

ETDK-dolgozat

Orbán Gergő

XXIX. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)

Informatika II.: innovatív számítástechnikai termékek, alkalmazások szekció

Kolozsvár, 2026. május 14–17.

LivingSmart

Intelligens jelenlét-szimuláció okosotthonokban



Szerzők:

Orbán Gergő

III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Témavezetők:

dr. Sulyok Csaba, egyetemi adjunktus,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Gálfi Csaba-Erik, szoftverfejlesztő,

SOFTECH SRL

Zsebe Ákos, szoftverfejlesztő,

SOFTECH SRL



XXIX. ERDÉLYI TUDOMÁNYOS DIÁKKÖRI KONFERENCIA REÁL- ÉS HUMÁNTUDOMÁNYOK

CONFERINȚA ȘTIINȚIFICĂ STUDENTEASCĂ DIN TRANSILVANIA, EDIȚIA A 29-A
29TH TRANSILVANIA STUDENTS' SCIENTIFIC CONFERENCE
ȘTIINȚE REALE ȘI UMANISTE // FORMAL AND EMPIRICAL SCIENCES



LivingSmart: Intelligens jelenlét-szimuláció okosotthonokban

Orbán Gergő, III. év, Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar

dr. Sulyok Csaba, egyetemi adjunktus,
Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar
Gálfi Csaba-Erik, szoftverfejlesztő,
SOFTECH SRL
Zsebe Ákos, szoftverfejlesztő,
SOFTECH SRL

A LivingSmart projekt célkitűzése egy olyan okosotthon-rendszer megtervezése és megvalósítása, amely egy helyi gateway architektúrára épül, csökkenti a felhőszolgáltatásoktól való függést, és egységes kezelést tesz lehetővé különböző gyártóktól származó, esetlegesen eltérő kommunikációs protokollokat alkalmazó eszközök számára. A rendszer képes a felhasználói szokások megfigyelésére, az ezekből képzett rutinok felépítésére, valamint azok későbbi reprodukálására, jelenlét-szimuláció céljából.

A projekt fő célja a jelenlét-szimuláció egy generált rutin alapján, amely hozzájárulhat lakásunk biztonságához távollét idején. A rutint a saját fejlesztésű Android alkalmazásunkból generálható, amely a manuálisan kapcsolható eszközök használati statisztikáján alapul. A szimuláció futtatását egy központi vezérlőegység oldja meg, amely a háztartásban található, így a szimuláció internet kapcsolat kiesése esetén is megszakításmentesen működik.

A másik fontos funkciója a rendszernek az az eszközök helyiségek szerinti csoportosítása, illetve azok csoportos vezérlése. Az azonos típusú eszközök csoportosan vezérelhetők, valamint a termosztátok akár ütemterv szerint is működtethetők, amit a gyártói cloud nem támogat.

A dolgozat a fentebb említett mobilalkalmazás és okos eszköz automatizáló rendszer architektúráját, technológiai megvalósítását, előnyeit és megvalósításának kihívásait mutatja be.



Témavezetői igazolás

Alulírott *dr. Sulyok Csaba, Gálfi Csaba-Erik és Zsebe Ákos* igazoljuk, hogy *Orbán Gergő* hallgató a(z) *LivingSmart: Intelligens jelenlét-szimuláció okosotthonokban* című dolgozatát az alulírottak szakmai irányításával készítette el, és javasoljuk az említett tudományos munka bemutatását a 2026-os XXIX. Reál- és Humántudományi Erdélyi Tudományos Diákköri Konferencián (ETDK).

Mindezek mellett igazoljuk, hogy a dolgozat megfelel a korszerű tudományos kritériumoknak, nem OTDK-n vagy nemzetközi szintű konferenciákon már bemutatott dolgozat, már megvédett államvizsga vagy magiszteri dolgozat, sem korábban már publikált dolgozat.

dr. Sulyok Csaba, egyetemi adjunktus,
Babeș–Bolyai Tudományegyetem, Matematika és Informatika Kar

Gálfi Csaba-Erik, szoftverfejlesztő,
SOFTECH SRL

Zsebe Ákos, szoftverfejlesztő,
SOFTECH SRL

Kolozsvár, 2026. október 16.

Tartalomjegyzék

Bevezető	1
1. A LivingSmart projektről	3
1.1. Funkcionalitások	3
1.2. Architektúra	4
2. Okoseszközök	5
2.1. Zigbee	5
2.2. Tasmota	6
2.3. Termosztát funkció	6
3. Gateway	8
3.1. Home Assistant	8
3.2. Backenddel való kommunikáció	8
3.3. Eszközök csatlakoztatása	9
3.4. Szokásfigyelés és lejátszás	10
3.4.1. Adatgyűjtés és profilozás	10
3.4.2. Rutin lejátszása	11
3.5. Termosztát ütemterv	11
4. Szerver	13
4.1. Funkcionalitások	13
4.1.1. Autentikáció és jogosultságkezelés	13
4.1.2. Szobák és képfeltöltés	14
4.1.3. Eszközkezelés	14
4.1.4. Csoportkezelés	15
4.1.5. Szimuláció vezérlése	16
4.1.6. Valós idejű állapot frissítések	16
4.2. Architektúra	17
4.3. Technológiák	17
5. Mobil alkalmazás	19
5.1. Funkcionalitások	19

5.1.1.	Bejelentkezés	19
5.1.2.	Főoldal és szobák	20
5.1.3.	Szoba hozzáadása és domináns szín	20
5.1.4.	Eszközök és eszköz hozzáadása	21
5.1.5.	Csoportok	21
5.1.6.	Termosztát ütemterv létrehozása	22
5.1.7.	Rutin generálás és futtatás	22
5.2.	Architektúra	22
5.2.1.	BLoC	23
5.2.2.	Hálózati réteg	23
5.2.3.	WebSocket	24
5.2.4.	Adatperzisztencia	25
5.2.5.	Dependency injection	25
5.2.6.	Nemzetköziesítés	26
5.2.7.	Flutter architektúra	26
6.	DevOps	30
6.1.	GitLab CI/CD	30
6.1.1.	Utilities szakasz	30
6.1.2.	Build szakasz	30
6.1.3.	Deploy szakasz	31
6.2.	Kubernetes	32
6.2.1.	Deployment	32
6.2.2.	Perzisztencia	33
6.2.3.	Service és Ingress	33
6.2.4.	Megbízhatóság és skálázhatóság	34
7.	Következtetések és továbbfejlesztési lehetőségek	35

Bevezető

Az okosotthon-piac az elmúlt évtizedben globális szinten is robbanásszerű növekedésen ment át. A Statista adatai szerint 2024-ben a globális okosotthon-piacon a felhasználók száma elérte az 1,47 milliárdot, és a folyamatos növekedési trend alapján ez a szám 2029-re várhatóan megközelíti a kétmilliárdot [17]. Az intelligens eszközök ilyen mértékű terjedése és a mindennapi életünkbe való integrációja jelentős új lehetőségeket kínál a lakásbiztonság és a vagyonvédelem terén.

A legfrissebb, 2026 tavaszán közzétett hivatalos adatok szerint a brit statisztikai hivatal (Crime Survey for England and Wales) 327 000 háztartási betörést regisztrált egyetlen év alatt [14]. Bár a trendek hosszú távon javulást mutatnak, az esetszám továbbra is jelentős kockázatot képvisel, ami indokolja a hatékony megelőzési technológiák, például az okosotthon-alapú rendszerek alkalmazását. A jelenlét-szimuláció hatékonyságát modern, virtuális valóság alapú kísérletek is igazolják: Van Sintemaartensdijk és munkatársai (2021) kutatása kimutatta, hogy az informális őrző-védők, illetve a lakottság látszatát keltő tényezők pusztán jelenléte (*mere presence*) önmagában is szignifikáns elrettentő erővel bír a betörőkkel szemben. A vizsgálat eredményei szerint a betörők elrettentéséhez nincs szükség aktív megfigyelésre vagy komplex beavatkozásra, mivel a lakottság pusztán érzékeltetése elegendő a kockázaterő növeléséhez [16]. Egy olyan rendszer, amely a lakó szokásos viselkedési mintáit utánozva kapcsolja az eszközöket, valószínűleg ható jelenlétet szimulál akkor is, ha a tulajdonos nincs otthon.

A LivingSmart projekt erre a biztonsági potenciálra építve kínál innovatív megoldást: egy rendszert, amely automatikusan tanulja meg és reprodukálja a felhasználó viselkedési szokásait jelenlét-szimuláció céljából. Ezzel párhuzamosan a projekt egy egységes platformot is biztosít, amely a különböző protokollú okoseszközöket egyetlen alkalmazásból teszi kezelhetővé. A rendszer helyi gateway architektúrán alapul, így az alapfunkciók külső internetkapcsolat nélkül is megbízhatóan működnek.

A jelenlét-szimuláció révén hozzájárul a lakás védelméhez, a csoportos vezérlés pedig leegyszerűsíti a felhasználói interakciókat, különösen nagyszámú okoseszkővel felszerelt otthonokban.

A LivingSmart projekt 2025 nyarán indult szakmai gyakorlatként a Codespring¹ mentorprogram keretein belül, Zsebe Ákos és Gálfi Csaba mentorálásával, Nagy Sámuel kollégával közös fejlesztésben. A projekt három fő célkitűzéssel indult. Az első cél egy olyan

¹<https://edu.codespring.ro/>

egységes platform megvalósítása, amely különböző protokollokat alkalmazó eszközöket képes egy rendszerbe integrálni, és egyetlen alkalmazásból kezelhetővé teszi azokat. A második cél a felhasználói szokások megfigyelése és a viselkedési rutin automatikus generálása, illetve visszajátzása, elsősorban a lakásbiztonság növelése érdekében. A harmadik célkitűzés pedig az eszközök csoportos vezérlése, ami lehetőséget ad arra, hogy a termosztátokat csoportosan, egy megadott ütemterv szerint vezéreljük, így az otthoni hőmérséklet-szabályozás is komfortosabbá és energiatakarékosabbá válik.

A dolgozat a következő fejezetekre tagolódik. Az 1. fejezet a rendszer általános funkcionalitásait és architektúráját tekinti át. A 2. fejezet az alkalmazott okoseszközöket és protokollokat mutatja be. A 3. fejezet a gateway belső működésével foglalkozik részletesen. A 4. fejezet a backend felépítését és szolgáltatásait ismerteti. Az 5. fejezet a mobilalkalmazás funkcióit és architektúráját tárgyalja. A 6. fejezet a DevOps megoldásokat mutatja be. A dolgozat továbbfejlesztési irányok felsorolásával zárul.

1. A LivingSmart projektről

A LivingSmart egy okoseszközöket összefogó rendszer, amelynek célja, hogy a felhasználó egyetlen alkalmazásból, egyszerűen kezelhesse otthona összes eszközeit, függetlenül azok gyártójától vagy kommunikációs protokolljától. Az alábbiakban a rendszer legfontosabb funkcióit és architektúráját tekintjük át.

1.1. Funkcionalitások

A rendszer felhasználói egy mobilalkalmazáson keresztül lépnek interakcióba a LivingSmart platformmal. Az alkalmazás az alábbi főbb funkciókat kínálja. Az eszközök vezérlése az alapvető funkció: a felhasználó ki- és bekapcsolhatja az okoskonnektorokat, kapcsolókat és reléket, beállíthatja a termosztátok célhőmérsékletét, és valós időben figyelheti az érzékelők (hőmérséklet-, mozgás- és egyéb érzékelők) aktuális értékeit.

