

POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica –
Indirizzo Software



**Politecnico
di Torino**

**Smartotum: progettazione di una mobile app
cross-platform con BFF e integrazione Home Assistant**

Supervisor

Prof. Luca ARDITO

Candidato

Alberto MARGAGLIA

Marzo 2026

Ringraziamenti

Desidero ringraziare i miei genitori per avermi supportato in questi 5 anni e mezzo. Mia mamma per non aver perso fiducia in me nonostante un inizio difficile e mio papà per avermi sempre spronato a dare il meglio di me anche quando mi sarei voluto accontentare.

Grazie alla mia ragazza Sasà/Patricia/Sassa per essermi sempre stata vicina tutte le volte che non passavo un appello importante, per tutte le sere passate insieme che facevano scomparire l'ansia, per avermi accompagnato agli esami, nonostante venisse importunata da stranieri che non parlavano la nostra lingua, e per tutte le foto portafortuna con Chupi.

Grazie alla mia famiglia e ai miei amici per esserci stati nei momenti di svago e spensieratezza che rompevano le monotone giornate di studio allgerendo le difficoltà con risate, consigli e sostegno che hanno reso questo cammino più bello e meno faticoso.

Ringrazio il Relatore, Prof. Ardito, per la guida attenta, la disponibilità e i preziosi consigli che hanno reso possibile la realizzazione di questo lavoro.

A tutti voi, grazie questo traguardo è anche vostro.

Indice

Elenco delle figure	IX
1 Introduzione	1
1.1 Contesto e motivazioni	1
1.2 Problema affrontato	2
1.2.1 Frammentazione tecnologica e duplicazione della logica . . .	2
1.2.2 Complessità di integrazione con servizi eterogenei	2
1.2.3 Vincoli operativi e aspettative di UX	3
1.3 Obiettivi della tesi	3
1.3.1 Obiettivo principale	3
1.3.2 Obiettivi specifici	4
1.4 Soluzione proposta: panoramica architetturale	4
1.4.1 Client mobile (React Native)	4
1.4.2 Backend For Frontend (BFF) in Express	5
1.4.3 Servizi upstream: Smartotum API e Home Assistant	5
1.5 Contributi principali	5
1.6 Metodologia e criteri di progettazione	6
2 Mobile app e Cross-platform	7
2.1 Mobile app: definizione e caratteristiche	7
2.1.1 Vincoli tipici e conseguenze architetturali	7
2.2 Approcci di sviluppo: nativo, web e ibrido	8
2.2.1 Applicazioni native	8
2.2.2 Applicazioni web e Progressive Web App	9
2.2.3 Applicazioni ibride WebView-based	9
2.2.4 Cross-platform moderni	10
2.3 Cross-platform: definizione e motivazioni	10
2.3.1 Definizione	10
2.3.2 Motivazioni generali	11
2.3.3 React Native: modello di esecuzione e architettura	11
2.3.4 React Native e Flutter: confronto sintetico	13

2.4	Confronto con lo sviluppo nativo	14
2.4.1	Criteri di confronto	14
2.4.2	Vantaggi del cross-platform	14
2.4.3	Svantaggi e rischi	15
2.4.4	Tabella di sintesi	15
2.5	Implicazioni per il progetto	16
2.5.1	Convergenza da soluzione multi-codebase	16
2.5.2	Ruolo del Backend For Frontend (BFF)	16
2.5.3	Home Assistant e requisiti real-time	16
2.5.4	Sicurezza e token: integrazione tra scelte client e scelte server	17
3	Analisi del problema e requisiti	18
3.1	Inquadramento del capitolo	18
3.2	Attori, confini del sistema e vincoli	18
3.2.1	Attori	18
3.2.2	Confini e responsabilità	19
3.2.3	Vincoli operativi	19
3.3	Casi d'uso principali	20
3.3.1	Vista d'insieme	20
3.3.2	UC-01: login e creazione sessione BFF	21
3.3.3	UC-02: refresh token e continuità della sessione	21
3.3.4	UC-03: selezione e cambio sistema attivo	21
3.3.5	UC-04/UC-05/UC-06: real-time e comandi Home Assistant .	22
3.3.6	UC-07: recupero statistiche da Home Assistant (REST) . . .	22
3.3.7	UC-08: ottenere URL stream camera (REST)	23
3.3.8	UC-09: registrazione push per notifiche Home Assistant . . .	23
3.3.9	UC-10: invio posizione e vincoli OS-level	24
3.3.10	UC-11: gestione schedule/programma temperature	24
3.3.11	UC-12: gestione automazioni e scene	25
3.3.12	UC-13: gestione sicurezza (codici allarme, sensori e ritardi) .	26
3.3.13	UC-14: logout e revoca sessione	27
3.4	Requisiti funzionali	27
3.4.1	Autenticazione e sessione	27
3.4.2	Gestione sistemi e funzionalità Smartotum	28
3.4.3	Integrazione Home Assistant	28
3.4.4	Push e location	29
3.5	Requisiti non funzionali	29
3.5.1	Sicurezza	29
3.5.2	Resilienza e continuità operativa	29
3.5.3	Osservabilità e manutenibilità	30
3.5.4	Requisiti quantitativi	30

3.6	Artefatti di supporto: interfacce e persistenza	30
3.6.1	Interfacce principali del BFF	30
3.6.2	Persistenza lato BFF: entità logiche	31
4	Architettura e progettazione della soluzione	32
4.1	Visione d'insieme e principi architetturali	32
4.1.1	Contratti e canali di comunicazione	32
4.2	Scomposizione in componenti	33
4.2.1	Componenti lato client	33
4.2.2	Componenti lato BFF	34
4.3	Razionale delle scelte architetturali	34
4.3.1	BFF come boundary tra mobile e servizi esterni	34
4.3.2	Token upstream lato server e minimizzazione delle credenziali sul client	35
4.3.3	Canale real-time mediato: WebSocket verso BFF invece che verso Home Assistant	35
4.3.4	React Native con layer nativo mirato	36
4.4	Allineamento requisiti–architettura	36
4.5	Flussi informativi e confini di persistenza (vista concettuale)	37
4.5.1	Dati persistiti sul dispositivo	37
4.5.2	Dati persistiti sul server BFF	38
4.5.3	Dati transienti e gestione degli eventi	38
4.6	Diagrammi di architettura	38
5	Backend for Frontend (BFF)	44
5.1	Inquadramento e responsabilità del BFF nel progetto	44
5.2	Architettura interna: moduli e flusso delle richieste	45
5.3	Modello di autenticazione e gestione token	46
5.3.1	Middleware Bearer e validazione access token	46
5.3.2	Token opachi, hashing e TTL	46
5.3.3	Rotazione del refresh token	47
5.3.4	Revoca sessione e logout	48
5.3.5	Token upstream: cifratura a riposo e rinnovo	48
5.4	Normalizzazione errori e gestione timeout	49
5.5	Contratti API esposti dal BFF	50
5.5.1	API di autenticazione (/auth/*)	50
5.5.2	API Smartotum (/smartotum/*)	50
5.5.3	API Home Assistant (REST) e webhook push	52
5.6	WebSocket come canale real-time centrale	53
5.6.1	WebSocket pubblico BFF–Client: handshake, auth e close codes	53

5.6.2	Protocollo messaggi: subscribe, snapshot, delta e service_call	55
5.6.3	Rate limiting e protezione del canale	56
5.6.4	Heartbeat e session liveness	57
5.6.5	WebSocket server-side BFF–Home Assistant: connessione condivisa e caching	57
5.6.6	Strategie di recovery: failure modes tipici	58
5.7	Struttura del database del BFF	58
5.7.1	Entità principali e responsabilità	59
5.7.2	Relazioni e cardinalità	59
5.7.3	Chiavi, vincoli e indici rilevanti	60
5.7.4	Lifecycle dei record (sintesi operativa)	60
5.7.5	Nota di progettazione: perimetro e non ridondanza dei dati .	60
5.7.6	Implicazioni architetturali del deployment per impianto . . .	61
5.8	Consolidamento: decisioni progettuali e trade-off	62

6 Frontend mobile: progettazione e implementazione del client React

Native		64
6.1	Obiettivo del capitolo	64
6.2	Struttura del progetto frontend	65
6.3	Bootstrap applicativo e navigazione	66
6.3.1	Entry point e inizializzazione	66
6.3.2	Root flow condizionale	66
6.3.3	Organizzazione della navigazione	67
6.4	Gestione della sessione lato client e contesto multi-sistema	68
6.4.1	AuthContext come coordinatore del ciclo di vita della sessione	68
6.4.2	Persistenza dei token BFF in storage nativo sicuro	68
6.4.3	Gestione del sistema attivo e multi-sistema	69
6.4.4	Scenari di unauthorized e ritorno allo stato non autenticato .	69
6.5	Layer API REST lato client	70
6.5.1	Ruolo del modulo <code>API.ts</code>	70
6.5.2	Bearer injection, timeout e normalizzazione errori	70
6.5.3	Refresh token e retry con strategia single-flight	71
6.6	Client WebSocket e gestione del real-time lato frontend	71
6.6.1	HomeAssistantProvider come perno del real-time	71
6.6.2	Handshake, subscribe e protocollo base lato client	72
6.6.3	Heartbeat, riconnessione e recovery della sessione	72
6.6.4	Impatto sulla UI e sincronizzazione stato	73
6.7	Stato applicativo, persistenza locale e sincronizzazione UI	73
6.7.1	Stato globale e stato locale	73
6.7.2	Persistenza locale: encrypted storage e SQLite	74
6.7.3	Coerenza tra stato, persistenza e lifecycle mobile	75

6.8	Integrazione nativa mirata	75
6.8.1	Razionalità dell'integrazione nativa nel client React Native	75
6.8.2	Secure storage nativo e interfaccia React Native	76
6.8.3	Location/background e continuità della sessione fuori dalla UI	76
6.9	Schermate principali e organizzazione funzionale della UI	77
6.9.1	Aree funzionali principali	77
6.9.2	Ruolo delle schermate nel flusso utente	77
6.9.3	Scalabilità della UI e riuso dei componenti	78
6.9.4	Home page e accesso rapido ai controlli	78
6.9.5	Navigazione per stanze e dettaglio contestuale	79
6.9.6	Gestione della sicurezza e configurazione dell'allarme	81
6.9.7	Programmi di temperatura e pianificazione stagionale	82
6.9.8	Automazioni dei dispositivi e schedulazione delle azioni	84
6.9.9	Cronologia e visualizzazione grafica dei sensori	87
6.9.10	Impostazioni applicative e selezione del sistema	88
6.9.11	Notifiche ed elenco degli eventi	89
6.10	Trade-off implementativi e considerazioni critiche	91
6.10.1	Centralizzazione vs modularità nel layer API	91
6.10.2	Complessità introdotta dal doppio livello di token e dal multi-sistema	91
6.10.3	Robustezza del provider real-time e costo della complessità	92
6.10.4	Valutazione complessiva delle scelte lato client	92
7	Test e validazione	93
7.1	Obiettivi e criteri di validazione	93
7.2	Strumenti e perimetro dei test automatici	94
7.2.1	Framework e organizzazione della suite	94
7.2.2	Comandi e riproducibilità	94
7.3	Ambiente di test	95
7.3.1	Configurazione e variabili d'ambiente	95
7.3.2	Database di test e gestione dello stato	95
7.3.3	Mock e stub degli upstream	95
7.4	Copertura dei test automatici sul BFF	96
7.4.1	Unit test	96
7.4.2	Integration test (HTTP e WebSocket)	96
7.5	Validazione funzionale tramite scenari ripetibili	97
7.5.1	Scenari su sessione e continuità operativa (A–C)	97
7.5.2	Scenari real-time su WebSocket (D–G)	98
7.5.3	Scenari su integrazione REST e gestione errori (H)	98
7.5.4	Tabella riassuntiva degli scenari	98
7.6	Evidenze e osservabilità	99

7.6.1	Discussione dei gap di copertura	99
8	Conclusioni	102
8.1	Bilancio della convergenza architetture	103
	Bibliografia	107

Elenco delle figure

4.1	Architettura a livelli	39
4.2	Sequenza: login e salvataggio token in storage nativo	40
4.3	Sequenza: richiesta protetta, refresh e retry	41
4.4	Sequenza integrata del canale WebSocket: apertura, sospensione in background, ripresa e recovery	42
4.5	Schema logico (semplificato)	43
6.1	Home page dell'applicazione e apertura di un modale di dettaglio per il controllo delle entità	79
6.2	Sezione <i>Stanze</i> : vista contestuale per ambiente e integrazione delle telecamere	80
6.3	Sezione <i>Sicurezza</i> : attivazione dei sensori, configurazione dei codici e impostazione delle tempistiche dell'allarme	81
6.4	Sezione <i>Programmi</i> : elenco dei programmi stagionali, creazione guidata e modifica degli slot orari	83
6.5	Sezione <i>Programmi</i> : gestione delle automazioni, creazione di una regola e assegnazione ai dispositivi	85
6.6	Visualizzazione storica dei dati dei sensori tramite grafici temporali	87
6.7	Sezione <i>Impostazioni</i> , dettaglio delle <i>App Settings</i> e selezione del sistema attivo	88
6.8	Sezione delle notifiche con filtri temporali e lista paginata degli eventi	90
7.1	Coverage dei test	100

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Smartotum si colloca nell’ambito dei sistemi *smart home*, dove l’utente si aspetta un controllo semplice e immediato di dispositivi e servizi eterogenei (illuminazione, climatizzazione, allarme, tapparelle, sensori, statistiche e consumi). In questi scenari l’esperienza d’uso è fortemente influenzata da due fattori: (i) la **coerenza** dell’interfaccia e dei flussi principali (login, selezione impianto, schermate di controllo), e (ii) la **reattività** del sistema nel riflettere lo stato reale dei dispositivi, idealmente in tempo reale.

Nel tempo, l’evoluzione del prodotto ha condotto a una situazione tipica dei sistemi cresciuti per stratificazione: l’applicazione risultava composta da più basi di codice e tecnologie, ciascuna con responsabilità parzialmente sovrapposte. In particolare, l’app presentava:

- una parte nativa Android sviluppata in Kotlin;
- una parte nativa iOS sviluppata in Swift;
- una componente web sviluppata con Vue;
- un forte affidamento su Home Assistant come piattaforma di integrazione e automazione.

Questa frammentazione aumenta il costo di manutenzione e rallenta l’evoluzione del prodotto: la stessa funzionalità tende a dover essere implementata più volte, con inevitabili divergenze tra piattaforme (comportamenti, bug, differenze di UI/UX e ritardi nei rilasci). Inoltre, in un contesto domotico, la gestione di connessioni intermittenti, aggiornamenti frequenti e credenziali sensibili amplifica i rischi tecnici legati a implementazioni duplicate.

L'obiettivo di questo lavoro è stato quindi guidato da una motivazione primaria: **convergere verso una sola codebase** per l'app mobile, capace di sostituire l'implementazione frammentata e di offrire un'esperienza coerente su Android e iOS, mantenendo solo moduli nativi mirati quando necessario (ad esempio per la persistenza sicura dei token e il tracking della posizione). La soluzione proposta si basa su una nuova applicazione in React Native e sull'introduzione di un *Backend For Frontend* (BFF) per centralizzare l'integrazione con i servizi upstream (Smartotum API e Home Assistant), mantenendo Home Assistant come elemento cardine dell'ecosistema ma rendendo l'app più semplice, sicura e manutenibile.

1.2 Problema affrontato

Il problema affrontato può essere descritto su tre piani: tecnologico, architetturale e operativo.

1.2.1 Frammentazione tecnologica e duplicazione della logica

Una base di codice separata per Android (Kotlin) e iOS (Swift) implica duplicazione della logica applicativa, in particolare per:

- autenticazione e gestione della sessione;
- navigazione e stato dell'app (selezione impianto, ritorno in schermata, notifiche);
- gestione errori e strategie di retry;
- gestione di aggiornamenti real-time e riconnessione.

La presenza di una componente web (Vue) introduce un ulteriore canale che richiede coerenza funzionale e semantica: nomenclatura delle funzionalità, disponibilità delle schermate, e uniformità dei flussi critici (ad esempio, il comportamento del login o la presentazione delle stanze e dei dispositivi). Questo tipo di architettura multi-codebase è sostenibile nel breve periodo, ma diventa fragile e costosa nel lungo periodo, soprattutto quando il dominio evolve (nuovi dispositivi, nuove entità, nuove regole di sicurezza).

1.2.2 Complessità di integrazione con servizi eterogenei

L'ecosistema Smartotum include almeno due domini principali:

- **Smartotum API** (*upstream* proprietario raggiunto via HTTP), protetto da token bearer e dotato di meccanismi di accesso e refresh;
- **Home Assistant** (HA), che espone sia API HTTP sia un canale WebSocket per lo streaming di eventi e stati, oltre a meccanismi di autenticazione basati su OAuth o token.

Integrare direttamente questi servizi dal client mobile presenta tre criticità ricorrenti:

1. **Sicurezza e gestione credenziali**: il client dovrebbe custodire e ruotare più token, con superfici d'attacco e complessità aumentate.
2. **Accoppiamento**: l'app diventerebbe dipendente dai dettagli dei servizi upstream (endpoint, errori, formati), rendendo più onerosa ogni evoluzione.
3. **Resilienza**: occorre gestire scadenze token, errori transitori e riconessioni, con alto rischio di divergenza.

1.2.3 Vincoli operativi e aspettative di UX

Un'applicazione mobile opera in condizioni non ideali: connettività variabile, passaggi tra Wi-Fi e rete cellulare, sospensione in background, limitazioni energetiche e di risorse. In ambito domotico, tuttavia, l'utente si aspetta immediatezza: *vedere* lo stato corretto dei dispositivi (ad esempio luci e allarme) e *comandare* attuatori con feedback affidabile. Questo richiede un canale real-time robusto e una gestione accurata dello stato lato client, soprattutto in presenza di aggiornamenti frequenti e potenziali disconnessioni. Nel progetto, tale requisito è affrontato tramite uno stream WebSocket mediato dal BFF verso Home Assistant e tramite strategie di riconnessione e heartbeat.

1.3 Obiettivi della tesi

1.3.1 Obiettivo principale

L'obiettivo principale del lavoro è la realizzazione di un'applicazione mobile basata su **un'unica base di codice** (React Native), in grado di fornire un'esperienza consistente su Android e iOS e di sostituire la frammentazione precedente (Kotlin/Swift/Vue), mantenendo al contempo l'integrazione con Home Assistant e includendo solo componenti nativi essenziali per storage sicuro e tracking della posizione.

1.3.2 Obiettivi specifici

All'obiettivo principale si affiancano obiettivi tecnici e progettuali, con impatto diretto su sicurezza, manutenibilità e qualità percepita:

1. **Cross-platform e uniformità dei flussi:** definire una struttura di navigazione chiara che separi bootstrap, autenticazione, selezione del sistema e area applicativa, con schermate principali coerenti.
2. **Gestione multi-sistema e continuità d'uso:** supportare la selezione di sistemi Smartotum e il passaggio tra di essi, preservando la continuità di navigazione.
3. **Introduzione di un BFF:** progettare e implementare un Backend For Frontend in Node/TypeScript (Express) che esponga un set di API unificato e protegga il client dai dettagli dei servizi upstream.
4. **Gestione robusta dell'autenticazione e dei token:** gestire access/refresh token e la loro rotazione con strategie che evitino instabilità e garantiscano separazione tra token BFF e token upstream.
5. **Real-time affidabile con Home Assistant:** garantire aggiornamenti in tempo reale sulle entità e invocazioni di servizi, con un canale stabile tra app e backend.
6. **Riduzione del coupling e arricchimento dati:** spostare nel BFF logiche di normalizzazione e arricchimento dei dati di Home Assistant, mantenendo il client più leggero.

