hodnot. Řídicí jednotka upravuje podmínky pomocí připojených zařízení.
  - Spouští ventilaci za účelem držení hladiny CO_2 pod žádanou hodnotou.
  - Spouští zvlhčovač pro udržení nastavené vlhkosti vzduchu.
  - Spouští topení, nebo klimatizaci za účelem udržení nastavené teploty.
- Řídicí jednotka spouští osvětlení podle nastavených časových intervalů.
- Řídicí jednotka zobrazuje naměřené hodnoty prostředí.
- Řídicí jednotka ukazuje nastavené cílové hodnoty.
- Řídicí jednotka se po spuštění připojí k internetu pomocí Wi-Fi.
  - Povede-li se připojení k internetu, probíhá dále popsaná komunikace se serverem.
  - Nezadaří-li se připojení, opakuje se pokus o připojení, a mezitím probíhá řízení dle naposledy stažených parametrů.
- Řídicí jednotka v pravidelném intervalu komunikuje se serverem.
  - Je-li připojena k internetu, odesílá aktuálně naměřené hodnoty.
  - Přijímá aktuálně nastavené cílové hodnoty prostředí.
- Řídicí jednotka si nastaví aktuální čas.
  - V případě úspěšného připojení k internetu získá čas pomocí NTP a aktualizuje čas připojeného RTC modulu.
  - V případě neúspěšného připojení k internetu si aktualizuje čas z připojeného RTC modulu.
- RTC modul pomocí baterie drží aktuální čas i v případě výpadku proudu nebo vypnutí.
- Nově spuštěná řídicí jednotka se registruje na serveru a získá své ID, pod kterým dále komunikuje.
- Po stisku tlačítka na řídicí jednotce se uživateli zobrazí kód pro spárování s uživatelským účtem.
- Server naměřené hodnoty ukládá do databáze pro budoucí vizualizaci.
- Server zprostředkovává správu uživatelských účtů a k nim vázaných řídicích jednotek pěstíren.

- Komunikace mezi serverem a řídicí jednotkou, serverem a mobilní aplikací probíhá skrze REST API.
- Komunikace je zabezpečena použitím protokolu HTTPS.
- Uživatel si v mobilní aplikaci musí vytvořit účet a následně má možnost se k němu přihlásit.
- Aplikace umožní tvorbu pěstebních profilů pro rychlé nastavení prostředí pěstírny.
- K uživatelskému účtu lze přidat jednu či více pěstíren.
- U připojených pěstíren si může uživatel zobrazit grafy stavu naměřených hodnot.
- Připojeným pěstírnám lze měnit cílové hodnoty prostředí.
- Systém umožní více uživatelům měnit nastavení jedné pěstírny.

3.2 Nefunkční požadavky

- Použitý hardware řídicí jednotky je cenově dostupný pro nasazení ve větším měřítku.
- Řídicí jednotka zachovává funkčnost (bez možnosti změny nastavení) i v případě že po spuštění nemá přístup k internetu.
- Řídicí jednotka umožňuje nepřetržitý provoz bez potřeby restartu.
- Mobilní aplikace umožní uživateli pohodlné odvládání jedné, či více pěstíren.

4 Architektura

Za účelem zlevnění a především škálovatelnosti celého systému jsem se rozhodl pro architekturu složenou ze tří dílčích částí systému. Takové rozložení systému umožní nejen dálkové ovládání pěstíren, ale také možnost, že jedna mobilní aplikace může řídit více pěstíren, nebo jedna pěstírna může být řízena více uživateli.

4.1 Řídicí jednotka

Řídicí jednotka se nachází přímo v místě pěstírny. Jejím úkolem je snímání hodnot klima pěstírny pomocí připojeného senzoru a následná úprava stavu klima pomocí připojených zařízení jako jsou zvlhčovač, ventilátor, topení a klimatizace. Jelikož zařízení nejsou integrována do systému, řídí se u nich pouze jejich napájení a nejedná se o zařízení se specifickým rozhraním.

Druhým úkolem řídicí jednotky je odesílání naměřených hodnot na server a přijímání aktuálně nastavených cílových hodnot pěstírny.

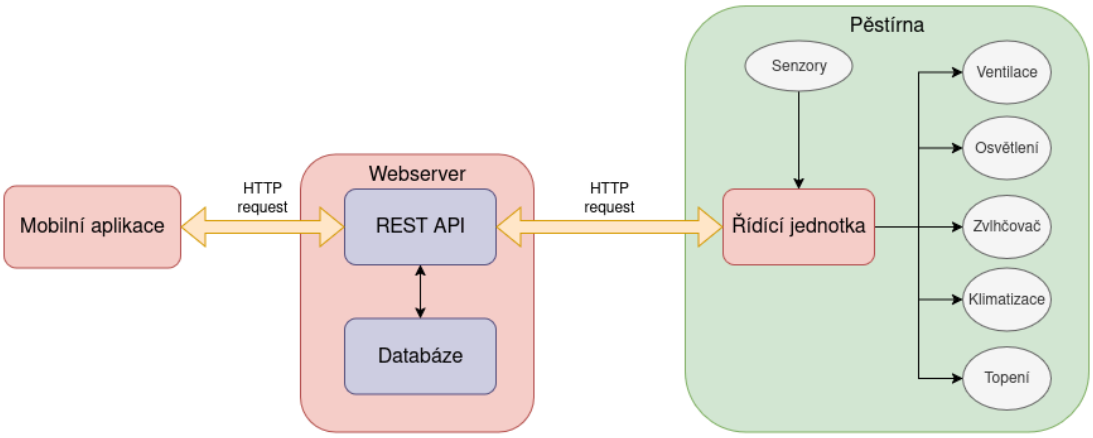
Jednotka musí také umožnit získání a zobrazení kódu pro spárování s uživatelským účtem.

4.2 Webserver

Serverová část systému slouží jako sdílená databáze pro všechny řídicí jednotky a uživatele. Zprostředkovává výměnu dat mezi řídicí jednotkou a mobilní aplikací.

4.3 Mobilní aplikace

Mobilní aplikace slouží jako rozhraní mezi uživatelem a jím spravovanými pěstírnami. Umožňuje jak zobrazení stavu prostředí pěstírny, tak změnu hodnot, které v pěstírně řídicí jednotka udržuje.



Obrázek 5: Architektura systému - Zdroj: vlastní

5 Řídicí jednotka

První a nejdůležitější částí systému je řídicí jednotka. Jejím úkolem není pouze čtení klimatických hodnot ze senzoru, ale i jejich ovlivňování pomocí připojených zařízení.

5.1 Hardware

Než začneme psát řídicí software, je potřeba určit, na jaké platformě budeme řídicí jednotku stavět a jaký další hardware budeme potřebovat. Tato rozhodnutí ovlivní nejen cenu zařízení, ale především složitost vývoje a možnosti funkcí, které půjde implementovat.

Pro řízení vstupů a výstupů máme na výběr z více možností, které nám umožňují výrazně rozdílný přístup k vývoji zařízení. Nejprve je potřeba si určit co od řídicí jednotky požadujeme za funkčnost.

Požadavky:

- Možnost řízení GPIO pinů
- I^2C komunikace pro připojení senzorů a dalších modulů
- Možnost připojení k internetu

Když máme základní požadavky na hardware, je nejprve potřeba rozhodnout, jestli postavit řídicí jednotku na jednodeskovém počítači (SBC), nebo na mikrokontroléru (MCU), jelikož naše požadavky splňují zástupci obou skupin. [29]

5.1.1 Jednodeskový počítač Raspberry Pi

Původním plánem pro tuto práci bylo využít pro řízení jednodeskového počítače Raspberry Pi. Jedná se o nejznámějšího zástupce single-board počítačů (SBC), který je hojně kutily používán pro automatizaci široké řady projektů. Jak už název kategorie napovídá, jedná se o kompletní počítač skládající se z jediné desky tištěných spojů. Jinak tomu je u modelů řady Pico, ty patří do dále zmíněné kategorie mikrokontrolérů. Raspberry Pi je postaven na architektuře ARM a jedná o se o počítač schopný provozovat plnohodnotnou distrubuci Linux, což je hlavní výhodou. Tato kombinace nabízí možnost spuštění široké škály programů vytvořených v libovolném programovacím jazyce jako je například Python. Umožňuje také vzdálené připojení pro administraci, nebo všeobecné užívání. [6]

Obrázek 6: Raspberry Pi 5 - Zdroj: [6]

Ve srovnání s mikrokontroléry popsanými v další části můžeme říci, že v závislosti na způsobu použití, můžou mít jednodeskové počítače několik nevýhod. Za hlavní z nich bych považoval větší velikost a vyšší pořizovací cenu. Další nevýhodou v automatizaci může být to, že u Raspberry Pi nemůžeme zaručit real-time vykonání operací, jelikož operační systém Linux, pod kterým naše aplikace běží, může způsobit zpoždění vykonání operací. Pokud je naším hlavním cílem práce s několika vstupy a výstupy, potom se může podobný počítač jevit jako příliš rozsáhlé zařízení, jehož potenciálu nevyužijeme a můžeme stejného efektu dosáhnout s levnějšími mikrokontroléry.

Jinak by tomu však bylo, pokud bychom na počítači provozovali i ostatní části systému, jako je server s databází a ovládací panel pro vzdálený přístup, jako tomu je u systému Mycodo. [5]

5.1.2 Mikrokontroléry

Pokud nepotřebujeme pro provoz počítač s plnohodnotným operačním systémem, jako je Linux, nepotřebujeme provozovat více programů zároveň a nemusíme dělat výpočetně náročné operace vyžadující více jader, nabízí se možnost využití mikrokontroléru. Stejně jako SBC, mikrokontroléry také obsahují procesor, operační paměť a periferie pro vstup a výstup. V tomto případě se však jedná o více integrovaný systém obsažený v jednom čipu. Tímto dosahují výrazně nižší ceny i menší velikosti, avšak způsob jejich využití je výrazně odlišný od SBC. Jejich výpočetní výkon a operační paměť jsou o poznání nižší a neumožňují provoz plnohodnotného operačního systému, jako je Linux. Namísto toho slouží k provozu jednoho konkrétního programu přímo na jednotce MCU, nebo ve speciálním real-time operačním systému. Jsou proto ideálním řešením v situacích, kdy potřebujeme provádět konkrétní řídicí

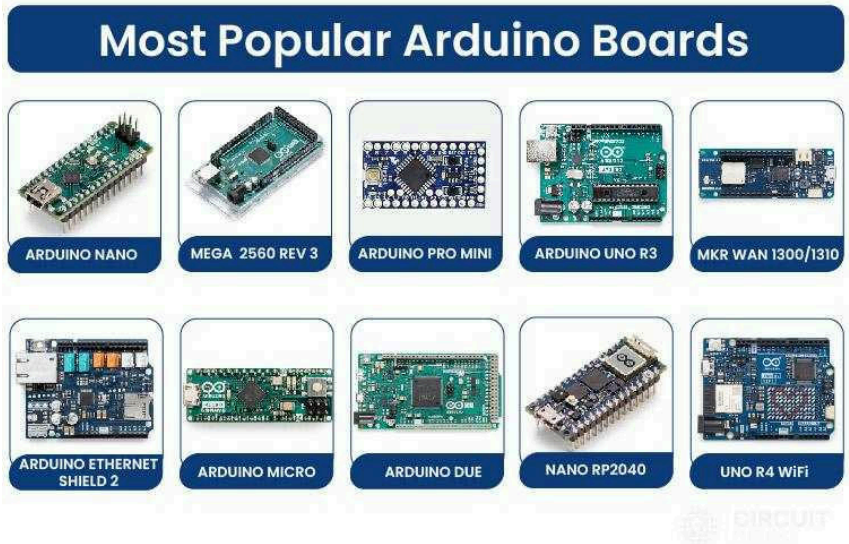
operace v přímém čase. Jednou nevýhodou MCU je omezený výběr programovacích jazyků, jaké lze na těchto platformách použít. Nejčastěji se používá C/C++.

V oblasti mikrokontrolérů máme na výběr z několika platforem nabízejících různou funkčnost. Některé z nich si proto uvedeme.

Arduino

Arduino je open-source projekt, který vznikl v roce 2005 a zabývá se vývojem mikrokontrolérů a softwaru pro jejich programování. Cílem je vytvoření cenově dostupných zařízení, která by i začátečníkům umožnila vytvářet systémy interagující s okolím skrze senzory a řízením motorů a dalších zařízení. Programování je možné v jazycích C a C++, MicroPython, nebo Arduino jazyce založeném na C++. Pro ulehčení programování Arduina lze použít integrované vývojové prostředí Arduino IDE ulehčující správu desek, knihoven a nabízející debugovací funkce. [30]

V době psaní této práce nabízí Arduino desítky různých desek zapadajících do 4 kategorií: Nano, MKR, Classic a Mega. Některé z produktů dnes již nabízí funkce, jako je Bluetooth, Wi-Fi, nebo zabudované teplotní, tlakové, vlhkostní senzory. Jednotlivé desky se dají ještě rozšířit o připojené senzory a rozhraní, některé z nich jsou dostupné přímo od organizace Arduino. [31]



Obrázek 7: Nejpopulárnější desky Arduino - Zdroj: [7]

Jedná se o velice populární řešení automatizačních projektů a díky velké komunitě uživatelů se jedná o vhodnou volbu pro širokou škálu projektů.

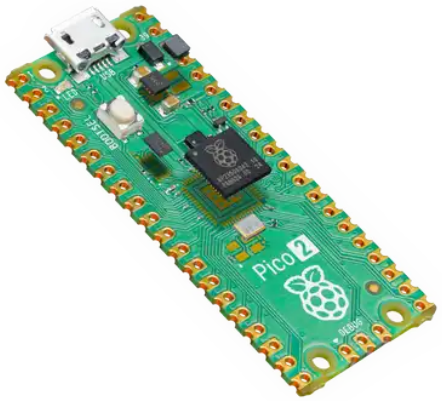
STM32

Další rodinou mikrokontrolérů je STM32. Jedná se o širokou řadu zařízení založených na 32 bitové architektuře Arm procesorů o různém výkonu od 24 až po 800MHz závisle na modelu. Díky velké nabídce modelů je možné vybrat ten správný pro daný projekt, ať už jde o STM32L4 nabízející velice nízkou spotřebu pro provoz z baterie, výkonnější STM32F7 pro náročnější výpočetní operace, nebo STM32WB umožňující bluetooth komunikaci. [32]

STM32 je vhodnou volbou pro automatizaci a nabízí se v mnoha verzích od cenově dostupných jednoduchých mikrokontrolérů až po výkonné modely vhodné pro neuronové sítě nebo grafické aplikace. Na rozdíl od Arduino jsou však komplexnější a práce s nimi je složitější na naučení. Pro můj projekt však STM32 není vhodnou volbou, jelikož internetové připojení nabízejí pouze některé modely, a to formou ethernetu, nikoli Wi-Fi.

Raspberry Pi Pico 2

Na rozdíl od většiny ostatních mikrokontrolérů nabízí Raspberry Pi Pico dvoujádrový procesor ARM Cortex-M33 nebo RISC-V Hazard3 operující na frekvenci 150MHz, umožňující lepší multitasking. Po startu operují vždy pouze jádra zvolené architektury. Spolu s 264kB operační paměti, až 16MB flash paměti se Raspberry Pi Pico 2 nabízí jako řešení i pro náročnější operace. Podporuje programování z jazycích C/C++ a MicroPython. Verze Raspberry Pi Pico 2 W nabízí možnost připojení pomocí Bluetooth a Wi-Fi, což ji dělá jednou z možných řešení mnou navrhovaného systému. [8]



Obrázek 8: Jednočipový počítač Raspberry Pi Pico 2 - Zdroj: [8]

Teensy

Na seznamu populárních mikrokontrolérů by neměly chybět ani počítače Teensy od společnosti PJRC. Nabízí výrazně výkonnější procesor než většina konkurenčních mikrokontrolérů a matematický koprocesor pro operace s čísly s pohyblivou řádovou čárkou. Výhodou je i kompatibilita s vývojovým prostředím a knihovnami určenými pro Arduino. Vhodným užitím jsou výpočtově náročnější operace, vývoj zařízení komunikujících přes USB, nebo zpracování audia. [33]

Pro tento projekt však platforma Teensy není vhodná volba, díky vyšší ceně a především pro nemožnost připojení k Wi-Fi bez rozšiřujícího modulu.

ESP32

ESP32 je rodina cenově dostupných mikrokontrolérů obsahujících podporu Wi-Fi a Bluetooth. Díky těmto možnostem bezdrátové komunikace jsou hojně využívány v oblasti IoT, automatizace domácností a dalších projektech, vyžadujících bezdrátové připojení.

Model	Architektura	Počet jader	Frekvence	SRAM	Flash
ESP32	Xtensa	2	240 MHz	520 KB	až 4 MB
ESP32-S2	Xtensa	1	240 MHz	320 KB	až 4 MB
ESP32-S3	Xtensa	2	240 MHz	512 KB	až 8 MB
ESP32-C3	RISC-V	1	160 MHZ	400 KB	až 4 MB
ESP32-C6	RISC-V	1	160 MHZ	512 KB	až 4 MB
ESP32-P4	RISC-V	2 + 1	400 MHZ, 40 MHz	768 KB	externí

Tabulka 3: Parametry vybraných verzí ESP32 - Zdroj: [9]

Programovat ESP mikrokontroléry je možné ve více jazycích. První možností je použít Arduino C/C++ uvnitř Arduino IDE. Další způsob je použití frameworku Espressif IDF, který v C/C++ zprostředkovává funkce hardware rozhraní a real-time operačního systému FreeRTOS. Mezi další možnosti patří MicroPython, JavaScript nebo LUA.[34]

Pro podporu IEEE 802.11b/g/n bez nutnosti rozšiřovacích modulů, nízkou prodejní cenu, velkou komunitu uživatelů a možnost použití existujících C/C++ knihoven jsem se rozhodl v mé práci použít právě ESP32-C3.