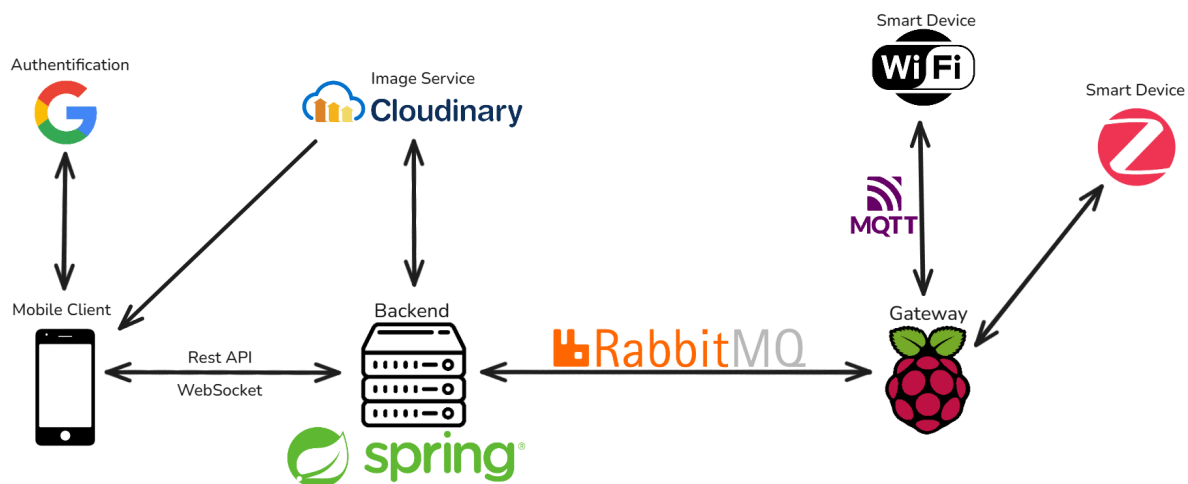
A szobaalapú szervezés segíti az áttekinthetőséget: a felhasználó szobákat hozhat létre, képpel és egyedi névvel láthatja el azokat, és az eszközöket szobákhoz rendelheti. Az alkalmazás főoldala a szobák rácsát mutatja, egy szobára koppintva pedig az abban található összes eszköz megjelenik.

A csoportos vezérlés lehetővé teszi, hogy a felhasználó több szobát egyetlen csoportba foglaljon össze, és a csoport összes, azonos típusú, eszközét egyidejűleg irányítsa. Ez különösen hasznos például akkor, ha az összes szoba összes világítóeszközét egyszerre kell ki- vagy bekapcsolni, vagy ha a teljes épület termosztátjait azonos hőmérsékletre kell állítani.

A jelenlét-szimuláló funkció megfigyeli a felhasználó eszközhasználati szokásait, és ezek alapján egy viselkedési rutint állít össze. Ezt a rutint a felhasználó elindíthatja, és a rendszer a szokásos minták szerint kapcsolgatja az eszközöket, kívülről nézve valós aktivitást imitálva, akár internetkapcsolat hiányában is. Ez a funkció elsősorban biztonsági célokat szolgál: ha a lakó hosszabb ideig nem tartózkodik otthon, a szimuláció aktívan kelthet "lakott" benyomást.

A termosztát-ütemezés lehetővé teszi, hogy a felhasználó a hét napjaira tetszőleges ütemtervet készítsen a kívánt termosztát állapotokkal. Emellett létre hozhat egy "felülíró" ütemtervet is, amely prioritást fog élvezni egyéb egyszerű ütemtervekkel szemben.

Az eszközpárosítás funkció segítségével a felhasználó könnyen új Zigbee és Wi-Fi-s eszközöket adhat a rendszeréhez.



1. ábra. A LivingSmart rendszer főbb alkotóelemei

1.2. Architektúra

A LivingSmart rendszer négy fő alkotóelemből áll, amelyek egymással jól meghatározott csatornákon kommunikálnak. Az 1. ábra a rendszer főbb komponenseit és a köztük lévő kapcsolatokat szemlélteti.

Az eszközök és a gateway között kétféle kommunikációs csatorna működik párhuzamosan. A Wi-Fi-s Tasmota-eszközök egy helyi MQTT-brokeren keresztül küldik állapotaikat és fogadják a vezérlési parancsokat. A Zigbee-eszközök a Sonoff Zigbee Bridge-en keresztül csatlakoznak, és a Home Assistant ZHA integrációján² át kerülnek be a rendszerbe.

A gateway és a backend közötti kommunikáció egy felhőalapú RabbitMQ MQTT-brokeren³ keresztül zajlik. A gateway 10 másodpercenként lekérdezi az összes eszköz állapotát, és ha változást érzékel, az állapotcsomagot publikálja az MQTT-brokerre. A backend fogadja ezt, elmenti az adatbázisba, és WebSocket-kapcsolaton keresztül azonnal értesíti a mobilklienseket.

A backend egy SQLite adatbázist használ az adatok tartós tárolásához. A szobákhoz rendelt képek egy Cloudinary CDN-alapú felhőszolgáltatáson⁴ tárolódnak, ami gyors tartalomkiszolgálást és alacsony backend-terhelést biztosít. Az alkalmazás bejelentkezése Google OAuth 2.0 alapú [4]: a felhasználó a meglévő Google-fiókjával lép be, így nincs szükség külön jelszó megjegyzésére. A Google által kiadott tokent a backend ellenőrzi, majd saját JWT-tokent állít ki, amelyet a mobilalkalmazás minden további kéréshez felhasznál.

²<https://www.home-assistant.io/integrations/zha/>

³<https://www.rabbitmq.com/docs>

⁴<https://cloudinary.com/documentation>

2. Okoseszközök

A LivingSmart rendszer jelenleg kétféle okoseszköz-típust támogat: Zigbee-alapú és Wi-Fi-alapú eszközök. Mindkét típus integrálható a Home Assistant platformba, így a gateway egységesen kezeli őket, függetlenül attól, hogy milyen protokollon kommunikálnak. Ez az egységesítés az egyik legfontosabb tervezési döntés volt, hiszen a valós otthonokban is tipikusan vegyes az eszközpark.

2.1. Zigbee

A Zigbee⁵ egy alacsony energiafogyasztású, rövid hatótávolságú vezeték nélküli kommunikációs szabvány, amelyet kifejezetten IoT-eszközökhöz fejlesztettek. Az egyik legfőbb előnye a mesh-hálózati topológia: a Zigbee-eszközök nem csupán a saját állapotukat tudják közvetíteni, hanem más eszközök csomagjait is képesek továbbítani, ha azok közvetlenül nem érnék el a koordinátort [7]. Ez a jelterjedést és a hálózat megbízhatóságát egyaránt javítja.

Egy Zigbee-hálózatban háromféle szerepkör létezik. A koordinátor az egyetlen eszköz, amelyik a hálózatot felépíti és adminisztrálja; a LivingSmart rendszerben ezt a szerepet a Sonoff Zigbee Bridge tölti be. A router-eszközök jelerősítőként és csomagtovábbítóként működnek; általában a hálózatra csatlakoztatott (nem akkumulátoros) eszközök látják el ezt a funkciót. A végeszközök (end device) azok a szenzorok és aktuátorok, amelyek a tényleges mérési vagy vezérlési feladatot végzik, és amelyek energiatakarékosság érdekében nagy részét az időnek alvó üzemmódban töltik.

A LivingSmart rendszerben a Zigbee-eszközök integrációja a Home Assistant ZHA modulján keresztül valósul meg. A ZHA (Zigbee Home Automation) egy Home Assistantbe beépített integráció, amely a koordinátoreszkőzzel kommunikálva az összes párosított Zigbee-eszközt Home Assistant entitásként regisztrálja. Ettől a ponttól a gateway Python-kódja számára teljesen mindegy, hogy egy adott entitás mögött Zigbee vagy Wi-Fi-s eszköz áll: mindkettő ugyanolyan formátumban érhető el a Home Assistant REST API-ján keresztül.

Az új Zigbee-eszköz párosítása a mobilalkalmazásból is elvégezhető. Az alkalmazás egy párosítási parancsot küld a backendnek, amely azt MQTT-n továbbítja a gateway-nek. A gateway megnyitja a ZHA párosítási ablakot 30 másodpercre, majd a párosítás befejeztével újratölti a ZHA integrációt, hogy az új eszköz megjelenjen a rendszerben. A Zigbee-eszközök tipikusan a kisebb,

⁵<https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf>

pontosabb méréseket igénylő funkciókat látják el a LivingSmart rendszerben: hőmérséklet- és páratartalom-érzékelők, mozgásérzékelők, nyitásérzékelők (ajtó, ablak). Ezek az eszközök akkumulátorról is hosszú ideig üzemelnek, köszönhetően az alacsony energiafogyasztásnak.

2.2. Tasmota

A LivingSmart Wi-Fi-s eszközei Sonoff gyártmányú, ESP8266 vagy ESP32 mikrovezérlőre épülő okoseszközök, amelyekre Tasmota⁶ nyílt forráskódú firmware van telepítve. A Tasmota egy önkéntes fejlesztői közösség által karbantartott, ingyenes alternatív firmware, amelynek célja a gyártói felhőfüggetlenség biztosítása. A Sonoff eszközök gyári firmware-e kizárólag a gyártó saját felhőszolgáltatásán keresztül kommunikál, és internetkapcsolat nélkül nem működik. A Tasmota firmware ezzel szemben lehetővé teszi a helyi hálózaton belüli MQTT-alapú kommunikációt, így az eszközök internet nélkül is vezérelhetők.

A Wi-Fi-t használó Sonoff eszközökön a gyártói firmware le lett cserélve erre a nyílt forráskódú Tasmota-ra, így tudtunk teljes, gyártófüggetlen kontrollt elérni az eszközök felett, amely lehetőséget adott arra, hogy megvalósítsuk a kívánt funkcionalitásokat.

A Tasmota-eszközök háromféle szerepkörben vesznek részt a LivingSmart rendszerben. Az S26R2 típusú okoskonnektorok egyszerű ki/be kapcsolási funkciót biztosítanak; ezek tipikusan lámpakapcsolók vagy egyéb fogyasztók vezérlésére szolgálnak. A 4CH Pro R3 négy csatornás vezérlő négy különálló relécsatornával rendelkezik, ami lehetővé teszi például több nagyobb fogyasztó irányított üzemeltetését. A THR320D típusú hőmérséklet-érzékelős relémodul, ami termosztát-funkcióra is alkalmas, amellyel a következő alfejezetben részletesen foglalkozunk.

A Tasmota az MQTT-n szabványos topicszerkezetet alkalmaz: minden eszköznek van egy egyedi topicja, amelyre az állapotüzeneteket publikálja, és egy parancs topicja, amelyről a vezérlési parancsokat fogadja. A Home Assistant MQTT-integrációja⁷ ezt a struktúrát automatikusan felismeri, és az eszközöket entitásként regisztrálja.

2.3. Termosztát funkció

A Sonoff THR320D eszköz alapvetően egy Wi-Fi-s relémodul amely bővíthető hőmérséklet- és páratartalom-érzékelővel, így otthonunk fűtés- illetve hűtésrendszerét is irányíthatjuk vele. Csak egyszerű ki/be kapcsolásra és hőmérsékletmérésre képes, de a Tasmota firmware termosztát

⁶<https://tasmota.github.io/docs/>

⁷<https://www.home-assistant.io/integrations/mqtt/>

driverével⁸ kiegészítve valódi hőmérséklet-szabályozóvá válik. A Tasmota termosztát-modul lényege, hogy a relét automatikusan vezérli az aktuális és a célhőmérséklet összehasonlítása alapján. Ha a mért hőmérséklet a célhőmérséklet alá süllyed, a relé bekapcsol (például fűtési szelepet nyit), ha fölé emelkedik, kikapcsol.

A LivingSmart rendszerben az egyedi Tasmota-build azért szükséges, mert az alap firmware nem teszi elérhetővé az összes termosztát-paramétert MQTT-n keresztül. A kiterjesztett build a célhőmérsékletet a Mem1 memóriaregiszterben tárolja, a termosztát-üzemmódot (automatikus vagy kézi) a Mem2 regiszterben. A gateway közvetlenül, API-hívással tudja ezeket a regisztereket módosítani.