1.4 Soluzione proposta: panoramica architetturale

La soluzione si basa su un'architettura a tre livelli che separa in modo netto responsabilità e domini:

1.4.1 Client mobile (React Native)

L'app mobile React Native è responsabile di:

- presentazione UI nativa e gestione della navigazione;
- orchestrazione della sessione applicativa (login, selezione sistema, logout);

- gestione dello storage sicuro e della continuità di sessione, con persistenza in storage nativo per i token;
- integrazione con un modulo nativo per il tracking della posizione quando richiesto;
- comunicazione verso il BFF tramite API REST e WebSocket.

1.4.2 Backend For Frontend (BFF) in Express

Il BFF rappresenta l'interfaccia unica e controllata tra app e servizi upstream. Esso:

- espone API REST protette per operazioni Smartotum e Home Assistant;
- gestisce le sessioni BFF e il ciclo di vita di access/refresh token BFF;
- mantiene e rinnova token upstream (Smartotum e HA), cifrandoli a riposo;
- espone un canale WebSocket per aggiornamenti real-time da Home Assistant e invio di comandi.

1.4.3 Servizi upstream: Smartotum API e Home Assistant

Il BFF dialoga con:

- Smartotum API via HTTP bearer, gestendo refresh e memorizzazione sicura dei token;
- Home Assistant via API e WebSocket, gestendo l'ottenimento del token (bootstrap OAuth) e mantenendo connessioni WS e logiche di subscribe/callService.

Questa separazione consente al client di non dipendere direttamente dai meccanismi di autenticazione e dalle specifiche dei servizi upstream: il client utilizza token BFF con privilegi e durata controllati, mentre il BFF conserva e rinnova i token upstream cifrati in database.

1.5 Contributi principali

I contributi di questo lavoro possono essere riassunti nei seguenti punti:

1. Progettazione e sviluppo di una nuova app mobile cross-platform in React Native

2. Introduzione di un BFF in Node/TypeScript (Express) come punto di ingresso unico per il client
3. Definizione e implementazione di una strategia di gestione token
4. Implementazione di un canale WebSocket per aggiornamenti real-time e invocazioni di servizi Home Assistant

1.6 Metodologia e criteri di progettazione

Lo sviluppo è stato guidato da criteri orientati alla manutenibilità e alla riduzione del rischio di divergenza tra piattaforme:

- **Single source of truth per il client:** definire una sola codebase per Android e iOS, in modo da rendere ogni modifica immediatamente disponibile su entrambe le piattaforme.
- **Boundary architetturale tramite BFF:** concentrare su server la complessità di integrazione e sicurezza, preservando nel client un ruolo focalizzato su UI e orchestrazione.
- **Separazione delle responsabilità:** isolare navigazione, autenticazione, API REST e client WebSocket in moduli dedicati, riducendo l'accoppiamento interno e facilitando test e debug.

Capitolo 2

Mobile app e Cross-platform

2.1 Mobile app: definizione e caratteristiche

Con il termine *mobile application* si indica un software progettato per essere eseguito su dispositivi mobili (smartphone e tablet) e distribuito attraverso canali specifici (store, enterprise distribution, o web). Le mobile app si distinguono dalle applicazioni desktop non solo per la dimensione del form factor, ma soprattutto per un insieme di vincoli e opportunità che influenzano l'intero processo di progettazione: input prevalentemente touch, schermi piccoli e in continua variazione (notch, aspect ratio, densità), presenza di sensori (GPS, accelerometro, fotocamera), connettività variabile e un ciclo di vita controllato dal sistema operativo (sospensione in background, ripresa, terminazione per pressione di memoria).

Dal punto di vista dell'ingegneria del software, questi elementi impattano sia i requisiti funzionali (ad esempio “mostrare lo stato di un impianto” o “inviare un comando”) sia quelli non funzionali: prestazioni, affidabilità, sicurezza, usabilità, manutenibilità e portabilità. Un riferimento consolidato per descrivere e valutare tali qualità è lo standard ISO/IEC 25010, che propone un modello di qualità del prodotto software e della *quality in use* [1].

2.1.1 Vincoli tipici e conseguenze architetturali

I vincoli del mobile si manifestano in modo particolarmente evidente in applicazioni che dipendono da servizi remoti e da aggiornamenti frequenti dello stato, come nel caso di applicazioni per smart home e domotica. In questi domini, l'app:

- gestisce un insieme di **sessioni** e **credenziali** (token di accesso e refresh) verso backend differenti;
- opera in condizioni di **rete intermittente**, con passaggi tra Wi-Fi e cellulare;

- deve garantire **reattività** e **coerenza** tra stato locale e remoto, spesso con canali real-time.

Ne derivano tre implicazioni architetturali ricorrenti, indipendenti dalla tecnologia specifica:

1. **Gestione dello stato e continuità d'uso:** l'app deve ripristinare rapidamente un contesto minimale (utente autenticato, impianto selezionato, schermata) per ridurre l'attrito e mantenere la percezione di continuità.
2. **Networking robusto:** timeouts, retry controllati, gestione di errori transitori e riconessioni diventano parte integrante del comportamento atteso.
3. **Sicurezza delle credenziali:** i token devono essere protetti a riposo (storage sicuro) e in transito (TLS), con strategie di rotazione e invalidazione coerenti.

Questi aspetti costituiscono il contesto entro cui si colloca la scelta dell'approccio di sviluppo: la tecnologia incide non solo sul “come” realizzare l'interfaccia, ma anche sulla sostenibilità della manutenzione e sulla capacità di garantire qualità uniforme su più piattaforme.

2.2 Approcci di sviluppo: nativo, web e ibrido

Lo sviluppo di applicazioni per dispositivi mobili può essere ricondotto a tre famiglie principali: *native app*, *web app* (incluso il paradigma PWA) e *app ibride*. A queste si affianca una quarta categoria, spesso considerata a sé, costituita dalle soluzioni *cross-platform* moderne che mirano a un'esperienza vicina al nativo.

2.2.1 Applicazioni native

Le applicazioni native sono sviluppate con linguaggi, toolchain e librerie ufficiali della piattaforma: Kotlin/Java per Android e Swift/Objective-C per iOS. Questo approccio offre:

- **Accesso completo alle API di sistema:** gestione avanzata dei permessi, servizi in background, secure storage (Keychain/Keystore), sensori e integrazione con componenti OS.
- **UI e performance:** utilizzo diretto dei componenti nativi e maggiore controllo su rendering, animazioni e risorse.
- **Conformità alle linee guida:** aderenza alle Human Interface Guidelines (iOS) e alle linee guida Material/Android [2, 3].

Il principale svantaggio risiede nel costo della moltiplicazione delle codebase: per supportare Android e iOS occorre sviluppare e mantenere due implementazioni, con inevitabile duplicazione di logiche (autenticazione, networking, gestione errori) e rischio di divergenza funzionale. In domini complessi, tale duplicazione tende ad aumentare nel tempo e ad allungare i cicli di rilascio.

2.2.2 Applicazioni web e Progressive Web App

Le *web app* sono applicazioni fruibili tramite browser, sviluppate con tecnologie web (HTML, CSS, JavaScript) e distribuite via URL. Il vantaggio principale è la portabilità: un'unica applicazione può raggiungere molteplici dispositivi senza dipendere da store e senza toolchain specifiche.

Il paradigma delle *Progressive Web App* (PWA) estende questo modello grazie a componenti come i *service worker*, che consentono caching avanzato, funzionamento offline e, in alcuni casi, integrazione con notifiche e installazione [4]. In termini architetturali, i service worker introducono un modello event-driven per attività in background e bootstrap dell'app anche in assenza di rete, avvicinando alcune caratteristiche operative a quelle delle app installabili.

Tuttavia, le web app rimangono vincolate alle capacità e alle policy del browser. Funzionalità come tracking posizione affidabile in background, accesso profondo a secure storage e gestione fine di connessioni persistenti possono risultare limitate o non uniformi tra dispositivi. In applicazioni con requisiti di integrazione OS-level e sicurezza stringente, tali limitazioni possono rendere il web un'opzione meno adeguata come canale principale.

2.2.3 Applicazioni ibride WebView-based

Le soluzioni ibride *WebView-based* (es. Cordova/PhoneGap e approcci simili) distribuiscono un'app installabile che esegue una web app all'interno di una WebView. L'accesso alle API native è ottenuto tramite plugin (bridge JavaScript-native). Il modello massimizza il riutilizzo di codice web e semplifica la distribuzione rispetto a una web app pura, ma introduce spesso:

- **overhead prestazionale** dovuto al rendering in WebView, con impatto su responsività e animazioni in UI complesse;
- **dipendenza dai plugin** per copertura e qualità delle API native;
- **rischio di incoerenza UX** rispetto alle convenzioni native.

In letteratura, i trade-off prestazionali e qualitativi delle soluzioni ibride e multi-platform sono stati discussi in studi empirici e di posizione [5, 6].

2.2.4 Cross-platform moderni

Accanto a web e ibrido WebView-based, si sono affermati framework cross-platform moderni che puntano a una UI e a prestazioni più vicine al nativo. In modo semplificato, si possono distinguere due grandi famiglie:

- **Native rendering:** l'app descrive la UI in un linguaggio/framework condiviso, ma il rendering finale avviene tramite componenti UI nativi (es. React Native).
- **Custom rendering:** il framework adotta un motore di rendering proprio e disegna i componenti (es. Flutter), riducendo dipendenze dalla toolkit UI nativa ma introducendo un runtime e un rendering engine dedicati.

Queste soluzioni si collocano in un punto intermedio tra il massimo controllo del nativo e la massima portabilità del web: condividono una porzione elevata di codice (logica, UI, networking), mantenendo la possibilità di estensioni native nei casi in cui sono necessarie. Tra i framework riconducibili a queste due famiglie, React Native e Flutter rappresentano due riferimenti particolarmente significativi. React Native adotta un modello di *native rendering*, nel quale la UI viene descritta in JavaScript/TypeScript e resa attraverso componenti nativi della piattaforma. Flutter, invece, si colloca nel paradigma del *custom rendering*: il framework utilizza un motore grafico proprio e disegna direttamente i widget, riducendo la dipendenza dalla toolkit UI nativa. Questa distinzione non è puramente tecnica, ma influenza in modo diretto trade-off quali coerenza con le convenzioni di piattaforma, modalità di accesso alle API native, prevedibilità del rendering e grado di uniformità visiva tra sistemi operativi.

2.3 Cross-platform: definizione e motivazioni

2.3.1 Definizione

Per *cross-platform development* si intende un insieme di tecniche e strumenti che permettono di realizzare applicazioni per più piattaforme target a partire da una codebase in larga misura condivisa. L'obiettivo non è esclusivamente “compilare per più piattaforme”, ma rendere sostenibile l'evoluzione del prodotto: ridurre duplicazioni, accelerare i rilasci e mantenere coerenza funzionale e qualitativa.

La letteratura propone classificazioni e criteri di confronto basati su prestazioni, copertura delle API, manutenibilità e produttività [7, 8]. In questo senso, “cross-platform” è un concetto pragmatico: l'efficacia dipende dall'allineamento tra vincoli del dominio (sicurezza, real-time, OS integration) e caratteristiche del framework scelto.

2.3.2 Motivazioni generali

Le motivazioni più frequenti per adottare il cross-platform includono:

- **Riduzione della duplicazione:** logiche comuni (autenticazione, networking, caching, gestione errori) vengono implementate una sola volta.
- **Allineamento dei rilasci:** feature e bug fix possono essere rilasciati simultaneamente su Android e iOS, riducendo periodi di inconsistenza.
- **Coerenza della UX:** flussi critici (login, onboarding, navigazione) risultano più uniformi.
- **Ottimizzazione delle competenze:** un team può focalizzarsi su un unico stack, mantenendo un perimetro nativo minimo per esigenze OS-level.

Questi vantaggi diventano più rilevanti quando l'app non è un semplice front-end statico, ma include logiche complesse di sicurezza e resilienza: ad esempio gestione token, refresh automatico, riconnessione WebSocket e trattamento uniforme degli errori di rete.

2.3.3 React Native: modello di esecuzione e architettura

Tra i framework cross-platform basati su *native rendering*, React Native si distingue per un principio chiave: consentire lo sviluppo dell'applicazione in JavaScript/TypeScript (seguendo il paradigma dichiarativo di React) producendo al contempo un'interfaccia utente composta da viste native di Android e iOS, evitando l'approccio WebView-based come soluzione primaria [9].

Runtime JavaScript e UI nativa

Il modello concettuale prevede due domini:

- un **runtime JavaScript** che esegue logica applicativa, gestione dello stato e orchestrazione degli eventi;
- un **runtime nativo** che costruisce e gestisce la gerarchia delle viste, input e integrazione con API di piattaforma.

La UI è descritta in forma dichiarativa tramite componenti; le variazioni di stato generano aggiornamenti della UI attraverso un processo di *reconciliation*, che determina quali operazioni applicare al layer nativo. Questo modello rende naturale condividere logica e struttura della UI, mantenendo un aspetto e un comportamento generalmente più vicini alle convenzioni native rispetto alle soluzioni basate su WebView.

Dalla “bridge architecture” alla New Architecture (JSI, Fabric, TurboModules)

Storicamente React Native ha utilizzato un bridge asincrono basato su scambio di messaggi serializzati tra JavaScript e nativo. Sebbene efficace, questo approccio può introdurre overhead e latenza nei passaggi frequenti attraverso il confine JavaScript-native.

La **New Architecture** di React Native mira a ridurre tali costi sostituendo il bridge con la *JavaScript Interface (JSI)*, che abilita un’interoperabilità più diretta tra runtime e riduce la necessità di serializzazione [9, 10]. In questo quadro si inseriscono:

- **Fabric**, un sistema di rendering rinnovato per migliorare scheduling e coerenza tra modello React e applicazione delle operazioni sulla UI nativa [9];
- **TurboModules**, un modello moderno per i moduli nativi con interfacce tipizzate e *lazy loading*, supportato da Codegen [11].

Dal punto di vista ingegneristico, questa evoluzione è significativa perché riduce uno dei trade-off classici del cross-platform: il costo del bridging, particolarmente visibile in scenari con eventi frequenti o aggiornamenti rapidi della UI.

Motore JavaScript: Hermes

React Native supporta diversi motori JavaScript; Hermes è un motore open source ottimizzato per React Native e, in molti scenari, migliora tempi di avvio, uso memoria e dimensione dell’app rispetto a JavaScriptCore [12, 13]. In ambito mobile, tali metriche incidono direttamente sulla percezione dell’utente (tempo per diventare interattiva dopo avvio o ripresa) e sull’efficienza complessiva, specialmente su dispositivi con risorse limitate.

Moduli nativi e principio “shared by default, native when needed”

Un aspetto critico delle soluzioni cross-platform è l’accesso a funzionalità OS-level. React Native consente l’estensione tramite moduli nativi, esposti al runtime JavaScript; nella New Architecture questa estensione è formalizzata tramite TurboModules e interfacce generate [11]. Ne deriva un principio pragmatico adottato frequentemente in progetti reali: massimizzare il codice condiviso e confinare il nativo a funzioni per cui è essenziale (ad esempio integrazione con secure storage o servizi di localizzazione affidabili). Tale impostazione consente di evitare la duplicazione completa del client, mantenendo però la possibilità di soddisfare requisiti non facilmente realizzabili in modo puramente cross-platform.

2.3.4 React Native e Flutter: confronto sintetico

Nel panorama cross-platform contemporaneo, React Native e Flutter rappresentano due approcci maturi ma concettualmente differenti. Entrambi consentono di sviluppare applicazioni Android e iOS a partire da una codebase in larga misura condivisa, ma differiscono nel modello di rendering, nella relazione con la piattaforma ospite e nelle implicazioni ingegneristiche che ne derivano.

React Native si basa su un approccio di *native rendering*: la UI è descritta tramite componenti React, ma il rendering finale avviene attraverso elementi nativi della piattaforma. Questo modello favorisce una maggiore vicinanza alle convenzioni di Android e iOS e rende naturale l'integrazione con moduli nativi quando siano necessarie funzionalità OS-level specifiche. In tale prospettiva, il client cross-platform non sostituisce completamente il livello nativo, ma ne riduce il perimetro secondo il principio *shared by default, native when needed*.

Flutter adotta invece un approccio di *custom rendering*: i widget sono disegnati tramite un motore grafico dedicato e non dipendono direttamente dai componenti UI nativi. Questo consente di ottenere un'elevata uniformità visiva tra piattaforme e un controllo molto stretto sul comportamento della UI. D'altra parte, l'integrazione con il sistema operativo e l'aderenza alle convenzioni native passano attraverso un layer di astrazione ulteriore, che può risultare vantaggioso in applicazioni fortemente centrate sulla coerenza grafica, ma meno allineato a scenari nei quali la relazione con API, lifecycle e servizi di piattaforma ha un peso architetturale rilevante.

Per il progetto Smartotum, i criteri decisivi non riguardavano soltanto la produttività cross-platform, ma anche la compatibilità con requisiti quali: persistenza sicura dei token, gestione del ciclo di vita mobile, interazione con funzionalità native mirate, continuità di sessione, aggiornamenti real-time e integrazione controllata con il backend. In questo contesto, React Native è stato ritenuto più coerente con una strategia che privilegia il riuso della logica applicativa e della maggior parte della UI, mantenendo al tempo stesso un accesso mirato e pragmatico alle capacità native quando necessario.

La Tabella 2.1 sintetizza i principali elementi di confronto.

Il confronto non implica una superiorità assoluta di React Native rispetto a Flutter. Piuttosto, evidenzia come i due framework rispondano in modo diverso a vincoli differenti. Nel caso di Smartotum, la scelta di React Native è risultata preferibile perché meglio allineata a un'applicazione che deve combinare condivisione del codice, accesso selettivo al nativo e integrazione stretta con un backend dedicato.

Tabella 2.1: Confronto sintetico tra React Native e Flutter

Aspetto	React Native	Flutter
Modello di rendering	Rendering tramite componenti UI nativi	Rendering tramite motore grafico proprio
Relazione con la piattaforma	Maggiore vicinanza alle convenzioni native	Maggiore uniformità visiva cross-platform
Accesso a funzionalità OS-level	Estensione tramite moduli nativi dedicati	Estensione tramite plugin e canali verso il codice nativo
Comportamento della UI	Più aderente al look-and-feel della piattaforma	Più controllabile e consistente tra piattaforme
Trade-off principale	Integrazione pragmatica con il nativo	Controllo più completo del rendering
Allineamento con Smartotum	Elevato, per la presenza di requisiti OS-level e real-time	Meno centrale rispetto ai vincoli specifici del progetto

2.4 Confronto con lo sviluppo nativo

2.4.1 Criteri di confronto

Il confronto tra nativo e cross-platform può essere condotto adottando criteri coerenti con qualità e costo totale di proprietà:

- prestazioni e responsività della UI;
- qualità dell'esperienza utente e aderenza alle linee guida di piattaforma;
- copertura e facilità di accesso alle API native;
- produttività e velocità di rilascio;
- manutenibilità (complessità, dipendenze, testabilità);
- rischio tecnologico (maturità framework, ecosistema, breaking changes).

Questi criteri sono allineati a modelli di qualità come ISO/IEC 25010 [1] e a framework di valutazione per strumenti cross-platform [8].

2.4.2 Vantaggi del cross-platform

Riduzione della duplicazione e coerenza Il vantaggio principale è l'alta percentuale di codice riusabile. In un'app con autenticazione e networking complesso, condividere:

- gestione dei token e dei flussi di refresh;
- strategie di retry e trattamento degli errori;
- logica di orchestrazione verso servizi remoti;
- gestione dello stato e caching,

riduce il rischio di comportamenti divergenti tra piattaforme e semplifica la manutenzione.

Time-to-market e rilasci sincronizzati Una codebase unica consente rilasci più allineati tra Android e iOS: bug fix e nuove funzionalità arrivano in modo più uniforme, riducendo finestre temporali in cui solo una piattaforma beneficia di una correzione critica.