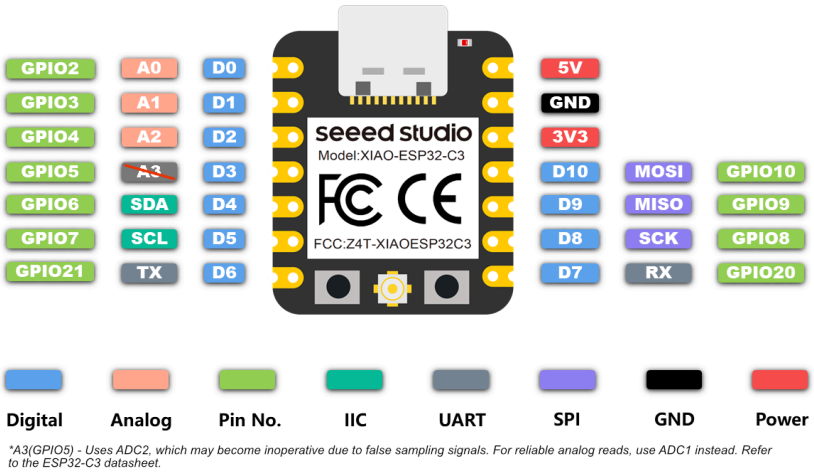
5.1.3 ESP32-C3

Pro provoz řídicí jednotky jsem nakonec z dostupných mikrokontrolérů vybral ESP32-C3, představený v roce 2020 společností Espressif. Jedná se o jednojádrový MCU s nízkou spotřebou a zabudovanou podporou 2.4 GHz Wi-Fi a Bluetooth 5. Na rozdíl od tradičních modelů, jako jsou ESP32, nebo ESP32-S3, je založen na architektuře RISC-V. Pro účely této práce hraje architektura minimální roli, jelikož je projekt tvořen v jazyce C, nikoli Assembly, tudíž rozdílné instrukce programátor nepocítí. Avšak díky tomu, že embedded zařízení poslední roky přecházejí z uzavřené architektury ARM k open-source architektuře RISC-V, rozhodl jsem se také jít touto cestou. [35]

Přesněji jsem se rozhodl použít vývojovou desku Seeed Studio XIAO ESP32-C3. Výhodou této desky je velice malý půdorys, tvořící pouze 21 x 17,8mm. Pro výběr tohoto modelu mě motivovaly dobré recenze ohledně kvality zpracování, ale především přítomnost externí antény pro bezdrátové připojení, která zaručí silnější signál než SMD anténa, jakou používá levnější model ESP32-C3 Super Mini. Jedná se o menší model vývojové desky, který zprostředkovává přístup k pouze 11 z 22 programovatelných GPIO pinů, kterými ESP32-C3 disponuje. To však pro účely této práce dostačuje. [10]

Procesor	ESP32-C3 32-bit RISC-V @160MHz
Paměť	400 KB SRAM, 4 MB Flash
Bezdrátové připojení	WiFi a Bluetooth 5 (BLE)
Rozhraní	I^2C/UART/SPI

Tabulka 4: Parametry ESP32-C3 - Zdroj: [10]



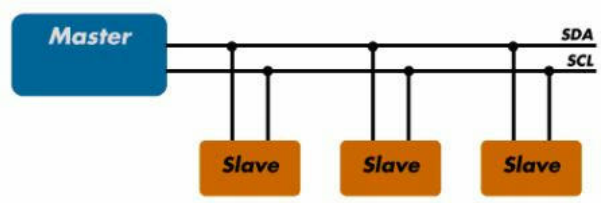
Obrázek 9: Základní rozložení GPIO pinů XIAO ESP32C3 - Zdroj: [10]

Další vlastnosti:

- Připojení skrze USB-C pro napájení, flashování a debugging
- 11 GPIO pinů s možností konfigurace I²C, SPI a UART na libovolný pin na software úrovni
- Možnost připojení, napájení a provozu z lithiové baterie
- Vstupní napětí 5V
- Pracovní napětí 3.3V

5.1.4 I²C sběrnice

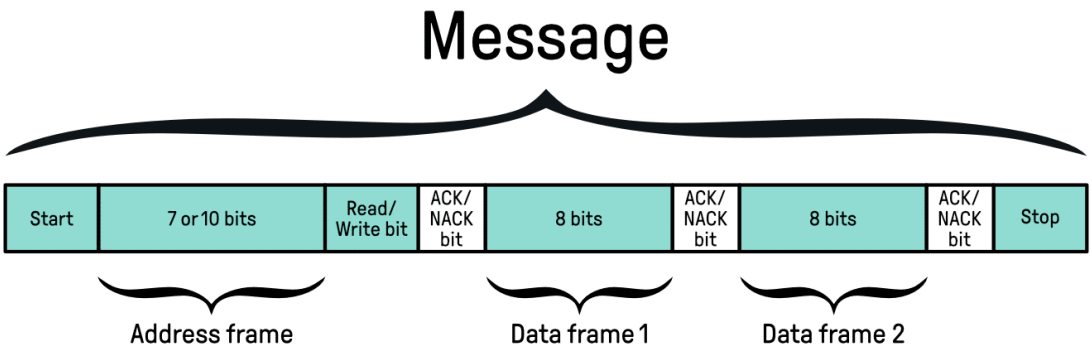
Ještě než si určíme všechny moduly a zařízení připojené k našemu mikrokontroléru, chtělo by to si něco říci o sběrnici I²C, kterou budeme používat ke komunikaci s nimi. Jedná se o sériovou sběrnici pracující na dvou vodičích. První je signálový vodič (**SDA** - serial data), druhý je hodinový vodič (**SCL** - serial clock). Komunikující zařízení jsou rozdělena způsobem master–slave s tím že master zařízení může být více, ale typicky bývá pouze jedno. V našem případě master bude ESP32-C3 a slave budou všechna zařízení s ním propojená pomocí sběrnice I²C. [11, 36]



Obrázek 10: Zapojení zařízení na I²C sběrnici - Zdroj: [11]

Zařízení v roli master má na starost vysílání hodinového signálu a je jediné, které může začít komunikaci. Každé zařízení podporující I²C má definovanou 7 bitovou adresu. Zařízení master

začíná komunikaci vysláním 7 bitové adresy slave zařízení zakončené 1 bitem určujícím, zda-li půjde o operaci zápisu (1), nebo čtení (0).



Obrázek 11: Složení I^2C komunikace - Zdroj: [12]

Zařízení v roli slave naslouchá na sběrnici a reaguje pouze na svoji adresu. Po odeslání těchto 8 bitů následuje odpověď v podobě jednoho ACK bitu, potvrzující přijetí předchozího bytu. Následně odesílatel (určen předchozím bitem) posílá 1 byte dat a příjemce mu na něj opět odpovídá jedním ACK bitem. Takto probíhá komunikace opakující se sekvencí 9 bitů.

Tímto způsobem může master zařízení komunikovat s více slave zařízeními na jedné sběrnici. I^2C nabízí i další funkce, jako je broadcast vysílání, nebo 10 bitové adresování. I^2C umožňuje přenos o frekvenci až 3.4 Mbps, ale většina zařízení dnes používá 100 Kbps nebo 400 Kbps a vyšší rychlosti nepodporuje.

5.1.5 Senzory

Pro potřeby řízení naší pěstírny potřebujeme snímat 3 hodnoty, těmi jsou teplota, vlhkost a hladina CO_2. Tato data bychom mohli získávat z jednotlivých senzorů, avšak to není potřeba, jelikož při použití senzoru, jako je SCD41 od společnosti Sensirion získáme měřením nejenom hladinu CO_2, ale také teplotu a vlhkost. Pokud hledáme cenově dostupné řešení přesného měření hladiny CO_2, tak nejlepší možností jsou senzory SCD30, SCD40 a SCD41 od společnosti Sensirion. [13–15]

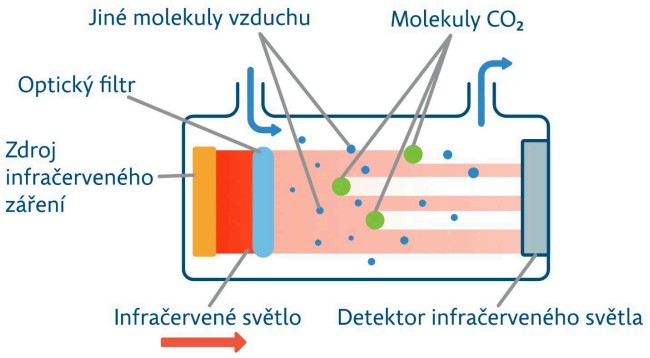
	SCD30	SCD40	SCD41
Technologie	NDIR	Fotoakustický NDIR	Fotoakustický NDIR
Rozsah	400 - 10000 ppm	400 - 2000 ppm	400 - 5000 ppm
Přesnost	±30.0 ppm ±3.0 %m.v.	±50.0 ppm ±5.0 %m.v.	±50.0 ppm ±3.0 %m.v.

Tabulka 5: Senzory CO_2 od společnosti Sensirion - Zdroj: [13–15]

5.1.5.1 SCD30

SCD30 je dvoukanálový senzor CO_2 založený na technologii NDIR (Nondispersive infrared sensor). [13]

Funguje na principu absorpce infračerveného světla molekulami CO_2. Do měřící komůrky vniká měřený vzduch, na jedné straně je zdroj infračerveného světla o vlnové délce, kterou pohlcují molekuly CO_2 a na opačném konci se nachází detektor infračerveného světla. Čím větší je koncentrace CO_2, tím méně infračerveného světla naměří detektor a z hodnot vypočítá hladinu CO_2. [16]



Obrázek 12: Princip fungování NDIR senzoru - Zdroj: [16]

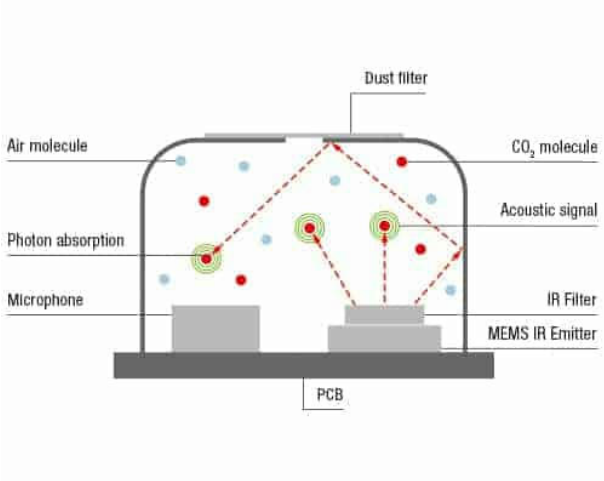
Tato metoda měření probíhá v prvním kanálu, zatímco druhý je uzavřený a složení plynu v něm se od výroby nemění. Jelikož se absorpční vlastnosti plynu nemění, senzor hodnoty z této druhé komory používá pro korekci naměřených hodnot v otevřené komoře. Tímto se zajistí, že senzor nepotřebuje tak častou kalibraci. [37]

Tato metoda měření je přesná a senzory jako SCD30 jsou i cenově dostupné, avšak technologie měření představuje omezení na minimální velikosti senzoru. Tento problém řeší fotoakustická NDIR metoda měření, jakou najdeme například na senzorech SCD40 a SCD41.

5.1.5.2 SCD40 a SCD41

První věc, která nás zaujme na senzorech SCD40 a SCD41, je samotná velikost. Ta tvoří pouhých 10,1 x 10,1 x 6,5 mm. Oba senzory mají stejný princip fungování a typ konstrukce, rozdíl tvoří především rozsah měřitelných hodnot, kde SCD41 vyhrává.

Zatímco tradiční NDIR senzory spoléhají na vyzařování a snímání infračerveného světla, fotoakustické NDIR senzory jako SCD40 a SCD41 používají zdroj infračerveného světla a mikrofon. Oba se nachází v takřka uzavřené měřící komoře zatěsněné pouze prachovým filtrem. Zdroj vyzařuje infračervené světlo o vlnové délce, jaká je pohlcována molekulami CO_2. Ty absorbují energii, zahřívají se a rozpínají, čímž způsobují zvýšení tlaku v měřící komoře. Modulací světla se způsobuje periodická změna tlaku, která je měřena mikrofonem. Z naměřených hodnot se opět vypočítá obsah CO_2.[17]



Obrázek 13: Princip fungování fotoakustických senzorů CO_2 - Zdroj: [17]

Jelikož senzor SCD41 disponuje dostatečnou přesností a zároveň je cenově výrazně dostupnější než přesnější alternativy od jiných výrobců, jevil se jako ideální řešením pro můj projekt. Navíc jedním měřením získáme všechny potřebné veličiny pro řízení pěstírny. Propojení řídicího ESP32-C3 a použitého senzoru bude realizováno pomocí I^2C sběrnice.

5.1.6 Display

Aby uživatel mohl vidět stav hodnot pěstírny i bez mobilní aplikace, připojil jsem k mobilní jednotce OLED modul SSD1306. Jedná se o jednobarevný OLED panel o úhlopříčce 0.96″ a rozlišení 128x64. Zápis zobrazených dat na modulu je opět prováděn skrze I^2C sběrnici. [38]

5.1.7 RTC modul

Jelikož pěstírna musí řídit i intervaly osvětlení, musí vědět aktuální čas. V bezporuchovém stavu se po spuštění přihlásí k Wi-Fi a pomocí Simple Network Time Protocol (SNTP) zjistí aktuální čas. Nastane-li však situace, kdy po spuštění řídicí jednotky nedojde k úspěšnému připojení k internetu a získání času, část řídicí osvětlení nebude vědět, jaký je zrovna čas a nemůže spolehlivě vykonávat svou práci.

Tomuto se dá zabránit použitím real-time clock (RTC) modulu. Jedná se o modul umožňující zápis a čtení aktuálního data a času, který pak udržuje pomocí krystalového oscilátoru. Konkrétně jsem vybral model DS1307, který nabízí pro naše účely dostatečnou přesnost za nízkou cenu. Součástí modulu je lithiová baterie LIR2032, na kterou v případě výpadku napájení přejde a pomocí ní dále udržuje aktuální čas. Ten si pak ESP32 může z modulu načíst, je-li potřeba. [39]

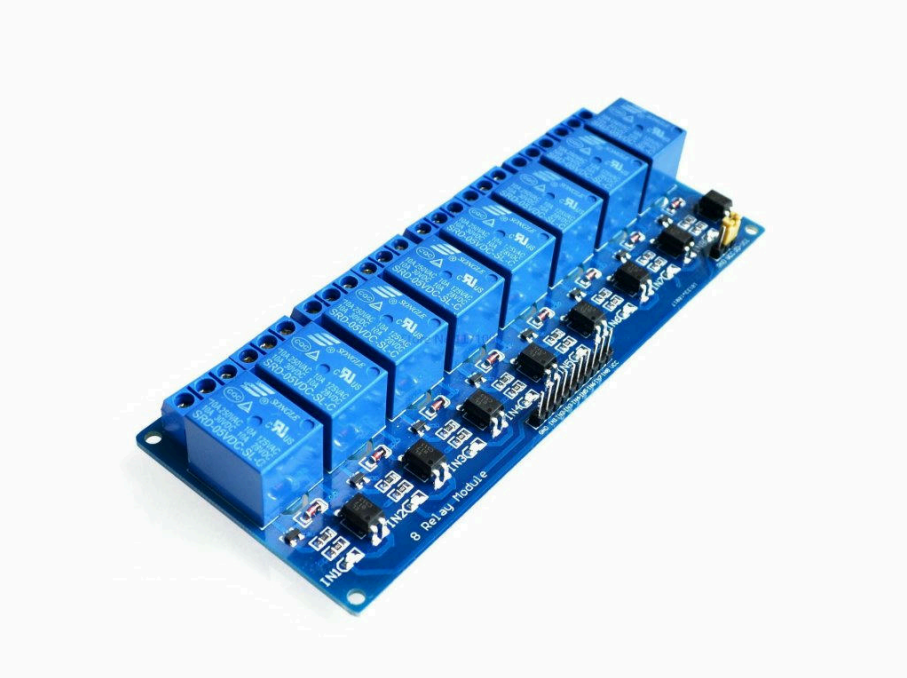
Nedostatkem tohoto modulu je nepřesnost způsobující posun hodin přibližně o 5 minut v měsíci. To v případě této práce však nevadí, jelikož hlavním úkolem modulu je udržet systém v provozu v případě neúspěšného připojení k internetu po výpadku a není cílem držet přesný čas po dlouhé intervaly neaktivity.

Komunikace s ESP32 probíhá opět pomocí I^2C sběrnice.