A kiterjesztett telemetriának köszönhetően a mobilalkalmazásban a termosztát-kártyán egyszerre jelenik meg az aktuális mért hőmérséklet, a célhőmérséklet, a termosztát aktuális üzemmódja és a relé pillanatnyi állapota. Ez az átlátható diagnosztikai kép segíti a felhasználót abban, hogy megítélje a fűtési rendszer állapotát anélkül, hogy akár a lakásban tartózkodna. Megjegyzendő, hogy a termosztát-funkció kizárólag termosztát-módban aktív: a felhasználó a mobilalkalmazásból bármikor visszakapcsolhat kézi módba, amelyben a relé kizárólag a felhasználói parancsokra reagál, és a hőmérséklet-szabályozás szünetel. Ez a rugalmasság fontos, ha például a fűtési rendszert átmenetileg nem szeretnénk automatikusan vezérelni.

⁸<https://tasmota.github.io/docs/Thermostat/>

3. Gateway

A gateway a LivingSmart rendszer legösszetettebb komponense. Egy Raspberry Pi 4 miniszámítógépen fut [10], amelyen Linux operációs rendszeren futnak a Python szolgáltatások, és Docker-konténerekben [4] üzemel rajta a Home Assistant. A gateway felelős az eszközökhöz való közvetlen kapcsolódásért, a szokásfigyelésért, a viselkedésszimuláció futtatásáért, a termosztát-ütemezésért és a backenddel való kommunikációért.

3.1. Home Assistant

A Home Assistant⁹ nyílt forráskódú okosotthon-platform, amelyet a LivingSmart rendszerben aggregációs réteggént alkalmazunk. Lényege, hogy a különböző protokollokon elérhető eszközöket, a Wi-Fi-s Tasmota-eszközöket és a Zigbee-eszközöket egyaránt, egységes entitásmodellben kezeli. Minden eszközből egy jól definiált entitás lesz, amelynek van azonosítója, aktuális állapota és attribútumai. A Home Assistant REST API-ján¹⁰ keresztül a gateway Python-kódja lekérdezheti bármely entitás aktuális állapotát, és szolgáltatáshívásokkal vezérelhet bármely eszközt. Ez az egységes interfész rendkívül fontos: a gateway oldalán nem kell külön kezelni, hogy egy adott eszköz Zigbee-s vagy Wi-Fi-s, mert mindkét esetben ugyanazon a Home Assistant API-n keresztül érhető el.

A Home Assistant Docker-konténerben fut a Raspberry Pi-n. A konténerizáció több előnnyel jár: a Home Assistant függőségei el vannak különítve a többi rendszerkomponenstől, a verziók kezelése egyszerűbb, és nem zavarja a Python service-k futását.

A Zigbee-eszközök a ZHA integráción keresztül kapcsolódnak a Home Assistanthez, amelyhez a Sonoff Zigbee Bridge fizikai koordinátoreszköz szükséges. Az MQTT-s Tasmota-eszközök az MQTT-integráción keresztül jelennek meg entitásokként, a helyi MQTT-broker közvetítésével. Mindkét esetben a végeredmény ugyanaz: a gateway Python-kódja a Home Assistant REST API-ját hívja, és protokollfüggetlen entitás-állapotokat kap vissza.

3.2. Backenddel való kommunikáció

A gateway fő Python-szkriptje, a `bridge.py` tartalmazza a rendszer koordinációs logikáját. Indulásakor csatlakozik a RabbitMQ MQTT-brokerhez és elindítja a polling-ciklust. A

⁹<https://www.home-assistant.io/docs>

¹⁰<https://developers.home-assistant.io/docs/api/rest/>

polling-ciklus 10 másodpercenként lekérdezi az entitásállapotokat a `fetcher.py` segítségével, összehasonlítja azokat az előző ciklus értékeivel, és csak akkor publikálja az adatokat a backendnek, ha valódi változás történt.

A gateway és a backend közötti kommunikáció egy felhőben üzemelő MQTT-brokeren (RabbitMQ) keresztül zajlik. Az MQTT (Message Queuing Telemetry Transport)¹¹ egy publish-subscribe elven működő üzenetprotokoll, amelyet kifejezetten IoT-eszközök és alacsony sávszélességű kapcsolatok számára terveztek.

A gateway négy MQTT-topicot használ. A `homeassistant/states` topic-ra publikálja az eszközállapot-csomagokat: minden csomag tartalmaz egy időbélyeget, a gateway egyedi azonosítóját és az összes entitás aktuális állapotát. A `homeassistant/commands` topic-ra feliratkozik, és az onnan érkező JSON-formátumú parancsokat hajtja végre. A `homeassistant/schedule` és a `homeassistant/thermostat_program` topicokra a gateway a jelenlét-szimuláció, illetve a termosztát-program aktuális állapotát publikálja, valahányszor az megváltozik. Ezeket a backend fogadja, és a mobilalkalmazás lekérdezheti.

Az MQTT-alapú aszinkron kommunikáció laza csatolást biztosít a gateway és a backend között. Ha a backend átmenetileg nem elérhető, az üzenetek a brokernél megőrződnek, és később kézbesítődnek. Ha a gateway kapcsolata megszakad, automatikusan újracsatlakozik, és folytatja a működést.

Az egyes parancsok kezelése szálbiztos: minden beérkező MQTT-üzenetre a Python service egy különálló daemon-szálat indít, amelyben a parancs feldolgozódik. Ez lehetővé teszi, hogy a fő polling-ciklus és a parancs-végrehajtás ne blokkolják egymást, a rendszer párhuzamosan tudja kezelni az eszközállapot-frissítéseket és a beérkező vezérlési kéréseket.

3.3. Eszközök csatlakoztatása

Új eszköz hozzáadása a rendszerhez a mobilalkalmazásból történik. Zigbee-eszközök esetén az alkalmazás a `backend /api/devices/pair` végpontját hívja, a backend ezt MQTT-parancson keresztül továbbítja a gateway-nek. A gateway megnyitja a ZHA párosítási ablakot 30 másodpercre, amelynek során az új Zigbee-eszközt a közelbe kell hozni, és a gyártó által meghatározott módon párosítási módba kell kapcsolni (általában egy gomb nyomva tartásával). A 30 másodperc elteltével a gateway újratölti a ZHA integrációt, hogy az újonnan párosított eszköz megjelenjen a Home Assistant entitás-listájában, és ezzel a rendszerben is

¹¹<https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>

elérhetővé váljon.

Wi-Fi-s Tasmota-eszközök csatlakoztatása a következő képpen történik, egy friss Tasmota-t használó eszköz először egy saját hálózatot hoz létre, amelynek a címe Tasmota-val kezdődik. Erre a hálózatra rá kell csatlakozzunk, amikor az alkalmazás utasít bennünket erre, ezután meg kell adjuk az otthoni Wi-Fi hálózatunk nevét illetve jelszavát, ezután az alkalmazás elvégzi az eszköz konfigurálását HTTP hívások segítségével, ezután pedig az eszköz csatlakozik a hálózatra, és Gatewayhez. A gateway és a backend a következő polling-ciklusban automatikusan felfedezi az új eszközt, és hozzáadja a nyilvántartáshoz. A backend az `/api/devices` PATCH végpontján keresztül manuálisan is kikényszeríthető az entitáslista frissítése, amelyet a `bridge.py` egy `refresh MQTT`-paranccsal hajt végre. Az eszközök nyilvántartása egy `devices.json` fájlban tárolódik a gateway-en, amelyet a `grouper.py` szkript tart naprakészen. Ez a fájl tartalmazza az entitás-azonosítók és az eszköazonosítók leképezését, amelyet a rendszer minden eszközinterakciónál használ.

3.4. Szokásfigyelés és lejátszás

A jelnlét-szimuláció a LivingSmart egyik legegységibb funkciója. A rendszer a felhasználó eszközhasználati szokásait automatikusan megfigyeli, ezekből egy ütemtervet állít össze, amelyet aztán a felhasználó a távolléte alatt visszajátszhat, valósan ható otthoni aktivitást szimulálva.

3.4.1. Adatgyűjtés és profilozás

Az adatgyűjtés alapját a Home Assistant entitásállapotok alkotják, amelyeket a `fetcher.py` 10 másodpercenként kérdez le. Az állapotnapló folyamatosan bővül a `bridge.py` polling-ciklusának köszönhetően, és az entítások ki/be állapotváltozásai időbélyeggel kerülnek rögzítésre. A profilozás a `user_profiling/main.py` szkripttel indul el, amelyet a `generate_schedule` MQTT-parancs vált ki. Ez a szkript feldolgozza a rögzített állapotnapló-adatokat, és napszakokra bontva statisztikát készít: minden entitáshoz és napszakhoz meghatározza a tipikus bekapcsolási és kikapcsolási időt, valamint azok szórását. Az eredmény egy `schedule.txt` szövegfájl, amelynek sorai az alábbi formátumban tartalmazzák az ütemtervet:

```
entity_id ON HH:MM:SS -> OFF HH:MM:SS
```

Egy sor tehát megadja, hogy egy adott entitásnak melyik időszakban kell bekapcsolt állapotban lennie. A fájl sorai az entitások napi aktivitási ablakait írják le: ha például a egy adott lámpa általában reggel 7:00 és 8:30 között van bekapcsolva, az adott entitáshoz egy sor kerül a fájlba ezekkel az időpontokkal. A profilozás befejeztekor a `bridge.py` azonnal elküldi a kész ütemtervet a backendnek a `homeassistant/schedule` topic-ra, ahol a mobilalkalmazás megtekintheti azt.

3.4.2. Rutin lejátszása

A jelenlét-szimuláció lejátszását a `schedule_runner.py` Python-szkript végzi, amelyet a `bridge.py` önálló folyamatként indít el. A szimulátor másodpercenként kiértékeli az összes ütemezési bejegyzést, és szükség esetén parancsokat küld az eszközöknek. A kiértékelés logikája a következő: minden ciklusban felépül egy `desired_state` szótár, amelybe minden entitáshoz meghatározódik, hogy az aktuális időpont szerint ki- vagy bekapcsolt állapotban kellene-e lennie. Ha több bejegyzés vonatkozik ugyanarra az entitásra, az OR logika érvényesül: az entitás akkor kell bekapcsolt legyen, ha bármelyik bejegyzés szerint aktívnak kellene lennie. Egy fontos részlet az éjfél-t átlépő időintervallumok kezelése: ha egy entitás például este 23:00 és másnap reggel 1:00 között aktív, a szokásos ($t_{on} < t_{off}$) feltétel nem működik helyesen. Ilyenkor a feltétel megfordul: az entitásnak akkor kell aktívnak lennie, ha az aktuális idő nagyobb vagy egyenlő t_{on} -nál, VAGY kisebb t_{off} -nál.

A szimulátor ezután összehasonlítja a kívánt állapotot a legutóbb küldött paranccsal. Ha eltérés van, kiadja a megfelelő kapcsolóparancsot a Home Assistant szolgáltatáshíváson keresztül. A szimuláció szándékosan nem percre pontos: a generált ütemterv már eleve tartalmaz némi szórást a tipikus bekapcsolási és kikapcsolási időpontok körül, ami természetesebb, életszerűbb viselkedést eredményez, mint egy merev, másodpercre pontos ismétlés.

A szimuláció leállítása a `stop_schedule` parancsra történik. Ilyenkor a `bridge.py` leállítja a `schedule_runner.py` folyamatot, majd beolvassa a `schedule.txt`-t, összegyűjti az összes érintett entitást, és mindegyiknek kikapcsolási parancsot küld. Ez biztosítja, hogy leállításkor ne maradjanak véletlenszerűen bekapcsolt eszközök.