Razionalizzazione del team e del processo Un unico stack riduce la necessità di duplicare competenze e procedure su due ecosistemi (Android/iOS) e facilita la standardizzazione di pratiche (code review, criteri di qualità, test). La presenza di moduli nativi mirati non nega il vantaggio: sposta il nativo in un perimetro circoscritto.

2.4.3 Svantaggi e rischi

Overhead e ottimizzazione Un framework cross-platform introduce runtime e livelli di astrazione. Sebbene spesso trascurabile per applicazioni business, l'overhead può diventare rilevante in UI molto complesse o in presenza di aggiornamenti ad alta frequenza. Ciò richiede maggiore attenzione a ottimizzazioni e profiling.

Dipendenza dall'ecosistema Framework e librerie possono introdurre breaking changes, deprecazioni e vincoli di compatibilità con nuove versioni di iOS/Android. La scelta cross-platform deve quindi considerare maturità, supporto della community e roadmap del framework [7].

Bridging per API OS-level Quando l'app richiede funzionalità specifiche (background, localizzazione affidabile, secure storage), possono essere necessari moduli nativi e bridging. Questo introduce complessità aggiuntiva e richiede competenze su entrambe le piattaforme, sebbene in misura ridotta rispetto a una doppia codebase completa.

2.4.4 Tabella di sintesi

Tabella 2.2: Confronto qualitativo tra sviluppo nativo e cross-platform (sintesi).

Criterio	Nativo	Cross-platform
Prestazioni UI	Massime, controllo fine	Buone nella maggioranza dei casi; possibile overhead del runtime
Accesso API OS	Completo, immediato	Ampio ma non totale; moduli nativi per casi avanzati
UX e linee guida	Aderenza naturale	Vicina al nativo; richiede cura per evitare incoerenze
Time-to-market	Più lento su doppia piattaforma	Più rapido con rilascio sincronizzato
Manutenibilità	Due codebase: costi e divergenze	Una codebase principale; attenzione a dipendenze e bridge
Rischio tecnologico	Dipende soprattutto da OS/-SDK	Dipende anche da framework ed ecosistema

2.5 Implicazioni per il progetto

2.5.1 Convergenza da soluzione multi-codebase

Come evidenziato nell'introduzione, la scelta cross-platform mira a convergere verso una sola codebase per l'app mobile, riducendo duplicazioni e divergenze tra piattaforme. In un dominio in evoluzione rapida, questa convergenza è un fattore chiave per mantenere coerenza funzionale e velocità di rilascio.

2.5.2 Ruolo del Backend For Frontend (BFF)

Il client comunica con un *Backend For Frontend* che media l'accesso ai servizi upstream e riduce l'accoppiamento tra app e backend. Il BFF consente un'API più stabile e concentra policy di sicurezza e integrazione [14]. I dettagli architetturali sono approfonditi nei capitoli successivi.

2.5.3 Home Assistant e requisiti real-time

L'integrazione con Home Assistant richiede aggiornamenti real-time dello stato e invocazione di azioni. Questo impone un canale persistente e meccanismi di riconnessione; la codebase unica aiuta a mantenere uniforme la logica di resilienza tra piattaforme.

2.5.4 Sicurezza e token: integrazione tra scelte client e scelte server

La gestione dei token e' un punto di contatto cruciale tra mobile e backend: il cross-platform consente logiche uniformi di sessione ed error handling, mentre i requisiti OS-level (secure storage, location) restano confinati in moduli nativi. I dettagli di sicurezza, rotazione/refresh e responsabilita' tra client e BFF sono formalizzati nel capitolo dei requisiti e approfonditi nei capitoli di architettura e integrazione. Nel progetto, la persistenza dei token e' demandata a storage nativo sicuro: Keychain su iOS e Encrypted Shared Preferences su Android, in modo da ridurre l'esposizione delle credenziali e rispettare le best practice delle piattaforme [15, 16].

Capitolo 3

Analisi del problema e requisiti

3.1 Inquadramento del capitolo

Il presente capitolo formalizza i requisiti del sistema e i principali casi d'uso che ne guidano la progettazione. Il contesto e le motivazioni generali sono già stati introdotti nei capitoli precedenti; qui l'attenzione è posta su *cosa* il sistema deve garantire e su quali vincoli operativi devono essere rispettati.

La formalizzazione dei requisiti è organizzata in due categorie:

- **Requisiti funzionali (RF):** funzionalità osservabili dall'utente o da sistemi esterni, esprimibili tramite casi d'uso.
- **Requisiti non funzionali (RNF):** proprietà trasversali di sicurezza, resilienza, manutenibilità e controllo del canale real-time.

Le scelte architetturali che implementano tali requisiti (separazione delle responsabilità tra app, BFF e servizi upstream) saranno approfondite nel capitolo successivo.

3.2 Attori, confini del sistema e vincoli

3.2.1 Attori

Gli attori rilevanti per la definizione dei requisiti sono:

- **Utente finale:** interagisce con la mobile app per autenticarsi, selezionare un impianto/sistema, consultare stati e inviare comandi.

- **App mobile:** client cross-platform che gestisce la UI e l'orchestrazione delle richieste verso il BFF; include un *sottile strato nativo* per funzioni OS-level.
- **Backend for Frontend (BFF):** punto di accesso unico per il client, responsabile di autenticazione, sessione e mediazione verso servizi upstream.
- **Servizi upstream: Smartotum API e Home Assistant,** considerati esterni al sistema e raggiunti tramite contratti di integrazione (REST e/o WebSocket).
- **Servizi di notifica:** canale push per notifiche device-side (es. Firebase/FCM), usato in alcuni flussi applicativi.

3.2.2 Confini e responsabilità

Ai fini della tesi, il sistema comprende **app mobile** e **BFF**; Smartotum API e Home Assistant sono dipendenze esterne. I requisiti vengono formulati senza assumere dettagli interni di tali servizi, ma solo le loro capacità (autenticazione, endpoint, stream eventi). La ripartizione delle responsabilità tra app e BFF segue il principio di boundary già illustrato nei capitoli precedenti.

3.2.3 Vincoli operativi

Dalla natura mobile e dal dominio smart home derivano vincoli che influenzano direttamente requisiti e casi d'uso:

- **Rete intermittente:** timeout, retry e riconnessioni devono essere considerati parte del comportamento atteso.
- **Ciclo di vita mobile:** backgrounding e ripresa possono interrompere sessioni e canali real-time; è richiesto ripristino rapido.
- **Real-time:** lo stato dei dispositivi deve aggiornarsi in modo reattivo; il canale WS richiede controlli dedicati (heartbeat, close codes, limiti).
- **Sicurezza credenziali:** i token dell'app devono essere protetti a riposo in storage nativo; lato server i token upstream devono essere protetti a riposo (cifatura) e i token BFF non devono essere memorizzati in chiaro.
- **Location OS-level:** il tracking posizione e l'invio della posizione (anche in background) richiedono integrazione nativa e gestione robusta della sessione associata.

3.3 Casi d'uso principali

I casi d'uso descrivono l'interazione tra attori e sistema; sono utili sia per derivare requisiti funzionali, sia per chiarire le condizioni di errore e le precondizioni operative.

3.3.1 Vista d'insieme

La Tabella 3.1 elenca i casi d'uso principali raggruppati per area. I flussi dettagliati (alternative, errori e postcondizioni) vengono ripresi in forma sintetica nei paragrafi successivi, mentre i dettagli implementativi sono rimandati ai capitoli dedicati ad architettura e integrazione.

Tabella 3.1: Casi d'uso principali del sistema (sintesi).

UC-ID	Descrizione	Area
UC-01	Login utente	Auth/Sessione
UC-02	Refresh token BFF e retry della richiesta	Auth/Sessione
UC-03	Selezione e cambio sistema attivo	App/Core
UC-04	Connessione e sottoscrizione alle entità via WebSocket (/ha/stream)	HA Real-time
UC-05	Invio comandi verso Home Assistant su canale real-time	HA Real-time
UC-06	Riconnessione WebSocket con backoff e gestione scadenza sessione	HA Real-time
UC-07	Consultazione statistiche/indicatori Home Assistant via REST	HA REST
UC-08	Ottenimento URL stream camera	HA REST
UC-09	Registrazione e gestione notifiche push (device ↔ BFF/HA)	Push
UC-10	Invio location (anche in background) e tag event verso Home Assistant	Location
UC-11	Gestione schedule/programma temperature (CRUD e attivazione)	Smartotum
UC-12	Gestione automazioni e scene (creazione, modifica, duplicazione, abilitazione)	Smartotum
UC-13	Gestione sicurezza: codici allarme, sensori e ritardi (CRUD/attivazione)	Smartotum
UC-14	Logout	Auth/Sessione

3.3.2 UC-01: login e creazione sessione BFF

Attori: Utente, App, Smartotum API, BFF.

Scopo: ottenere una sessione applicativa valida e i token necessari al funzionamento dell'app.

Flusso principale (sintesi):

1. L'utente inserisce le credenziali.
2. L'app ottiene token upstream (Smartotum).
3. L'app richiede al BFF la creazione di una sessione applicativa.
4. Il BFF conserva in modo sicuro i token upstream e rilascia token BFF (access/refresh).
5. L'app salva i token BFF in **storage nativo** associandoli al sistema selezionato.

3.3.3 UC-02: refresh token e continuità della sessione

Attori: App, BFF.

Scopo: garantire continuità delle operazioni protette quando l'access token scade.

Flusso principale (sintesi):

1. Una richiesta protetta fallisce con errore di autorizzazione.
2. L'app avvia un refresh *single-flight* (evitando refresh concorrenti).
3. Il BFF ruota la coppia access/refresh.
4. L'app aggiorna i token in storage nativo e ritenta la richiesta.

3.3.4 UC-03: selezione e cambio sistema attivo

Attori: Utente, App mobile, BFF.

Scopo: consentire all'utente di selezionare (o cambiare) il sistema attivo e garantire che il contesto applicativo rimanga coerente con la selezione.

Precondizioni:

- Utente autenticato e sessione applicativa valida.
- Lista dei sistemi disponibile.

Flusso principale (sintesi):

1. L'utente seleziona uno o più sistemi e sceglie quello da rendere attivo.

2. L'app aggiorna il `systemId` attivo e salva localmente la lista sistemi associata.
3. L'app sincronizza il cambio contesto aggiornando i token/contesto Smartotum sul BFF per il nuovo sistema.

Alternative/Errore:

- Se il salvataggio o l'aggiornamento del contesto fallisce, l'app ripristina il sistema precedente e notifica l'errore.

Postcondizioni:

- `systemId` attivo aggiornato.
- Token e contesto applicativo coerenti con il sistema selezionato.

3.3.5 UC-04/UC-05/UC-06: real-time e comandi Home Assistant

Attori: App, BFF, Home Assistant.

Scopo: mantenere lo stato delle entità aggiornato e consentire azioni operative.

Flusso principale (sintesi):

- L'app apre una connessione WebSocket autenticata verso il BFF e sottoscrive lo stream delle entità (UC-04).
- Il canale invia una snapshot iniziale e aggiornamenti incrementali (eventi).
- L'utente invia comandi (service call) che vengono inoltrati a Home Assistant (UC-05).
- In caso di disconnessione o sessione scaduta, l'app riconnette con backoff ed eventualmente esegue refresh prima di ristabilire lo stream (UC-06).

Nota: data la natura persistente del canale, la gestione delle disconnessioni e dei limiti di messaggistica è trattata anche come requisito non funzionale.

3.3.6 UC-07: recupero statistiche da Home Assistant (REST)

Attori: App mobile, BFF, Home Assistant.

Scopo: ottenere serie temporali/statistiche relative a una o più entità di Home Assistant tramite chiamata REST mediata dal BFF.

Precondizioni:

- Access token BFF valido.

- Parametri di richiesta disponibili (insieme di `entityIds` e intervallo/periodo).

Flusso principale (sintesi):

1. L'app invia una richiesta `GET /ha/stats` specificando `entityIds` e periodo.
2. Il BFF interroga Home Assistant e, se necessario, aggrega/normalizza i dati restituiti.
3. L'app riceve la serie temporale e la rende disponibile per la visualizzazione.

Postcondizioni:

- Dati statistici disponibili nel client.

3.3.7 UC-08: ottenere URL stream camera (REST)

Attori: App mobile, BFF, Home Assistant.

Scopo: richiedere l'URL di streaming associato a una camera esposta come entità di Home Assistant.

Precondizioni:

- Access token BFF valido.
- Identificativo dell'entità camera disponibile (`entity_id`).

Flusso principale (sintesi):

1. L'app invia una richiesta `POST /ha/camera` includendo `entity_id`.
2. Il BFF richiede a Home Assistant l'URL di streaming corrispondente.
3. L'app riceve l'URL e lo utilizza per la visualizzazione dello stream.

Postcondizioni:

- URL dello stream disponibile lato app.

3.3.8 UC-09: registrazione push per notifiche Home Assistant

Attori: App mobile, BFF, Home Assistant, Firebase/FCM.

Scopo: registrare il dispositivo per ricevere notifiche push relative a Home Assistant, mantenendo un mapping server-side tra device e token push.

Precondizioni:

- Token push disponibile sul dispositivo (FCM).

- Access token BFF valido.

Flusso principale (sintesi):

1. L'app recupera il token FCM dal provider di notifiche.
2. L'app invia POST `/ha/push/register` al BFF con il token push e i metadati necessari.
3. Il BFF registra il device sul lato Home Assistant (o sul meccanismo previsto) e salva il mapping in database.

Alternative/Errore:

- Token push mancante: errore di validazione e registrazione non completata.

Postcondizioni:

- Dispositivo registrato per ricevere notifiche push e mapping persistito lato server.

3.3.9 UC-10: invio posizione e vincoli OS-level

Attori: App (modulo nativo), BFF, Home Assistant.

Scopo: inviare la posizione (e/o tag event) verso Home Assistant in modo affidabile anche quando l'app non è in foreground.

Flusso principale (sintesi):

1. Il modulo nativo acquisisce aggiornamenti di posizione secondo le policy OS.
2. L'app invia la posizione verso il BFF, che la inoltra a Home Assistant.
3. Se la sessione risulta non valida, il sistema deve tentare refresh e retry, nel rispetto dei vincoli del background.

Osservazione: questo caso d'uso giustifica il mantenimento di un perimetro nativo per funzioni OS-level, senza reintrodurre una duplicazione completa dell'app.

3.3.10 UC-11: gestione schedule/programma temperature

Attori: Utente, App mobile, BFF, Smartotum API.

Scopo: consentire la gestione dei programmi di temperatura (schedule), includendo creazione, modifica, eliminazione e attivazione/disattivazione, garantendo coerenza con il sistema attivo e con la sessione applicativa.

Precondizioni:

- Utente autenticato e sessione BFF valida.
- Sistema attivo selezionato (UC-03).

Flusso principale (sintesi):

1. L'utente accede alla sezione schedule e visualizza l'elenco dei programmi disponibili.
2. L'utente crea un nuovo programma o seleziona un programma esistente per modificarlo.
3. L'app invia al BFF le operazioni richieste (creazione/modifica/eliminazione/attivazione).
4. Il BFF inoltra la richiesta a Smartotum API utilizzando i token upstream associati all'utente e al sistema attivo.
5. L'app riceve esito e dati aggiornati e aggiorna la UI.

Postcondizioni:

- Programma creato/modificato/eliminato secondo l'azione richiesta.
- Stato di attivazione/disattivazione coerente con la selezione dell'utente.

3.3.11 UC-12: gestione automazioni e scene

Attori: Utente, App mobile, BFF, Smartotum API.

Scopo: gestire le automazioni del sistema (creazione, modifica, attivazione/disattivazione), includendo operazioni di duplicazione e l'abilitazione di scene, garantendo coerenza con il sistema attivo.

Precondizioni:

- Utente autenticato e sessione BFF valida.
- Sistema attivo selezionato (UC-03).

Flusso principale (sintesi):

1. L'utente visualizza l'elenco delle automazioni e il loro stato (abilitata/disabilitata).
2. L'utente crea una nuova automazione o seleziona un'automazione esistente per modificarla.
3. L'utente può duplicare un'automazione esistente per creare rapidamente una variante.

4. L'utente abilita/disabilita automazioni o scene associate, in base alle esigenze operative.
5. L'app invia le operazioni al BFF, che le inoltra a Smartotum API e restituisce l'esito.

Postcondizioni:

- Automazioni/scenes create o aggiornate; stato abilitato/disabilitato coerente.
- Duplicazioni effettuate con creazione di una nuova automazione derivata.

3.3.12 UC-13: gestione sicurezza (codici allarme, sensori e ritardi)

Attori: Utente, App mobile, BFF, Smartotum API.

Scopo: consentire la gestione delle funzionalità di sicurezza del sistema, includendo codici di allarme/accesso, abilitazione/disabilitazione sensori e configurazione dei ritardi dell'allarme, garantendo coerenza con il sistema attivo e con la sessione applicativa.

Precondizioni:

- Utente autenticato e sessione BFF valida.
- Sistema attivo selezionato (UC-03).
- Permessi utente adeguati alle operazioni di sicurezza (ove previsti dal servizio).

Flusso principale (sintesi):

1. L'utente accede alla sezione sicurezza e visualizza codici, sensori e configurazioni disponibili.
2. L'utente esegue una o più operazioni:
 - creazione/rimozione di un codice di sicurezza;
 - attivazione/disattivazione di sensori;
 - impostazione dei ritardi di ingresso/uscita (delays) dell'allarme.
3. L'app invia la richiesta al BFF, includendo i dati necessari all'operazione.
4. Il BFF inoltra la richiesta a Smartotum API utilizzando i token upstream associati all'utente e al sistema attivo.
5. L'app riceve l'esito e aggiorna la UI con lo stato corrente.

Postcondizioni:

- Codici/sensori/ritardi aggiornati coerentemente con l'operazione richiesta.
- Stato di sicurezza persistito lato servizio e visibile nelle successive consultazioni.

3.3.13 UC-14: logout e revoca sessione

Attori: Utente, App mobile, BFF.

Scopo: terminare la sessione dell'utente in modo esplicito, revocando la sessione lato BFF e rimuovendo le credenziali dal dispositivo.

Precondizioni:

- Utente autenticato (oppure sessione localmente presente).

Flusso principale (sintesi):

1. L'app invia `POST /auth/logout` includendo l'access token.
2. Il BFF revoca la sessione (marcandola come non valida).
3. L'app cancella i token dallo storage nativo e pulisce lo stato locale associato.

Postcondizioni:

- Sessione revocata lato server (se esistente) e token rimossi dal dispositivo.

3.4 Requisiti funzionali

I requisiti funzionali sono derivati dai casi d'uso e dall'interfaccia esposta dal BFF. Per chiarezza, sono raggruppati per area.

3.4.1 Autenticazione e sessione

- **RF-01** L'app deve consentire l'autenticazione dell'utente e l'avvio di una sessione applicativa.
- **RF-02** Il sistema deve creare una sessione BFF con token access/refresh.
- **RF-03** I token BFF devono essere persistiti sul dispositivo in **storage nativo sicuro**, con supporto al contesto multi-sistema.
- **RF-04** Il sistema deve gestire refresh e rotazione dei token BFF, con retry automatico delle richieste fallite.
- **RF-05** Il sistema deve supportare logout con revoca della sessione e pulizia delle credenziali locali.