5.1.8 Relé modul

Koncept řízení pěstírny nepočítal s integrací zařízení ovlivňujících klimatické podmínky, ale předpokládal, že bude použito libovolných produktů, pro které bude řídicí jednotka pouze zapínat a vypínat napájení. Jelikož potřebujeme spouštět střídavé napětí 230V a naše ESP32-C3 má výstupy pouze 3.3V s maximálním odběrem 700mA, budeme potřebovat elektromagnetické relé.

Zapojení si můžeme usnadnit pomocí relé modulů obsahující jedno, nebo více nezávisle řízených relé. Jelikož potřebujeme řídit celkem 5 samostatných zařízení (topení, klimatizace, zvlhčovač, ventilátor a osvětlení), vybral jsem modul s nejbližším vyšším počtem relé, což bylo 8. To umožňuje přivést napětí 5V, kterým je napájeno ESP32 a použít ho pro napájení cívek potřebných k aktivaci relé. Dále přivedeme 5 vodičů logicky reprezentující 5 ovládaných zařízení z GPIO pinů našeho ESP32 na 5 vstupních pinů na relé modulu. Logická 0 na těchto vodičích spíná relé. Každé relé má vlastní LED diodu reprezentující jeho stav. [18]



Obrázek 14: Použitý relé modul od LaskaKit - Zdroj: [18]

Skrze jednotlivá relé poté můžeme zapojit střídavé napětí 230V do zásuvek, ke kterým připojíme naše řízená zařízení.

5.1.9 Připojená zařízení

Hlavním cílem práce je navrhnout systém schopný řídit připojená zařízení za základě naměřených hodnot a nastavených cílových hodnot. Není cílem vytvořit integrovaný systém pěstební komory, který by v sobě obsahoval jak řídicí systém, tak samotná zařízení regulující klima. Proto tato práce počítá s výstupem řídicí jednotky v podobě relé modulů, kde každý představuje jedno zařízení, jako je topení, nebo ventilátor. To uživateli vybavujícímu pěstírnu umožní buď přímé zapojení napájení skrze výstupní relé moduly řídicí jednotky, nebo zapojení zásuvek napájených skrze řízené relé moduly a následné zapojení řízených elektrospotřebičů do zásuvek.

Tento způsob přináší výhodu v možnosti použít jakákoli zařízení pro regulaci prostředí, a díky tomu je možné se přizpůsobit potřebám různých pěstebních prostor. Také se nabízí možnost automatizace pěstíren, které již nějaké vybavení měly, pouze bylo připojeno například skrze časové spínače.

Toto řešení však přináší i nevýhody spojené s potřebou nalezení vhodné konfigurace pro daný prostor a následným laděním jak zařízení ovlivňující klima, tak řídicí jednotky, aby byla schopná stabilně udržovat požadované hodnoty.

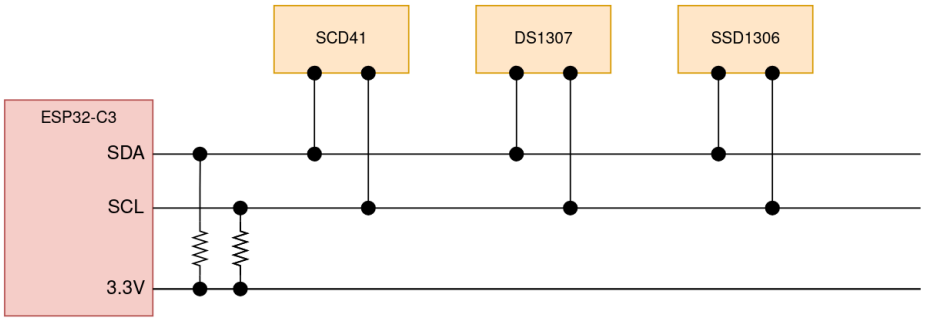
Další nevýhodou tohoto řešení je nemožnost plynulé regulace řízených elektrospotřebičů. Například u ventilátoru by se dalo střídavé zapínání a vypínání nahradit řízením pomocí pulzně šířkové modulace (PWM). U motorů, topných těles, nebo osvětlení se jedná o populární způsob řízení výkonu spočívající v pravidelném spínání a odepínání napájení o frekvenci od desítek po desetitisíce Hz. Poměr napájeného a nenapájeného času poté tvoří procento výkonu zařízení. Tímto bychom mohli dosáhnout, že například ventilátor by mohl běžet trvale a pouze podle potřeby měnit rychlost. [40]

Tento způsob řízení by na ESP32 šel implementovat, jelikož například ESP32-C3 má 6 kanálů pro PWM výstup. Jelikož však nemůžeme takové zařízení zapojit přímo k ESP32, bylo by třeba použít driver pro řízení motoru. [35]

5.1.10 Zapojení

Pro správné připojení modulů k ESP32 je potřeba řídit se především piny 5V, GND a 3,3V, ostatní mohou mít speciální funkci, ale ta se pomocí přepínatelné matice dá nastavit na libovolný pin až v softwaru.

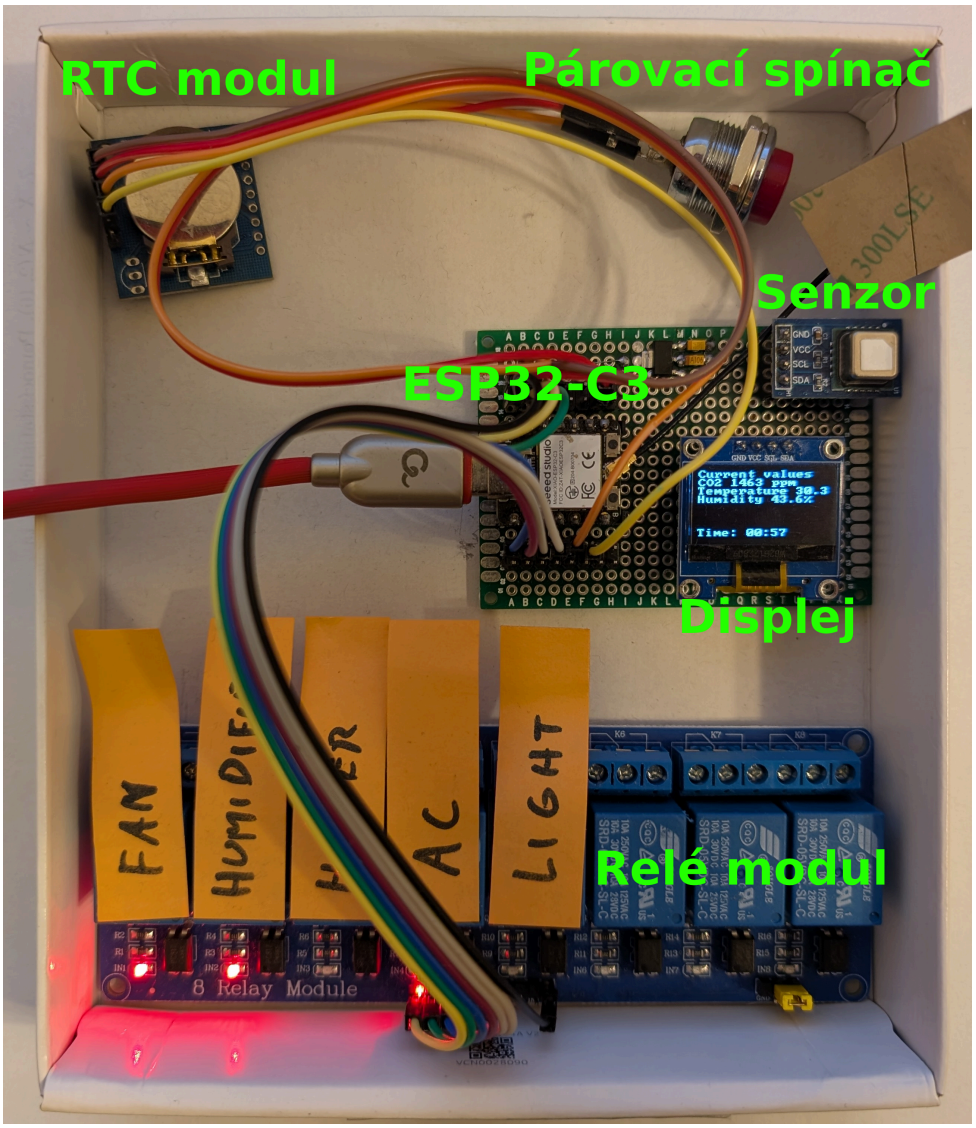
Důležitá pro tento projekt byla sběrnice I^2C. Tu jsem ponechal na GPIO pinech 6 a 7. Pro její fungování je potřeba přidat *pullup* rezistory spojující SDA a SCL vodič s 3,3V linkou. Dále již každý modul stačí připojit k SDA a SCL lince a I^2C rozpozná moduly podle jejich adresy.



Obrázek 15: Zapojení I^2C sběrnice - Zdroj: vlastní

Pro spínání relé modulu byly použity GPIO 2-5 a 10. Ty však lze v konfiguračním souboru změnit.

Spínač sloužící k párování je připojen mezi GPIO 20 a GND.



Obrázek 16: Zapojení prototypu řídicí jednotky - Zdroj: vlastní

5.2 Software

Pro programování mikrokontroléru ESP32-C3 jsem zvolil Espressif IoT Development Framework (ESP-IDF). Jedná se o oficiální vývojový framework pro ESP32, který v jazycích C a C++ nabízí knihovny pro práci s hardwarem ESP32. Výrazně tak ulehčuje operace, jako jsou práce s Wi-Fi, Bluetooth, nebo se vstupními, či výstupními periferiemi. Programátor se tak může soustředit na svůj projekt a nemusí řešit operace spojené s komunikací s hardwarem. [41]

Tyto funkce byly pro mou práci klíčové, jelikož tato práce zakládá na připojení k Wi-Fi a následné komunikaci pomocí HTTP, která je zabezpečena pomocí TLS certifikátu. Toto vše lze realizovat pomocí knihoven obsažených v ESP-IDF.

Další důležitou částí ESP-IDF je operační systém FreeRTOS. Jedná se o drobný operační systém reálného času, který dnes již podporuje i dvoujádrové mikrokontroléry jako je ESP32-S3, nebo ESP32-P4. Mezi hlavní činnosti patří správa tasků (vlákno definované funkcí), plánování procesů, synchronizace a správa paměti. [42]

Pomocí správy tasků jsem mohl vytvořit několik na sobě nezávislých modulů, kde každý má své vlákno a prioritu. Pokud jsem tedy potřeboval mezi vykonáváním nějaké cyklické akce čekat, ostatní moduly systému to nijak neovlivnilo.

Díky několika současně běžícím modulům systému bylo potřeba využít další funkce FreeRTOS, jako jsou semafory pro synchronizaci přístupu ke sdílené paměti, nebo zasílání upozornění mezi tasky.

5.2.1 Vývojové prostředí

Po úvodních komplikacích s konfigurací ESP-IDF ve vývojovém prostředí CLion od společnosti Jetbrains jsem byl donucen přejít do editoru VSCode.

Po instalaci editoru jsem do něj instaloval rozšíření ESP-IDF, které ulehčí vytvoření a nastavení nového projektu pro ESP32. Do editoru přidává menu pro volání základních operací potřebných k vývoji, jako je building nebo flashing. Umožňuje také monitorování výstupu mikrokontroléru po sériové lince, kterou posílá přes USB. [43]

Obrázek 17: Menu přidané rozšířením ESP-IDF - Zdroj: vlastní

5.2.2 Komponenty

Architektura programu se skládá z jednotlivých komponentů, kde každý má na starost určitou funkčnost systému. Pokud má komponent vykonávat cyklicky nějakou operaci po celou dobu provozu systému, spouštíme jeho hlavní funkci pomocí funkce *xTaskCreate*, čímž se vytvoří vlastní vlákno pro funkci a nebude blokovat ostatní tasky. Toto se nám hodí například v komponentu, kde chceme pravidelně načítat hodnoty ze senzoru.

```
xTaskCreate(register_device_task, "register_device_task", 4096, NULL, 5,
NULL);
```

Zdrojový kód 1: Vytvoření tasku pro registraci zařízení

Jelikož jsem ESP32 programoval v jazyce C, každý komponent, který není součástí frameworku ESP-IDF, musel být součástí zdrojových kódů projektu. To probíhalo typicky pomocí příkazu *git clone* a adresy repozitáře, ve kterém se zdrojové kódy nachází. Zdrojové kódy pro obsluhu jednotlivých částí systému jsou umístěny v individuálních adresářích pod adresářem components.

Ke staženým komponentům bylo potřeba napsat funkce, které je budou využívat a budou tvořit funkčnost dané části systému. Tyto zdrojové kódy jsem ponechal v adresáři s komponentem, který využívají.

Každý komponent má vlastní soubor *CMakeLists.txt*, ve kterém jsou specifikovány zdrojové kódy ke kompilaci a závislosti na dalších komponentech, které jsou v daném komponentu používány. Tyto soubory říkají CMake, jak má program sestavit.

```
set(EXTRA_COMPONENT_DIRS components/esp-idf-lib/components)

idf_component_register(SRCS "main.c"
                       INCLUDE_DIRS "."
                       REQUIRES scd4x ssd1306 driver i2c wifi spiffs
environment_control DS1307 communication state)
spiffs_create_partition_image(storage ../partition FLASH_IN_PROJECT)
```

Zdrojový kód 2: Ukázka CMakeLists.txt kořenového komponentu main

5.2.3 Perzistence dat

Jelikož řídicí jednotka potřebuje pro operaci řadu konfiguračních dat, jako jsou údaje o přístupovém bodu Wi-Fi, nebo adresa serveru, se kterým komunikuje, bylo vhodné tyto informace někde spravovat a nemít je pouze ve zdrojovém kódu.

Pro tento účel byl použit jednoduchý souborový systém SPIFFS, který umožňuje udržovat menší soubory ve flash paměti zařízení. Ve vstupní funkci tak proto jako první vytváříme SPIFFS oddíl, který následně budeme používat pro čtení a zápis konfigurace pěstírny. [44]

```
esp_vfs_spiffs_conf_t config = {
    .base_path = "/storage",
    .partition_label = NULL,
    .max_files = 5,
    .format_if_mount_failed = true
};

esp_err_t result = esp_vfs_spiffs_register(&config);
```

Zdrojový kód 3: Vytvoření oddílu uložiště

V tomto uložišti jsem vytvořil celkem 4 soubory:

- **config.json** - Základní nastavení jako je přístupový bod Wi-Fi, nebo adresa serveru
- **environment_settings.json** - Cílové hodnoty prostředí pěstírny
- **output_map.json** - Mapa GPIO pinů použitých pro řízení konkrétního zařízení (ventilátor, topení, …)
- **cert.pem** - Vygenerovaný TLS certifikát serveru

Soubory slouží nejen jako přehlednější konfigurace, ale také se do nich dá zapisovat. Takže pokud řídicí jednotka přijme ze serveru změnu konfigurace, zapíše ji do konfiguračního souboru a v případě restartu a neúspěšném připojení k internetu může nadále pracovat podle poslední přijaté konfigurace.

5.2.4 Správa konfigurace

Jelikož konfigurace zařízení a cílové hodnoty pěstírny jsou hodnoty, které potřebuje více částí systému, nebylo by vhodné, aby pokaždé, když je například potřeba *device_id* musel systém otevírat *config.json* soubor a získávat z něj hledanou hodnotu. Proto jsem v *config.h* vytvořil strukturu, do které se hodnoty při startu načtou a ostatní části systému je pak budou moci využít.

Protože na systému běží několik vláken zároveň, bylo potřeba hodnoty ošetřit strukturou mutex. Ta zajistí, že se sdílenou pamětí nemůže manipulovat více vláken zároveň.

```
typedef struct{
    char wifi_ssid[32];
    char wifi_password[32];
    char device_id[37];
    char server_url[128];
    int sensor_post_interval;
} device_config_t;

typedef struct{
    device_config_t device_config;
    SemaphoreHandle_t mutex;
} safe_device_config_t;
```

Zdrojový kód 4: Struktura představující konfiguraci za běhu systému

Tuto strukturu bylo nutno doplnit o funkce, které s ní pracují. Funkce *config_init* zajistí vytvoření mutexu a načtení dat ze souboru do struktury. Funkce *config_get* bezpečně vykopíruje obsah struktury a slouží ostatním částem systému pro získání aktuální konfigurace. Dále bylo potřeba doplnit funkce měnící část konfigurace na zadanou hodnotu.

Načítání hodnot z JSON souborů bylo realizováno pomocí funkcí obsažených v knihovně *cJSON.h*.

```
cJSON *json= NULL;
    read_json_from_file("/storage/config.json",&json);
    if (xSemaphoreTake(g_safe_device_config.mutex, pdMS_TO_TICKS(100))) {
        cJSON *wifi_ssid = cJSON_GetObjectItem(json, "wifi_ssid");
        if (cJSON_IsString(wifi_ssid)) {
            strlcpy(g_safe_device_config.device_config.wifi_ssid,
wifi_ssid->valuestring,
                    sizeof(g_safe_device_config.device_config.wifi_ssid));
        }
        xSemaphoreGive(g_safe_device_config.mutex);
    }
    cJSON_Delete(json);
```

Zdrojový kód 5: Ukázka načítání hodnot z JSON souboru do struktury

Obdobným způsobem byla implementována práce se strukturou *safe_environment_values_t* představující cílové hodnoty prostředí.