3.5. Termosztát ütemterv

A termosztát-ütemezési funkció lehetővé teszi, hogy a felhasználó napszakhoz kötött hőmérséklet-programokat állítson be a termosztát-eszközökhöz. A program önállóan,

felhasználói beavatkozás nélkül fut a gateway-en, és automatikusan állítja a termosztátok célhőmérsékletét a megadott menetrend szerint. Az ütemezési-programot egy JSON-fájl (`thermostat_program.json`) tárolja a gateway-en. A fájl tartalmazza a napokra és időpontokra lebontott programpontokat: minden bejegyzés megadja, hogy melyik entitásnak mikor milyen célhőmérsékleten és milyen üzemmódban kell üzemelnie. A struktúra lehetővé teszi, hogy különböző helyiségekben eltérő programok fussanak egyidejűleg.

A standard, illetve az override ütemtervet a mobilalkalmazás termosztát-ütemezési oldaláról lehet létrehozni és aktiválni. A backend ezután készít egy egyszerű, összesített termosztát-programot, amely követi a rendszer logikáját. Tehát minden csoport ütemterve össze lesz foglalva az eszközök szerinti vonatkozásban, és prioritást ad az override ütemtervnek a standard felett. Az alkalmazás a backenden keresztül PUT kéréssel elküldi ezt az egyszerűsített tervet, a backend ezt `thermostat_program_set` MQTT-parancsként továbbítja a gateway-nek, a gateway menti a JSON-fájlt, majd leállítja és újraindítja a `thermostat_program_runner.py` folyamatot. A `thermostat_program_runner.py` szkript percenként ellenőrzi az aktuális nap és idő alapján, hogy melyik programpont van érvényben. Megkeresi az aktuális időpontnál legutoljára érvénybe lépett bejegyzést, és ha az eltér az utoljára alkalmazottól, API-hívásokon keresztül frissíti a Tasmota-eszköz `Mem1` (célhőmérséklet) és `Mem2` (termosztát-mód) regisztereit. Ezek a regiszterek közvetlenül befolyásolják a Tasmota termosztát-modul viselkedését.

A mobilalkalmazás termosztát-ütemezési oldala két ütemezési fület tartalmaz: normál ütemezést és override ütemezést. A normál ütemezés a hétköznapi működést írja le, az override ütemezés aktiválásával átmenetileg (például szabadság idején) egy eltérő program léphet érvénybe anélkül, hogy a normál programot felül kellene írni. Az oldalon a felhasználó szabadon aktiválhatja a tetszés szerinti programokat. A termosztát-program induló értéke automatikusan visszaáll, ha a Raspberry Pi újraindul: a `bridge.py` elinduláskor megvizsgálja, hogy létezik-e `thermostat_program.json`, és ha igen, azonnal elindítja a `thermostat_program_runner.py`-t. Ez biztosítja, hogy az ütemezési funkció egy esetleges áramkimaradás után is automatikusan helyreáll, felhasználói beavatkozás nélkül.

4. Szerver

A backend a LivingSmart rendszer központi adatkezelő és koordináló komponense. REST API-t és WebSocket-csatornát biztosít a mobilkliens számára, MQTT-n keresztül kommunikál a gateway-jel, és SQLite¹² adatbázisban tárolja az adatokat [3][13][15]. A következő alfejezetekben a szerver főbb funkcionálisait, adatbázisát, valós idejű kommunikációját, technológiáit és architektúráját mutatjuk be.

4.1. Funkcionalitások

A szerver service-interfészei tisztán körülhatárolt felelősségi területeket fednek le. Az alábbiakban ezeket az interfészeket és a mögöttük rejlő üzleti logikát mutatjuk be részletesen.

4.1.1. Autentikáció és jogosultságkezelés

A bejelentkezés az `/auth/login` végponton keresztül történik. A mobilalkalmazás a Google Sign-In SDK-val egy `idToken`-t szerez, amelyet elküld erre a végpontra. A szerver az `AuthService.signIn()` metódusával ellenőrzi a Google-token a Google API-n keresztül, azonosítja a felhasználót, és ha az adott e-mail-cím még nem szerepel az adatbázisban, a `findOrCreateUser()` metódus automatikusan létrehozza a felhasználói rekordot. Sikeres ellenőrzés esetén a `JwtService.generateToken()` metódusa kiad egy saját JWT-token¹³ [5], amelyet a kliens minden további kérésben `Authorization: Bearer` fejlécként csatol.

A JWT-token a *Subject* mezőjében a felhasználó belső adatbázis-azonosítóját tartalmazza. Az `isTokenValid()` metódus minden védett végponton ellenőrzi a token lejáratát és aláírását. A `/user/info` végponton a bejelentkezett felhasználó adatai kérhetők le, a Spring Security¹⁴ a `SecurityContextHolder`-ből adja vissza az aktuálisan azonosított felhasználó azonosítóját.

Az `/auth/login` végpont nincs JWT-védelemmel ellátva, hiszen ez az a végpont, ahol a token kiadása történik. Minden más `/api/**` végpont érvényes token igényel, amelyet a Spring Security konfigurációja kényszerít ki.

¹²<https://sqlite.org/docs.html>

¹³<https://javadoc.io/doc/io.jsonwebtoken/jjwt-api/latest/index.html>

¹⁴<https://docs.spring.io/spring-security/reference/index.html>

4.1.2. Szobák és képfeltöltés

A szobák kezelése az `/api/rooms` végpontokon keresztül valósul meg. A szoba létrehozásakor és frissítésekor a kérés `multipart/form-data` formátumban érkezik: tartalmaz egy nevet, egy színkódot és egy képfájlt. A képfeltöltés folyamata a következő: a `RoomService` fogadja a `MultipartFile`-t, amelyet a Cloudinary Java SDK¹⁵ `upload()` metódusával tölt fel a CDN-re. Az API elfogad helyi fájllelési útvonalat, HTTP-linket és Base64 formátumot is, a mobilalkalmazásból küldött képek Base64 formátumban érkeznek. A Cloudinary által visszaadott CDN-URL kerül az adatbázisba a szoba `imageUrl` mezőjeként, ezt használja a mobilalkalmazás a kép megjelenítéséhez.

A `GET /api/rooms/{id}/devices` végpont egy `DevicesDto` objektumot ad vissza, amelynek mezői típus szerint csoportosítják az eszközöket: `temperatureSwitches`, `singleSwitches`, `multiSwitches`, `temperatureSensors`, `movementSensors` és `binarySensors`. Ez a struktúra közvetlenül megfelel a mobilalkalmazás `DevicesResponse` modelljének, ami leegyszerűsíti a deszerializálást és a megjelenítést.

4.1.3. Eszközkezelés

Az eszközök kezelése a `DeviceController`-en és az alatta lévő `DeviceService`-en keresztül valósul meg. Az eszközök állapotát alapvetően a gateway tartja naprakészen az MQTT-állapotfrissítéseken keresztül: amikor a backend fogadja az állapotcsomagot, az adatbázisban frissíti az érintett eszközök állapotát.

A kapcsolóeszközök ki/be állapotának változtatása az `/api/devices/toggle/{id}` végponton keresztül, illetve a `GroupService`-n keresztül csoportosan is elvégezhető. A `toggle` parancsra a backend az `MQService` `sendToggleCommand(entityId)` metódusát hívja, amely egy MQTT-üzenetet küld a gateway-nek. A gateway fogadja a parancsot, és a Home Assistant megfelelő service hívásával végrehajtja a kapcsolást. Miután a fizikai eszköz állapota megváltozott, a következő polling-ciklusban a gateway publikálja a frissített állapotot, amelyet a backend fogad és a klienseknek továbbít.

A termosztát-eszközök paraméterezése az `/api/devices/{id}/thermostat` PATCH végponton történik. A `DeviceThermostatPatchRequest` DTO tartalmaz egy opcionális `targetTemperature` és egy opcionális `thermostatModeOn` mezőt. Ha a célhőmérséklet be van állítva, a service az `MQService` `sendSetTempCommand()`-ot hívja, ha az üzemmód

¹⁵https://cloudinary.com/documentation/java_integration

van megadva, a `sendSetThermostatModeCommand()`-ot. Mindkét esetben az `MQService` MQTT-n üzen a gateway-nek, amely HTTP-n közvetlenül a Tasmota Mem1 illetve Mem2 regiszterét állítja be.

Az `/api/devices` PATCH végpont (paraméter nélkül) a `sendRefreshCommand()` hívásán keresztül egy kézi frissítési ciklust indít a gateway-en: a `fetcher.py` lefut, és az eredmény azonnal publikálódik a `homeassistant/states` topic-ra.

4.1.4. Csoportkezelés

A `GroupService` interfész a rendszer egyik legtöbb metódust tartalmazó service-e, ami tükrözi a csoportos vezérlés összetettségét. A csoportok szobák gyűjteményei, így a csoportbeli szobák összes ugyanolyan típusú eszközt egyszerre vezérelhetjük.

A `switchAllDevicesByType()` metódus a csoport összes, adott típusú eszközének ki/be kapcsolását egy MQTT-üzenetben képes elvégezni: az összes érintett entitásazonosítót összegyűjti, és egyetlen `sendToggleCommand`-sorozatot küld a gateway-nek. Ez lényegesen hatékonyabb, mint minden eszközhöz külön-külön parancsot küldeni.

A `setTempForAllDevices()` metódus az összes termosztát-eszköz célhőmérsékletét állítja be egyidejűleg. A metódus `NotThermostatException`-t dob, ha a csoportban nem `TEMPERATURE_SWITCH` típusú eszközhöz próbálnánk hőmérsékletet beállítani, amit a `GroupController` validál, mielőtt a service-t meghívna.

A termosztát-ütemezési metódusok (`getThermostatSchedule`, `saveThermostatSchedule`, `setThermostatScheduleActive`, `setThermostatScheduleOverrideActive`) a csoport ütemezési programját kezelik. A `saveThermostatSchedule()` a teljes `GroupThermostatScheduleDTO`-t elmenti az adatbázisba, majd a mentett adatot JSON-ba szerializálva meghívja az `MQService` `sendThermostatProgramSet()` metódusát, ami a gateway-en elindítja az ütemező újrakonfigurálását. A `setThermostatScheduleActive()` és `setThermostatScheduleOverrideActive()` metódusok csak a jelzőket módosítják az adatbázisban, és az `MQService` `sendThermostatProgramGet()`-en keresztül frissítik a gateway állapotát.

4.1.5. Szimuláció vezérlése

A `ScheduleController` a `HouseholdService` és az `MQService` kombinációjával valósítja meg a jelenlét-szimuláció vezérlését. A `GET /api/schedules` végpont a legutóbbi, gateway-től fogadott schedule-csomag alapján összeállítja az aktuális ütemtervet a mobilkliens számára.

A `POST` végpontok mindegyike az `MQService` egy-egy metódusát hívja: a `/generate` a `sendGenerateScheduleCommand()`-ot, a `/start` a `sendRunScheduleCommand()`-ot, a `/stop` a `sendStopScheduleCommand()`-ot, a `/demo` a `sendGenerateDemoCommand()`-ot. Minden hívás egy egyszerű `try-catch` blokkban fut: ha az MQTT-küldés kivételt dob, a végpont 500-as hibakóddal tér vissza, különben 200-assal.