3.4.2 Gestione sistemi e funzionalità Smartotum

- **RF-06** L'utente deve poter ottenere la lista sistemi disponibili e selezionare/-cambiare il sistema attivo.
- **RF-07** Il BFF deve gestire la persistenza sicura dei token Smartotum necessari alle chiamate.
- **RF-08** Il sistema deve esporre funzionalità Smartotum per lettura e modifica di informazioni operative (sistemi, parametri casa, automazioni e operazioni correlate).
- **RF-09** Il sistema deve supportare operazioni operative tipiche (ad esempio gestione switch, temperature, override, schedulazioni) in modo coerente con la sessione BFF.
- **RF-10** Il sistema deve consentire la consultazione e gestione di notifiche (lettura, risorse correlate, cancellazione).
- **RF-11** Il sistema deve consentire la gestione dei programmi di temperatura (schedule), includendo creazione, modifica, eliminazione e attivazione/disattivazione, in coerenza con il sistema attivo e la sessione BFF.
- **RF-12** Il sistema deve consentire la gestione delle automazioni, includendo creazione, modifica, duplicazione e attivazione/disattivazione, nonché l'abilitazione di scene dove supportato.
- **RF-13** Il sistema deve supportare la gestione delle funzionalità di sicurezza (codici di allarme/accesso, attivazione/disattivazione sensori e configurazione dei ritardi), in coerenza con il sistema attivo e la sessione BFF.

3.4.3 Integrazione Home Assistant

- **RF-14** Il sistema deve consentire una connessione real-time autenticata (WebSocket) mediata dal BFF.
- **RF-15** Il sistema deve supportare sottoscrizione allo stream entità e ricezione aggiornamenti incrementali.
- **RF-16** Il sistema deve consentire invio di comandi (service call) verso Home Assistant attraverso il canale previsto.
- **RF-17** Il sistema deve gestire i token Home Assistant lato backend senza esporre credenziali upstream all'app oltre quanto strettamente necessario.
- **RF-18** Il sistema deve esporre endpoint REST per funzionalità complementari (ad esempio statistiche e risorse multimediali, ove disponibili).

3.4.4 Push e location

- **RF-19** Il sistema deve consentire registrazione e gestione di notifiche push e badge (quando applicabile).
- **RF-20** Il sistema deve consentire invio della posizione (anche in background) tramite moduli nativi, garantendo robustezza in presenza di scadenze token.
- **RF-21** Il sistema deve supportare l'inoltro di notifiche/eventi verso il canale push previsto.

3.5 Requisiti non funzionali

I requisiti non funzionali definiscono proprietà trasversali che determinano l'affidabilità del sistema in scenari reali (rete instabile, background, real-time). Sono raggruppati per categoria.

3.5.1 Sicurezza

- **RNF-01 (Token a riposo lato client)** I token devono essere conservati in **storage nativo sicuro** su iOS/Android, evitando persistenze non protette.
- **RNF-02 (Token upstream a riposo lato server)** I token verso servizi upstream devono essere cifrati a riposo nel database del BFF.
- **RNF-03 (Token BFF non in chiaro)** I token BFF non devono essere memorizzati in chiaro lato server (uso di hash).
- **RNF-04 (Protezione endpoint)** Gli endpoint del BFF devono applicare autenticazione obbligatoria sulle risorse protette.
- **RNF-05 (TTL e revoca)** Le sessioni devono avere scadenze definite e supportare revoca esplicita.

3.5.2 Resilienza e continuità operativa

- **RNF-06 (Resilienza WS)** Il canale real-time deve includere meccanismi di heartbeat e riconnessione per mitigare disconnessioni e cambi rete.
- **RNF-07 (Rate limiting WS)** Il canale real-time deve applicare limiti di messaggistica per prevenire overload o abusi.
- **RNF-08 (Timeout e abort)** Le richieste di rete devono avere timeout e meccanismi di abort per evitare blocchi indefiniti.

- **RNF-09 (Gestione errori coerente)** Il sistema deve normalizzare e mappare gli errori in modo coerente tra componenti (client/BFF), facilitando la gestione e la diagnosi.

3.5.3 Osservabilità e manutenibilità

- **RNF-10 (Logging)** Il sistema deve fornire osservabilità di base tramite logging lato server e, dove utile, lato client.
- **RNF-11 (Compatibilità moduli nativi)** Il client deve mantenere compatibilità con moduli nativi dedicati (location e token storage), senza reintrodurre una doppia codebase completa.
- **RNF-12 (Graceful shutdown)** Il server deve supportare chiusura ordinata e pulizia delle connessioni, in particolare per il canale WebSocket.

3.5.4 Requisiti quantitativi

Nel progetto non sono formalizzati SLA quantitativi (ad esempio latenza massima in millisecondi o percentuali di uptime). In questa tesi tali aspetti vengono quindi trattati come obiettivi qualitativi (reattività percepita, robustezza alle disconnessioni) e verranno discussi in termini di strategie progettuali e validazione nei capitoli successivi.

3.6 Artefatti di supporto: interfacce e persistenza

Questa sezione raccoglie elementi utili a contestualizzare i requisiti, senza entrare nei dettagli implementativi.

3.6.1 Interfacce principali del BFF

Dal punto di vista del contratto con il client, le interfacce del BFF si articolano in:

- **Sessione/Auth:** login, refresh, logout.
- **Smartotum:** risorse e operazioni legate al sistema/impianto e alle funzioni applicative.
- **Home Assistant REST:** risorse informative (ad esempio statistiche, metadati e risorse multimediali).
- **Home Assistant WS:** endpoint di stream per entità e service call.

La descrizione dettagliata degli endpoint e delle scelte di protezione sarà affrontata nel capitolo di architettura e in quello di integrazione.

3.6.2 Persistenza lato BFF: entità logiche

La persistenza lato BFF supporta in particolare requisiti di sicurezza e sessione. In termini logici, il modello include:

- una rappresentazione dell'**utente** (identità logica);
- una o più **sessioni** associate all'utente, con scadenza e revoca;
- contenitori per **token upstream** (Smartotum e Home Assistant) protetti a riposo;
- una struttura per **push** (associazione tra device e token di notifica).

La modellazione concreta e la gestione delle chiavi/strategie crittografiche verranno approfondite nei capitoli successivi.

Capitolo 4

Architettura e progettazione della soluzione

4.1 Visione d'insieme e principi architetturali

L'architettura adottata separa in modo netto il canale mobile dalle complessità di integrazione e sicurezza, distribuendo responsabilità in tre livelli:

- **App mobile (React Native + moduli nativi mirati)**: interfaccia utente, stato applicativo e orchestrazione delle chiamate; conserva i token BFF in storage nativo sicuro, e include componenti nativi per funzioni OS-level (tracking posizione e gestione robusta della sessione in background).
- **Backend for Frontend (BFF) in Express/TypeScript**: punto di ingresso unico per il client; espone API REST e un canale WebSocket; gestisce sessioni e token BFF; conserva in modo sicuro i token dei servizi esterni, curando il refresh e la mediazione verso Smartotum API e Home Assistant.
- **Servizi esterni**: Smartotum API (HTTP), Home Assistant (HTTP e WebSocket), e Firebase/FCM per la consegna delle notifiche push.

Il disegno complessivo applica il principio di *boundary* tra client e servizi esterni: il client utilizza esclusivamente credenziali BFF con privilegi e durata controllati, mentre il BFF custodisce le credenziali dei servizi esterni, normalizza errori e riduce l'accoppiamento con i dettagli di integrazione.

4.1.1 Contratti e canali di comunicazione

La comunicazione tra app e BFF avviene tramite due canali distinti:

- **REST** per operazioni applicative (autenticazione, funzioni Smartotum, query a Home Assistant, gestione push e location).
- **WebSocket** per requisiti real-time (stream dello stato entità Home Assistant e invocazioni di servizi).

Questa separazione consente di trattare il canale real-time come un componente con policy proprie (heartbeat, rate limiting, backoff), senza vincolare la semantica delle API REST.

4.2 Scomposizione in componenti

La soluzione è organizzata in componenti coerenti con le responsabilità descritte sopra. Di seguito si sintetizzano i principali moduli lato client e lato server, evidenziando la logica di separazione interna.

4.2.1 Componenti lato client

L'app React Native è strutturata secondo una suddivisione tra UI, gestione stato/sessione, networking e real-time:

- **UI e navigazione:** schermate e componenti riusabili; gestione del flusso applicativo (bootstrap, login, selezione sistema, area principale).
- **Contesto di autenticazione e sessione:** mantiene le informazioni minime per l'esperienza utente (utente, sistema attivo, stato login) e coordina l'accesso al layer API.
- **Layer API:** incapsula le chiamate REST al BFF, applica timeout e mapping errori, e implementa il comportamento di refresh token con retry automatico.
- **Client WebSocket Home Assistant:** gestisce connessione a `/ha/stream`, subscribe allo stream, ping/pong, backoff e resubscribe.
- **Integrazione nativa mirata:** storage sicuro dei token BFF per `systemId` e bridge dedicati a location/background per garantire continuità operativa.

Storage sicuro lato dispositivo

La persistenza dei token BFF avviene tramite storage nativo sicuro, con separazione per `systemId`. Tale scelta evita di memorizzare credenziali in store non protetti e permette di integrare correttamente i vincoli di piattaforma (Keychain su iOS, storage cifrato/Keystore-backed su Android). In aggiunta, la presenza di un bridge

nativo per il refresh in scenari background (in particolare legati alla location) consente di gestire correttamente richieste effettuate fuori dal ciclo di vita tipico della UI, mantenendo un coordinamento con il single-flight di React Native.

4.2.2 Componenti lato BFF

Il BFF espone un set di interfacce coerenti con le aree funzionali e include moduli trasversali di sicurezza, persistenza e integrazione:

- **Routing REST:** endpoint di autenticazione, proxy/aggregazione per Smartotum, endpoint dedicati a Home Assistant (REST), e gestione push/location.
- **Gestione sessioni BFF:** creazione sessione, verifica access token, rotazione refresh token, revoca sessione.
- **Vault crittografico:** cifratura a riposo dei token dei servizi esterni (Smartotum e Home Assistant) tramite schema AES-GCM e `key_id`.
- **Integrazione Home Assistant:** componente server-side che mantiene la connessione verso Home Assistant e fornisce snapshot+delta e invocazioni di servizi.
- **WebSocket manager:** accettazione client `/ha/stream`, autenticazione, rate limiting, heartbeat, gestione close codes.
- **Persistenza (Prisma):** schema dati per utente, sessioni, token esterni cifrati e mapping per notifiche push.

4.3 Razionale delle scelte architetturali

Le scelte architetturali adottate mirano a ridurre la frammentazione della soluzione e a rendere più controllabili sicurezza, resilienza e integrazione con servizi eterogenei. In un dominio smart home, infatti, la complessità non deriva soltanto dalla UI, ma soprattutto dalla necessità di orchestrare più canali di comunicazione (REST e real-time), gestire credenziali sensibili, e garantire continuità operativa anche in presenza di rete instabile e vincoli del ciclo di vita mobile.

4.3.1 BFF come boundary tra mobile e servizi esterni

L'introduzione di un Backend for Frontend costituisce una decisione centrale: invece di collegare direttamente l'app ai servizi esterni, il BFF definisce un confine applicativo che stabilizza l'interfaccia verso il client e concentra la logica di integrazione. In particolare, il BFF:

- riduce l'accoppiamento del client rispetto a dettagli e variazioni degli upstream (contratti, errori, timeouts);
- consente di applicare policy trasversali (sicurezza, normalizzazione errori, controllo delle richieste) in un unico punto;
- rende più semplice mantenere coerenti i comportamenti tra piattaforme, evitando duplicazioni (in passato tipiche di codebase separate).

Dal punto di vista progettuale, il BFF non sostituisce i servizi esterni, ma ne *media* l'accesso: l'app interagisce con un contratto coerente e orientato al mobile, mentre la complessità di orchestrazione e protezione delle credenziali è confinata lato server.

4.3.2 Token upstream lato server e minimizzazione delle credenziali sul client

Una scelta ulteriore è la minimizzazione delle credenziali necessarie sul dispositivo. L'app utilizza token BFF a privilegi e durata controllati, memorizzati in storage nativo sicuro; i token verso Smartotum e Home Assistant sono invece custoditi dal BFF. Questa separazione consente di:

- ridurre l'esposizione delle credenziali upstream (più sensibili) al perimetro del dispositivo;
- centralizzare il rinnovo dei token upstream e la gestione delle relative anomalie;
- consentire revoca e controllo di sessione in modo uniforme.

La scelta è particolarmente rilevante in scenari di perdita del device o compromissione dell'ambiente di esecuzione: pur non eliminando il rischio, la superficie di credenziali disponibili localmente viene ridotta al minimo necessario.

4.3.3 Canale real-time mediato: WebSocket verso BFF invece che verso Home Assistant

Per soddisfare requisiti real-time, una possibile alternativa sarebbe connettere l'app direttamente al WebSocket di Home Assistant. Tuttavia, l'approccio adottato privilegia un WebSocket mediato dal BFF, con due benefici principali:

- **controllo di sicurezza:** l'app non deve gestire direttamente credenziali Home Assistant né conoscere dettagli del protocollo o delle modalità di autenticazione upstream;

- **controllo operativo:** il BFF può applicare limiti e policy dedicate al canale (gestione riconnessione, controllo del traffico, regole di sostituzione connessioni), migliorando prevedibilità e diagnosticabilità.

Inoltre, il BFF consente di fornire al client un protocollo applicativo più semplice e orientato ai casi d'uso (subscribe, aggiornamenti, invocazioni di servizi), evitando che il client debba adattarsi ad evoluzioni o complessità del protocollo upstream.

4.3.4 React Native con layer nativo mirato

Dal punto di vista del client, la scelta di React Native è motivata dalla necessità di ricondurre l'applicazione a una singola codebase cross-platform, riducendo duplicazioni e divergenze tra implementazioni. Al contempo, alcune funzionalità non risultano adatte a essere gestite esclusivamente nel runtime JavaScript:

- persistenza sicura delle credenziali (integrazione con meccanismi nativi di secure storage);
- tracking della posizione con vincoli OS-level e possibili esecuzioni in background.

Per questa ragione, l'app mantiene un *sottile* strato nativo mirato, evitando però di reintrodurre una duplicazione completa dell'applicazione. In termini architetturali, si tratta di un compromesso che conserva i vantaggi del cross-platform senza rinunciare alle garanzie necessarie in ambiti specifici.

4.4 Allineamento requisiti–architettura

La progettazione è guidata dai requisiti funzionali e non funzionali formalizzati nel capitolo precedente. La Tabella 4.1 mostra una mappatura sintetica tra requisiti rilevanti e componenti architetturali che concorrono a soddisfarli. La tabella non intende esaurire i dettagli implementativi, ma chiarire la corrispondenza tra *vincoli* e *scelte di struttura*.

Tabella 4.1: Mappatura sintetica tra requisiti e componenti architetturali.

Requisito	Obiettivo	Componenti coinvolti
RNF-01	Protezione token a riposo sul dispositivo	Secure storage nativo; contesto sessione client
RNF-02	Protezione token upstream a riposo sul server	Vault crittografico; persistenza BFF
RNF-03	Evitare token BFF in chiaro lato server	Gestione sessione BFF; persistenza sessioni
RNF-06	Resilienza del canale real-time	WS client; WS manager; policy di reconnect/backoff
RNF-07	Limitazione del traffico WS	WS manager; rate limiting applicativo
RNF-08	Evitare blocchi indefiniti su rete instabile	Layer API client; timeout lato BFF verso upstream
RNF-09	Gestione errori coerente e diagnosticabile	Layer API client; normalizzazione errori nel BFF
RF-02/RF-04	Sessione BFF con refresh e retry	Session manager BFF; contesto auth client
RF-11/RF-12/RF-13	Stream entità e comandi real-time	WS /ha/stream; integrazione HA server-side; WS client
RF-17	Invio location anche in background	Moduli nativi; orchestrazione sessione/refresh

4.5 Flussi informativi e confini di persistenza (vista concettuale)

Per chiarire le responsabilità senza anticipare dettagli implementativi, è utile distinguere i flussi informativi tra componenti e la natura dei dati trattati (*persistenti* vs *transienti*). In generale:

- Il canale **REST** veicola richieste/risposte puntuali (autenticazione, operazioni Smartotum, interrogazioni Home Assistant, gestione push e invio location).
- Il canale **WebSocket** veicola uno stream continuo di aggiornamenti (stato entità) e invocazioni di servizi real-time.

4.5.1 Dati persistiti sul dispositivo

L'app mantiene localmente un insieme minimo di dati necessari a garantire continuità dell'esperienza utente, con particolare attenzione alla sicurezza:

- token di sessione del BFF (access/refresh), conservati in storage nativo sicuro;
- metadati di contesto (ad esempio `systemId` attivo e informazioni necessarie al bootstrap della UI).

I dati relativi allo stream real-time (snapshot e delta delle entità) non vengono tipicamente persistiti come fonte primaria di verità, ma trattati come stato applicativo aggiornabile.

4.5.2 Dati persistiti sul server BFF

Il BFF persiste ciò che è necessario per la sicurezza e la gestione delle integrazioni:

- sessioni BFF (con scadenze e revoca) e identificativi associati;
- credenziali verso Smartotum e Home Assistant protette a riposo;
- mapping dei dispositivi per le notifiche push.

Al contrario, lo stato delle entità Home Assistant trasmesso sul canale WebSocket è trattato come dato *transiente* per inoltro e caching a breve termine, evitando di duplicare nel DB informazioni già gestite dai sistemi esterni.

4.5.3 Dati transienti e gestione degli eventi

Lo stream real-time è, per natura, un flusso di eventi: il sistema privilegia una gestione *event-driven* in cui l'app riceve una snapshot iniziale seguita da aggiornamenti incrementali. La persistenza è riservata alle credenziali e ai metadati necessari a garantire autenticazione, continuità e riconnessione, mentre il contenuto informativo dello stream (stati e attributi delle entità) rimane transiente e aggiornabile.

4.6 Diagrammi di architettura

In questa sezione si riportano i diagrammi utilizzati per rappresentare la soluzione. I diagrammi sono espressi in sintassi Mermaid, utile per mantenere le figure versionabili insieme al codice.