5.2.5 I^2C sběrnice

Jelikož k ESP32 jsou připojené 3 zařízení komunikující přes I^2C sběrnici bylo nutné udělat sdílenou inicializaci a jednotlivá zařízení na sběrnici poté přidávat. Této funkce využívá modul senzoru scd4x, RTC modul DS1307 a modul displeje ssd1306. Knihovny pro práci s jednotlivými moduly již obsahují inicializaci I^2C sběrnice, ale jelikož používám modulů více, bylo potřeba ji z jednotlivých funkcí odebrat a volat pouze jednou pro všechny moduly.

Opakované volání sdílené inicializace vrací pouze kód *ESP_OK* značící úspěšně inicializovanou sběrnici.

5.2.6 Wi-Fi

Základní podmínkou pro připojení k Wi-Fi je úspěšné načtení konfigurace obsahující SSID a heslo přístupového bodu, ke kterému se řídicí jednotka pokouší připojit. Konfigurace se nastavuje v souboru config.json a nahrává se spolu s firmwarem. Tento postup by pro skutečný produkt nebyl dostačující. Jedno z možných řešení je dále popsáno v kapitole 5.2.14.

Pro inicializaci, připojení, odpojení a deinicializaci jsem použil kód z tutoriálu na webu Espressif [45]. K inicializaci jsem přidal pouze globální proměnnou *g_wifi_connected*, která umožňuje kontrolu, zda-li je zařízení úspěšně připojené k Wi-Fi a v případě, že není, zabrání spuštění funkcí využívajících Wi-Fi.

Připojení k Wi-Fi není voláno přímo z funkce *app_main*, ale má vlastní task, který se v případě neúspěchu opakovaně pokouší připojit. To funguje i v případě, kdy řídicí jednotka připojení ztratí po tom, co ho již měla.

5.2.7 Získání reálného času

Jednou z funkcí pěstírny je spouštění osvětlení v zadaném časovém intervalu. K tomu však potřebuje znát přesný čas. Toho lze dosáhnout dvěma způsoby. Primární způsob je pomocí SNTP protokolu a jako záloha slouží připojený Real Time Clock (RTC) modul.

V ideální případě, kdy se po spuštění řídicí jednotka úspěšně připojí k Wi-Fi, je pro získání přesného času použit Simple Network Time Protocol (SNTP). Jedná se o protokol, kde klient (naše zařízení) může získat od serveru aktuální UTC čas. Na rozdíl od NTP nepoužívá složité algoritmy pro dosažená přesnosti a bezpečnosti. V případě této práce to je však dostačující řešení, které je zároveň méně výpočetně náročné. [46]

V případě, kdy pěstírna po spuštění neuspěje s připojením k Wi-Fi nastane problém, kdy nelze spolehlivě řídit intervaly osvětlení, jelikož ESP32 nezná čas. Pro tento případ jsem na sběrnici I^2C připojil RTC modul, který za pomocí vlastní baterie udržuje aktuální čas i v době, kdy je řídicí jednotka vypnutá.

Konkrétně jsem použil modul DS1307. Ten udržuje čas v podobě vteřin, minut, hodin, data v měsíci, měsíce i s kompenzací přechodných roků. Je napájen skrze ESP32, ale v případě ztráty napájení přepne na vlastní baterii a dále drží aktuální čas. [39]

Pro komunikaci s RTC modulem jsem použil knihovnu DS1307 [47]. Konkrétně v ní obsažení port určený pro ESP-IDF. V ní jsem musel upravit inicializaci I^2C sběrnice, aby používala společnou inicializaci pro všechny moduly.

Další potřebná změna byl způsob I^2C komunikace. Jelikož knihovny některých modulů používaly starší knihovnu *driver/i2c.h* a jiné již aktuální a zjednodušenou *driver/i2c_master.h*, musel jsem moduly sjednotit. Proto jsem funkce zápisu a čtení dat z DS1307 změnil, aby používaly *driver/i2c_master.h*.

Když byla vyřešena komunikace, stačilo již v případě úspěšného připojení k Wi-Fi zapsat čas získaný pomocí SNTP do RTC modulu. V případě neúspěšného připojení k Wi-Fi se načte čas z RTC modulu a nastaví podle něj čas zařízení.

5.2.8 Čtení dat ze senzoru

Pro čtení hodnot ze senzoru SCD41 jsem použil knihovnu esp32-scd4x [48]. Ta již byla určena pro ESP32, ale musel jsem provést několik změn. První bylo použití společné inicializace I^2C. Dále bylo stejně jako u knihovny RTC modulu potřeba předělat I^2C komunikaci z legacy knihovny *driver/i2c.h* na aktuální *driver/i2c_master.h*. Tím se složitost funkcí pro zápis a čtení výrazně snížila.

Funkce pro čtení senzoru obsahuje nekonečnou smyčku, která se pokouší získat data ze senzoru každých 50ms.

```
for(;;) {
    scd4x_sensors_values_t sensors_values = {
        .co2 = 0x00,
        .temperature = 0x00,
        .humidity = 0x00
    };
    vTaskDelay(50 / portTICK_PERIOD_MS);

    if(scd4x_read_measurement(&sensors_values) != ESP_OK) {
        continue;
    }
    ...
  }
```

Zdrojový kód 6: Pokus o čtení dat ze senzoru

Senzor má vlastní task, který se stará o periodické čtení aktuálních hodnot. Tasku je v parametrech předána fronta, do které zapisuje získané hodnoty a *TaskHandle_t* od dále popsaného tasku displeje, který využívá pro upozornění na změnu dat. Z fronty si poté můžou ostatní části programu data načíst.

```
if (xQueueOverwrite(sensor_queue, &sensors_values) != pdTRUE) {
    ESP_LOGE(SENSOR_TAG, "Failed to write to queue");
}
```

Zdrojový kód 7: Zápis dat do fronty

```
if (display_task_handle != NULL) {
    xTaskNotify(display_task_handle, 0, eNoAction);
}
```

Zdrojový kód 8: Zaslání notifikace, na kterou čeká task vypisující data na displej

5.2.9 Zobrazení dat na displeji

Aby uživatel nemusel pro kontrolu aktuálního stavu používat aplikaci, je součástí řídicí jednotky i malý displej ukazující aktuální hodnoty. Ten je použit i k zobrazení kódu pro spárování s uživatelským účtem, více v kapitole 6.7.2.

Pro komunikaci s modulem SSD1306 jsem použil knihovnu esp-idf-ssd1306 [49]. Specificky zdrojové kódy z adresáře *components*. Na rozdíl od předchozích knihoven, zde již byla pro

I²C komunikaci použita aktuálně podporovaná knihovna *driver/i2c_master.h*, takže jedinou změnu, co jsem musel udělat, bylo vyjmutí inicializace I²C. Součástí knihovny jsou funkce pro zobrazení textu, obrázků a základní font, ve kterém se text zobrazuje.

Funkce ovládající displej má opět vlastní task, který opakovaně volá funkci zobrazující aktuální hodnoty a v případě, že je zařízení v režimu párování, dočasně ukáže obrazovku s párovacím kódem.

```
void display_task(void *pvParam) {
  ssd1306_clear_screen(&dev, false);
  while(1){
    display_current_values(pvParam);
    if(is_pairing_active()){
      display_pairing_code();
    }
  }
}
```

Zdrojový kód 9: Task obsluhující displej

Tasku je v parametrech předán ukazatel *TaskHandle_t*, pod kterým senzor posílá notifikaci značící nově zapsaná data do fronty. Cyklus zobrazování dat vždy čeká na příchozí notifikaci, než načte a zobrazí data.

```
ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
```

Zdrojový kód 10: Blokující funkce čekající na *xTaskNotify* ze strany senzoru

Dále dostává task v parametrech frontu, do které načítá senzor data a z té si je funkce displeje bere k zobrazení.

```
if (xQueuePeek(sensor_queue, &sensors_values, 0) == pdTRUE) {
    // Vykreslí aktuální hodnoty
    display_sensor_values(sensors_values.co2, sensors_values.temperature,
sensors_values.humidity,'c');
    // Vykreslí aktuální čas
    display_time();
}
```

Zdrojový kód 11: Vykreslení hodnot získaných z fronty

Pokud je řídicí jednotka v režimu párování, volá se funkce *display_pairing_code*, která zobrazí párovací kód. Více o jeho vytvoření v kapitole 6.7.2.

5.2.10 Komunikace s API

Pro funkce, které vytvářejí POST nebo GET requesty, jsem vytvořil společnou funkci *api_request_json*, které se předává relativní URL, pointer na odesílaný payload ve formě cJSON dat, pointer na cJSON představující návratové hodnoty a typ HTTP metody (POST/GET).

Tato funkce nejprve sestaví z relativní cesty a adresy serveru absolutní URL. Dále alokuje buffer pro odpověď ze serveru a načte TLS certifikát z flash paměti. Pokud vše končí úspěchem, vytvoří HTTP klienta, pod kterým poté může poslat request.

```
esp_http_client_config_t config = {
    .url = full_url,
    .method = method,
    .cert_pem = (const char *)g_cert_pem,
    .event_handler = _http_event_handler2,
    .user_data = &response_data,
};

client = esp_http_client_init(&config);
```

Zdrojový kód 12: Vytvoření HTTP klienta

V případě, že se jedná o POST request, který má tělo, převede JSON data na string a přidá je to requestu.

Následně je voláno provedení HTTP requestu pomocí *esp_http_client_perform*. Značí-li návratová hodnota úspěch, a odpověď není prázdná, převádí se na JSON a vrací volajícímu.

Na konci těchto operací, ale i v případě selhání jakékoli části je potřeba uvolnit alokovanou paměť. Jelikož dochází k několikanásobné alokaci při každém volání, neprovedeme-li správně dealokaci, systém po naplnění paměti přiřazené tasku nebude schopen dále alokovat paměť a funkce s ní spojené nebudou fungovat.

5.2.11 Funkce komunikující s API

První volání API, které řídicí jednotka provede, je pokus o registraci zařízení. Pokud jednotka zjistí, že nemá uložené žádné *device_id*, pod kterým se může hlásit, odesílá serveru POST

request obsahující svou MAC adresu v jeho těle. Server odpovídá vrácením *device_id*, které si jednotka ukládá a dále se pod ním hlásí.

```
cJSON *root = cJSON_CreateObject();
cJSON_AddStringToObject(root, "mac_address", mac_str);

cJSON *response = NULL;
esp_err_t err = api_request_json(CONFIG_URL_PATH, root, &response,
HTTP_METHOD_POST);
cJSON_Delete(root);
```

Zdrojový kód 13: Vytvoření API requestu obsahujícího MAC adresu

Další POST request vzniká, každých 30 vteřin při odesílání naměřených hodnot prostředí pěstírny. Zde se zapisují hodnoty prostředí do JSON formátu a odesílají v těle requestu.

V případě získávání aktuálně nastavených cílových hodnot pěstírny je odesílán GET request, kde je na konec celé URL přidána hodnota hodnota *device_id*, ke které se poptávané hodnoty vážou. Není zde tak žádné tělo a na rozdíl od předchozích POST requestů se zde převádí navrácená data do JSON formátu, ze kterého už se načítají jednotlivé hodnoty.

```
char full_url[128];
const char *device_id = config.device_id;

// Add /{device_id} to the base_url
snprintf(full_url, sizeof(full_url), "%s/%s", ENVIRONMENT_URL_PATH,
device_id);
```

Zdrojový kód 14: Sestavení URL obsahující _device_id_

Posledním použití komunikace s API je ve funkci párování. Ta je aktivována stisknutím tlačítka připojeného k řídicí jednotce. Odesílá na server request obsahující *device_id* a obratem získává párovací kód, který po dobu 60s zobrazí na displeji řídicí jednotky pro uživatele k opsaní.

5.2.12 Řízení klima přestírny

Ve chvíli, kdy je načtená veškerá konfigurace a senzor pravidelně čte hodnoty prostředí, můžeme konečně vytvořit cyklus regulující prostředí pěstírny.

Opět se jedná o nový task, který v parametrech dostává frontu, do které senzor načítá data. Po spuštění tasku se načte konfigurace výstupních GPIO pinů (na kterém je připojeno které zařízení). Proběhne reset a nastavení základní úrovně vybraných pinů.

Poté je již spuštěn nekonečný cyklus, který vždy načte aktuální cílové hodnoty, získá data senzoru z fronty a předává je jednotlivým funkcím, které rozhodují, zda-li je třeba pustit, nebo vypnout některé připojené zařízení. To se opakuje v intervalu 5s.

```
while (1) {
    load_environment_settings();
    if (xQueuePeek(sensor_queue, &sensor_values, 0) == pdPASS) {
        control_co2(sensor_values.co2);
        control_humidity(sensor_values.humidity);
        control_temperature(sensor_values.temperature);
        control_light();
    } else {
        ESP_LOGI("Control", "Queue empty");
    }
    vTaskDelay(WAIT_PERIOD / portTICK_PERIOD_MS);
}
```

Zdrojový kód 15: Smyčka řídicí připojená zařízení

Samotné funkce ovlivňující jednotlivé parametry už jen kontrolují překročení mezních hodnot a podle toho mění stav GPIO pinů.

```
void control_co2(int co2_value){
    if(co2_value>co2_max && !fan_on){
        ESP_LOGI("Control","Turning the fan on.");
        gpio_set_level(GPIO_fan, RELAY_ON);
        fan_on = true;
    }else if(co2_value<co2_min && fan_on){
        ESP_LOGI("Control","Turning the fan off.");
        gpio_set_level(GPIO_fan, RELAY_OFF);
        fan_on = false;
    }
}
```

Zdrojový kód 16: Funkce ovládající ventilaci na základě hladiny CO_2

5.2.13 Posloupnost operací po spuštění

1. Vytvoření SPIFFS oddílu uložiště
2. Načtení konfigurace ze souboru
3. Načtení cílových hodnot prostředí ze souboru
4. Připojení k Wi-Fi
5. Registrace zařízení na serveru, pokud již nemá *device_id*
6. Získání času pomocí SNTP a aktualizování času v RTC modulu
7. Spuštění čtení hodnot ze senzoru
8. Spuštění zobrazování hodnot na displeji
9. Spuštění regulace prostředí pěstírny
10. Odesílání naměřených hodnot na server
11. Získávání aktuálního nastavení prostředí ze serveru
12. Spuštění párování, pokud uživatel stiskne tlačítko

Některé z výše vypsaných kroků běží paralelně díky rozdělení na tasky. Proto není možno určit přesné pořadí vykonávání operací, pouze ošetřit počátečními podmínkami pro vstup.

```
I (4072) SNTP: Time synchronized: Thu Aug 14 18:39:29 2025

I (4072) DS1307: DS1307 I2C device initialized successfully
I (4072) DS1307: RTC updated with SNTP time
I (4072) WIFI_TASK: Wi-Fi connection successful.
I (5422) CERT: Certificate loaded, length: 1119
I (5892) SCD41: CO₂ 1750 ppm - Temperature 35.6 °C - Humidity 41.5%
I (6052) Control: Turning the humidifier on.
I (6052) Control: Turning the AC on.
I (6352) API: HTTP request successful, status code: 200
I (6362) HTTP: Extracted Device ID: 9f798beb-196e-4c4e-9fbe-f38404c76f44
I (6392) TASK: Device registration succeeded
```

Obrázek 18: Ukázka kontrolních výpisů po spuštění - Zdroj: vlastní

Pokud řídicí jednotka neuspěje s připojením k internetu, nebo o něj v průběhu času přijde, funkce měření, zobrazení hodnot a regulace prostředí fungují beze změny, pouze do až do obnovy připojení nelze měnit nastavení pěstírny.

5.2.14 Možná další rozšíření

Práce na řídicí jednotce se nakonec ukázala být rozsáhlejší, než jsem původně předpokládal, avšak stále existují rozšíření, která jsem nerealizoval a pro finální produkt by byla vhodná, ne-li dokonce nutná.

Hlavním nedostatkem mého systému je nemožnost nastavení a změny konfigurace přístupového bodu Wi-Fi. Jelikož veškeré změny nastavení provádím skrze komunikaci se serverem, při absenci připojení k internetu je jediná možnost změny konfigurace skrze úpravu config.json a nahrání nového firmwaru.

To by se však dalo obejít použitím další funkce mikrokontrolérů ESP32. Ty totiž nabízí mimo Wi-Fi i možnost použití Bluetooth. Proto by šlo realizovat nastavení pomocí bezdrátové komunikace mezi mobilní aplikací a řídicí jednotkou. To jsem však v rámci této práce neimplementoval.

Dalším možným rozšířením by byla podpora PWM řízení motorů. U ventilátoru by se tak dalo dosáhnout plynulé regulace rychlosti a stabilnějšího stavu ovzduší. V závislosti na použitých zařízeních by se podobná plynulá regulace dala využít i pro topení a klimatizaci, ale tam už by byla potřeba větší integrace zařízení do systému.

6 Web server

Jelikož uživatelé a pěstírny nemohou komunikovat přímo, a zároveň je potřeba někde uchovávat data, je nutné jako prostředníka vytvořit web server. Ten se chová jako rozhraní mezi pěstírnami a uživateli. Server se dále stará o zabezpečení přístupu k datům pomocí uživatelských účtů a autentizace.

6.1 Architektura

Serverová část se bude skládat ze dvou částí, kterými budou samotná aplikace serveru a relační databáze. Ta zajistí persistenci dat a jejich logické vazby. K databázi má přistup pouze server a všechna ostatní zařízení k ní přistupují skrze funkce serveru.