4.1.6. Valós idejű állapot frissítések

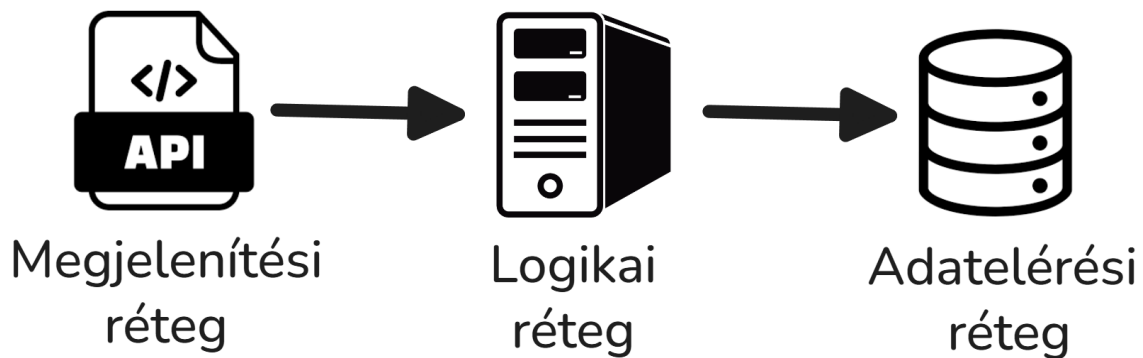
A `DevicePushService` interfész a `WebSocket`-alapú állapotváltozások továbbításáért felelős. Négy metódusa van, amelyek különböző szituációkra reagálnak.

A `pushAllToUser(userId)` az adott felhasználó összes szobájának összes eszközét felküldi a kliensnek. Ez az alkalmazás első megnyitásakor vagy egy teljes frissítéskor hasznos.

A `pushRoomDevices(userId, room)` egy adott szoba eszközeit küldi el a kliensnek. Ez hívódik meg akkor, amikor a backend MQTT-n fogadja a gateway állapotfrissítő csomagját: a backend meghatározza, hogy az érkező entitásazonosítók melyik szobához tartoznak, és csak az érintett szoba frissített eszközlistáját juttatja el a kliensnek a `/user/queue/rooms/{roomId}` STOMP topicra.

A `pushScheduleNotifier(userId)` a `/user/queue/schedule` STOMP topicra küld értesítést, amikor a gateway egy új schedule-csomagot publikál. A mobilalkalmazásban a `WsService connectScheduleReceiver()` metódusa iratkozik fel erre a topicra, amikor az üzenet megérkezik, az alkalmazás automatikusan újra lekérdezi az `/api/schedules` végpontot, és frissíti a megjelenített ütemtervet.

A `pushDeviceAddedNotifier(userId, name)` a `/user/queue/paired` topicra küld értesítést az új eszköz nevével, amikor a Zigbee-párosítás sikeresen befejeződik. A mobilalkalmazásban a `WsService _pairingUnsubscribe` változója tárolja ezt a feliratkozást, és a szoba oldalán az `onDeviceConnected` callback egy felugró dialógusban értesíti a felhasználót.



2. ábra. Háromrétegű architektúra

4.2. Architektúra

A LivingSmart backend rétegzett, háromrétegű (three-tier) architektúrát követ(2): prezentációs (controller), logikai (service) és adatelérési (repository) rétegekből áll. Ez a struktúra elősegíti a karbantarthatóságot, a tesztelhetőséget és a moduláris felépítést.

A prezentációs réteg felelős a HTTP-kérések kezeléséért és a REST API végpontok biztosításáért [9]. A Spring Boot keretrendszer `@RestController` annotációval jelölt osztályai fogadják a kliens (mobilalkalmazás) kéréseit, majd továbbítják azokat a service réteg felé.

A logikai réteg tartalmazza az alkalmazás üzleti logikáját. Itt történik az adatok feldolgozása, az adatok validálása, valamint az egyes műveletek összehangolása. Ez a réteg biztosítja, hogy az üzleti logika elkülönüljön a webes és adatbázis-specifikus megvalósítástól, ezáltal a rendszer könnyebben tesztelhető és bővíthető.

Az adatelérési réteg felelős az adatbázis-kezelésért. A Spring Data JPA technológia segítségével definiált repository interfészek biztosítják az entitások perzisztenciáját [8]. A CRUD (Create, Read, Update, Delete) műveletek automatikusan generált metódusokon keresztül érhetők el, míg az összetettebb lekérdezések egyedi metódusokkal valósíthatók meg.

4.3. Technológiák

A Spring Boot keretrendszer gyors alkalmazásfejlesztést tesz lehetővé előre konfigurált, production-ready komponensekkel [18]. A Spring Security modul védi a végpontokat: a `JwtAuthenticationFilter` a beérkező kérések `Authorization` fejlécéből kinyeri és validálja a JWT-tokent, majd beállítja a Spring Security kontextusát.

A Lombok¹⁶ könyvtár nagymértékben csökkenti a boilerplate-kódot. A `@RequiredArgsConstructor` automatikusan generálja a konstruktort a `final` mezőkből, ami biztosítja a konstruktor-alapú dependency injectiont. A `@Slf4j` annotáció automatikusan létrehoz egy `logger` mezőt, a `@Getter` és `@Setter` a hozzáférő metódusokat.

Az Eclipse Paho¹⁷ Java MQTT-kliens biztosítja az MQTT-kommunikációt. Az `MQService` implementációja egy `singleton MqttClient` példányt tart fenn, amelyen keresztül az összes MQTT-üzenet küldése és fogadása történik. A `getClient()` metódus lehetővé teszi, hogy szükség esetén más komponensek is hozzáférjenek az MQTT-klienshez.

A Cloudinary Java SDK¹⁸-n keresztül történik a képfeltöltés a szoba létrehozásakor és frissítésekor. A képájl Base64 kódolással kerül feltöltésre, a visszakapott CDN-URL tárolódik az adatbázisban. A CDN-alapú kiszolgálás alacsony backend-terhelést és gyors tartalomkiszolgálást biztosít.

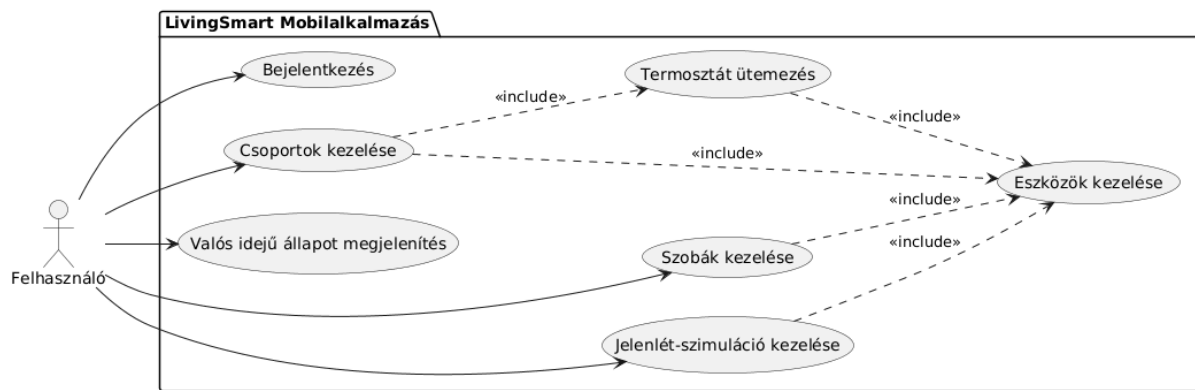
A `json-webtoken`¹⁹ (jjwt) könyvtár kezeli a JWT-tokenek generálását és validálását a `JwtService` implementációjában. A tokenek RS256 algoritmussal vannak aláírva, a `subject` mezőben a felhasználó belső adatbázis-azonosítója szerepel.

¹⁶<https://projectlombok.org/features/val>

¹⁷<https://eclipse.dev/paho/clients/java/>

¹⁸https://cloudinary.com/documentation/java_integration

¹⁹<https://javadoc.io/doc/io.jsonwebtoken/jjwt-api/latest/index.html>



3. ábra. A mobil alkalmazás funkcionalitásai

5. Mobil alkalmazás

A felhasználó egy cross-platform alkalmazás segítségével férhet hozzá a LivingSmart rendszerhez, ami lehetővé teszi az eszközök irányítását, szobák, illetve csoportok kezelését, csoport-ütemterv felállítását, és a jelenlét-szimuláció kezelését.

5.1. Funkcionalitások

A 3. ábra jól szemlélteti, hogy a felhasználó milyen funkcionalitásokat érhet el a mobilalkalmazásból. Az alábbiakban az alkalmazás funkcionalitásait és architektúráját tárgyalja a dolgozat.

5.1.1. Bejelentkezés

Az alkalmazás első indításakor a SignInPage(4. ábra) fogadja a felhasználót. Az oldal az alkalmazás logóját és egy Google Sign-In gombot tartalmaz. A gomb megnyomásakor elindul a Google OAuth folyamat: a `google_sign_in`²⁰ csomag egy natív fiókválasztót jelenít meg.

A sikeres Google-bejelentkezés után az alkalmazás elküldi a kapott `idToken`-t a backend `/auth/login` végpontjára az `AuthService.sendAuthRequest()` metódusán keresztül. A backend JWT-token-t ad vissza, amelyet az alkalmazás a `UserRepository`-ban tárol el (`shared_preferences`²¹ csomag). Sikeres bejelentkezés esetén az alkalmazás a főoldalra navigál, ellenkező esetben hibaüzenetet kapunk.

Azok a felhasználók, akiknek érvényes JWT-tokenjük van tárolva a `SharedPreferences`-be, a következő alkalmazásindításkor automatikusan

²⁰https://pub.dev/packages/google_sign_in

²¹https://pub.dev/packages/shared_preferences

átkerülnek a főoldalra anélkül, hogy újra be kellene jelentkezniük. Ezt a `UserRepository.getUserData()` metódusa ellenőrzi: ha a tárolt JSON érvényes `UserModel`-t tartalmaz, a token visszaadható, ellenkező esetben a hibás adatot törli és `null`-t ad vissza.

5.1.2. Főoldal és szobák

A főoldal (5. ábra) a felhasználó szobáit két-oszlopos rácsban jeleníti meg, ahol minden szoba egy `RoomCard` widgetként jelenik meg a szoba nevével és egyedi színével. Egy szobakártyára koppintva az alkalmazás átnavigál a szoba részletoldalára (7. ábra). Az oldal alján egy "Add room" gomb található ami átnavigál az `AddRoomPage`-re (6. ábra). A képernyő legalján pedig egy navigációs sávot láthatunk, amellyel 4 különböző oldal között navigálhatunk: Főoldal (Szobák), Csoportok, Rutinok, Profil.

5.1.3. Szoba hozzáadása és domináns szín

A szoba hozzáadási oldal (6. ábra) lehetővé teszi a szoba nevének megadását és egy borítókép kiválasztását. A névmező maximum 15 karaktert fogad el. A kép kiválasztása az `image_picker`²² csomag `ImagePicker().pickImage()` metódusával történik, `gallery` forrásból.

A szoba létrehozásának egyik érdekessége a domináns szín automatikus kinyerése a feltöltött képből. A `DominantColor` segédosztály a `palette_generator`²³ csomagot alkalmazza erre a célra. A `PaletteGenerator.fromImageProvider()` legfeljebb 20 különböző színmintát kér le a kép egy 200×200 pixeles verziójából. A kapott `PaletteColor` listából a legjobb szín kiválasztása egy egyedi pontozási képlettel történik: $\text{pontszám} = \text{telítettség} \times \text{világosság} \times \text{pixel-előfordulás}$. A pontozás a túl sötét ($\text{lightness} < 0.2$) és a túl világos ($\text{lightness} > 0.85$) színeket kiszűri, így csak a képre jellemző, élénk, de nem szélsőséges árnyalatok kerülhetnek a végeredménybe. Ez a szín kerül mentésre a szoba rekordja mellé az adatbázisba, és jelenik meg a szoba kártyáján.

²²https://pub.dev/packages/image_pickers

²³https://pub.dev/documentation/palette_generator

5.1.4. Eszközök és eszköz hozzáadása

A szoba oldal (7. ábra) a szoba részletes nézetét mutatja. A képernyő felső harmadát a szoba képe foglalja el, amelyet a `Network.Image` widget tölt be a Cloudinary CDN URL-ről.