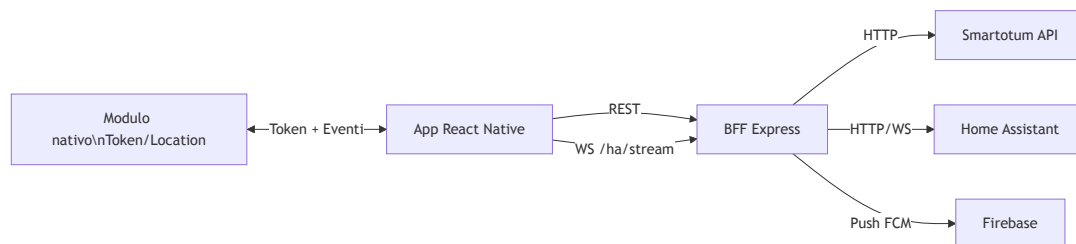


Figura 4.1: Architettura a livelli

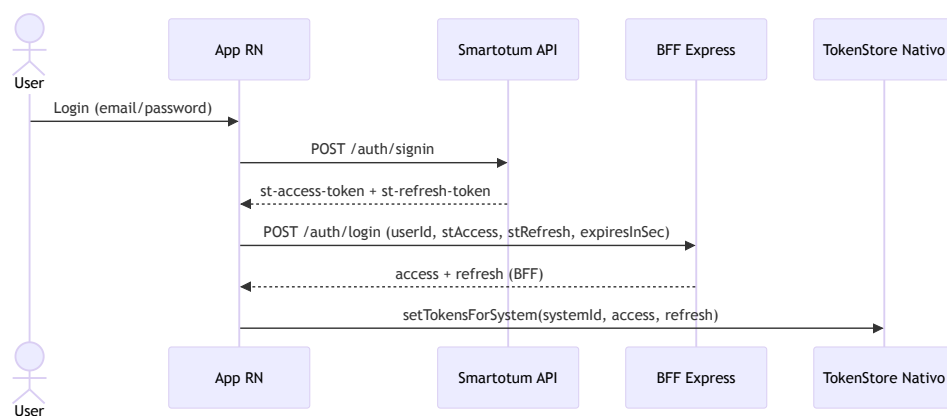


Figura 4.2: Sequenza: login e salvataggio token in storage nativo

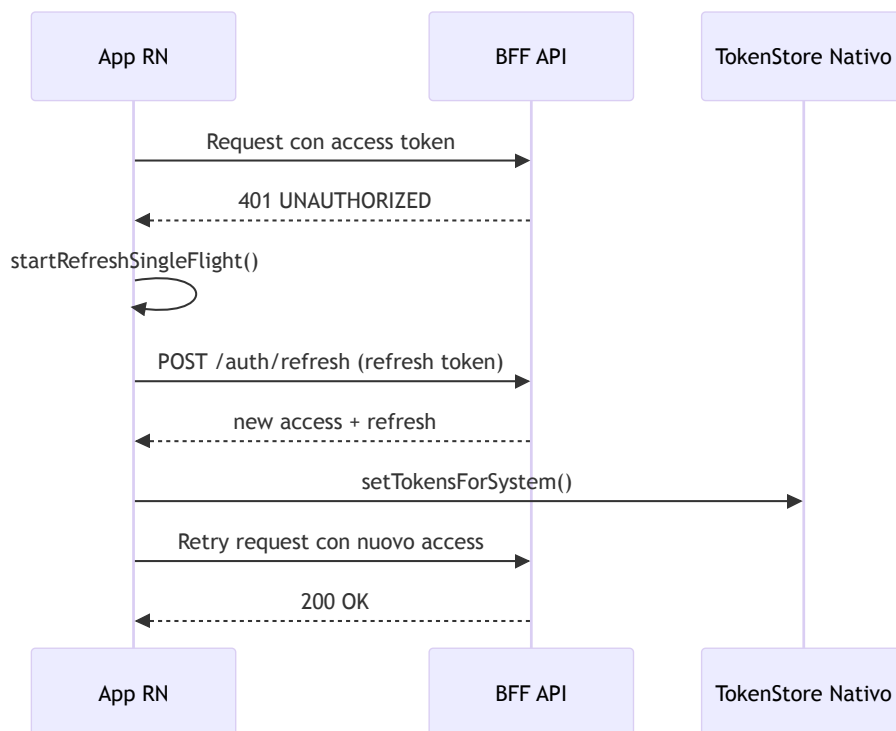


Figura 4.3: Sequenza: richiesta protetta, refresh e retry

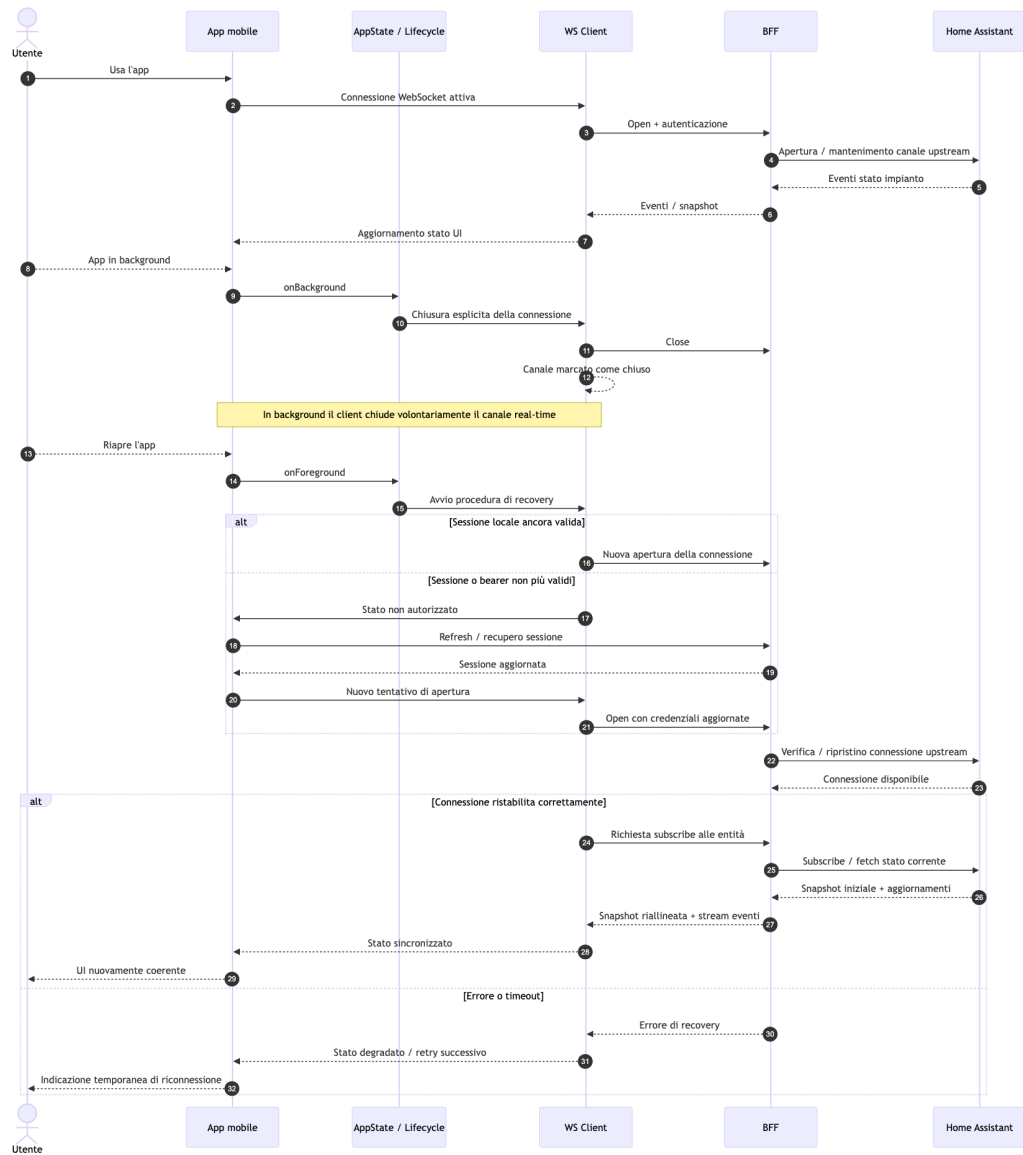


Figura 4.4: Sequenza integrata del canale WebSocket: apertura, sospensione in background, ripresa e recovery

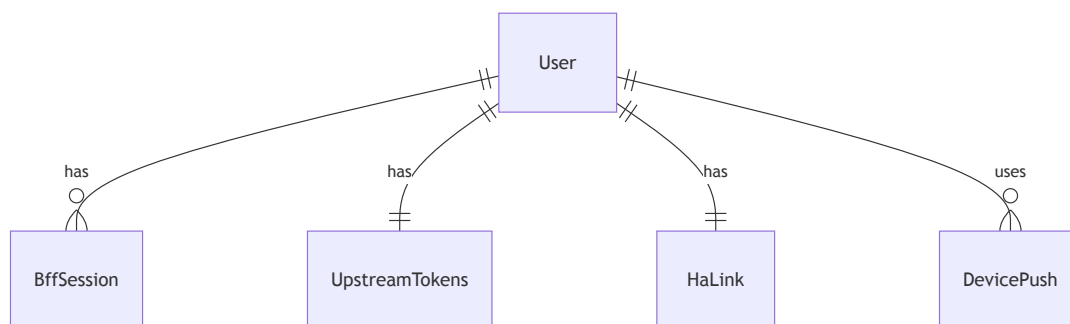


Figura 4.5: Schema logico (semplificato)

Capitolo 5

Backend for Frontend (BFF)

5.1 Inquadramento e responsabilità del BFF nel progetto

Nel contesto di SmartotumLite il *Backend for Frontend* (BFF) costituisce il punto di accesso unico per il client mobile: espone un insieme di API REST e un canale WebSocket pensati specificamente per le esigenze dell'applicazione, schermando la complessità e l'eterogeneità dei servizi esterni. In termini architetturali, il BFF realizza un confine netto tra: (i) la logica di presentazione e le responsabilità tipiche del mobile (interazione utente, gestione dello stato locale, gestione di rete in condizioni instabili), e (ii) l'integrazione con servizi esterni, che include autenticazione, rinnovo credenziali, normalizzazione degli errori e gestione di un canale real-time.

Nel progetto le responsabilità principali del BFF possono essere sintetizzate come segue:

- **Boundary unico per il client:** tutte le chiamate applicative transitano attraverso endpoint dedicati (`/auth`, `/smartotum`, `/ha`) e un canale real-time (`/ha/stream`).
- **Gestione sessioni BFF:** emissione e validazione di token opachi (`access/refresh`), rotazione del refresh token e revoca sessione.
- **Custodia e rinnovo dei token upstream:** i token dei servizi esterni (Smartotum e Home Assistant) sono gestiti lato server, cifrati a riposo e rinnovati quando necessario.
- **Mediazione e normalizzazione:** le API esterne vengono invocate con timeout e mapping degli errori in codici coerenti per il client.

- **Real-time come componente centrale:** il BFF funge da *bridge* tra il WebSocket client e la connessione server-side verso Home Assistant.

È importante sottolineare anche ciò che il BFF *non* fa: non implementa l'autenticazione primaria verso Smartotum (che avviene lato client), non contiene logica di UI e non sostituisce la logica di dominio dei servizi esterni; piuttosto orchestra chiamate e applica policy coerenti (sicurezza, timeouts, rate limiting, ecc.) su un confine controllato.

5.2 Architettura interna: moduli e flusso delle richieste

L'implementazione del BFF è organizzata secondo una struttura modulare tipica di un server Node/Express:

- **Router REST:** separazione per macro-aree (**Auth**, **Smartotum**, **HomeAssistant**).
- **Librerie applicative:** gestione sessione e token (**Session**), cifratura (**Vault**), invocazioni upstream e rinnovi (**Upstream**).
- **Gestione WebSocket:** un manager di connessioni e un modulo per la connessione server-side verso Home Assistant.
- **Persistenza:** DB gestito tramite ORM (Prisma), con tabelle focalizzate su sessioni e credenziali.

Il flusso generale di una richiesta REST protetta è:

1. il client invia una richiesta includendo il token BFF in **Authorization: Bearer**;
2. un middleware verifica il token e arricchisce il contesto della richiesta (**userId**, **sessionId**);
3. l'handler invoca eventuali funzioni di *ensure* per ottenere token upstream validi;
4. la chiamata verso i servizi esterni viene eseguita con timeout e mapping errori;
5. la risposta viene normalizzata e restituita al client.

Nel caso del canale WebSocket, la struttura è analoga ma la verifica del token avviene all'handshake e viene riapplicata periodicamente tramite *session liveness check* (Sez. 5.6.1).

5.3 Modello di autenticazione e gestione token

Il progetto adotta un modello a più livelli:

- **Token BFF (client-facing)**: coppia *access/refresh* opachi, emessi dal BFF e usati dal mobile per autenticare tutte le chiamate verso il BFF stesso.
- **Token upstream (server-facing)**: token necessari per parlare con Smartotum e Home Assistant, custoditi lato BFF.

Questo approccio consente di mantenere il client “leggero” rispetto alle credenziali più sensibili: il mobile non necessita di conservare token upstream, riducendo l’esposizione e centralizzando il controllo sul server.

5.3.1 Middleware Bearer e validazione access token

La protezione delle API REST è realizzata tramite un middleware che: (i) estrae il Bearer token dall’header, (ii) rifiuta la richiesta se assente, (iii) verifica l’access token e recupera la sessione, (iv) popola il contesto della richiesta con `userId` e `sessionId`.

Listing 5.1: Middleware di autenticazione REST (`bffAuth`).

```
1 function bffAuth(req: any, res: express.Response, next: express.  
  NextFunction) {  
2   const hdr = req.header('authorization') || '';  
3   const tok = hdr.startsWith('Bearer ') ? hdr.slice(7) : '';  
4   if (!tok) return res.status(401).json({ code: 'UNAUTHORIZED' });  
5   verifyAccess(tok).then(s => {  
6     if (!s) return res.status(401).json({ code: 'BFF_TOKEN_EXPIRED'  
       });  
7     req.userId = s.userId; req.sessionId = s.id; next();  
8   }).catch(next);  
9 }
```

L’uso di token opachi riduce la quantità di informazione contenuta nel token stesso: la validazione richiede una verifica lato server (hash e scadenze in DB), rendendo più semplice la revoca e l’evoluzione del formato.

5.3.2 Token opachi, hashing e TTL

I token BFF non vengono memorizzati in chiaro nel database: vengono salvati esclusivamente come hash (ad es. SHA-256 in Base64). In questo modo, un’eventuale compromissione del DB non fornisce direttamente credenziali utilizzabili.

In termini di validità temporale, il sistema differenzia:

- **Access token:** TTL breve (circa un'ora), per limitare la finestra di abuso.
- **Refresh token:** TTL lungo (circa trenta giorni), per garantire sessioni persistenti senza richiedere login frequenti.

5.3.3 Rotazione del refresh token

La rotazione del refresh token è implementata come aggiornamento atomico della sessione: presentando un refresh valido, il server genera *nuova coppia* access/refresh, aggiorna hash e scadenze nella stessa sessione, e restituisce i nuovi token al client. L'approccio di rotazione (anziché semplice rinnovo dell'access) consente di:

- ridurre l'impatto di un refresh token eventualmente intercettato;
- rendere possibile l'invalidazione per “consumo” (un refresh token vecchio non è più valido dopo rotazione);
- mantenere una traccia univoca del lifecycle della sessione.

Listing 5.2: Rotazione del refresh token lato BFF.

```

1 export async function rotateRefresh(oldRefresh: string) {
2   const hash = sha256Base64(oldRefresh);
3   const s = await prisma.bffSession.findFirst({
4     where: { refresh_hash: hash, revoked_at: null }
5   });
6   if (!s || s.refresh_expires_at.getTime() <= Date.now()) return null
7   ;
8   const { value: access, hash: access_hash } = genOpaqueToken();
9   const { value: refresh, hash: refresh_hash } = genOpaqueToken();
10
11   await prisma.bffSession.update({
12     where: { id: s.id },
13     data: {
14       access_hash,
15       access_expires_at: new Date(Date.now() + ACCESS_TTL),
16       refresh_hash,
17       refresh_expires_at: new Date(Date.now() + REFRESH_TTL),
18     }
19   });
20
21   return {
22     access, refresh,
23     access_expires_in: ACCESS_TTL/1000,
24     refresh_expires_in: REFRESH_TTL/1000,
25     sessionId: s.id
26   };

```

27 }

5.3.4 Revoca sessione e logout

Il logout revoca la sessione marcandola come non valida (`revoked_at`). Da quel momento, ogni tentativo di usare l'access token fallisce anche se formalmente non ancora scaduto; questo allinea il comportamento del sistema alle aspettative dell'utente e fornisce un meccanismo robusto in scenari di compromissione o device smarrito.

5.3.5 Token upstream: cifratura a riposo e rinnovo

I token per Smartotum e Home Assistant vengono memorizzati lato BFF in forma cifrata. L'implementazione adotta AES-256-GCM con un `keyId` associato, che consente anche *key rotation* e versionamento della chiave senza migrazioni invasive.

Listing 5.3: Cifratura/decifratura token upstream tramite AES-GCM.

```

1 export function encryptToken(plain: string, keyId = process.env.
  VAULT_KEY_ID || 'v1') {
2   const key = keyFromEnv(keyId);
3   const iv = randomBytes(12);
4   const aad = Buffer.from(keyId);
5   const c = createCipheriv('aes-256-gcm', key, iv); c.setAAD(aad);
6   const enc = Buffer.concat([c.update(plain, 'utf8'), c.final()]);
7   const tag = c.getAuthTag();
8   return `${keyId}:${iv.toString('base64')}:${enc.toString('base64')}
    :${tag.toString('base64')}`;
9 }
10
11 export function decryptToken(blob: string) {
12   const [keyId, ivB64, dataB64, tagB64] = blob.split(':');
13   const key = keyFromEnv(keyId);
14   const iv = Buffer.from(ivB64, 'base64');
15   const data = Buffer.from(dataB64, 'base64');
16   const tag = Buffer.from(tagB64, 'base64');
17   const aad = Buffer.from(keyId);
18   const d = createDecipheriv('aes-256-gcm', key, iv); d.setAAD(aad);
19   d.setAuthTag(tag);
20   const out = Buffer.concat([d.update(data), d.final()]);
21   return out.toString('utf8');
  }
```

Il *rinnovo* dei token upstream viene incapsulato in funzioni di tipo `ensure*`: l'handler richiede “un token valido”, e la funzione provvede a: (i) verificare la scadenza, (ii) eseguire refresh se necessario, (iii) aggiornare il DB con token e

scadenze aggiornate, (iv) ritornare un access token utilizzabile per la chiamata immediatamente successiva.

Questo pattern riduce la duplicazione di logica tra endpoint e rende più controllabili i *failure modes*.

5.4 Normalizzazione errori e gestione timeout

In un sistema che integra più servizi esterni, un requisito fondamentale è mantenere coerenti gli errori percepiti dal client, anche quando le cause originarie sono eterogenee (HTTP status, errori applicativi, timeouts, disconnessioni).

Il BFF adotta un insieme di codici applicativi (es. `UNAUTHORIZED`, `TIMEOUT`, `VALIDATION_ERROR`, `UPSTREAM_ERROR`, `SERVER_ERROR`) e li restituisce tramite payload JSON essenziale. Due esempi rappresentativi:

Listing 5.4: Esempio di errore di autorizzazione REST.

```
1 { "code": "UNAUTHORIZED" }
```

Listing 5.5: Esempio di errore di timeout verso un upstream.

```
1 { "code": "TIMEOUT", "message": "Upstream timeout" }
```

Lato implementativo, le chiamate verso gli upstream vengono eseguite con una primitiva di timeout (ad es. `fetchWithTimeout`); se il timeout scatta, l'errore viene ricondotto a un codice BFF stabile.

Listing 5.6: Proxy Smartotum con timeout/abort e mapping errori.

```
1 router.get('/systeminfo', async (req: any, res, next) => {
2   try {
3     const access = await ensureSmartotumAccess(req.userId);
4     const id = process.env.SYSTEM_ID;
5     const r = await fetchWithTimeout(`${API}/service/systems/${id}`,
6     {
7       headers: { Authorization: `Bearer ${access}`, Accept: '
      application/json' },
8       timeoutMs: 15000,
9     });
10    const body = await parseBody(r);
11    if (!r.ok) {
12      const msg = typeof body === "object" ? (body as any)?.message :
      undefined;
13      throw new UpstreamHttpError(r.status, msg ?? null);
14    }
15    res.json(body ?? []);
16  } catch (e: any) {
17    if (e instanceof HttpError) return next(e);
```

```

17 |     if (e?.name === "AbortError") return next(new TimeoutError);
18 |     return next(new InternalServerError)
19 |   }
20 | })

```

Questa strategia produce due benefici pratici: (i) il client gestisce un numero limitato di casi e può implementare retry/refresh in modo prevedibile; (ii) la logica di resilienza e mapping rimane centralizzata nel BFF, evitando di replicare comportamenti su più piattaforme.

5.5 Contratti API esposti dal BFF

Questo paragrafo sintetizza le API esposte al client, raggruppandole per dominio. L'obiettivo non è descrivere la logica di business degli upstream, ma esplicitare l'*interfaccia stabile* che il BFF presenta al mobile.

5.5.1 API di autenticazione (/auth/*)

Le API di autenticazione sono il punto di bootstrap della sessione BFF: il client invia i token Smartotum ottenuti in precedenza e riceve in risposta la coppia access/refresh BFF. La rotazione avviene tramite /auth/refresh, mentre /auth/logout revoca la sessione.

Tabella 5.1: Endpoint di autenticazione del BFF.