Na straně serveru jsem zvolil architekturu REST API, která umožňuje komunikaci pomocí HTTP requestů a nevyžaduje žádné uchovávání stavu. Každý požadavek má adresu zdroje, ke kterému přistupuje, v podobě URL adresy. Proto lze každý požadavek vykonat nezávisle na předchozím stavu. Další výhodou je komunikace ve formátu JSON. To umožní jednoduchou implementaci přijímání a vytváření requestů napříč platformami. Také to ulehčí vývoj a testování, jelikož se jedná o člověkem lehce čitelnou formu dat. [50]

6.2 Databáze

Při výběru databáze je důležité si určit, jakým způsobem ji plánujeme používat. V případě tohoto projektu půjde především o správu uživatelů, pěstíren, nastavených hodnot prostředí a vzájemných relací mezi tabulkami. Na takové užití se nabízejí tradiční relační databáze, které uchovávají data v tabulkách a jejich relace vytváří pomocí primárních a sekundárních klíčů.

Oproti běžným databázovým aplikacím má tato práce však ještě jednu hodnotu, kterou je potřeba uchovávat, a to jsou data naměřená senzory pěstíren. K těmto hodnotám se přidává informace o času záznamu, čímž vzniká časová řada dat. Při intervalu zápisu každých 30s vznikne 2880 nových záznamů denně za každou pěstírnu. Pokud chceme pracovat s velkým množstvím časových dat a chtěli bychom je dále analyzovat, tradiční relační databáze nebudou nejúčinnější řešení.

Proto jsem se rozhodl použít relační databázi PostgreSQL s rozšířením TimescaleDB. Jedná se o open-source rozšíření poskytující optimalizace pro uchovávání, získávání a analýzu velkého množství dat obsahujících časovou hodnotu. Tímto získám jednoduchost a tradiční syntaxi relačních databází spolu s možností efektivního uchování a zpracování časových dat. [51]

6.2.1 Vytvoření

Pro účely této práce provozuji databázi v Docker kontejneru, což mi zajistí jednoduchost zprovoznění, izolaci od systému a opakovatelnost na libovolném zařízení. Pro ještě lehčí replikovatelnost jsem si vytvořil *docker-compose.yml* soubor definující nastavení docker kontejneru. Díky tomu lze celý kontejner odstranit a vytvořit nový se stejnou konfigurací spuštěním příkazu *docker-compose up -d*.

```
services:
  timescaledb:
    image: timescale/timescaledb-ha:pg17
    container_name: timescaledb
    restart: always
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: timescaledb
      POSTGRES_PASSWORD: password
    volumes:
      - ./data/timescaledb:/home/postgres/pgdata/data
```

Zdrojový kód 17: Soubor docker-compose.yml

Zároveň v souboru *docker-compose.yml* mám nastavené mapování adresáře, ve kterém databáze uchovává data na adresář *data* mimo kontejner databáze. Tím dosáhnu toho, že databáze uchovává data mimo svůj kontejner a v případě odstranění a znovuvytvoření kontejneru databáze nepřijde o žádná data.

6.2.2 Datový model

Před tím, než začneme databázi plnit daty, je potřeba navrhnout model tabulek a jejich relací, které budou databázi tvořit.

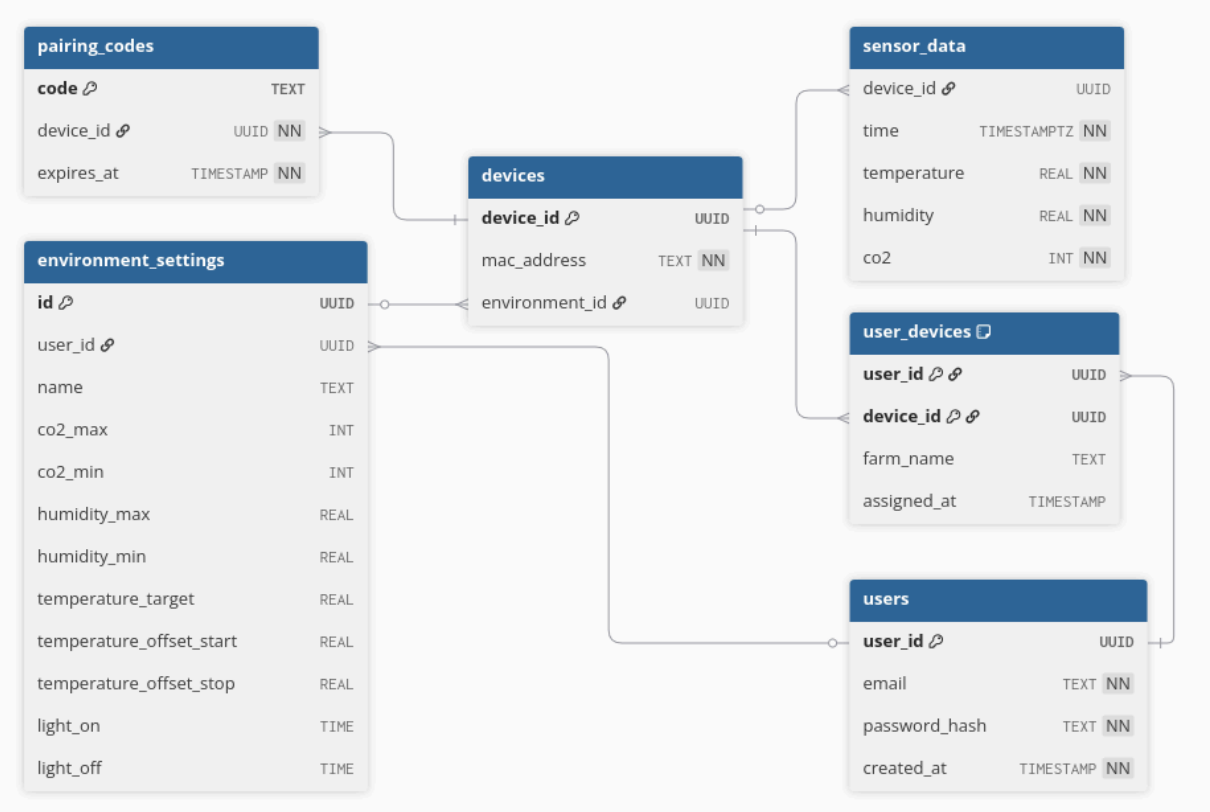
Pro uchování řídicích jednotek byla vytvořena tabulka *devices* , která má primární klíč *device_id* generovaný funkcí *gen_random_uuid()*. Dále má textovou hodnotu pro uložení MAC adresy a sekundární klíč odkazující na id v tabulce *environment_settings*. Ten představuje nastavení prostředí pěstírny, jaké řídicí jednotka aktuálně používá. Jelikož je TimescaleDB pouze rozšíření PostgreSQL, vytváříme tabulky tradičním DDL skriptem.

```
CREATE TABLE devices(
  device_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  mac_address TEXT NOT NULL UNIQUE,
  environment_id UUID REFERENCES environment_settings(id)
);
```

Zdrojový kód 18: Vytvoření tabulky devices

Obdobně to vypadá u tabulky users, představující uživatele, tabulky sensor_data uchovávající naměřené hodnoty a tabulky environment_settings, která obsahuje hodnoty, podle kterých řídicí jednotka pracuje.

Speciální je tabulka user_devices, která je spojující tabulkou pro N:M relaci mezi devices a users. Umožňuje jednomu uživateli spravovat více pěstíren, ale také více uživatelům spravovat jednu pěstírnu. Jelikož je vázaná na uživatele a vzniká při párování, uchovává také jméno, pod jakým má uživatel pěstírnu uloženou.



Obrázek 19: Diagram databáze, vytvořeno pomocí [19]

Poslední, co musíme udělat při vytváření databáze, je použít TimescaleDB funkci *create_hypertable* na tabulku s daty ze senzoru.

```
SELECT create_hypertable(
   'sensor_data',
   'time',
   'device_id',
   4,
   chunk_time_interval => INTERVAL '1 day'
);
```

Zdrojový kód 19: Převod tabulky *sensor_data* na typ *hypertable*

Tento příkaz říká, že tabulku *sensor_data* databáze bude dělit podle hodnot ve sloupci *time*. Dále jí rozdělí podle *device_id* na 4 oddíly. Dělení podle času bude v intervalu jednoho dne. Tímto se zajistí rychlejší vyhledávání záznamů ze stejného dne a pod stejným *device_id*.

6.3 Server

Pro vytvoření REST API jsem zvolil framework Axum, který je založený na jazyce Rust a umožňuje vytvoření výkonného API v jazyce, který zachytí většinu chyb v době kompilace. Jedná se o lehký framework umožňující jednoduché vytvoření REST API pomocí handler funkcí obsluhující jednotlivé requesty a Routeru, který mapuje URL adresy na jednotlivé handler funkce. [52]

Nejprve je ve funkci main potřeba vytvořit pool, přes který budeme dále přistupovat do databáze. Toho dosáhneme pomocí funkce *PgPoolOptions::new()* z knihovny SQLx.

```
let db_pool = PgPoolOptions::new()
    .max_connections(16)
    .connect(&database_url)
    .await
    .expect("Failed to connect to postgres");
```

Zdrojový kód 20: Vytvoření connection poolu pomocí SQLx

Dále je potřeba vytvořit Router, který bude API requesty z určitých URL adres směrovat na příslušné handler funkce.

```
let app = Router::new().route("/api/v1/healthcheck",
get(health_check_handler));
```

Zdrojový kód 21: Konfigurace přístupových bodů API - Zdroj: [20]

Poté již stačí aplikaci spustit na zadané adrese. Jelikož se však v tomto případě nejedná o minimální provedení API a bylo použito zabezpečení pro komunikaci přes HTTPS, zbytek konfigurace je popsán v kapitole 6.4.1.

Realizace minimální handler funkce může vypadat například viz Zdrojový kód 22.

```
pub async fn health_check_handler() -> impl IntoResponse {
    const MESSAGE: &str = "API Services";
    let json_response = serde_json::json!({
        "status": "ok",
        "message": MESSAGE
    });
    Json(json_response)
}
```

Zdrojový kód 22: Minimální handler funkce - Zdroj: [20]

6.4 Zabezpečení

Jelikož komunikace se serverem obsahuje informace vázané na uživatele včetně citlivých dat, jako je e-mail, nebo heslo, je potřeba komunikaci zabezpečit. V základu by web server vytvořený pomocí frameworku Axum neobsahoval žádné zabezpečení.

6.4.1 HTTPS

Není-li použito dalších knihoven, server odesílá data v nešifrované formě pomocí HTTP protokolu. To znamená, že každý, kdo se ke komunikaci dostane, může číst její obsah včetně uživatelských hesel, nebo session tokenů. Díky tomu nemáme jistotu, že naši komunikaci někdo neodposlouchává, nebo že data nebudou během cesty mezi klientem a serverem někým pozměněna.

Proto poslední roky takřka všechny veřejně přístupné webové služby přešly z HTTP na HTTPS. Data mají stále stejnou formu jako u HTTP, ale posílána jsou v šifrované formě. Toho je dosaženo pomocí Transport Layer Security (TLS). Server má pro účely šifrování dva klíče, veřejný klíč dostupný všem klientům a soukromý klíč, který má pouze server a využívá ho pro dešifrování dat šifrovaných veřejným klíčem. [53]

Než však vůbec dojde k přenosu dat mezi serverem a klientem, musí server klientu zaslat svůj certifikát obsahující veřejný klíč. Klient certifikát ověří u certifikační autority (CA), a pokud zjistí, že server je věrohodný, dojde k výměně informací a navázání komunikace.

TLS dále zajišťuje pomocí kontrolních součtů, že zpráva přišla celá a beze změny. Tím se zajistí, že s daty nemůže nikdo manipulovat.

Pro bezpečnou komunikaci mezi serverem a klienty je tedy potřeba TLS certifikát. Pro prototypy, jako je tato práce nastává však jeden problém. TLS certifikáty jsou placená služba poskytovaná jednotlivými CA. Pro tyto účely testování si však můžeme vygenerovat vlastní TLS certifikát pomocí *openssl*.

To uděláme tak, že si nejdříve vytvoříme soubor s parametry svého certifikátu.

```
[ req ]
default_bits       = 2048
distinguished_name = req_distinguished_name
x509_extensions    = v3_req
prompt             = no

[ req_distinguished_name ]
CN = 10.0.1.44

[ v3_req ]
keyUsage = critical, digitalSignature, keyEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[ alt_names ]
IP.1 = 10.0.1.44
```

Zdrojový kód 23: Konfigurační soubor openssl.cnf

Následně použijeme příkaz *openssl* pro vygenerování certifikátu a privátního klíče na základě konfiguračního souboru.

```
openssl req -x509 -nodes -days 365 \
  -newkey rsa:2048 \
  -keyout key.pem \
  -out cert.pem \
  -config openssl.cnf\
  -extensions v3_req
```

Zdrojový kód 24: Příkaz pro generování certifikátu

Oba získané soubory je potřeba dát serveru, který je použije k vytvoření HTTPS komunikace.

V případě frameworku Axum to je realizováno pomocí knihovny rustls, poskytující kryptografické zabezpečení. Nejprve je potřeba vytvořit konfiguraci obsahující certifikát a klíč.

```
let config = RustlsConfig::from_pem_file(
    "cert.pem",  // Path to your certificate
    "key.pem",   // Path to your private key
)
    .await
    .expect("Failed to load TLS certificates");
```

Zdrojový kód 25: konfigurace Rustls - Zdroj: [21]

Poté, když již máme nastavené přístupové body pomocí *Router* a vytvořenou adresu serveru, můžeme spustit server i s TLS konfigurací.

```
axum_server::bind_rustls(addr, config)
    .serve(app.into_make_service())
    .await
    .unwrap();
```

Zdrojový kód 26: Spuštění serveru - Zdroj: [21]

Nevýhoda tohoto řešení s vlastním vygenerovaným certifikátem je, že musíme vložit kopii certifikátu do každé aplikace, která má se serverem komunikovat. Proto se jedná pouze o řešení vhodné pro vývoj a nemělo by se dostat do produkce.

6.4.2 Autentizace

Druhý použitý způsob zabezpečení je autentizace uživatelů. Tím zabráníme tomu, že někdo bude bez platného přihlášení získávat data ze serveru. Dosáhneme toho pomocí JSON Web Tokenů (JWT). Jedná se o bezstavové ověření uživatele pomocí tokenu. [54]

Ve chvíli, kdy se uživatel registruje, nebo přihlásí, server z jeho id vygeneruje token. Jeho součástí je i informace o konci jeho platnosti. Server generuje token pomocí svého tajného klíče. Tento token je poté poslán aplikaci klienta, která ho na svojí straně ukládá a následně přidává do hlavičky requestů.

Ve chvíli, kdy serveru přijde request vyžadující autentizaci, bere si z hlavičky token a pomocí svého klíče ho opět dekóduje pro zjištění platnosti.

Na serverové části toto řeším pomocí knihovny jsonwebtoken [55].

6.5 Struktury

Protože výměna dat mezi serverem a klientem probíhá formou JSON souborů, je potřeba zajistit nějakou konverzi do datových typů, s jakými již aplikace serveru zvládne pracovat. Toho dosáhneme pomocí knihovny *Serde*, která nám zajistí automatickou konverzi struktur označených atributem *Serialize* nebo *Deserialize*.

```
#[derive(Deserialize)]
pub struct LoginRequest {
    pub email: String,
    pub password: String,
}
#[derive(Serialize)]
pub struct LoginResponse {
    pub token: String,
    pub user_id: String,
}
```

Zdrojový kód 27: Struktury použité k načtení a vytvoření těl HTTP requestů

Máme-li vytvořenou odpověď *response* ve formě *LoginResponse*, stačí zavolat funkci *Json(response)* pro převod do formátu JSON. Tímto způsobem jsou řešená těla všech příchozích a odchozích requestů.

6.6 Směrování

Request	URL	handler funkce
GET	/api/healthcheck	health_check_handler
POST	/api/sensor_data	insert_sensor_data_handler
GET	/api/sensor_data/{device_id}	get_sensor_data_handler
POST	/api/devices/register	register_device_handler
GET	/api/devices/environment/{device_id}	get_device_environment_handler
POST	/api/devices/environment/{device_id}	set_device_environment_handler
GET	/api/devices/{device_id}	get_device_handler
GET	/api/get_environment/{device_id}	get_environment_handler
POST	/api/set_environment/{environment_id}	set_environment_handler
POST	/api/set_environment/	insert_environment_handler
GET	/api/user_environments/{user_id}	get_environments_of_user_handler
POST	/api/login	login_handler
POST	/api/signup	signup_handler_handler
GET	/api/me	me_handler
POST	/api/generate_pairing_code	generate_pairing_code_handler
POST	/api/pair_device	use_pairing_code_handler
GET	/api/user_devices/{user_id}	get_user_devices_handler
GET	/api/chart_data/{device_id}	get_chart_data_handler

Tabulka 6: Mapování adres na obslužné funkce

Takto vypadá tabulka všech přístupových bodů a funkcí, které na nich provádí operace s databází. S rostoucími požadavky na funkce systému poroste i množství handler funkcí. Například ke kompletnosti databázové aplikace zde ještě chybí funkce pro odebírání prvků z databáze.