Az eszközök a `DevicesResponse` modell szerint típusonként vannak csoportosítva. A kapcsolható eszközök (`temperatureSwitches`, `singleSwitches`, `multiSwitches`) ábécé-sorrendben, egymás alatt kártyás elrendezésben jelennek meg. Az érzékelők (`temperatureSensors`, `movementSensors`, `binarySensors`) szintén ábécérendben, de pontosan 2 oszlopból álló elrendezésben.

Az eszközök megjelenítése Dart típusminták (`switch-case`) alapján történik: a `switch(deviceData)` blokk minden típushoz a megfelelő widgetet adja vissza. A `TemperatureSwitchCard` megmutatja a mért hőmérsékletet, a célhőmérsékletet és az üzemmódot; a `SingleSwitchCard` egyszerű kapcsolóval kezel ki/be állapotot; a `MultiSwitchCard` négycsatornás eszközök minden csatornáját külön mutatja. Az érzékelő-kártyák csak olvasható adatokat jelenítenek meg.

Új eszköz hozzáadása a szoba oldalon elhelyezett "Add device" gombbal indítható el. Ilyenkor egy panelen jelennek meg a szobához adott eszközök (8. ábra), és a még szabadon maradt eszközök. Az eszközöket el lehet távolítani a mellettük található "-" gombbal, illetve a még szabad eszközöket a "+" gombbal lehet hozzáadni a szobához. Ha teljesen új eszközt szeretnénk a rendszerhez csatlakoztatni, azt a panel alján megjelenő "Add device" gombbal lehet, ezután ki kell választani az eszköz típusát (Zigbee eszköz vagy Tasmota eszköz), és követni kell a képernyőn található utasításokat.

5.1.5. Csoportok

A csoportok listáját a `GroupsPage` mutatja (9. ábra), két-oszlopos rácsban. Minden `GroupCard` mutatja a csoport nevét és a hozzá tartozó szobák számát.

Új csoport létrehozása az oldal alján lévő "Add group" gombbal lehetséges, ahol csak a csoport nevét kell megadni. A csoport részletoldalán (10. ábra) az összes csoport-szoba eszköze egységes listában jelenik meg, egy szűrőszávvval, amellyel az eszközöket típusuk szerint szűrhetjük.

Az ugyanolyan típusú eszközblokk felett egy kártya jelenik meg, amellyel a típus specifikus vezérléseket el tudjuk végezni.

Szobák csoporthoz adása hasonlóan a `RoomPage` oldalhoz, a képernyő alján egy gombbal

valósítható meg, ahol majd ugyanúgy "+" és "-" gombokkal tudjuk hozzáadni, illetve eltávolítani a szobákat a csoportból.

5.1.6. Termosztát ütemterv létrehozása

A csoport oldalán található egy "Thermostat schedule" gomb, amellyel átnavigálhatunk arra az oldalra, ahol tudjuk szerkeszteni a termosztátok ütemtervét(11. ábra).

Itt majd 2 kapcsoló található, amelyekkel aktiválhatjuk, a kívánt ütemtervet, az alatta lévő 2 fülben pedig szerkeszthetjük azokat. Mindkét fülön a napok (hétfő-vasárnap) egy DropdownButton-ból választhatók ki. A kiválasztott naphoz tartozó bejegyzések szerkeszthető listán jelennek meg: minden bejegyzés tartalmaz egy időpont-beviteli mezőt és egy hőmérséklet-beviteli mezőt, az "Add entry" gombbal pedig mi is tudunk bejegyzést hozzáadni.

A "Save" gomb megnyomásakor az alkalmazás lementi az ütemtervet az adatbázisba, és eljuttatja a gatewayhez.

5.1.7. Rutin generálás és futtatás

A jelenlét-szimuláció vezérlési oldala (12. ábra) a navigációs sáv harmadik lapján érhető el. Az oldal a DailyRoutineCard widget segítségével jeleníti meg az aktuális ütemtervet és a vezérlő gombokat.

A "Generate" gombra koppintva az alkalmazás elindítja a profilozást a gateway-en. Az eredmény megérkezetről a `WsService.connectScheduleReceiver()` értesít: a `/user/queue/schedule` STOMP destination-ről érkező üzenetre az alkalmazás automatikusan újra lekérdezi az ütemtervet. A kapcsolóval aktiválni a jelenlét-szimulációt, a "Load demo" gomb, egy előre megírt ütemtervet tölt be.

5.2. Architektúra

Az alkalmazás háromrétegű architektúrára épül(13. ábra): a megjelenítési rétegre (Widgetek 5.2.7), az üzleti logikai rétegre (BLoC 5.2.1) és a hálózati rétegre 5.2.2 (Retrofit, STOMP). Az alábbiakban ezeket a rétegeket tárgyaljuk részletesen.

5.2.1. BLoC

A BLoC²⁴ (Business Logic Component) tervezési minta esemény-állapot modellen alapul: a megjelenítési réteg eseményeket küld a BLoC-nak, a BLoC feldolgozza ezeket és állapotokat bocsát ki, a BlocBuilder widget az állapotváltozásra újrarajzolja a megfelelő widgetet. Ez a szétválasztás tesztelhetővé és karbantarthatóvá teszi az alkalmazást, mivel az üzleti logika teljesen független a UI-tól. Az 1. táblázat bemutatja, hogy milyen esetekben használtunk BLoC-ot:

- Bejelentkezés
- Főoldal
- Szobák
- Csoportok
- Eszközök

A táblázatból látszanak a tisztán elhatárolt állapotok, amelyeket az osztály a különböző események hatására vesz fel.

5.2.2. Hálózati réteg

A hálózati réteg alapját a Dio²⁵. A DioFactory segédosztály egy statikus createDio() metódussal hozza létre a konfigurált Dio-példányokat. Fejlesztési módban (isDevelopment = true) egy LogInterceptor csatolódik, amely a kérések és válaszok részleteit naplózza, megkönnyítve a hibakeresést.

Az alkalmazásban két különböző Dio-példány létezik, amelyeket névvel regisztrálunk a get_it konténerbe. A simpleClient az /auth végponthoz használatos, és nem tartalmaz AuthorizationInterceptor-t, mivel a bejelentkezési kéréshez még nincs JWT-token. Az authorizedClient az AuthorizationInterceptor-val kiegészített példány, amelyet minden egyéb API-híváshoz alkalmazunk.

A Retrofit²⁶ annotáció-alapú API-generátor lehetővé teszi, hogy az API-osztályok pusztán függvény-aláírásokból álljanak. A ha_service.dart tartalmazza az összes backend végpontot:

²⁴<https://bloclibrary.dev/>

²⁵<https://pub.dev/packages/dio> HTTP-kliens alkotja

²⁶<https://pub.dev/packages/retrofit>

1. táblázat. Az alkalmazás BLoC osztályai, eseményeik és állapotaik.

BLoC osztály	Esemény	Kibocsátott állapotok
AuthBloc	AuthSignInRequested, AuthSignOutRequested	AuthInitial, AuthLoading, AuthSuccess, AuthError
HomeBloc	LoadRooms	HomeInitial, RoomsLoading, RoomsLoaded, HomeError
RoomDevicesBloc	LoadRoomDevices, RefreshRoomDevices	RoomDevicesInitial, RoomDevicesLoading, RoomDevicesLoaded, RoomDevicesError
GroupDevicesBloc	LoadGroupDevices, RefreshGroupDevices	GroupDevicesInitial, GroupDevicesLoading, GroupDevicesLoaded, GroupDevicesError
GroupsBloc	LoadGroups	GroupsInitial, GroupsLoading, GroupsLoaded, GroupsError
AddRoomBloc	CreateRoom, ReloadPage	AddRoomInitial, AddRoomLoading, AddRoomSuccess, AddRoomFailure
AccountPageBloc	— (indulásnál tölt)	AccountPageStateInitial
DevicesBloc	LoadDevices	DevicesInitial, DevicesLoading, DevicesLoaded, DevicesError

szoba-, csoport-, eszköz- és szimuláció-végpontokat egyaránt. A `@RestApi(baseUrl: ...)` osztályannotáció megadja az alap URL-t. A `@GET`, `@POST`, `@PUT`, `@PATCH`, `@DELETE`, `@MultiPart` annotációk megadják a HTTP-metódust és az útvonalat. A `@Path` paraméterek az URL változó részét, a `@Query` paraméterek a query string értékeit, a `@Body` a kérés testét, a `@Part` a multipart mezőket jelölik. A `build_runner` a `.g.dart` fájlban implementálja a tényleges HTTP-hívásokat.

5.2.3. WebSocket

A valós idejű kommunikációhoz a `WsService` osztály a `stomp_dart_client`²⁷ csomag `StompClient`-jét alkalmazza. A `WsService` négyféle kapcsolatot kezel. A `connect()` szoba-szintű frissítésekhez a `/user/queue/rooms/{roomId}` destination-re iratkozik fel, és a

²⁷https://pub.dev/packages/stomp_dart_client

/user/queue/paired destinationre az eszközpárosítási értesítésekhez. A `connectGroup()` a csoport-szintű frissítésekhez a /user/queue/groups/{groupId} destination-re iratkozik. A `connectScheduleReceiver()` az /user/queue/schedule destination-re iratkozik, és szimuláció-frissítéskor automatikus újralekérdezést vált ki. A `disconnect()` az aktív kapcsolatot bontja; ezt hívja meg a vissza gomb lenyomásakor a szoba és csoport oldal.

Minden STOMP-kapcsolat tartalmazza az `Authorization: Bearer` fejléct mind a STOMP handshake-ben (`stompConnectHeaders`), mind a WebSocket szinten (`websocketConnectHeaders`), biztosítva, hogy a Spring Security mindkét rétegen ellenőrizze a token.

5.2.4. Adatperzisztencia

Az alkalmazás adatperzisztenciáját a `UserRepository` osztály biztosítja `SharedPreferences`-szel (`shared_preferences`²⁸ csomag). A `SharedPreferences` az Android platform kulcs-érték tárolója, amellyel kis méretű adatok tárolhatók tartósan, alkalmazás-újraindítás után is megőrzött formában.

A `UserRepository` a `UserModel`-t JSON-ba szerializálva tárolja a `user_data` kulcs alatt. A `saveUser()` metódus JSON-kódolással menti a modellt; a `getUserData()` metódus visszatöltéskor JSON-dekódolást végez, és hiba esetén törli a sérült adatot. A `getToken()` metódus először a memóriában lévő `_user` változót ellenőrzi, majd ha szükséges, a `SharedPreferences`-ből tölti vissza a token.

A `UserRepository` a kijelentkezéskor a `clearUser()` metódussal törli a tárolt adatokat, biztosítva, hogy sem a token, sem a felhasználói adatok ne maradjanak az eszközön.

5.2.5. Dependency injection

Az alkalmazás dependency injectionje a `get_it`²⁹ csomagon alapul. Az `injectable.dart` `configureDependencies()` aszinkron függvénye az alkalmazás indulásakor, a `MaterialApp` felépítése előtt regisztrálja az összes függőséget a `GetIt` konténerbe. A függőségek három életciklus-stratégia valamelyikével kerülnek regisztrálásra.

A `registerSingleton` az alkalmazás teljes futásideje alatt egyetlen példányt tart fenn, és azonnal létrehozza azt. Ez a `ThemeManager`, a `SharedPreferences`, a `UserRepository`

²⁸https://pub.dev/packages/shared_preferences

²⁹https://pub.dev/packages/get_it

és a két Dio-példány esetén alkalmazott stratégia. A `registerLazySingleton` szintén egyetlen példányt tart fenn, de az első `getIt<T>()` lekérésig elhalasztja a létrehozást (`AuthService`, `HaService`, `TasmotaService`, `WsService`, `Logger`). A `registerFactory` minden `getIt<T>()` hívásra új példányt hoz létre (`DevicesBloc`). A `GoogleSignIn` `registerSingletonAsync`-kal regisztrálódik, mert az `initialize()` aszinkron inicializálást igényel.