Met.	Path	Scopo / Output
POST	/auth/login	Crea sessione BFF e salva token Smartotum. Output: access , refresh , TTL.
POST	/auth/refresh	Ruota refresh token BFF. Output: nuova coppia access/refresh e TTL.
POST	/auth/logout	Revoca sessione BFF e rimozione device lato HA (se applicabile). Output: 204.

5.5.2 API Smartotum (/smartotum/*)

Gli endpoint /smartotum/* fungono da proxy verso Smartotum API, offrendo al client un insieme di operazioni orientate a casi d'uso: recupero sistemi, lettura e modifica parametri/temperature, gestione utenti, topic e notifiche.

Per mantenere leggibile la catalogazione, si riporta una tabella riassuntiva (metodo, path, scopo). La semantica dei payload rimane allineata al contratto BFF.

Tabella 5.2: Endpoint Smartotum mediati dal BFF.

Met.	Path	Scopo
GET	/smartotum/systems	Lista sistemi
GET	/smartotum/systeminfo	Informazioni sistema
POST	/smartotum/changeSwitch	Toggle parametri (es. <code>turn_off_all_enabled</code>)
GET	/smartotum/roomTargetTemperature	Lettura target temperatura stanza (<code>roomId</code> , <code>tempMode</code>)
POST	/smartotum/roomTargetTemperature	Impostazione target temperatura stanza
POST	/smartotum/disableRoomOverride	Disabilita override stanza
POST	/smartotum/activateSensor	Attiva sensore
POST	/smartotum/deactivateSensor	Disattiva sensore
POST	/smartotum/addSecCode	Aggiunge codice sicurezza
POST	/smartotum/removeSecCode	Rimuove codice sicurezza
POST	/smartotum/saveAlarmDelays	Salva ritardi allarme
GET	/smartotum/getAlarmDelays	Legge ritardi allarme
GET	/smartotum/getAllAutomation	Lista automazioni
GET	/smartotum/getAllAccessCodeConfigurations	Configurazioni access code
GET	/smartotum/getAllNotifications	Lista notifiche (con supporto media)
DELETE	/smartotum/deleteNotification	Elimina notifica e media associati
POST	/smartotum/saveLastNotificationsViewVisit	Salva timestamp “ultima visita”
GET	/smartotum/getAllTemperatureSchedule	Lista schedule temperatura
GET	/smartotum/getRoomTemperatureSetting	Lista setting temperatura stanza
POST	/smartotum/updateTopic	Aggiorna topic
POST	/smartotum/saveDoorBellButtonSetting	Imposta doorbell settings
POST	/smartotum/deleteTopic	Elimina topic
POST	/smartotum/handleDeviceSchedule	Collega schedule a device
GET	/smartotum/tokens	Legge token Smartotum (proxy)
GET	/smartotum/getUsers	Lista utenti

Continua nella pagina successiva

Met.	Path	Scopo
POST	/smartotum/deleteUser	Cancella utente
POST	/smartotum/inviteUser	Invita utente
POST	/smartotum/addUserToHouse	Aggiunge utente alla casa
GET	/smartotum/houseParameters	Parametri casa
GET	/smartotum/maintenanceMode	Stato maintenance
PUT	/smartotum/joinNetwork	Abilita join network
PUT	/smartotum/maintenanceMode	Imposta maintenance mode
PUT	/smartotum/wifiCredentials	Cambia credenziali Wi-Fi
GET	/smartotum/currentSoftwareVersion	Versione software corrente
POST	/smartotum/setSoftwareVersion	Imposta versione software
POST	/smartotum/checkIfSmartotumUserExists	Verifica esistenza utente
POST	/smartotum/smartotumTokens	Salva token Smartotum

5.5.3 API Home Assistant (REST) e webhook push

Gli endpoint `/ha/*` coprono funzionalità strettamente necessarie al client: statistiche e camera, informazioni utente, gestione push (registrazione/reset), eventi *location* e *tag*. È inoltre presente un webhook (`/ha/push`) pensato per ricevere notifiche da Home Assistant e inoltrarle a FCM; per ragioni di sicurezza, questo endpoint non usa Bearer token ma una chiave condivisa.

Tabella 5.3: Endpoint Home Assistant mediati dal BFF (REST e webhook).

Met.	Path	Scopo / Note	Auth
POST	/ha/token	Bootstrap token HA lato server.	Bearer BFF
GET	/ha/stats	Statistiche entità (<code>entityIds</code> , <code>start</code> , <code>end</code>).	Bearer BFF
POST	/ha/camera	URL stream camera (body <code>entity_id</code>).	Bearer BFF
GET	/ha/user/name	Nome e cognome utente.	Bearer BFF
POST	/ha/push/register	Registra device push HA.	Bearer BFF
POST	/ha/push/reset	Reset badge push.	Bearer BFF
POST	/ha/location	Invia evento location.	Bearer BFF
POST	/ha/tag	Invia evento tag.	Bearer BFF
POST	/ha/push?key=...	Webhook inbound: riceve push da HA e inoltra a FCM.	Chiave

5.6 WebSocket come canale real-time centrale

Una caratteristica distintiva del BFF nel progetto è l'uso del WebSocket come componente centrale per la fruizione real-time dello stato della smart home: il client non si connette direttamente al WebSocket di Home Assistant, ma usa un canale `/ha/stream` esposto dal BFF, che a sua volta gestisce una connessione server-side a Home Assistant.

Questo *dual-channel design* ha due obiettivi principali:

- **Riduzione del coupling:** il client parla un protocollo applicativo semplice (`subscribe/updates/service_call`), indipendente dalle specifiche di HA.
- **Controllo di sicurezza e resilienza:** autenticazione BFF, rate limiting, heartbeat e refresh token rimangono sotto il controllo del server e del client, senza esporre credenziali HA.

5.6.1 WebSocket pubblico BFF–Client: handshake, auth e close codes

Il WebSocket pubblico è esposto su WS `/ha/stream`. L'autenticazione avviene durante l'handshake mediante token BFF inviato:

- come header `Authorization: Bearer <token>`, oppure
- come query string `?token=<token>`.

Se il token è mancante o invalido, il server invia un messaggio applicativo **unauthorized** e chiude la connessione con un close code dedicato (es. 4001 per token mancante, 4003 per token invalido/scaduto). Inoltre, se un client apre una nuova connessione con la stessa sessione, la connessione precedente viene chiusa con close code 4000 (rimpiazzata da una nuova).

Listing 5.7: Handshake WS: autenticazione e gestione connessioni duplicate.

```

1 wss.on("connection", async (ws, req) => {
2   const tok = extractToken(req);
3   if (!tok) {
4     safeSend(ws, { type: "unauthorized" });
5     ws.close(4001, "Missing BFF token");
6     return;
7   }
8
9   let sessionId: string;
10  let userId: string;
11  try {
12    const session = await verifyAccess(tok);
13    if (!session?.id) throw new NotFoundError;
14    sessionId = session.id;
15    userId = session.userId;
16  } catch (e) {
17    safeSend(ws, { type: "unauthorized" });
18    ws.close(4003, "Invalid BFF token");
19    return;
20  }
21
22  for (const existing of clients) {
23    if (existing.sessionId === sessionId && existing.userId ===
24      userId) {
25      try { existing.ws.close(4000, "Replaced by new connection"); }
26      catch { }
27      cleanupClient(existing);
28      break;
29    }
30  }
31  });

```

Questa scelta progettuale è particolarmente utile su mobile, dove cambi rete, riprese da background o riaperture dell'app possono generare connessioni duplicate o half-open: rendere esplicita la politica “una sessione = un canale WS attivo” riduce gli stati incoerenti.

5.6.2 Protocollo messaggi: subscribe, snapshot, delta e service__call

Il protocollo applicativo adottato sul canale WS è volutamente essenziale. I principali message type sono:

- **Client** → **BFF**: `subscribe_entities`, `service_call`, `ping`.
- **BFF** → **Client**: `ready`, `entities_update`, `pong`, `error`, `unauthorized`.

La connessione segue un comportamento *ready-then-subscribe*: dopo l'handshake, il server invia `ready`, e il client esplicita l'intenzione di ricevere stream inviando `subscribe_entities`. A quel punto il server:

1. assicura che le credenziali HA siano valide;
2. crea o riusa la connessione server-side verso HA;
3. invia uno **snapshot** iniziale (`entities_update`);
4. sottoscrive gli update su HA e inoltra **delta** come ulteriori `entities_update`.

Listing 5.8: Messaggi WS: `ready` e `subscribe_entities`.

```

1 { "type": "ready" }
2
3 { "type": "subscribe_entities", "systemId": "system-123" }
```

Listing 5.9: Messaggio WS: snapshot iniziale (`entities_update`).

```

1 {
2   "type": "entities_update",
3   "data": {
4     "light.living_room": {
5       "entity_id": "light.living_room",
6       "state": "off",
7       "attributes": { "friendly_name": "Luce soggiorno" }
8     },
9     "sensor.temp_kitchen": {
10      "entity_id": "sensor.temp_kitchen",
11      "state": "21.3",
12      "attributes": { "unit_of_measurement": "\u00b0C" }
13    }
14  }
15 }
```

Listing 5.10: Messaggio WS: aggiornamento delta (`entities_update`).

```

1 {
2   "type": "entities_update",
3   "data": {
4     "light.living_room": {
5       "entity_id": "light.living_room",
6       "state": "on",
7       "attributes": { "friendly_name": "Luce soggiorno" }
8     }
9   }
10 }

```

Per l'esecuzione di comandi, il client invia `service_call`, che viene inoltrato dal BFF a Home Assistant:

Listing 5.11: Messaggio WS: richiesta di `service_call`.

```

1 {
2   "type": "service_call",
3   "domain": "light",
4   "service": "turn_on",
5   "data": { "entity_id": "light.living_room" }
6 }

```

Un aspetto rilevante del protocollo è la mancanza di un meccanismo esplicito di correlazione richiesta/risposta (ad esempio un `requestId` con `ack`): l'azione è considerata riuscita in assenza di errore; eventuali problemi vengono comunicati con `error`. Questa scelta semplifica il protocollo, ma sposta sul client la necessità di gestire la percezione di conferma tramite aggiornamenti di stato (tipicamente osservando il delta successivo).

5.6.3 Rate limiting e protezione del canale

Un canale WS esposto pubblicamente può diventare un vettore di abuso involontario (bug lato client, loop di retry) o intenzionale. Per questo il server applica un rate limiting sui messaggi ricevuti dal client (es. 30 messaggi/minuto). Superata la soglia, il server può rifiutare messaggi e/o inviare `error`.

Listing 5.12: Rate limiting applicativo lato WS.

```

1 function checkRateLimit(ctx: ClientCtx): boolean {
2   const now = Date.now();
3   if (now - ctx.lastMessageTime > RATE_LIMIT_WINDOW) {
4     ctx.messageCount = 0;
5     ctx.lastMessageTime = now;
6   }
7
8   ctx.messageCount++;

```

```

9
10 if (ctx.messageCount > RATE_LIMIT_MAX) {
11   console.warn("[HA-WS] Rate limit exceeded", { userId: ctx.userId
12   });
13   return false;
14 }
15
16 return true;
17 }

```

Questa protezione si combina con il fatto che il server mantiene una sola connessione WS attiva per sessione (chiusura con 4000 in caso di rimpiazzo), riducendo ulteriormente la possibilità di moltiplicare carico e stati inconsistenti.

5.6.4 Heartbeat e session liveness

Il sistema adotta una strategia di keepalive a due livelli:

- **Heartbeat WebSocket:** ping/pong per mantenere viva la connessione e rilevare half-open.
- **Session liveness check:** una verifica periodica lato server che la sessione BFF associata alla connessione sia ancora valida (non scaduta, non revocata). In caso negativo, il server forza **unauthorized** e chiusura (4003).

Su mobile, queste scelte sono particolarmente importanti perché l'app può essere sospesa, perdere connettività o cambiare rete; un heartbeat troppo aggressivo aumenterebbe il consumo energetico, mentre uno troppo blando aumenterebbe il tempo di rilevazione degli errori. La soluzione implementata bilancia reattività e overhead mantenendo un ping server regolare e una verifica sessione più frequente.

5.6.5 WebSocket server-side BFF–Home Assistant: connessione condivisa e caching

Sul lato upstream, il BFF apre una connessione WebSocket verso Home Assistant usando una libreria dedicata (**home-assistant-js-websocket**). Tale connessione:

- viene autenticata usando token HA ottenuti e rinnovati dal BFF;
- supporta operazioni di **getStates** (snapshot), **subscribeEntities** (stream) e **callService** (comandi);
- viene gestita con una cache per coppia **userId::sessionId**, così da riusare la connessione quando più client (o più subscribe) insistono sullo stesso contesto.

La logica di caching e subscription è rilevante perché evita di aprire una connessione HA per ogni client WS. In particolare:

1. al primo `subscribe_entities` il BFF crea una connessione HA e calcola uno snapshot;
2. gli update successivi vengono inoltrati come delta al client;
3. quando non ci sono più subscriber, la connessione HA può essere chiusa e rimossa per liberare risorse.

In caso di disconnessione o errori di reconnect, il BFF adotta una ricostruzione con un backoff breve (ordine di 1–2 secondi). Inoltre, in assenza di un heartbeat esplicito sul canale HA, può utilizzare un probe rapido (es. `getStates`) per verificare vitalità e decidere se ricostruire.

5.6.6 Strategie di recovery: failure modes tipici

Il design dual-channel consente di confinare e gestire in modo esplicito diversi failure mode:

- **Token BFF scaduto durante WS client:** rilevato dal session liveness check → `unauthorized` + `close 4003`.
- **Token HA scaduto:** rilevato da `ensureHABearerForSystem` → `refresh`; se fallisce → `error` al client.
- **HA down / WS HA disconnesso:** fallimento nella creazione o nella subscription → `error` e tentativi di rebuild lato server.
- **Rete mobile instabile:** il client gestisce riconnessione e backoff; il server limita gli effetti con close codes chiari e rate limiting.
- **Shutdown del server:** chiusura con close code 1001 (server shutdown) per distinguere eventi amministrativi da errori applicativi.

5.7 Struttura del database del BFF

Il database del BFF ha lo scopo di supportare la sicurezza e il corretto funzionamento delle integrazioni, non di replicare lo stato completo degli impianti. In particolare, persiste: (i) identità logica utente, (ii) sessioni BFF e relativo lifecycle, (iii) credenziali upstream cifrate a riposo (Smartotum e Home Assistant), e (iv) mapping dei dispositivi per notifiche push.

5.7.1 Entità principali e responsabilità

La Tabella 5.4 sintetizza le entità logiche principali e il loro ruolo.

Tabella 5.4: Entità principali del database del BFF e ruolo.

Entità	Ruolo nel sistema
User	Identità logica dell'utente nel perimetro del BFF. Costituisce la radice delle relazioni con sessioni, token upstream e dispositivi push.
BffSession	Sessione applicativa del BFF. Conserva gli hash dei token BFF (access/refresh), le scadenze (TTL) e l'eventuale revoca (<code>revoked_at</code>). Supporta rotazione del refresh token e invalidazione immediata.
UpstreamTokens	Contenitore dei token Smartotum. I valori sono memorizzati cifrati a riposo (schema AES-GCM con <code>key_id</code>) insieme alle relative scadenze.
HaLink	Contenitore dei token Home Assistant. Analogamente a <code>UpstreamTokens</code> , i token sono cifrati a riposo con <code>key_id</code> e scadenze.
DevicePush	Associazione tra dispositivo mobile e token di notifica (es. FCM), con metadati utili alla registrazione e alla gestione badge/reset.

5.7.2 Relazioni e cardinalità

La modellazione adotta relazioni semplici, coerenti con l'assunzione progettuale che ogni sistema abbia un'istanza BFF dedicata con database separato:

- `User` → `BffSession`: relazione 1–N (un utente può avere più sessioni, ad esempio per dispositivi diversi o login multipli).
- `User` → `UpstreamTokens`: relazione 1–1 (un set di credenziali Smartotum per utente nel contesto dell'istanza).
- `User` → `HaLink`: relazione 1–1 (un set di credenziali Home Assistant per utente nel contesto dell'istanza).
- `User` → `DevicePush`: relazione 1–N (un utente può associare più dispositivi push).

5.7.3 Chiavi, vincoli e indici rilevanti

Al fine di garantire efficienza e sicurezza, risultano rilevanti i seguenti vincoli (espressi qui in forma concettuale, indipendente dall'ORM):

- **Token BFF non in chiaro:** in `BffSession` vengono memorizzati `access_hash` e `refresh_hash`, non i token in chiaro.
- **Lookup efficiente dei token:** indici su `access_hash` e `refresh_hash` per validazione e rotazione.
- **Unicità dei link token upstream:** vincoli di unicità su `userId` in `UpstreamTokens` e `HaLink` (relazione 1-1).
- **Gestione revoca:** campo `revoked_at` (o equivalente) in `BffSession` per invalidazione immediata.
- **Cifratura versionata:** presenza di `key_id` in `UpstreamTokens` e `HaLink` per supportare rotazione/aggiornamento chiave senza migrazioni invasive del modello.

5.7.4 Lifecycle dei record (sintesi operativa)

Dal punto di vista del lifecycle, le entità seguono un comportamento prevedibile:

- **BffSession:** creazione al login; aggiornamento su refresh (rotazione); marcatura revoca su logout; scadenza naturale su TTL.
- **UpstreamTokens e HaLink:** creazione/aggiornamento durante bootstrap o quando il BFF rinnova le credenziali; aggiornamento periodico delle scadenze e dei valori cifrati durante refresh upstream.
- **DevicePush:** creazione/aggiornamento quando il dispositivo registra o aggiorna il token push; eventuale rimozione o invalidazione su logout o reset del device.

5.7.5 Nota di progettazione: perimetro e non ridondanza dei dati

Il database del BFF non persiste lo stato delle entità di Home Assistant (snapshot/delta) né i payload transienti del canale real-time: tali dati sono gestiti in memoria per finalità di inoltro e caching a breve termine. Questa scelta riduce la complessità e limita la superficie di dati sensibili a riposo.

5.7.6 Implicazioni architetturali del deployment per impianto

L'assunzione secondo cui ogni sistema Smartotum disponga di un'istanza BFF dedicata con database separato non va letta soltanto come una scelta di isolamento logico dei dati, ma come una conseguenza diretta dell'architettura fisica e distribuita del dominio applicativo. In Smartotum, infatti, ciascun impianto è associato a un gateway locale che costituisce il punto di accesso di prossimità al sistema domestico. Collocare il BFF in corrispondenza di tale gateway consente di mantenere allineati il perimetro software e il perimetro fisico dell'impianto.

Questa scelta abilita una proprietà importante dal punto di vista operativo: quando l'applicazione mobile rileva che il dispositivo dell'utente si trova sulla rete locale dell'abitazione, ad esempio attraverso la connessione Wi-Fi, le richieste possono essere instradate verso un endpoint interno esposto dal gateway stesso. In tal modo il client non dipende necessariamente da un percorso remoto verso infrastrutture esterne per l'esecuzione delle operazioni supportate in ambito locale. Il deployment per impianto risponde quindi anche a un requisito di continuità operativa: quando il gateway è raggiungibile sulla rete locale, l'applicazione può continuare a utilizzare il sistema tramite il BFF residente sull'impianto, riducendo la dipendenza dalla connettività esterna.

Da questo punto di vista, la presenza di un BFF dedicato per sistema non rappresenta una duplicazione arbitraria di componenti, ma una conseguenza della necessità di supportare sia l'accesso remoto sia l'accesso locale. Il backend non è soltanto un mediatore verso servizi esterni, ma anche il punto di ingresso coerente attraverso cui il client può interagire con l'impianto quando si trova nel suo contesto di rete naturale. L'istanza per impianto consente inoltre di mantenere separati token, sessioni, configurazioni e stato operativo in modo coerente con i confini reali del sistema controllato.