6.7 Handler funkce

Pokaždé, když uživatel volá API, je jeho request podle URL adresy směrován na některou z handler funkcí. Handler funkce jsou asynchronní funkce s návratovým typem, který musí jít převést na HTTP odpověď. Dále může mít handler funkce další vstupní parametry, jako je stav aplikace obsahující pool databáze, tělo POST requestu, nebo identifikátor zdroje z URL requestu.

```
pub async fn get_user_devices_handler(
    Path(user_id): Path<Uuid>,
    State(state): State<Arc<AppState>>,
    TypedHeader(Authorization(bearer)): TypedHeader<Authorization<Bearer>>,
) -> Result<Json<Vec<UserDevice>>, (StatusCode, String)> {
    if !verify_jwt_for_user(state.jwt_secret.clone(), bearer.token(),
user_id)
    {
        return Err((StatusCode::UNAUTHORIZED, "Invalid token".into()));
    }
    let rows = sqlx::query_as::<_, UserDevice>(
        "SELECT device_id, farm_name FROM user_devices WHERE user_id = $1"
    )
        .bind(user_id)
        .fetch_all(&state.db_pool)
        .await
        .map_err(|_| (StatusCode::INTERNAL_SERVER_ERROR, "Failed to fetch
devices".into()))?;

    Ok(Json(rows))
}
```

Zdrojový kód 28: Příklad handler funkce pro získání všech zařízení uživatele

První vstupní parametr je *user_id* získané z URL adresy, to značí u GET requestu zdroj, k jakému přistupujeme, a není zapotřebí vytvářet tělo requestu s dalšími daty. Dále obsahuje proměnnou state, která v sobě nese pool databáze a klíč použitý u JWT tokenů. Posledním parametrem je header obsahující JWT token pro autentizaci.

Návratová hodnota značí, že funkce v případě úspěchu vrací vektor typu *UserDevice* serializovaný do JSON formátu. V případě neúspěchu vrací stavový kód HTTP doplněný o chybovou hlášku.

U funkcí, kde máme například id uživatele a hledáme k němu vztažené hodnoty, provedeme kontrolu, zdali je token platný, a jestli byl vygenerován pro žádajícího uživatele. Tím zajistíme, že k datům nemá přístup nikdo bez oprávnění.

Pro získání dat z databáze je použita knihovna SQLx, pomocí které vytváříme dotaz, a získané řádky automaticky převádíme na formát struktury UserDevice. K tomuto automatickému mapování je potřeba strukturu označit atributem *sqlx::FromRow* viz zdrojový kód 29.

```
#[derive(Debug, Serialize, sqlx::FromRow)]
pub struct UserDevice {
    pub device_id: Uuid,
    pub farm_name: String,
}
```

Zdrojový kód 29: Struktura umožňující automatickou konverzi z řádku tabulky

Selže-li dotaz, vrací funkce HTTP kód 500 *Internal Server Error* spolu s chybovou hláškou. Pokud dotaz uspěje, funkce vrací vektor získaných řádků ve formě JSON souboru spolu s HTTP kódem 200 *OK*.

Pokud handler funkce obsluhuje requesty s tělem, získáváme jeho hodnotu opět v parametrech ve formě JSON dat, které automaticky deserializujeme na datový typ, se kterým dále můžeme pracovat.

Některé funkce mohou být složitější než pouze získání dat z databáze, nebo jejich vložení. Například funkce *register_device_handler*, která slouží k registraci řídicí jednotky, nejdříve zjistí, zda-li se jednotka už v databázi nenachází, pokud ano tak vrací její informace. Pokud ji v databázi nenajde, tak ji vkládá a opět vrací její informace.

6.7.1 Registrace a přihlášení

Abychom zabránili uchovávání hesel v databázi v původní textové formě, vkládá server do databáze hash uživatelem zadaného hesla. To uděláme pomocí hash funkcí v knihovně argon2.

V případě přihlášení se porovná hash nově příchozího hesla s hash hodnotou uloženou v databázi.

```rust
pub fn hash_password(password: &str) -> Result<String,
password_hash::Error> {
    let salt = SaltString::generate(&mut OsRng);
    let argon2 = Argon2::default();

    let password_hash = argon2.hash_password(password.as_bytes(), &salt)?;
    Ok(password_hash.to_string())
}

pub fn verify_password(password: &str, hashed: &str) -> bool {
    let parsed_hash = PasswordHash::new(hashed).unwrap();
    Argon2::default()
        .verify_password(password.as_bytes(), &parsed_hash)
        .is_ok()
}
```

Zdrojový kód 30: Funkce pro bezpečné uchování hesel

Pokud registrace nebo přihlášení proběhne úspěšně, server vygeneruje nový JWT a vrací ho v těle odpovědi.

6.7.2 Párování

Pro propojení řídicí jednotky a uživatelského účtu slouží dvě handler funkce.

První funkce *generate_pairing_code_handler* je volána samotnou řídicí jednotkou po stisku tlačítka. Ta vygeneruje náhodný alfanumerický řetězec o 8 znacích, ten následně spolu s *device_id* řídicí jednotky a časem vypršení platnosti vloží do databáze. Vygenerovaný kód poté vrátí řídicí jednotce, která ho zobrazí.

Uživatel poté opíše kód do mobilní aplikace, vyplní jméno, pod jakým chce mít řídicí jednotku uloženou a stiskem tlačítka *Přidat pěstírnu* volá druhou handler funkci *use_pairing_code_handler*. Pokud funkce zadaný kód najde v databázi, vloží záznam do spojující tabulky *user_device*, čímž se vytvoří relace mezi uživatelským účtem a řídicí jednotkou.

6.7.3 Data pro zobrazení grafu

Pro potřebu zobrazení naměřených dat v grafu by bylo odesílání v podobě listu JSON objektů nevhodné kvůli velkému množství opakujících se dat spojených s JSON formátem. Proto získaná data za určitý časový interval server odesílá jako jeden objekt, kde každá hodnota je vektor hodnot z jednoho sloupce tabulky.

```
#[derive(Serialize)]
pub struct AggregatedSensorDataBatch {
    pub bucket: Vec<DateTime<Utc>>,
    pub avg_temperature: Vec<f32>,
    pub avg_humidity: Vec<f32>,
    pub avg_co2: Vec<f64>,
}
```

Zdrojový kód 31: Struktura pro návratové hodnoty dat do grafu

Jelikož řídicí jednotka odesílá naměřená data každých 30s, všechna data v intervalu jednoho dne by stále byla dost objemná. Navíc pro zobrazení denního intervalu nepotřebujeme informace za každých 30s, proto server z databáze vytváří průměrné hodnoty za každých 5 minut a v případě chybějících dat je nahrazuje hodnotou 0.

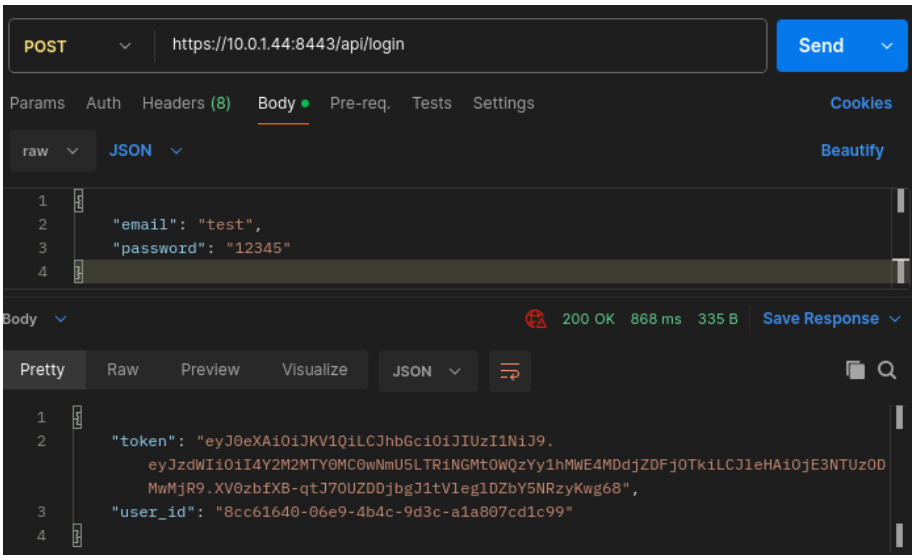
```
SELECT
    time_bucket_gapfill('5 minutes', time, now() - interval '24 hours',
now()) AS bucket,
    COALESCE(AVG(temperature), 0) AS avg_temperature,
    COALESCE(AVG(humidity), 0) AS avg_humidity,
    COALESCE(AVG(co2)::float8, 0) AS avg_co2
FROM sensor_data
WHERE device_id = $1
  AND time >= now() - interval '24 hours'
GROUP BY bucket
ORDER BY bucket ASC
```

Zdrojový kód 32: SQL dotaz pro získání senzorových dat za poslední den

V tuto chvíli funkce vždy vrací průměrná data po 5 minutách za posledních 24 hodin, ale bylo by určitě možné ji rozšířit podle potřeb aplikace. Například by mohla získávat data v zadaném časovém rozmezí a podle jeho délky přizpůsobit, po jakých časových úsecích vytváří průměry.

6.8 Testování

Pro účely testování API jsem používal nástroj Postman, ten mi umožnil rychlé a opakovatelné volání API s různým obsahem requestu a sledování odpovědi.



Obrázek 20: Testování úspěšného přihlášení v aplikaci Postman - Zdroj: vlastní

7 Mobilní aplikace

Ve chvíli, kdy máme řídicí jednotku a web server spravující data, je potřeba vytvořit ještě nějaký způsob, jakým může uživatel se systémem komunikovat, ovládat nastavení pěstírny a sledovat naměřená data.

K realizaci takového rozhraní se nabízejí dvě možnosti: webová aplikace, kterou uživatel ovládá v prohlížeči, nebo nativní desktop/mobilní aplikace. Vzhledem k tomu, že mám blíže k tvorbě desktop a mobilních aplikací, rozhodl jsem se řízení realizovat pomocí mobilní aplikace pro systém Android.

7.1 Požadavky

Jelikož je mobilní aplikace jediný způsob přístupu k systému, musí podporovat všechny operace potřebné k uživatelské správě pěstírny.

Mezi ně patří:

- Vytvoření uživatelského účtu
- Přihlášení uživatele
- Odhlášení uživatele
- Spárování účtu s pěstírnou
- Zobrazení všech pěstíren uživatele
- Vytvoření klimatického profilu pěstírny
- Úprava profilu pěstírny
- Nastavení profilu, kterým se pěstírna řídí
- Zobrazení aktuálního stavu vybrané pěstírny
- Zobrazní grafu historie hodnot pěstírny

7.2 Platforma a vývojové prostředí

Řídicí mobilní aplikaci jsem vyvíjel v jazyce Kotlin, který je dnes již primárním jazykem používaným pro vývoj Android aplikací. Jako vývojové prostředí jsem použil Android Studio, jelikož se jedná o oficiální nástroj pro vývoj Android aplikací, poskytující všechny potřebné nástroje k vývoji, sestavení a testování aplikace.

7.3 Architektura

Pro přehlednost kódu a menší závislost mezi jednotlivými částmi jsem zvolil návrhový vzor Model View ViewModel (MVVM). Jedná se o populární vzor architektury pro vývoj grafických aplikací. Hlavním principem MVVM je rozdělení aplikace do tří částí. [56]

- **Model**

Představuje naše data, jako jsou lokálně uchovávané objekty, se kterými aplikace pracuje, nebo přístup k databázi, ze které aplikace data získává. Dále může obsahovat logiku práce s daty.

- **View**

Tvoří grafické rozhraní, které vidí uživatel. Definuje, z čeho se skládá obrazovka, ale neuchovává v sobě žádná data, ani logiku aplikace.

- **ViewModel**

Tvoří spojující prvek mezi **modelem** a **view**. Pracuje s **modely** a uchovává v sobě data, která jsou zobrazována pomocí **view**. Dále obsahuje funkce, které uživatel skrze **view** aktivuje a které tvoří logiku aplikace.

Díky tomuto rozdělení na sobě nejsou jednotlivé vrstvy aplikace závislé a je možné zachovat funkcionalitu i stav dat, zatím co se změní view, například při změně orientace obrazovky. Dále se toho dá využít při testování, kdy lze vytvořit testy pro ViewModel bez potřeby ovládání skrze View.

7.3.1 Jetpack Compose

Tradiční způsob navržení uživatelského rozhraní android aplikací je pomocí XML souborů obsahujících vnořené XML prvky, definující, co aplikace zobrazí. V posledních letech však vznikla i modernější alternativa. Jedná se o toolkit Jetpack Compose, který umožňuje definovat grafické rozhraní pomocí Kotlin funkcí značených atributem *@Composable*. Každý prvek ve scéně je pak dalším voláním *@Composable* funkce. Parametry zobrazených prvků měníme skrze atributy volané funkce, kde například u funkce *Text()* použijeme parametr *text* pro určení zobrazovaného textového řetězce. [57]

```
Text(
    text = "Přihlášení",
    style = MaterialTheme.typography.headlineMedium
)
```

Zdrojový kód 33: Ukázka volání composable funkce

Pro funkce představující prvky, které mohou mít potomky, jako je například *Column()* je pro vložení využito bloku kódu s lambda výrazy. Tím dosáhneme stromové struktury zobrazených prvků, jako tomu bylo u XML.

```
@Composable
fun MyScreen() {
    Column(
        modifier = Modifier.padding(16.dp)
    ) {
        Text("Hello")
        Text("World")
        Button(onClick = { /* do something */ }) {
            Text("Click Me")
        }
    }
}
```

Zdrojový kód 34: Vnořené volání funkcí

Díky tomuto rozdělení scény na funkce představující jednotlivé grafické prvky můžeme pak funkce opakovaně volat v místech, kde chceme daný prvek zobrazit.

Pokud zobrazujeme data z ViewModelu, a chceme, aby *@Composable* funkce reagovala na jejich změnu, používáme metodu *collectAsState()*, pro převod na typ *State*. *Composable* funkce zobrazující tuto proměnnou je poté při změně dat ve ViewModelu automaticky aktualizována.

```
val count: State<Int> = viewModel.count.collectAsState()
Text("Count: ${count.value}")
```

Zdrojový kód 35: Zobrazení dat z ViewModelu

7.3.2 Dependency injection

ViewModely často potřebují pro fungování instance jiných tříd, jako jsou třídy uchovávající data stavu aplikace, nebo třídy zprostředkující komunikaci s databází. Abychom tuto potřebu ViewModelu naplnili, museli bychom tyto objekty manuálně vytvořit a předat konstruktoru ViewModelu. Toto řešení by bylo složité a nepřehledné.

Lepší způsob je využití dependency injection poskytnuté knihovnou Dagger Hilt. Ta nám umožní pomocí atributu *@Inject* označit třídy, které má Hilt poskytovat. [58]

```
@Singleton
class RuntimeData @Inject constructor(
) {
...
}
```

Zdrojový kód 36: Třída poskytovaná Hilt

Třídu našeho ViewModelu označíme attributem *@HiltViewModel*, aby Hilt věděl, že třídě má poskytnout třídy, na kterých je závislá.

```
@HiltViewModel
class FarmStateViewModel @Inject constructor(
    private val userPreferences: UserPreferences,
    private val runtimeData : RuntimeData,
    private val repository: DataRepository,
    ) : ViewModel(){
      ...
}
```

Zdrojový kód 37: Označení třídy ViewModelu

Pak je již potřeba jen získat instanci ViewModelu do *Composable* funkce představující naše View. Toho dosáhneme voláním funkce *hiltViewModel()* v parametrech *Composable funkce*.

```
@Composable
fun FarmStateScreen(
    viewModel: FarmStateViewModel = hiltViewModel(),
){ ... }
```

Zdrojový kód 38: Získání ViewModelu uvnitř View funkce

Poté již View má referenci na ViewModel, který používá a ten má všechny potřebné závislosti. Hilt zajišťuje životnost vkládaných závislostí a ve chvíli, kdy končí životnost View, odstraní i jeho ViewModel.

7.3.3 Navigace

Řídicí aplikace k funkci potřebuje několik obrazovek, mezi kterými uživatel může přepínat. Ty jsou tvořeny jednotlivými *@Composable* funkcemi, které představují naše Views. Potřebujeme

však mezi jednotlivými Views přecházet a implementace pomocí volání funkce otevíraného View by byla dosti omezující.

Naštěstí Jetpack Compose má na toto řešení v podobě tříd *NavController* a *NavHost*. Třída NavController funguje na principu zásobníku. Když navigujeme z jedné obrazovky na druhou, pomocí funkce *navigate("route")*, tak přidá nově otevřenou obrazovku na vrchol zásobníku. Uchovává tedy aktuální obrazovku, ale i všechny před ní. Díky tomu pokud uživatel zařízení udělá krok zpět, aplikace ví, na kterou obrazovku se má vrátit. Takto do nekonečna přidávat nové obrazovky na zásobník, by však nebylo vhodné řešení, proto nabízí i další funkce pro navigaci. Funkce *popBackStack()* vrací o krok zpátky a funkce *popUpTo("route")* vrací na specifikovanou obrazovku, přičemž odstraňuje všechny, co byly po ní.