5.2.6. Nemzetköziesítés

Az alkalmazás az `easy_localization`³⁰ csomag segítségével van kétnyelvűsítve: magyarul és angolul érhető el. A csomag a JSON fordítási fájlokból a `build_runner` segítségével automatikusan generálja az összesítő fileokat.

A `flutter pub run easy_localization:generate -f keys -o locale_keys-g-dart` terminálban futtatható parancsra generálja a nyelvi fájlokat. Az `EasyLocalization` widget a `MaterialApp` fölé kerülve biztosítja a lokalizációs kontextust. Az alkalmazás az eszköz rendszernyelvét követi; ha a rendszernyelv nem támogatott (nem magyar vagy angol), akkor az angol az alapértelmezetten választott nyelv.

5.2.7. Flutter architektúra

A Flutter architektúrája³¹ réteges felépítésű, ahol minden réteg az alatta lévő komponensekre épül. A legalsó szint a platform-specifikus *embedder*, amely az operációs rendszer és az alkalmazás közötti kommunikációt biztosítja, kezelve például a bemeneti eseményeket, az ablakkezelést és a natív rendszerhívásokat.

Erre épül a C++ nyelven írt *Flutter engine*, amely felelős a grafikai megjelenítésért (a Skia grafikus motor segítségével), az animációk futtatásáért, valamint a Dart futtatókörnyezet biztosításáért. Az engine gondoskodik arról, hogy az alkalmazás különböző platformokon egységes módon jelenjen meg.

A legfelső réteg a Dart nyelven megvalósított *Flutter framework*, amely magas szintű fejlesztői eszközöket és absztrakciókat kínál. A framework alapvető építőelemei a *widgetek*. A Flutterben minden felhasználói felületi elem widgetként jelenik meg, legyen szó egyszerű szövegről, gombokról vagy összetett elrendezésekről. A widgetek hierarchikus fa-struktúrába

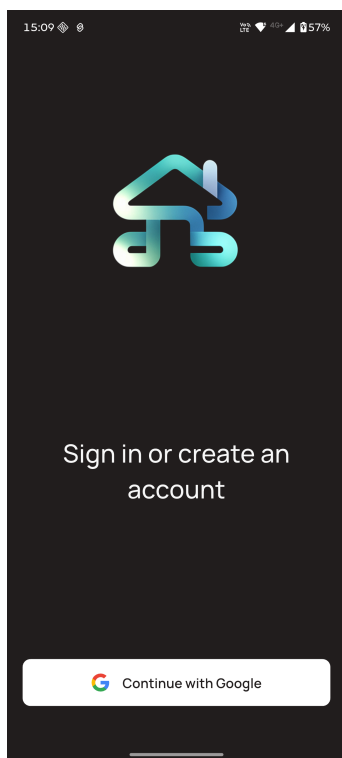
³⁰https://pub.dev/packages/easy_localization

³¹<https://docs.flutter.dev/resources/architectural-overview>

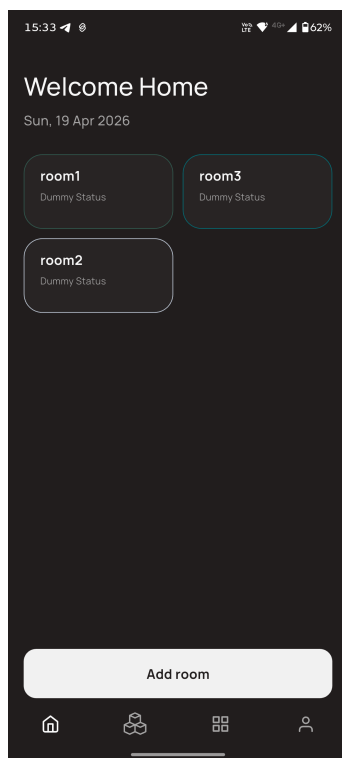
rendeződnek (widget tree), amely meghatározza a felhasználói felület szerkezetét és viselkedését. A framework tartalmazza az elrendezési mechanizmusokat, az animációs rendszert, valamint a Material és Cupertino komponenskönyvtárakat is.

A Dart programozási nyelv képezi a Flutter³² alapját [11][1]. Fejlesztési környezetben a Dart Virtual Machine (DVM) biztosítja a *hot reload* funkciót, amely lehetővé teszi, hogy a kódmódosítások az alkalmazás újraindítása nélkül, azonnal megjelenjenek a futó alkalmazásban. Ez jelentősen gyorsítja a fejlesztési folyamatot és támogatja az iteratív alkalmazásfejlesztést.

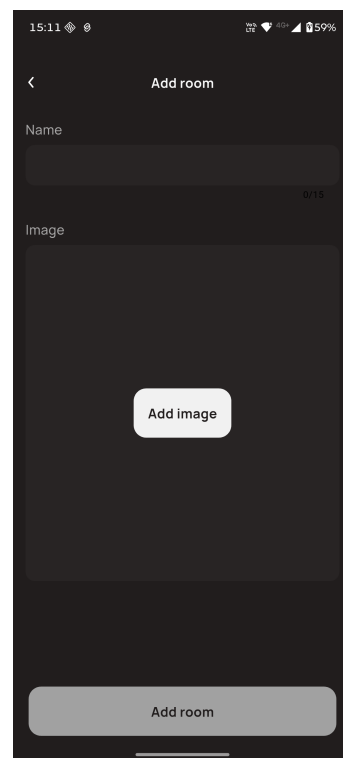
³²<https://docs.flutter.dev/>



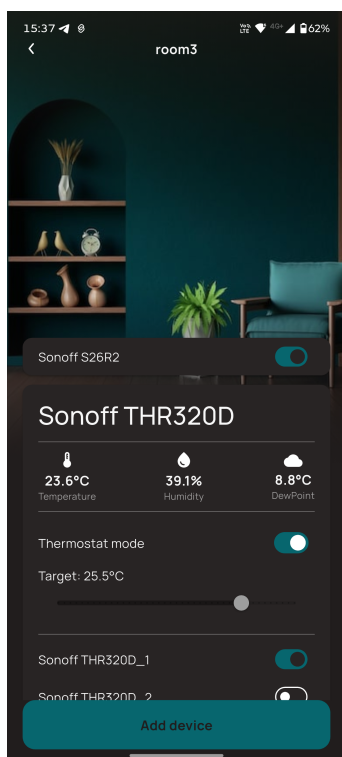
4. ábra. Bejelentkezés



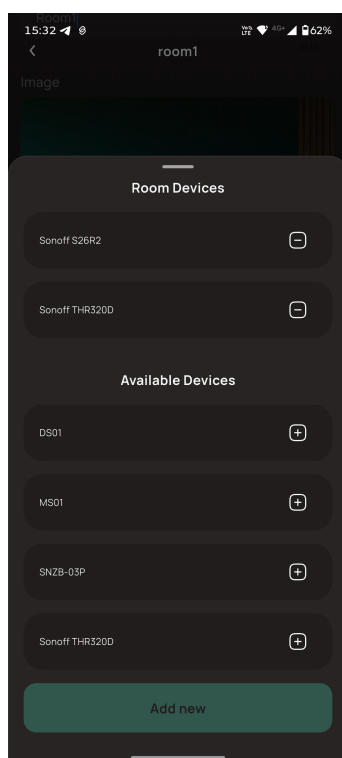
5. ábra. Főoldal - Szobák



6. ábra. Szoba hozzáadása



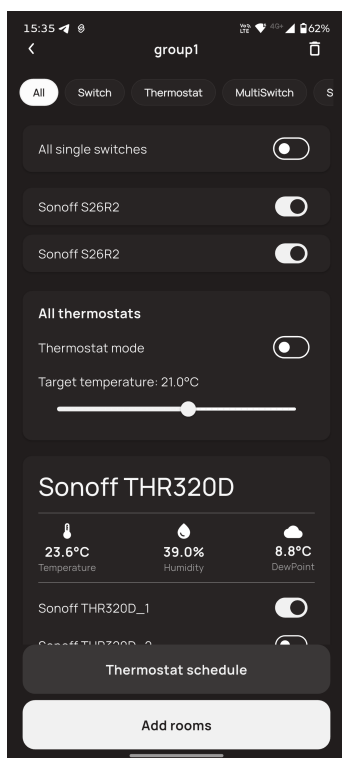
7. ábra. Szoba oldala



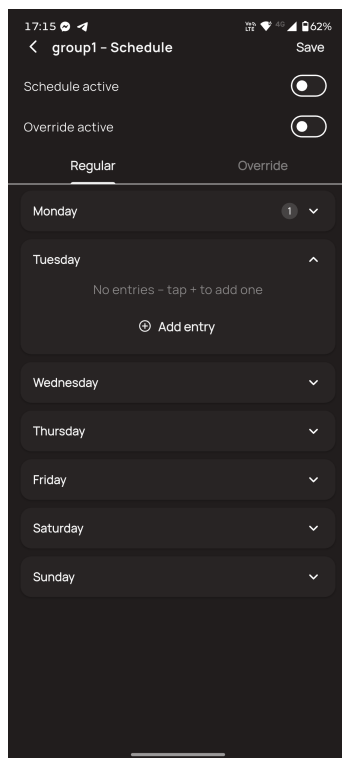
8. ábra. Eszközök hozzáadása a szobához



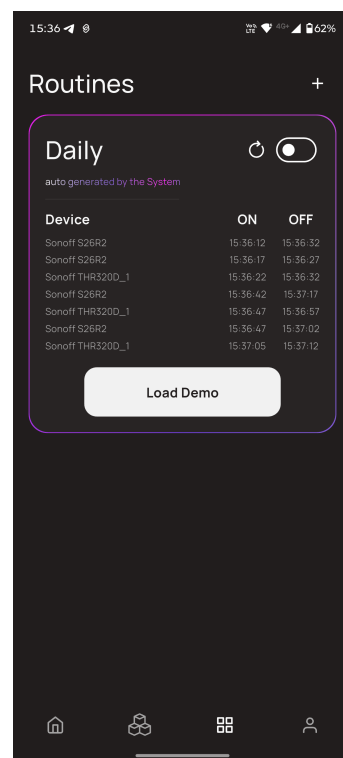
9. ábra. Csoportok



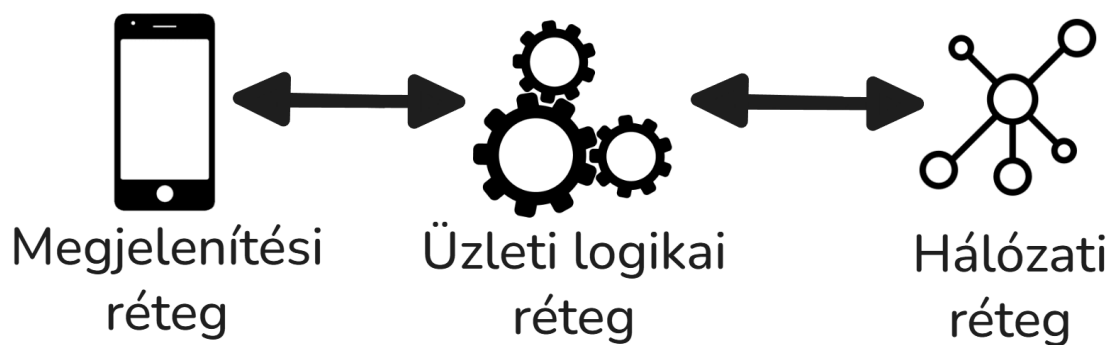
10. ábra. Csoport oldala



11. ábra. Termosztát
ütemtervek



12. ábra. Rutinok



13. ábra. Alkalmazás architektúra

6. DevOps

A LivingSmart backend folyamatos integrációja és kitelepítése GitLab CI/CD³³ és Kubernetes³⁴ segítségével valósul meg. Az automatizált pipeline biztosítja, hogy a kódmódosítások ellenőrzöten, reprodukálható módon és emberi beavatkozás nélkül kerüljenek a futó környezetbe.