Questa impostazione ha naturalmente implicazioni in termini di scalabilità. In uno scenario con un numero elevato di impianti, ad esempio nell'ordine delle centinaia o delle migliaia, non sarebbe realistico gestire manualmente una istanza BFF per ciascun gateway. Il deployment multi-istanza deve quindi essere interpretato come un problema di automazione infrastrutturale: ogni istanza deve essere generata, configurata, aggiornata e monitorata a partire da procedure standardizzate, variando soltanto i parametri specifici dell'impianto. In altri termini, il modello corretto non è quello di una moltiplicazione manuale dei backend, ma quello di una famiglia di istanze omogenee, distribuite sui gateway e gestite tramite provisioning automatico, configurazione parametrica e osservabilità centralizzata.

I vantaggi principali di questa scelta sono i seguenti:

- coerenza tra il perimetro logico del backend e il perimetro fisico dell'impianto;

- possibilità di instradamento locale delle richieste quando l'utente si trova nella rete domestica;
- riduzione della dipendenza dalla connettività esterna per le funzionalità supportate dal gateway;
- isolamento di sessioni, credenziali e configurazioni tra sistemi distinti;
- contenimento degli effetti di eventuali fault entro il perimetro del singolo impianto.

A fronte di tali benefici, il modello introduce alcuni costi architetturali, principalmente legati alla gestione operativa di molte istanze:

- necessità di meccanismi di provisioning e aggiornamento automatizzati;
- maggiore complessità nelle attività di monitoraggio, logging e diagnostica distribuita;
- esigenza di coordinare in modo centralizzato rollout e configurazioni;
- gestione controllata dei segreti e dei parametri specifici di ciascun gateway.

Nel contesto di questa tesi, tali aspetti non vengono sviluppati come problema di orchestrazione infrastrutturale completa, ma la scelta progettuale risulta comunque motivata dal dominio. Il deployment di un BFF per impianto va infatti inteso come una conseguenza dell'architettura Smartotum e del requisito di operatività locale, più che come una semplice opzione implementativa.

5.8 Consolidamento: decisioni progettuali e trade-off

Per rendere esplicite le scelte implementative, la Tabella 5.5 riassume i trade-off principali.

Tabella 5.5: Decisioni progettuali nel BFF e trade-off.

Decisione	Alternative	Trade-off / Impatto
Token BFF opachi + lookup server	JWT self-contained	Opachi: revoca semplice e minor leakage; costo: lookup DB per validazione.
Hash token BFF nel DB	Token in chiaro cifrati	Hash: DB leak meno impattante; limite: impossibile "recuperare" token (non necessario).

Rotazione refresh token	Refresh statico	Rotazione: riduce rischio replay; richiede gestione corretta di race e single-flight lato client.
Token upstream solo lato server	Token upstream nel client	Riduce esposizione credenziali e coupling con upstream; aumenta responsabilità del BFF.
AES-GCM con keyId	Cifratura senza versionamento	Supporta key rotation e audit; richiede gestione sicura delle chiavi applicative.
WebSocket BFF-Client (protocollo custom)	Connessione diretta client-HA	BFF controlla sicurezza, rate limiting, e shield; costo: sviluppo e manutenzione del bridge.
Close codes applicativi (4000/4001/4003)	Close generico	Close codes: diagnosi chiara e recovery migliore; richiede coordinamento con client.
Rate limiting WS	Nessun limite	Protegge da bug/abusi; rischio: bloccare burst legittimi (va calibrato).
Snapshot + delta con <code>entities_update</code>	Solo delta	Snapshot semplifica bootstrap e recupero dopo reconnect; costo: payload iniziale potenzialmente grande.
Assenza di ack con requestId nel WS	Protocollo con correlazione/ack	Protocollo semplice e leggero; limite: feedback utente basato su update di stato, non su ack esplicito.

Capitolo 6

Frontend mobile: progettazione e implementazione del client React Native

6.1 Obiettivo del capitolo

Dopo aver definito il contesto, i requisiti e l'architettura complessiva della soluzione, questo capitolo descrive il **client mobile** sviluppato in React Native dal punto di vista della sua organizzazione interna e delle principali scelte implementative. L'attenzione è posta sul *come* il frontend realizza, in pratica, i flussi applicativi e i vincoli introdotti nei capitoli precedenti.

In particolare, il capitolo analizza:

- la struttura del progetto frontend e la suddivisione in moduli;
- la navigazione e il flusso di bootstrap/login/selezione sistema;
- la gestione della sessione lato client e dei token BFF in storage nativo sicuro;
- il layer API REST e la gestione uniforme di timeout, errori e refresh;
- il client WebSocket verso `/ha/stream` e la sincronizzazione dello stato real-time;
- l'integrazione nativa mirata (secure storage e location/background);
- i trade-off emersi nell'implementazione.

6.2 Struttura del progetto frontend

Il progetto frontend è organizzato in una struttura modulare orientata a separare le responsabilità principali del client: bootstrap dell'applicazione, gestione sessione, navigazione, comunicazione REST, canale real-time, schermate di dominio e integrazioni native. È presente una struttura centrata sul percorso `client/`, con moduli React Native, componenti condivisi e cartelle native distinte per Android e iOS.

Una vista semplificata delle macro-aree è la seguente:

- **Entry e bootstrap:** `index.js`, `App.tsx`;
- **Sessione/Auth:** `auth/AuthContext.tsx`;
- **Networking REST:** `API/API.ts`;
- **Real-time Home Assistant:** `ha_client/HA_client.tsx`;
- **Navigazione:** cartella `navigation/` (tab e stack);
- **Schermate:** cartella `screens/`;
- **Componenti riusabili:** cartella `components/`;
- **Persistenza locale e impostazioni:** cartella `settings/` (SQLite, DAO, location);
- **Notifiche e integrazioni dispositivo:** `notification/`, `WIFI/`;
- **Strato nativo:** cartelle `android/` e `ios/` con moduli per token store, refresh bridge e location core.

Questa organizzazione riflette una scelta progettuale coerente con il principio “*shared by default, native when needed*” introdotto nei capitoli precedenti: la logica applicativa e la maggior parte dei flussi sono mantenuti nel codice condiviso React Native, mentre le funzionalità strettamente OS-level sono confinate in moduli nativi mirati.

Un aspetto rilevante è la presenza di un **layer API centralizzato** (`API.ts`) e di due provider principali (`AuthProvider` e `HomeAssistantProvider`), che costituiscono di fatto il nucleo applicativo lato client: il primo governa sessione e sistema attivo, il secondo governa il canale real-time e lo stato delle entità domestiche. Questa impostazione riduce la dispersione della logica di rete all'interno delle schermate, che restano maggiormente focalizzate sulla presentazione e sulle interazioni di dominio.

Nel complesso, la struttura del progetto mostra una separazione pragmatica tra:

1. **responsabilità applicative trasversali** (auth, API, real-time, storage);

2. **responsabilità di dominio/UI** (home, sicurezza, schedulazioni, notifiche, impostazioni);
3. **responsabilità di piattaforma** (token store nativo, location/background, bridge refresh).

Questa suddivisione risulta particolarmente utile in un progetto multi-funzionale come Smartotum, dove il client non è un semplice “viewer” ma un componente che coordina flussi di autenticazione, aggiornamenti real-time, operazioni CRUD/configurazione e integrazione con servizi di dispositivo.

6.3 Bootstrap applicativo e navigazione

6.3.1 Entry point e inizializzazione

L’entry point React Native registra il componente principale dell’applicazione, mentre la logica di bootstrap è concentrata in `App.tsx`. In fase di avvio, il client inizializza innanzitutto la persistenza locale (database SQLite) e successivamente esegue il bootstrap della sessione tramite `AuthProvider`, leggendo:

- il `systemId` attivo;
- l’elenco dei sistemi salvati;
- i token BFF memorizzati nel token store nativo per il sistema attivo.

L’inizializzazione non ha solo una funzione tecnica, ma anche di UX: permette di ricostruire rapidamente il contesto minimo necessario per decidere il flusso iniziale dell’app senza richiedere all’utente interazioni ridondanti (ad esempio una nuova autenticazione quando la sessione è ancora valida).

6.3.2 Root flow condizionale

La navigazione iniziale è implementata tramite un **gate condizionale** nel componente principale. In modo semplificato, il flusso dipende da due variabili di stato:

- stato di autenticazione (`isLoggedIn`);
- presenza del sistema attivo (`systemId`).

Da questa combinazione derivano tre macro-scenari:

1. **Bootstrap in corso** (`isLoggedIn == null`): visualizzazione di una schermata di splash;

2. **Sessione assente/non valida:** accesso al *navigator* di autenticazione (login);
3. **Sessione valida ma sistema non selezionato:** accesso al *navigator* di selezione sistema;
4. **Sessione e sistema disponibili:** accesso all'area principale applicativa.

Questa impostazione traduce direttamente, sul piano implementativo del client, l'obiettivo di separare bootstrap, autenticazione, selezione del sistema e area operativa, già esplicitato a livello progettuale nei capitoli iniziali. La logica è volutamente centralizzata per evitare che ogni schermata debba gestire in modo autonomo precondizioni di sessione o contesto attivo.

6.3.3 Organizzazione della navigazione

La navigazione è articolata in una combinazione di **tab** e **stack**:

- un *root navigator* condizionale per i macro-flussi (auth, selezione sistema, app principale);
- un *bottom tab navigator* per le aree principali;
- stack dedicati per le schermate secondarie o di dettaglio.

Il **BottomBar** include le aree più frequenti d'uso (ad esempio dashboard, stanze, sicurezza, schedulazioni, storico), mentre sezioni come notifiche e impostazioni risultano navigabili ma non necessariamente esposte come tab principali, in modo da evitare sovraccarico dell'interfaccia.

Dal punto di vista ingegneristico, questa scelta bilancia due esigenze:

- **accesso rapido** alle funzionalità core;
- **scalabilità della navigazione** per un'app che include numerose schermate di configurazione, diagnostica e gestione dispositivi.

La presenza di stack di dominio consente inoltre di mantenere una buona località del codice: schermate correlate (ad esempio gestione automazioni, programmi temperatura, dettagli dispositivo) possono evolvere con minore impatto sulle aree non coinvolte.

6.4 Gestione della sessione lato client e contesto multi-sistema

6.4.1 AuthContext come coordinatore del ciclo di vita della sessione

La gestione della sessione è centralizzata in `AuthContext.tsx`, che mantiene informazioni come stato di login, `systemId` attivo e lista dei sistemi disponibili. Questa scelta è coerente con l'uso di React Context per stato globale a bassa/media complessità, evitando l'introduzione di uno state manager esterno (Redux, Zustand, MobX).

Il ruolo dell'`AuthProvider` non si limita alla memorizzazione di variabili di stato, ma include:

- bootstrap della sessione a partire dalla persistenza locale;
- registrazione di handler per scenari di unauthorized;
- coordinamento delle operazioni di login, switch di sistema e logout;
- transizione coerente del root flow tra splash, auth, selezione sistema e area principale.

In altre parole, il contesto di autenticazione funziona come **orchestratore del ciclo di vita della sessione applicativa lato client**, concentrando in un unico punto le transizioni che avrebbero altrimenti un elevato rischio di duplicazione tra schermate.

6.4.2 Persistenza dei token BFF in storage nativo sicuro

Una scelta importante, già motivata nei capitoli precedenti ma qui osservabile nell'implementazione, è la **persistenza dei token BFF tramite storage nativo sicuro**. In particolare, il client usa un modulo nativo esposto a React Native (`NativeModules.TokenStore`) per leggere/scrivere i token runtime per sistema:

- su Android: `EncryptedSharedPreferences`;
- su iOS: `Keychain`.

Questa soluzione presenta due vantaggi pratici:

1. consente di trattare i token BFF come credenziali sensibili, evitando storage non protetti;

2. permette la collaborazione con i moduli nativi di background (in particolare location), che possono aver bisogno di accedere alla sessione anche fuori dal normale ciclo di vita della UI React Native.

Accanto al token store nativo, il progetto utilizza anche `react-native-encrypted-storage` per persistere *metadati applicativi* (ad esempio `systemId`, lista sistemi e altri dati di supporto) e token temporanei impiegati in alcuni flussi di login/switch. Tale doppio livello di persistenza aumenta la complessità implementativa, ma consente di distinguere meglio tra credenziali operative BFF e dati applicativi di contesto.

6.4.3 Gestione del sistema attivo e multi-sistema

Il client supporta un contesto **multi-sistema**: l'utente può avere più sistemi associati e selezionare quale rendere attivo. L'informazione sul sistema corrente è persistita localmente (`systemId`), mentre l'elenco dei sistemi è conservato sia in encrypted storage sia in una tabella locale SQLite dei server/metadati.

La transizione di sistema (funzione di *switch*) non è un semplice cambio di UI, ma una vera transizione di contesto che coinvolge:

- aggiornamento del `systemId` attivo;
- caricamento/rigenerazione dei token BFF e del contesto HA per il nuovo sistema;
- riallineamento del provider real-time e delle schermate dipendenti dallo stato delle entità.

Dal punto di vista della robustezza, questa è una delle aree più sensibili del client: il supporto multi-sistema introduce casi limite aggiuntivi (switch in presenza di WebSocket attivo, token in scadenza, logout parziale di un solo sistema, ecc.) che richiedono una gestione coordinata tra stato React, storage locale e bridge nativi.

6.4.4 Scenari di unauthorized e ritorno allo stato non autenticato

Quando il refresh fallisce o la sessione non è più recuperabile, il client registra un handler di *unauthorized* che conduce a un flusso di ritorno al login (`backLogin()`). Tale flusso include pulizia del token store nativo, pulizia della persistenza applicativa (con eccezioni mirate, ad esempio identificativi push da preservare), pulizia dei metadati nel DB locale e aggiornamento dello stato di autenticazione.

Questa gestione esplicita è importante perché evita uno stato applicativo “semi-autenticato” (ad esempio UI ancora in area principale ma token invalidi), che in sistemi con real-time e background tasks tende a generare errori difficili da

diagnosticare. La centralizzazione nel contesto auth riduce inoltre il rischio che singole schermate implementino strategie di recovery incoerenti.

6.5 Layer API REST lato client

6.5.1 Ruolo del modulo `API.ts`

Le chiamate REST verso il BFF risultano accentrate in un modulo principale (`client/API/API.ts`). La scelta ha una motivazione pragmatica: mantenere in un unico punto il comportamento comune relativo a:

- aggiunta del bearer token;
- timeout delle richieste;
- intercettazione dei 401;
- refresh e retry;
- mapping degli errori in tipi applicativi.

Tale centralizzazione facilita uniformità e manutenzione del comportamento di rete, soprattutto in un progetto in cui molte schermate invocano endpoint diversi ma condividono policy identiche di autenticazione e gestione errori. Il rovescio della medaglia, come discusso anche nei trade-off, è la crescita progressiva del modulo e l'aumento di accoppiamento tra domini applicativi distinti.

6.5.2 Bearer injection, timeout e normalizzazione errori

Il *wrapper* di fetch autorizzato (`authorizedFetch`) recupera il token access BFF dal token store nativo del sistema attivo e lo inserisce nell'header `Authorization: Bearer <access>`. In questo modo le funzioni applicative che invocano endpoint del BFF non devono conoscere direttamente i dettagli di persistenza dei token.

Si evidenzia inoltre:

- uso sistematico di `AbortController` con timeout standard (tipicamente 15s);
- un mapping errori verso tipi applicativi (`ApiError`, `TimeoutError`, `NetworkError`);
- una funzione di conversione da risposta HTTP a codice errore applicativo.

Questa normalizzazione è particolarmente utile per la UI, perché permette di presentare comportamenti coerenti tra schermate diverse (messaggi, retry, fallback) anche quando le risposte del backend variano per endpoint o dominio funzionale.

Dal punto di vista architetturale, si osserva una continuità con la scelta lato BFF di normalizzare errori e timeout: il client eredita un contratto più uniforme e aggiunge un ulteriore livello di adattamento orientato alla UX e alla gestione dello stato locale.

6.5.3 Refresh token e retry con strategia single-flight

Uno dei punti più rilevanti dell'implementazione del client è la gestione del refresh con strategia **single-flight**. In presenza di 401, `authorizedFetch`:

1. tenta il refresh tramite una funzione dedicata;
2. attende il completamento di un'unica promessa condivisa se un refresh è già in corso;
3. rilegge il nuovo access token dal token store nativo;
4. ritenta la richiesta una sola volta.

La presenza di una variabile condivisa (es. `refreshingPromise`) evita il problema classico dei *refresh concorrenti*, molto frequente nelle app con più chiamate REST in parallelo o con eventi real-time che innescano aggiornamenti simultanei.

Un elemento particolarmente interessante è il coordinamento tra runtime JavaScript e layer nativo tramite `ApiBridge/RefreshGuard`. Ciò risponde a un'esigenza pratica del progetto: non solo la UI React Native, ma anche i moduli nativi di background possono dover interagire con la stessa sessione, e quindi devono evitare collisioni nei refresh.

Questa scelta implementativa rappresenta una concretizzazione importante dei requisiti di continuità e robustezza della sessione definiti nel capitolo 3, senza demandare completamente il problema al solo backend.

6.6 Client WebSocket e gestione del real-time lato frontend

6.6.1 HomeAssistantProvider come perno del real-time

Il canale real-time lato client è gestito da `client/ha_client/HA_client.tsx`, dove è implementato `HomeAssistantProvider`. Questo provider funge da **strato di adattamento tra WebSocket e UI**: mantiene la connessione a `/ha/stream`, espone lo stato delle entità Home Assistant e fornisce azioni utilizzabili dalle schermate di dominio.

Questa architettura consente di evitare che ogni schermata apra o gestisca un proprio socket. Invece, il canale è unico e centralizzato, mentre l'interfaccia utente consuma lo stato condiviso tramite hook/context. Il vantaggio principale è duplice:

- minore duplicazione della logica di connessione/riconnessione;
- maggiore coerenza dello stato mostrato simultaneamente in schermate diverse.

6.6.2 Handshake, subscribe e protocollo base lato client

Il client apre la connessione WebSocket passando in query string:

- il token access BFF;
- il `systemId` attivo.

Dopo il messaggio di `ready`, il provider invia una richiesta di subscribe allo stream delle entità (`subscribe_entities`) e inizia a ricevere messaggi di tipo `entities_update`.

Sul piano implementativo client-side, la gestione del protocollo viene mantenuta relativamente semplice:

- coda/invio dei messaggi quando il canale è pronto;
- aggiornamento dello stato locale delle entità a partire dai messaggi ricevuti;
- gestione di messaggi di errore/`unauthorized`.

L'aggiornamento delle entità avviene tramite sostituzione della mappa locale (`setEntities(msg.data || {})`);.

6.6.3 Heartbeat, riconnessione e recovery della sessione

Il provider implementa meccanismi di resilienza coerenti con i requisiti real-time:

- heartbeat ping/pong (con intervalli e timeout dedicati);
- riconnessione con exponential backoff e jitter;
- trattamento specifico di alcuni close code (ad esempio riavvio server);
- gestione di messaggi `unauthorized` e close code di sessione scaduta.

Uno degli aspetti più interessanti è l'integrazione tra **recovery del canale WS** e **refresh token lato client**. In caso di sessione non valida, il provider non si limita a chiudere il canale, ma tenta il refresh (tramite la stessa logica single-flight condivisa con il layer REST) e poi la riconnessione. Solo dopo un certo numero di fallimenti consecutivi viene eseguito il fallback verso l'handler di unauthorized generale.

Questa integrazione è significativa perché evita la duplicazione di policy tra REST e real-time: il client adotta una strategia unificata di recupero della sessione e la riutilizza in più canali di comunicazione. In termini di manutenibilità, ciò riduce il rischio di divergenze (ad esempio REST che riesce a recuperare la sessione mentre WS forza il logout, o viceversa).