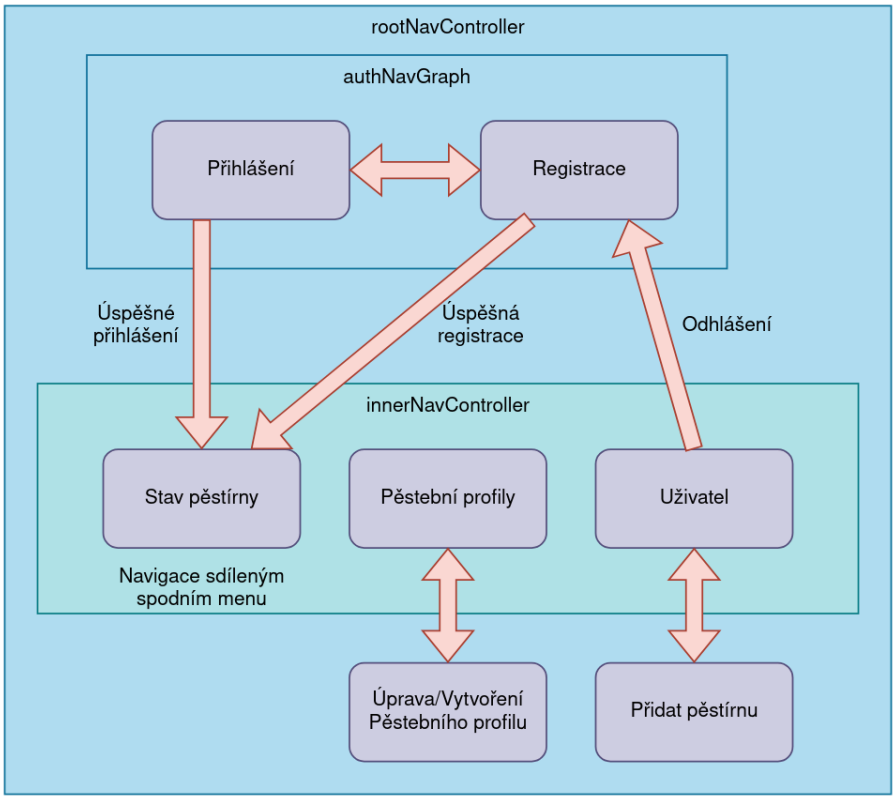
Abychom mohli využít navigačních funkcí, které nabízí *NavController*, musíme funkce představující jednotlivé obrazovky dát do *@Composable* funkce *NavHost*. Té předáme vytvořený *NavController*, podle jehož stavu bude přepínat zobrazený obsah. Ve funkci *NavHost* je dále potřeba registrovat jednotlivé cíle, do kterých může navigovat, spolu s funkcí tvořící zobrazený obsah.

```
val navController = rememberNavController()
NavHost(
    navController = navController,
    startDestination = "login"
) {
    composable("login") { LoginScreen() }
    composable("signup") { SignUpScreen() }
    composable("main") { MainScreen() }
}
```

Zdrojový kód 39: Základní podoba navigačního grafu

Pro mojí aplikaci jsem použil vrstveného navigačního grafu. Ten vedle hlavních obrazovek aplikace obsahuje druhý navigační graf, tvořený obrazovkami přihlášení a registrace. Aplikace se spouští v tomto autentizačním grafu, kde vstupní obrazovka je ta přihlašovací. Odtamtud je pomocí navControleru možné navigovat do dalších částí aplikace.

Obrazovky Stav, Profily a Uživatel jsou sdruženy pod hlavní obrazovkou, mají vlastní NavController a přechody mezi obrazovkami jsou realizovány sdíleným komponentem *NavigationBar*, který tvoří menu na spodní hraně obrazovky.



Obrázek 21: Navigační schéma aplikace - Zdroj: vlastní

7.3.4 Komunikace s API

Pro vytvoření HTTP klienta jsem použil knihovnu OkHttp. Klient má za úkol navázání spojení s web serverem, použití TLS certifikátu, zapsání hlavičky a těla requestů, nebo také načítání příchozích odpovědí. Ve chvíli, kdy je uživatel přihlášen, aplikace uchovává jeho JWT token a OkHttpClient ho přidává pro autentizaci do hlavičky requestů. Vytvoření klienta opět vyžadovalo dependency injection, což je řešeno pomocí Dagger Hilt podobně jako u ViewModelů viz kapitola 7.3.2.

```
    @Provides
    @Singleton
    fun provideOkHttpClient(
        authInterceptor: AuthInterceptor
    ): OkHttpClient {
        return OkHttpClient.Builder()
            .addInterceptor(authInterceptor)
            .build()
    }
```

Zdrojový kód 40: Vytvoření HTTP klienta

Další úroveň abstrakce tvoří knihovna Retrofit. Ta pracuje s již vytvořeným HTTP klientem a umožní nám vytvořit rozhraní, odpovídající přístupovým bodům našeho API. Funkce pro jednotlivé requesty mají určený typ requestu, ale také datové typy reprezentující tělo requestu a o tělo odpovědi. Ty jsou při zasílání serializovány do formátu JSON.

```
interface AuthApi {
    @POST("/api/login")
    suspend fun login(@Body request: LoginRequest): Response<AuthResponse>
}
```

Zdrojový kód 41: Rozhraní představující API request pro přihlášení

Z těchto rozhraní pro komunikaci s API poté Retrofit generuje funkce provádějící samotnou konverzi a výměnu dat.

Tyto Retrofit funkce volám z funkcí, které mi navrácenou hodnotu podle jejího obsahu převedou na jeden z typů třídy Resource, podle které už může aplikace reagovat na výsledek API volání.

```
sealed class Resource<T> {
    class Loading<T> : Resource<T>()
    data class Success<T>(val data: T) : Resource<T>()
    data class Error<T>(val message: String) : Resource<T>()
}
```

Zdrojový kód 42: Resource třída obalující hodnotu navrácenou z API

7.4 Funkce

7.4.1 Přihlášení a registrace

První obrazovka, kterou uživatel po spuštění aplikace uvidí, mu nabídne přihlášení. Rozložení obrazovky je jednoduché, tvořené pouze funkcí *Column* obsahující funkci *Text* pro nadpis, *Spacer* pro vytvoření mezery, *OutlinedTextField* pro vstupní pole a tlačítka typu *Button* a *TextButton*. *OutlineTextField* pro zadání hesla má nastavenou transformaci skrývající heslo viz Zdrojový kód 43.

Při stisku tlačítka *Přihlásit* aplikace odesílá serveru request pro přihlášení obsahující zadané hodnoty. Pokud odpověď značí úspěch, aplikace ukládá získaný JWT do *DataStore* aplikace, kde je uchován i po vypnutí.

Při úspěšném přihlášení aplikace zobrazí hlavní obrazovku se stavem pěstírny.

Textové tlačítko *Nemám účet* uživatele přepne do obrazovky Registrace.

Obrázek 22: Přihlašovací obrazovka - Zdroj: vlastní

```
OutlinedTextField(
    value = uiState.password,
    onValueChange = { viewModel.updatePassword(it) },
    label = { Text("Heslo") },
    modifier = Modifier.fillMaxWidth(),
    visualTransformation = PasswordVisualTransformation()
)
```

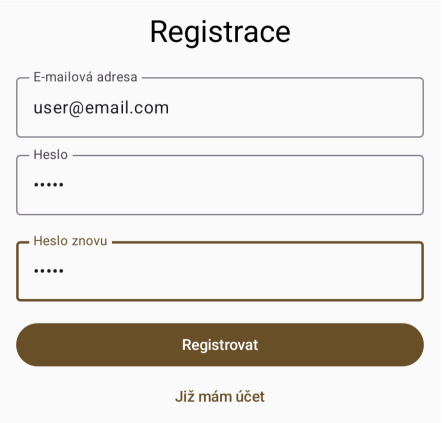
Zdrojový kód 43: Vstupní pole pro zadání hesla

Vstupní pole jsou vázána na hodnoty uvnitř ViewModelu, které se v případě pokusu o přihlášení použijí k vytvoření HTTP requestu.

Obrazovka registrace je realizovaná stejným způsobem, jako obrazovka přihlášení. Jediným rozdílem jsou dvě pole pro zadání hesla, u jejichž obsahu je podmínka stejnosti před odesláním requestu pro registraci.

Tlačítko *Již mám účet* uživatele opět vrací na přihlašovací obrazovku.

Stav obrazovky a jejích prvků je ve ViewModelu uchováván pomocí třídy *SignUpUiState*.



Obrázek 23: Registrační obrazovka - Zdroj: vlastní

```
data class SignUpUiState(
    val email: String = "",
    val passwordFirst: String = "",
    val passwordSecond: String = "",
    val isLoading: Boolean = false,
    val errorMessage: String? = null,
    val isSignedUp: Boolean = false
)
```

Zdrojový kód 44: Stav registrační obrazovky

7.4.2 Hlavní obrazovka

Po úspěšném přihlášení, nebo registraci se aplikace přepne do hlavní obrazovky, která je sestavená z obrazovek *Stav pěstírny*, *Pěstební profily* a *Uživatel*. Ty spojuje společné menu na spodní hraně obrazovky, pomocí kterého lze mezi obrazovkami přecházet.

To je realizováno pomocí funkce *Scaffold* a jejího parametru *bottomBar*. Zde jsou vkládány jednotlivé funkce *NavigationBarItem* reprezentující položky menu. Těm je potřeba nastavit cestu obrazovky, kterou má NavHost otevírat a další formátování, jako je text, nebo ikona.

```
Scaffold(
    bottomBar = {
        NavigationBar {
            NavigationBarItem(
                selected = currentRoute == Routes.FARM_STATE,
                onClick = {
                    innerNavController.navigate(Routes.FARM_STATE) {

popUpTo(innerNavController.graph.findStartDestination().id) {
                            saveState = true
                        }
                        launchSingleTop = true
                        restoreState = true
                    }
                },
                icon = { Icon(Icons.Default.Info, "Farm State") },
                label = { Text("Stav") }
            )
```

Zdrojový kód 45: Menu hlavní obrazovku

Dále je potřeba *Scaffold* dát funkci NavHost, definující zobrazované obrazovky spolu s výběrem vstupní obrazovky, která se zobrazí po inicializaci. V tomto případě hlavní obrazovka při spuštění otevírá obrazovku *Stav pěstíny*.

7.4.3 Stav pěstírny

Pomocí obrazovky stavu pěstírny může uživatel vybrat jednu z pěstíren spojených se svým účtem. U vybrané pěstírny má možnost zvolit pěstební profil, kterým se řídí. Výběr pěstírny i výběr pěstebního profilu jsou realizovány vlastní *@Composable* funkcí založenou na *ExposedDropdownMenuBox*.

Obrázek 24: Výběr použitého pěstebního profilu - Zdroj: vlastní

Po vybrání jiné pěstírny se automaticky aktualizují všechny hodnoty s ní spojené. Po vybrání pěstebního profilu se odesílá request na změnu nastavení pěstírny.

Další část obrazovky tvoří stavové informace prostředí pěstírny, doplněné o čas pořízení zobrazených dat. Pomocí této časové informace by v případě výpadku serveru, nebo řídicí jednotky uživatel poznal, že data nejsou aktuální.

Obrázek 25: Zobrazení aktuálního stavu - Zdroj: vlastní

Pro vykreslení stavových hodnot je použita vlastní *@Composable* funkce používající funkci *Card*, a v ní *Column* obsahující *Text* prvky označení a naměřené hodnoty. K naměřeným hodnotám je pomocí *String.format()* přidána jednotka. Pro zobrazení času používám funkci převádějící *timestamp* hodnoty do *dd.MM.yyyy HH:mm* formátu.

Stavové hodnoty se automaticky získávají ze serveru při otevření obrazovky, nebo při změně vybrané pěstírny. Aby však zůstávaly aktuální, probíhá i pravidelná aktualizace každých 30s. O to se stará asynchronně běžící cyklus vytvořený pomocí *viewModelScope.launch* při vytvoření ViewModelu.

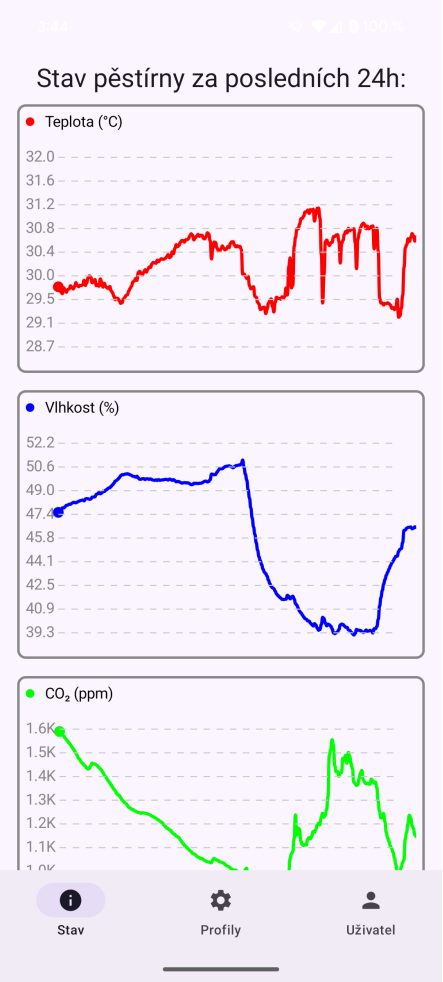
```
init {
    startPeriodicUpdates(30_000L)
}
fun startPeriodicUpdates(intervalMs: Long) {
    viewModelScope.launch {
        while (isActive) {
            loadCurrentValues()
            delay(intervalMs)
        }
    }
}
```

Zdrojový kód 46: Cyklus periodicky získávající nová data

Poslední částí obrazovky jsou grafy zobrazující stav naměřených hodnot za posledních 24 hodin. Jejich zdrojem je pole číselných hodnot pro každou veličinu. Vytvořením těchto hodnot se zabývá kapitola 6.7.3. Díky většinu objemu dat a délce zobrazovaného intervalu není potřeba

provádět pravidelnou aktualizaci a aplikace si vystačí se získáním dat při otevření obrazovky. K vytvoření grafů jsem použil knihovnu AAY-chart[59].

Funkce knihovny byly jednoduché na použití, ale ukázalo se, že pro vykreslení podobných dat není knihovna vhodná. Je tomu tak, protože funkce LineChart osu x nebere jako souřadnice, ale pouze jako popis zanesené hodnoty. To pro nás znamená, že pokud máme mezeru v datech, musíme ji nahradit například hodnotou 0, bez toho by úsek bez dat graf vůbec nezohlednil. Větším problémem bylo, že popisy osy x a osy y jsou zobrazeny pro každý bod zobrazený v grafu. Jelikož při vzorkování po 5 minutách máme 288 hodnot v intervalu 24 hodin, graf již nebyl schopný popisy os ani zobrazit.



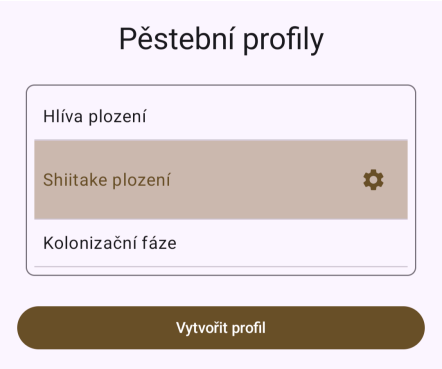
Obrázek 26: Zobrazení grafů - Zdroj: vlastní

Částečným řešením bylo vytvoření druhého grafu s výrazně menším množstvím vzorků ve stejném rozsahu jako hlavní graf. Jeho křivku jsem nastavil na transparentní a grafy překryl přes sebe, čímž vznikl popis os hlavního grafu. Toto řešení však bylo dosti omezující a na ose x nepřesné, jelikož se hlavní graf při větším množství hodnot roztahuje a ujíždí mimo vlastní zobrazovací prostor.

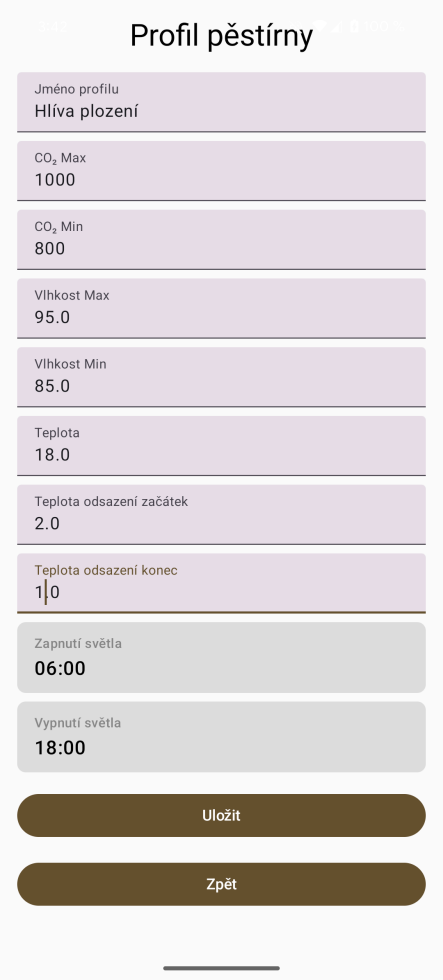
Zobrazované hodnoty získané řídicí jednotkou nejsou stálé, jelikož k jejímu výstupu nejsou připojena žádná řízená zařízení, a pouze snímá své okolí. Ve skutečném nasazení jednotky by grafy měly ukazovat, jak se prostředí drží okolo zadaných cílových hodnot prostředí.

7.4.4 Pěstební profily

Pro správu pěstebních profilů jsem vytvořil obrazovku obsahující seznam profilů vázaných na uživatele. Obrazovka umožňuje editaci stávajícího profilu, nebo vytvoření nového.