A DevOps architektúra célja a reprodukálhatóság, az automatizáltság és a biztonság biztosítása [6]. A teljes infrastruktúra deklaratív konfigurációkon alapul, amelyek verziókezeltek, így minden módosítás visszakövethető.

6.1. GitLab CI/CD

A CI/CD folyamatot a repository gyökerében található `.gitlab-ci.yml` konfiguráció írja le. A pipeline három fő szakaszból áll:

`stages:`

- `utilities`
- `build`
- `deploy`

6.1.1. Utilities szakasz

A *utilities* szakaszban egy Python-alapú script generálja a környezeti konfigurációs fájlt:

`generate-env-file:`

```
stage: utilities
image: python:3.9-slim
script:
  - python3 scripts/generate-env.py
```

Az így létrejövő `.env` fájl artifactként kerül továbbadásra a későbbi pipeline lépésekhez.

6.1.2. Build szakasz

Merge request esetén a rendszer Gradle segítségével lefordítja az alkalmazást és elkészíti a futtatható `bootJar` fájlt:

³³<https://docs.gitlab.com/ee/ci/>

³⁴<https://kubernetes.io/docs/home/>

```
build:
  stage: build
  image: gradle:8.14.3-jdk17
  script:
    - gradle bootJar
```

Amennyiben a develop ágra történik push, Docker-kép épül és feltöltésre kerül a GitLab Container Registry-be:

```
docker build -t $CONTAINER_IMAGE .
docker push $CONTAINER_IMAGE
```

A Docker alkalmazása biztosítja, hogy az alkalmazás és annak összes függősége egységes, izolált környezetben fusson [12]. A konténer image neve környezeti változóként kerül definiálásra:

```
variables:
  CONTAINER_IMAGE: $CI_REGISTRY/livingsmart/livingsmart-backend:latest
```

6.1.3. Deploy szakasz

A deploy szakasz a Kubernetes klaszterrel kommunikál kubectl segítségével [2]. A CI környezetben tárolt KUBE_CONFIG változó base64 formátumban kerül tárolásra, majd futáskor dekódolásra:

```
echo $KUBE_CONFIG | base64 -d > kubeconfig
export KUBECONFIG=$PWD/kubeconfig
```

A registry hitelesítési adatokat a pipeline Kubernetes Secret erőforrásként hozza létre:

```
kubectl create secret docker-registry gitlab-registry \
  --docker-server=r.edu.codespring.ro \
  --docker-username=$CI_REGISTRY_USER \
  --docker-password=$CI_REGISTRY_PASSWORD \
  --dry-run=client -o yaml | kubectl apply -f -
```

Ezután a rendszer alkalmazza a Kubernetes manifest fájlokat:

```
kubectl apply -f k8s/database-pvc.yaml
kubectl apply -f k8s/database-init-configmap.yaml
kubectl apply -f k8s/backend-deployment.yaml
kubectl apply -f k8s/backend-service.yaml
kubectl apply -f k8s/ingress.yaml
```

Végül a `rollout restart` és `rollout status` parancsok biztosítják, hogy az új verzió sikeresen elinduljon:

```
kubectl rollout restart deployment/living-smart-backend
kubectl rollout status deployment/living-smart-backend --timeout=180s
```

A státuszellenőrzés garantálja, hogy hibás verzió nem marad futásban.

Az érzékeny konfigurációs adatok (adatbázis-jelszavak, registry-hitelesítési adatok) nem részei a Docker-képnek, CI-változóként vagy Kubernetes Secretként kerülnek injektálásra futásidőben.

6.2. Kubernetes

A backend egy Kubernetes-klaszteren fut, amely konténerizált alkalmazások automatizált üzemeltetésére szolgál. A klaszter kezelése Rancher felületen keresztül történik.

6.2.1. Deployment

A backend futtatását Deployment erőforrás biztosítja:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: living-smart-backend
spec:
  replicas: 1
```

Jelenleg a backend egy példányban fut, azonban a `replicas` paraméter növelésével horizontális skálázás valósítható meg.

A Deployment tartalmaz egy `initContainer`-t is, amely az adatbázis inicializálását végzi:

`initContainers:`

- `name: database-init`
`image: busybox:1.35`
`command: ["/bin/sh"]`
`args: ["/scripts/init-db.sh"]`

Ez biztosítja, hogy az SQLite adatbázis megfelelő verzióban létezzen még a fő konténer indulása előtt.

6.2.2. Perzisztencia

Az adatbázis állandó tárolását `PersistentVolumeClaim` biztosítja:

`kind: PersistentVolumeClaim`

`spec:`

`accessModes:`

- `ReadWriteOnce`

`resources:`

`requests:`

`storage: 1Gi`

Ez garantálja, hogy a pod újraindítása esetén az adatbázis tartalma ne vesszen el.

6.2.3. Service és Ingress

A Service erőforrás biztosítja a klaszteren belüli elérést, míg az Ingress külső hozzáférést biztosít TLS titkosítással:

`annotations:`

`kubernetes.io/ingress.class: traefik`

`cert-manager.io/cluster-issuer: letsencrypt-prod`

A TLS tanúsítvány automatikusan generálódik a Let's Encrypt szolgáltatáson keresztül, biztosítva a REST API és WebSocket végpontok HTTPS-alapú védelmét.

6.2.4. Megbízhatóság és skálázhatóság

A Kubernetes a következő előnyöket biztosítja:

- automatikus konténer-újraindítás hiba esetén,
- rolling update verzióváltáskor,
- horizontális skálázhatóság,
- deklaratív infrastruktúra-kezelés,
- namespace-alapú izoláció.

A jelenlegi architektúra kis- és közepes terhelésre optimalizált, azonban a rendszer könnyen bővíthető több replika, külső adatbázis vagy autoscaling mechanizmus bevezetésével.

7. Következtetések és továbbfejlesztési lehetőségek

A LivingSmart egy olyan egységes okosotthon-rendszer, amely sikeresen integrálja a Wi-Fi-s Tasmota-eszközöket és a Zigbee-eszközöket egyetlen, helyi gateway architektúrára épülő platformba. A rendszer a lakásbiztonsághoz aktívan hozzájárul: a szokásmegfigyelésen alapuló jelenlét-szimuláció valósan ható aktivitást kelt a tulajdonos távolléte alatt, és ez a funkció internetkapcsolat nélkül is működőképes marad. A termosztát-ütemezési funkció napszakhoz kötött, csoportos hőmérséklet-szabályozást biztosít, amelyet a legtöbb gyártói felhőmegoldás nem támogat. Az három-rétegű architektúrán alapuló Spring Boot backend, a BLoC-alapú Flutter-mobilalkalmazás, a Retrofit-generált API-osztályok és a STOMP WebSocket-csatorna egy jól strukturált, rétegelt rendszert alkotnak.

A projekt egy tanulságos problémát is felvetett: a különböző protokollú eszközök egységes kezelése a Home Assistant-en mint aggregációs rétegen keresztül leegyszerűsítette a gateway logikáját, mivel a Python-kódnak nem kell ismernie az adott eszköz fizikai kommunikációs protokollját. Ez az absztrakciós döntés visszaigazolódott a fejlesztés során, amikor az első Zigbee-eszközöket hozzáadtuk a már működő Tasmota-infrastruktúrához.

Számos területen azonosíthatók továbbfejlesztési lehetőségek. Az adatgyűjtés hatékonyabbá tehető eseményalapú megközelítéssel: a jelenlegi REST-polling-ciklus kiváltható esemény alapú frissítésekkel, amely azonnal értesíti a fogyasztókat minden állapotváltozásról, csökkentve a gateway CPU-terhelését és javítva a reakcióidőt.

A viselkedési modell pontosítható hétköznapi/hétvége megkülönböztetéssel és szezonális mintázatok figyelembevételével. Emellett a jelenléti minták (mikor van ténylegesen otthon a felhasználó) is beépíthetők a profilozásba, ami életszerűbbé tenné a szimulációt. A push-értesítések bevezetése értesítené a felhasználót, ha a szimuláció befejeződik, ha egy eszköz elérhetetlenné válik, vagy ha az ütemterv frissül.

Hosszabb távon egy könnyűsúlyú adatbázis SQLite lehetővé teszi, hogy a backend lokális változata a gateway-en is fusson, és akár teljes felhőfüggetlenség is elérhető legyen. Egy Partner API kialakítása lehetővé tenné a LivingSmart integrálását más okosotthon-platformokkal, bővítve az elérhető eszközök körét.

Hivatkozások

- [1] Gilad Bracha. *The Dart Programming Language*. Addison-Wesley Professional, 2015. ISBN: 978-0321927705.
- [2] Brendan Burns, Joe Beda és Kelsey Hightower. *Kubernetes: Up and Running*. 2. kiadás. O'Reilly Media, 2019. ISBN: 978-1492046530.
- [3] Ian Fette és Alexey Melnikov. *The WebSocket Protocol*. RFC 6455. 2011. dec. DOI: 10.17487/RFC6455. URL: <https://www.rfc-editor.org/info/rfc6455>.
- [4] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Internet Engineering Task Force, 2012. URL: <https://datatracker.ietf.org/doc/html/rfc6749>.
- [5] Michael Jones, John Bradley és Nat Sakimura. *JSON Web Token (JWT)*. RFC 7519. Internet Engineering Task Force, 2015. DOI: 10.17487/RFC7519. URL: <https://datatracker.ietf.org/doc/html/rfc7519>.
- [6] Gene Kim és mások. *The DevOps Handbook*. IT Revolution Press, 2016. ISBN: 978-1942788003.
- [7] Mikael Kohvakka és mások. „Performance Analysis of IEEE 802.15.4 and ZigBee for Large-Scale Wireless Sensor Network Applications”. *Proceedings of the 3rd ACM International Workshop on Performance Evaluation of Wireless Ad Hoc, Sensor and Ubiquitous Networks (PE-WASUN)*. ACM, 2006, 48–57. oldal. DOI: 10.1145/1163610.1163619.
- [8] Heckler Mark. *Spring Boot: Up and Running*. O'Reilly Media, 2021. ISBN: 978-1492076988.
- [9] Mark Masse. *REST API Design Rulebook*. O'Reilly Media, 2012. ISBN: 978-1449310509.
- [10] Simon Monk. *Raspberry Pi Cookbook: Software and Hardware Problems and Solutions*. 4. kiadás. O'Reilly Media, 2023. ISBN: 978-1098130923.
- [11] Marco L. Napoli. *Beginning Flutter: A Hands On Guide to App Development*. <https://digilib.stekom.ac.id/dokumen/ebook>. John Wiley & Sons, 2019. ISBN: 978-1119550822.
- [12] Jeff Nickoloff és Stephen Kuenzli. *Docker in Action*. 2. kiadás. Shelter Island, NY: Manning Publications, 2019. ISBN: 978-1617294761.
- [13] OASIS. *MQTT Version 5.0*. OASIS Standard. Organization for the Advancement of Structured Information Standards, 2019. márc. URL: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>.
- [14] Office for National Statistics (ONS). *Crime in England and Wales: year ending December 2025*. Statistical bulletin. 2026. ápr.
- [15] Mike Owens. *The Definitive Guide to SQLite*. Apress, 2006. ISBN: 978-1590596739.

- [16] Iris van Sintemaartensdijk és mások. „Mere presence of informal guardians deters burglars: a virtual reality study”. *Journal of Experimental Criminology* 17.4 (2021), 657–676. oldal. DOI: 10.1007/s11292-020-09430-1.
- [17] Statista. *Number of users of smart homes worldwide from 2018 to 2029 (in millions)*. Statista Graph. Hozzáférve 2026, Április 23, <https://www.statista.com/forecasts/887613/number-of-smart-homes-in-the-smart-home-market-in-the-world/>. 2025. júl.
- [18] Craig Walls. *Spring in Action*. 4. kiadás. Manning Publications, 2014. ISBN: 978-1617291203.