Obrázek 27: Seznam pěstebních profilů - Zdroj: vlastní



Obrázek 28: Tvorba pěstebního profilu - Zdroj: vlastní

Vytvoření nového pěstebního profilu a úprava stávajícího jsou realizovány stejnou obrazovkou. Při vytváření nového aplikace vyplní základní hodnoty, které si uživatel následně upraví podle potřeby.

Při uložení aplikace odesílá POST request webserveru, tam se na základě přítomnosti id rozhodne, zda-li jde o úpravu stávajícího profilu, nebo o vložení nového. Po vložení se aplikace vrátí na seznam pěstebních profilů, který obnoví a ten by již měl obsahovat změněné nebo nově vytvořené hodnoty.

7.4.5 Uživatel

Obrazovka uživatel zobrazuje e-mail aktuálně přihlášeného uživatele. Ten získá pomocí třídy *UserPreferences*, která zprostředkovává přístup k datům uchovaným v *DataStore* aplikace. Pokud uživatel zvolí možnost *Odhlásit*, proběhne odstranění uchovávaných informací o uživateli v *DataStore* a aplikace zobrazí přihlašovací obrazovku.

Obrázek 29: Obrazovka Uživatel - Zdroj: vlastní

Dále se zde nachází seznam pěstíren spravovaných uživatelem. Uživatel má možnost přidat novou pomocí funkce párování.

Obrázek 30: Obrazovka Uživatel - Zdroj: vlastní

Pro úspěšné spárování musí uživatel zadat název pěstírny, pod jakým ji dále uvidí v seznamu pěstíren.

Dále musí z řídicí jednotky opsat kód, který mu zobrazila po stisku tlačítka. Pokud toto splní a stiskne tlačítko *Přidat pěstírnu*, aplikace odešle serveru request k párování, obsahující jméno, párovací kód a id uživatele. Pokud server najde párovací kód a je stále aktivní, vloží nový záznam do tabulky *user_devices*, čímž vytvoří relaci mezi uživatelem a pěstírnou.

7.5 Možná další rozšíření

Výsledná implementace aplikace byla spíše minimální pro potřeby ovládání systému a v případě nasazení do reálného provozu by určitě bylo vhodné aplikaci ještě rozšířit.

Operace změny jména pěstíren, nebo odebírání pěstírny a pěstebního profilu by určitě byly další v seznamu očekávaných funkcionalit.

Další částí k rozšíření by byly zobrazované grafy. K vykreslování grafů by bylo vhodné doplnit výběr intervalu, aby se například mimo posledního dne, dalo zobrazit rozmezí vybrané hodiny.

Dále by bylo vhodné, aby řídicí jednotky odesílaly i chybové stavy, které by se poté zobrazovaly v mobilní aplikaci. To by se dalo použít i pro vytvoření notifikací, které by uživatele včas upozornily na problém.

Pro účinnou správu většího množství pěstíren by bylo možné vytvořit obrazovku s jejich přehledem obsahujícím ukazatel stavu a aktuálně používaný pěstební profil.

ZÁVĚR

Zadáním této práce bylo navrhnout a alespoň částečně implementovat systém pro řízení klimatických podmínek pro pěstování hub. Takový systém je tvořen ze tří částí. Každá část představovala novou sadu problémů, které bylo potřeba vyřešit.

Nejprve bylo potřeba nastudovat problematiku samotného pěstování hub, abych mohl zdůvodnit potřebu řídicího systému a mohl určit požadavky, jaké musí splňovat. V teoretické části byly popsány fáze růstu hub a podmínky, jaké vyžadují. Dále jsem provedl průzkum konkurenčních produktů pro zhodnocení potřeby vývoje nového systému a případnou úpravu požadavků.

V části zabývající se samotnou řídicí jednotkou jsem provedl průzkum nabízených mikrokontrolérů, abych mohl vybrat takový, co nejlépe splňuje požadavky tohoto projektu. Kromě mikrokontroléru bylo potřeba vybrat i moduly tvořící funkčnost řídicí jednotky. Jelikož je tato práce určena pro obor informačních technologií, zapojení řídicí jednotky zůstalo ve formě prototypu a důraz byl kladen především na softwarovou stránku. V softwarové části šlo o nalezení a úpravy knihoven pro práci s připojenými moduly a následný návrh komponent systému, které provádějí jednotlivé úkony tvořící funkčnost řídicí jednotky. Zde bylo důležité zajistit vzájemnou spolupráci jednotlivých částí systému.

Pro serverovou část byla s přihlédnutím na charakter uchovávaných dat vybrána databáze TimescaleDB vhodná pro práci s časovými daty. Jako prostředníka mezi uživatelem, databází a řídicí jednotkou jsem vytvořil REST API realizované pomocí frameworku Axum. V této části bylo kromě přístupu do databáze nutné implementovat i způsoby zabezpečení komunikace a ověřování uživatele.

V poslední řadě bylo potřeba vytvořit mobilní aplikaci, pomocí které může uživatel ovládat sebou spravované pěstírny. To bylo realizováno v podobě Android aplikace s grafickým rozhraním vytvořeným pomocí Jetpack Compose.

Pro skutečné nasazení v provozu by bylo vhodné systém dále rozvíjet pro doplnění nedostatků. Avšak podoba systému v době odevzdání plní všechny určené požadavky a v případě nasazení do provozu by měla vyhovovat potřebám pro automatizaci pěstebního prostředí.

LITERATURA

[1] STAMETS, Paul. *Growing gourmet and medicinal mushrooms.* 3rd. Berkeley, Calif. : Ten Speed Press, 2000. ISBN 978-1-58008-175-7.

[2] SHROOLY. Shrooly growing device. [online]. 2025. [cit 2025-07-22]. Dostupné z: https://eu.shrooly.com/pages/product-generic

[3] NORTHSPORE. 'BoomRoom II' Automated Martha Tent Mushroom Grow System. [online]. 2025. [cit 2025-07-21]. Dostupné z: https://northspore.com/products/boomroom-ii-automated-martha-tent-mushroom-grow-system

[4] FANCOM. Growing exotic mushrooms with automated mushroom growing. [online]. 2025. [cit 2025-07-22]. Dostupné z: https://www.fancom.com/system/growing-exotic-mushrooms#page

[5] GABRIEL, Kyle. Mycodo. [online]. 2025. [cit 2025-07-22]. Dostupné z: https://github.com/kizniche/Mycodo

[6] RASPBERRY PI. Raspberry Pi 5. [online]. 2025. [cit 2025-07-24]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-5/

[7] JOSEPH, Jobit. Types of Arduino Boards – From Basic Embedded to Advanced AI Boards. [online]. 2025. [cit 2025-07-26]. Dostupné z: https://circuitdigest.com/article/different-types-of-arduino-boards

[8] RASPBERRY PI. Raspberry Pi Pico 2. [online]. 2025. [cit 2025-07-26]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-pico-2/

[9] ESPBOARDS.DEV. Exploring the ESP32 versions. Differences and similarities. [online]. 2024. [cit 2025-07-27]. Dostupné z: https://www.espboards.dev/blog/esp32-soc-options/

[10] SEEED STUDIO. Exploring the ESP32 versions. Differences and similarities. [online]. [cit 2025-07-27]. Dostupné z: https://wiki.seeedstudio.com/XIAO_ESP32C3_Getting_Started/

[11] TIŠNOVSKÝ, Pavel. Komunikace po sériové sběrnici I^2C. [online]. 2009. [cit 2025-07-27]. Dostupné z: https://www.root.cz/clanky/komunikace-po-seriove-sbernici-isup2supc/

[12] SOLDERED.COM. What is the I^2C communication protocol. [online]. 2023. [cit 2025-07-27]. Dostupné z: https://soldered.com/learn/what-is-the-i2c-communication-protocol/

[13] SENSIRION. SCD30. [online]. [cit 2025-07-27]. Dostupné z: https://sensirion.com/products/catalog/SCD30

[14] SENSIRION. SCD40. [online]. [cit 2025-07-27]. Dostupné z: https://sensirion.com/products/catalog/SCD40

[15] SENSIRION. SCD41. [online]. [cit 2025-07-27]. Dostupné z: https://sensirion.com/products/catalog/SCD41

[16] PROTRONIX. Co je princip NDIR a proč ho používáme v našich čidlech?. [online]. [cit 2025-07-27]. Dostupné z: https://www.cidla.cz/clanky/cidla-kvality-vzduchu/co-je-princip-ndir-a-proc-ho-pouzivame-v-nasich-cidlech/

[17] EMARIETE. Sensirion SCD41 and SCD40 CO_2 Sensors. [online]. 2021. [cit 2025-07-27]. Dostupné z: https://emariete.com/en/sensor-co2-sensirion-scd40-scd41/

[18] LASKAKIT. 8-kanálů relé modul 5VDC 250VAC 10A. [online]. [cit 2025-07-28]. Dostupné z: https://www.laskakit.cz/8-kanalu-rele-modul-5vdc-250vac-10a/

[19] HOLISTICS SOFTWARE. dbdiagram.io. [online]. 2025. [cit 2025-08-15]. Dostupné z: https://dbdiagram.io/home

[20] HERNANDA, Radityo. Build CRUD REST API with Rust and MySQL using Axum & SQLx. [online]. 2024. [cit 2025-08-15]. Dostupné z: https://medium.com/@raditzlawliet/build-crud-rest-api-with-rust-and-mysql-using-axum-sqlx-d7e50b3cd130

[21] RUST. Axum_server. [online]. 2025. [cit 2025-08-15]. Dostupné z: https://docs.rs/axum-server/latest/axum_server/tls_rustls/index.html

[22] KANSARA, Karan, ZAVERI, Vishal, SHAH, Shreyans, DELWADKAR, Sandip and JANI, Kaushal. Sensor based automated irrigation system with IOT: A technical review. *International Journal of Computer Science and Information Technologies*. 2015. Vol. 6, no. 6, p. 5331–5333.

[23] LACHHAB, Fadwa, BAKHOUYA, Mohamed, OULADSINE, Radouane and ESSAAIDI, Mohammed. A context-driven platform using Internet of things and data stream processing for heating, ventilation and air conditioning systems control. *Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering*. [online]. 2019. Vol. 233, no. 7, p. 877–888. DOI 10.1177/0959651819841534.

[24] Začínáme s domácím pěstováním hub. [online]. 2013. [cit 2025-07-22]. Dostupné z: https://www.ohoubach.cz/clanky/detail/12/disk-pridani-prispevku/

[25] TESLENKOVA, Larisa. How do you grow a second flush of oyster mushrooms?. [online]. 2024. [cit 2025-07-22]. Dostupné z: https://veshenka-expert.info/en/second-flush-of-oyster-mushrooms/

[26] FANCOM. Lumina 762 climate computer for exotic mushrooms. [online]. 2025. [cit 2025-07-22]. Dostupné z: https://ss-usa.s3.amazonaws.com/c/308478790/media/188167a1ce21ac2bf05001573871426/762%20Exotic%20mushroom%20computer%20Factsheet_gb.pdf

[27] GABRIEL, Kyle. Automated Hydroponic System Build. [online]. 2020. [cit 2025-07-22]. Dostupné z: https://kylegabriel.com/projects/2020/06/automated-hydroponic-system-build.html

[28] GABRIEL, Kyle. Automated Hydroponic System Build. [online]. 2015. [cit 2025-07-22]. Dostupné z: https://kylegabriel.com/projects/2015/10/artificial-bat-cave.html

[29] VEMEKO. The Differences between MCU and SBC. [online]. 2024. [cit 2025-07-26]. Dostupné z: https://www.vemeko.com/blog/the-differences-between-mcu-and-sbc.html

[30] ARDUINO. What is Arduino?. [online]. 2025. [cit 2025-07-26]. Dostupné z: https://www.arduino.cc/en/Guide/Introduction/

[31] ARDUINO. Arduino Hardware. [online]. 2025. [cit 2025-07-26]. Dostupné z: https://www.arduino.cc/en/hardware/

[32] BARROZO, Jharwin. STM32: Things You Need to Know. [online]. 2024. [cit 2025-07-26]. Dostupné z: https://www.flux.ai/p/blog/stm32-things-you-need-to-know

[33] PJRC. Teensy® 4.1 Development Board. [online]. [cit 2025-07-26]. Dostupné z: https://www.pjrc.com/store/teensy41.html

[34] SANTOS, Rui. Getting Started with the ESP32 Development Board. [online]. 2024. [cit 2025-07-27]. Dostupné z: https://randomnerdtutorials.com/getting-started-with-esp32/

[35] ESPRESSIF. ESP32-C3 Series Datasheet Version 2.1. [online]. 2025. [cit 2025-07-27]. Dostupné z: https://www.espressif.com/sites/default/files/documentation/esp32-c3_datasheet_en.pdf

[36] KALINSKY, David and KALINSKY, Roee. Introduction to I^2C. *Embedded Systems Programming*. 2001. Vol. 14, no. 8, p. 1101–1105.

[37] EMARIETE. Sensirion SCD30 dual-channel NDIR CO_2 Sensor. [online]. 2024. [cit 2025-07-27]. Dostupné z: https://emariete.com/en/sensor-co2-ndir-sensirion-scd30-dual-channel/

[38] SOLOMON SYSTECH LIMITED. SSD1306. [online]. 2008. [cit 2025-07-28]. Dostupné z: https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf

[39] MAXIM INTEGRATED PRODUCTS. DS1307. [online]. 2015. [cit 2025-07-28]. Dostupné z: https://www.analog.com/media/en/technical-documentation/data-sheets/ds1307.pdf

[40] DIETRICH, Shawn. Understanding the Basics of Pulse Width Modulation (PWM). [online]. 2022. [cit 2025-08-13]. Dostupné z: https://control.com/technical-articles/understanding-the-basicsof-pulse-width-modulation-pwm/

[41] ESPRESSIF. Official IoT Development Framework. [online]. [cit 2025-08-13]. Dostupné z: https://www.espressif.com/en/products/sdks/esp-idf

[42] ESPRESSIF. FreeRTOS (IDF). [online]. [cit 2025-08-13]. Dostupné z: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos_idf.html

[43] IGNACIO, Brian A. ESP-IDF Extension for VS Code. [online]. [cit 2025-08-13]. Dostupné z: https://github.com/espressif/vscode-esp-idf-extension/blob/master/README.md

[44] ESPRESSIF. SPIFFS Filesystem. [online]. [cit 2025-08-14]. Dostupné z: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/storage/spiffs.html

[45] YI, Cheah Hao. Getting Started with Wi-Fi on ESP-IDF. [online]. 2024. [cit 2025-08-14]. Dostupné z: https://developer.espressif.com/blog/getting-started-with-wifi-on-esp-idf/

[46] SNTP: All You Need To Know About Simple Network Time Protocol. [online]. 2020. [cit 2025-08-14]. Dostupné z: https://timetoolsltd.com/ntp/sntp-overview/

[47] HOSSEIN.M. DS1307 Library. [online]. 2023. [cit 2025-08-14]. Dostupné z: https://github.com/Hossein-M98/DS1307

[48] MA-LWA-RE. ESP32 SCD4x Library. [online]. 2022. [cit 2025-08-14]. Dostupné z: https://github.com/ma-lwa-re/esp32-scd4x

[49] NOPNOP2002. esp-idf-ssd1306. [online]. [cit 2025-08-14]. Dostupné z: https://github.com/nopnop2002/esp-idf-ssd1306

[50] GUPTA, Lokesh. What is REST?. [online]. 2025. [cit 2025-08-14]. Dostupné z: https://restfulapi.net/

[51] TIMESCALEDB. TimescaleDB. [online]. 2025. [cit 2025-08-15]. Dostupné z: https://github.com/timescale/timescaledb

[52] axum. [online]. 2025. [cit 2025-08-15]. Dostupné z: https://github.com/tokio-rs/axum

[53] SSL.COM. What is HTTPS?. [online]. 2025. [cit 2025-08-15]. Dostupné z: https://www.ssl.com/faqs/what-is-https/

[54] JWT.IO. Introduction to JSON Web Tokens. [online]. [cit 2025-08-15]. Dostupné z: https://www.jwt.io/introduction#what-is-json-web-token

[55] PROUILLET, Vincent. jsonwebtoken. [online]. [cit 2025-08-15]. Dostupné z: https://github.com/Keats/jsonwebtoken

[56] CHUGH, Anupam. Android MVVM Design Pattern. [online]. 2022. [cit 2025-08-16]. Dostupné z: https://www.digitalocean.com/community/tutorials/android-mvvm-design-pattern

[57] YIĞIT, Mustafa. Say Hello to Jetpack Compose and Compare with XML. [online]. 2021. [cit 2025-08-16]. Dostupné z: https://blog.kotlin-academy.com/say-hello-to-jetpack-compose-and-compare-with-xml-6bc6053aec13

[58] ANDROID. Dependency injection with Hilt. [online]. [cit 2025-08-16]. Dostupné z: https://developer.android.com/training/dependency-injection/hilt-android

[59] CHANCE, The. AAY-chart. [online]. 2022. [cit 2025-08-17]. Dostupné z: https://github.com/TheChance101/AAY-chart

PŘÍLOHY

PŘÍLOHA A: Zdrojové kódy

PŘÍLOHA A: Zdrojové kódy

Archív zdrojových kódů všech částí systému zabalený v souboru source.zip