6.6.4 Impatto sulla UI e sincronizzazione stato

Le schermate principali (dashboard, stanze, sicurezza, storico) consumano lo stato delle entità dal provider real-time. Questo produce un modello in cui:

- il WebSocket è la fonte principale per lo *stato corrente* delle entità;
- le API REST sono usate maggiormente per operazioni di configurazione, CRUD o query specialistiche;
- dopo alcune mutazioni REST, le schermate eseguono refetch mirati.

Ne deriva un pattern ibrido, pragmatico e comune nelle app real-time:

1. **stream** per lo stato dinamico ad alta frequenza;
2. **REST** per operazioni puntuali, configurazioni e risorse non adatte al canale persistente.

Dal punto di vista della progettazione del client, questa separazione aiuta a mantenere le schermate relativamente semplici: la UI non deve conoscere dettagli di protocollo WebSocket, ma si limita a consumare uno stato già sincronizzato e invocare azioni di dominio.

6.7 Stato applicativo, persistenza locale e sincronizzazione UI

6.7.1 Stato globale e stato locale

Il frontend adotta una strategia basata su **Context + hooks**, con due contesti principali:

- **AuthContext**, per sessione, sistema attivo e stato auth;
- **HAContext** (tramite **HomeAssistantProvider**), per entità real-time e azioni correlate.

La UI di ciascuna schermata mantiene invece stato locale tramite hook React (**useState**, **useMemo**, **useCallback**, **useEffect**, **useFocusEffect**) per aspetti come filtri, modali, form, paginazione, loading locale e dettagli di interazione.

Questa separazione consente di evitare sia:

- uno stato globale eccessivamente ampio e difficile da mantenere;
- sia una frammentazione totale della logica, con duplicazione di stato sessione o stato real-time tra schermate.

In altre parole, il client adotta un modello intermedio in cui il globale contiene solo ciò che deve essere condiviso stabilmente, mentre la maggior parte della logica UI rimane locale ai componenti di dominio.

6.7.2 Persistenza locale: encrypted storage e SQLite

Accanto al token store nativo e alla persistenza di chiavi applicative in encrypted storage, il client utilizza un database locale SQLite (attraverso cartelle **db/** e **dao/**) per metadati come server/sistemi e storico location.

La presenza di SQLite nel client è utile per due ragioni principali:

1. gestire dati strutturati che beneficino di query locali o relazioni semplici;
2. evitare di sovraccaricare gli storage key-value con insiemi di metadati più articolati.

L'uso combinato di persistenze diverse (token store nativo, encrypted storage, SQLite) riflette una scelta pratica che privilegia l'adeguatezza dello strumento al tipo di dato:

- **credenziali sensibili**: token store nativo;
- **chiavi applicative e metadati semplici**: encrypted storage;
- **dati strutturati locali**: SQLite/DAO.

Questo approccio aumenta il numero di componenti da coordinare, ma migliora la chiarezza del perimetro di ciascuna persistenza e riduce l'uso improprio di un solo meccanismo per dati eterogenei.

6.7.3 Coerenza tra stato, persistenza e lifecycle mobile

Una difficoltà tipica delle app mobili è mantenere coerenza tra:

- stato in memoria (React);
- persistenza locale;
- stato dei canali di rete (REST/WS);
- ciclo di vita del dispositivo (foreground/background, terminazione, ripresa).

Nel client Smartotum, questa coerenza viene perseguita centralizzando:

- bootstrap sessione e routing in `AuthProvider`;
- canale real-time in `HomeAssistantProvider`;
- auth/refresh/retry nel layer API;
- accesso ai token sensibili nel token store nativo.

La scelta non elimina tutta la complessità, ma la concentra in moduli chiaramente identificabili, rendendo più prevedibili gli effetti di una modifica e più tracciabili gli scenari di errore.

6.8 Integrazione nativa mirata

6.8.1 Razionalità dell'integrazione nativa nel client React Native

I capitoli precedenti hanno già motivato la scelta di mantenere un perimetro nativo limitato. Nel frontend questo principio è applicato concretamente in due aree:

1. **secure storage dei token BFF;**
2. **location/background tracking e invio posizione.**

Questa configurazione preserva il vantaggio della codebase unica per UI e logica applicativa, senza rinunciare a requisiti OS-level che richiedono maggiore controllo o integrazione diretta con le piattaforme.

6.8.2 Secure storage nativo e interfaccia React Native

Sul lato JavaScript, il client accede ai token BFF tramite un modulo nativo (`TokenStore`) esposto a React Native. Le implementazioni di piattaforma usano meccanismi differenti ma concettualmente equivalenti:

- **iOS:** Keychain;
- **Android:** `EncryptedSharedPreferences` con chiavi gestite dal sistema.

Questa astrazione consente al resto del client (`AuthContext`, layer API, provider WS) di non dipendere dai dettagli di piattaforma. L'uso di un'interfaccia comune è importante anche in ottica di test e manutenzione: gli aggiornamenti a una singola implementazione nativa non richiedono modifiche pervasive nel codice React Native, purché il contratto del modulo resti stabile.

6.8.3 Location/background e continuità della sessione fuori dalla UI

La seconda area nativa rilevante è la gestione della location in background. Nel client è presente una distinzione esplicita tra:

- orchestrazione JS (permessi, configurazioni, integrazione con preferenze/DB locale);
- core nativi (Android/iOS) responsabili di geofence, aggiornamenti significativi, trigger periodici/background e invio verso il BFF.

Sono inoltre presenti differenze di piattaforma coerenti con i vincoli dei rispettivi sistemi operativi:

- su Android: uso di componenti come `WorkManager` per task periodici e receiver per il riavvio del tracking dopo reboot;
- su iOS: integrazione con background fetch, significant location changes e limiti specifici (ad esempio numero massimo di zone).

Un aspetto particolarmente significativo è che, in scenario background, i moduli nativi possono gestire il refresh della sessione BFF (su risposta 401) e ritentare l'invio della posizione, coordinandosi con il runtime JS tramite `ApiBridge/RefreshGuard`.

Questa soluzione evita di dipendere esclusivamente dalla presenza dell'interfaccia React Native attiva in foreground e rende più robusto un caso d'uso critico del progetto. Allo stesso tempo, introduce complessità di sincronizzazione che giustifica la presenza di primitive di coordinamento tra i due livelli.

6.9 Schermate principali e organizzazione funzionale della UI

6.9.1 Aree funzionali principali

Il frontend include un insieme ampio di schermate che coprono sia funzioni core di controllo domotico sia funzioni di configurazione e gestione. Senza entrare nei dettagli implementativi di ogni singolo componente, è utile raggrupparle in macro-aree:

- **Bootstrap e accesso:** Splash, Login, Systems;
- **Controllo e monitoraggio:** Home, Rooms, Security, History;
- **Programmazione e automazione:** Schedules, schermate collegate per automazioni e programmi temperatura;
- **Notifiche:** lista notifiche e gestione contenuti correlati;
- **Impostazioni e configurazione:** Settings e sottoschermate (utenti, Wi-Fi, aggiornamento software, location, tastierini/codici, NFC, ecc.).

Questa articolazione è coerente con i casi d'uso formalizzati nel capitolo 3: il frontend non si limita a un pannello di controllo real-time, ma include funzioni di configurazione persistente, sicurezza, automazioni e gestione eventi.

6.9.2 Ruolo delle schermate nel flusso utente

A livello di UX e orchestrazione tecnica, si possono distinguere tre tipologie di schermate:

1. **schermate di transizione di stato** (es. login, selezione sistema), che modificano il contesto applicativo globale;
2. **schermate operative real-time** (es. home, stanze, sicurezza), che consumano principalmente stato da WebSocket;
3. **schermate di configurazione/CRUD**, che invocano endpoint REST e spesso eseguono refetch dopo mutazioni.

Questa distinzione è utile anche in termini implementativi, poiché giustifica la coesistenza dei due canali (WS e REST) e dei due contesti globali (auth e real-time): non tutte le schermate hanno lo stesso profilo di interazione con la rete e con lo stato condiviso.

6.9.3 Scalabilità della UI e riuso dei componenti

La presenza di una cartella `components/` dedicata a componenti riusabili di dominio (ad esempio elementi di controllo per luci, sensori, allarme, card stanza, modali) suggerisce una strategia di composizione della UI basata su building blocks condivisi.

In un'applicazione con numerose schermate e dispositivi eterogenei, il riuso dei componenti è fondamentale non solo per la produttività, ma anche per la coerenza dell'esperienza utente: pattern di interazione simili (attivazione/disattivazione, visualizzazione stato, input di parametri) possono essere mantenuti uniformi in aree diverse dell'applicazione.

Al tempo stesso, parte della logica di `fetch/loading/error` rimane distribuita tra più schermate, con `refetch` manuali dopo mutazioni. Ciò indica una soluzione pragmatica e funzionante, ma anche un possibile spazio di evoluzione futura verso pattern più uniformi di data fetching lato client.

6.9.4 Home page e accesso rapido ai controlli

La Home page costituisce il punto di ingresso principale all'applicazione e riassume in un'unica vista le macro-funzionalità più rilevanti del sistema, privilegiando una logica di controllo rapido e consultazione sintetica.

La Figura 6.1 evidenzia due aspetti complementari dell'interfaccia. A sinistra è mostrata la schermata principale, progettata come dashboard sintetica delle principali aree funzionali dell'impianto. L'utente può agire direttamente sui controlli rapidi, ad esempio selezionando lo stato della climatizzazione o richiamando azioni aggregate su tapparelle, luci e prese. La barra di navigazione inferiore consente inoltre l'accesso alle altre sezioni dell'applicazione, mantenendo coerente la struttura globale dell'interfaccia.

A destra è invece mostrato l'utilizzo di un modale contestuale, aperto a partire dalla sezione relativa alle luci. Questo pattern permette di approfondire una categoria funzionale senza abbandonare la Home page, mantenendo il contesto della schermata principale e riducendo il costo cognitivo della navigazione. Lo stesso meccanismo viene adottato anche per altre categorie, come prese, tapparelle e sensori, e consente di combinare una vista compatta a livello globale con un livello di dettaglio selettivo sulle singole entità controllabili.

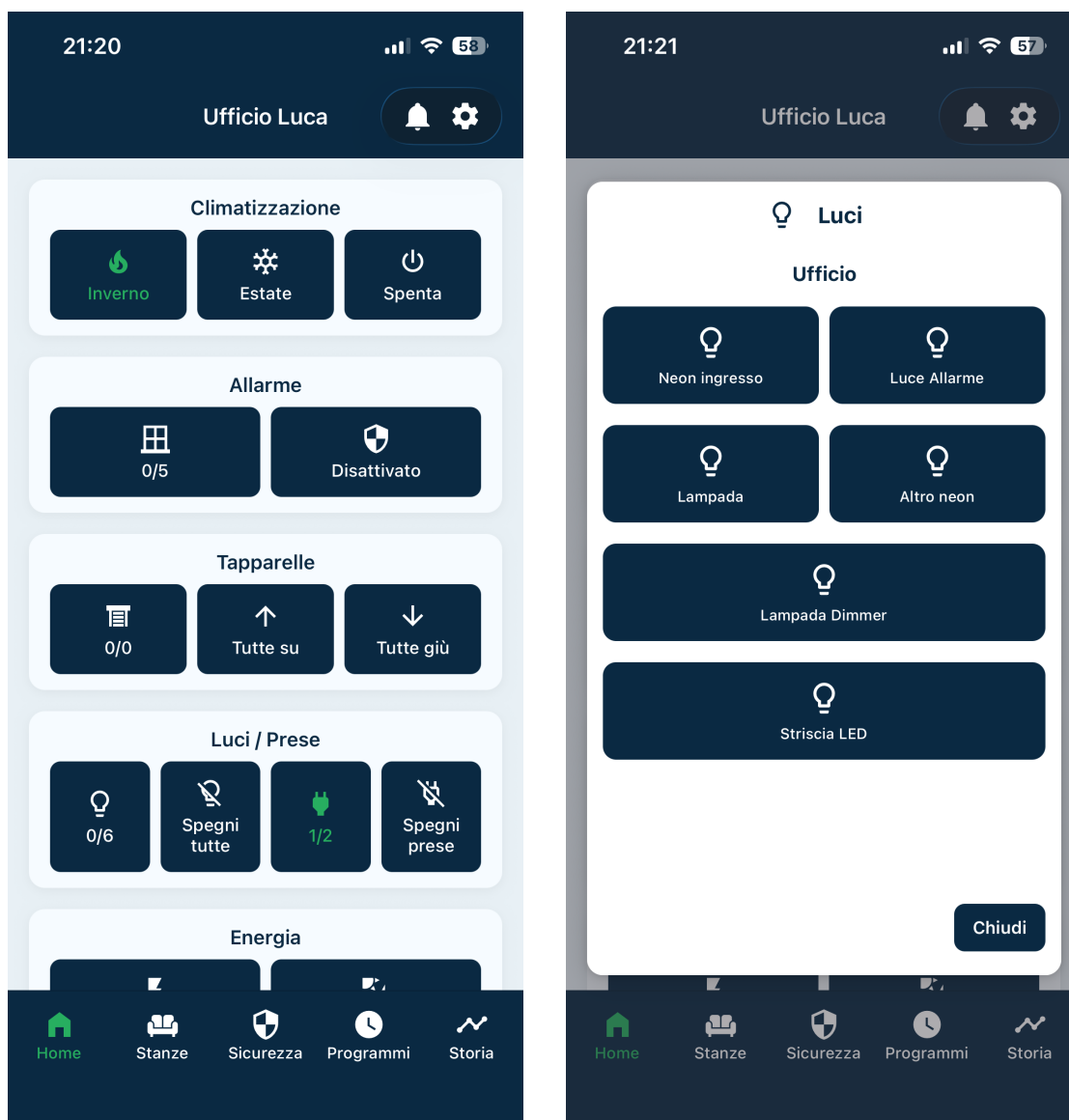


Figura 6.1: Home page dell'applicazione e apertura di un modale di dettaglio per il controllo delle entità

6.9.5 Navigazione per stanze e dettaglio contestuale

La sezione *Stanze* organizza le funzionalità dell'applicazione secondo una logica spaziale, raggruppando controlli e dati in base agli ambienti dell'impianto e offrendo una consultazione più contestualizzata rispetto alla dashboard globale della Home page.

La Figura 6.2 mostra due esempi rappresentativi della sezione dedicata alle stanze.

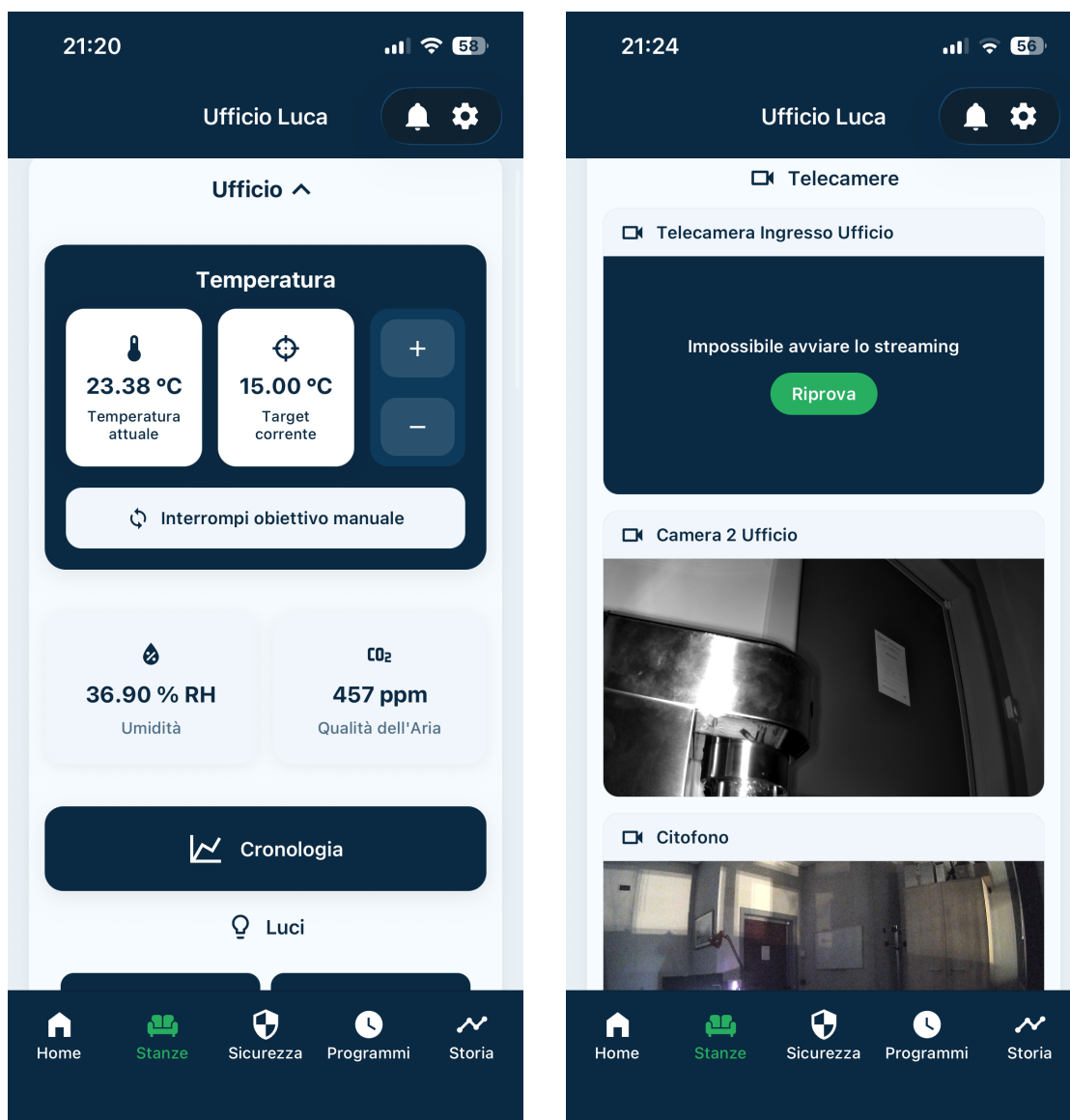


Figura 6.2: Sezione *Stanze*: vista contestuale per ambiente e integrazione delle telecamere

A sinistra è presentata la vista relativa a un singolo ambiente, nella quale vengono mostrati dati ambientali come temperatura, umidità e qualità dell'aria, insieme ai controlli associati alla climatizzazione della stanza. Questa schermata evidenzia come l'applicazione combini informazione di stato e azione diretta, permettendo all'utente di leggere il contesto corrente e intervenire immediatamente sui parametri disponibili.

A destra è invece mostrata l'integrazione delle telecamere all'interno della stessa logica per ambiente. Oltre alla visualizzazione dei flussi disponibili, la schermata documenta anche la gestione di un caso non nominale, nel quale uno streaming non può essere avviato e l'interfaccia propone un'azione esplicita di ripetizione. Questo aspetto è rilevante perché mostra che la UI non si limita a presentare i controlli in condizioni ideali, ma include anche meccanismi di feedback e recupero coerenti con l'uso reale del sistema. Le altre categorie già accessibili dalla Home, come luci, prese, tapparelle e sensori, sono riproposte nella sezione *Stanze* secondo la stessa logica contestualizzata, evitando tuttavia ridondanze nella documentazione visiva della tesi.

6.9.6 Gestione della sicurezza e configurazione dell'allarme

La sezione *Sicurezza* raccoglie le funzionalità di configurazione e controllo dell'impianto antintrusione, mettendo a disposizione in un'unica area sia la gestione operativa dei sensori, sia gli elementi di configurazione persistente legati ai codici di attivazione, alle azioni automatiche e ai tempi di inserimento dell'allarme.

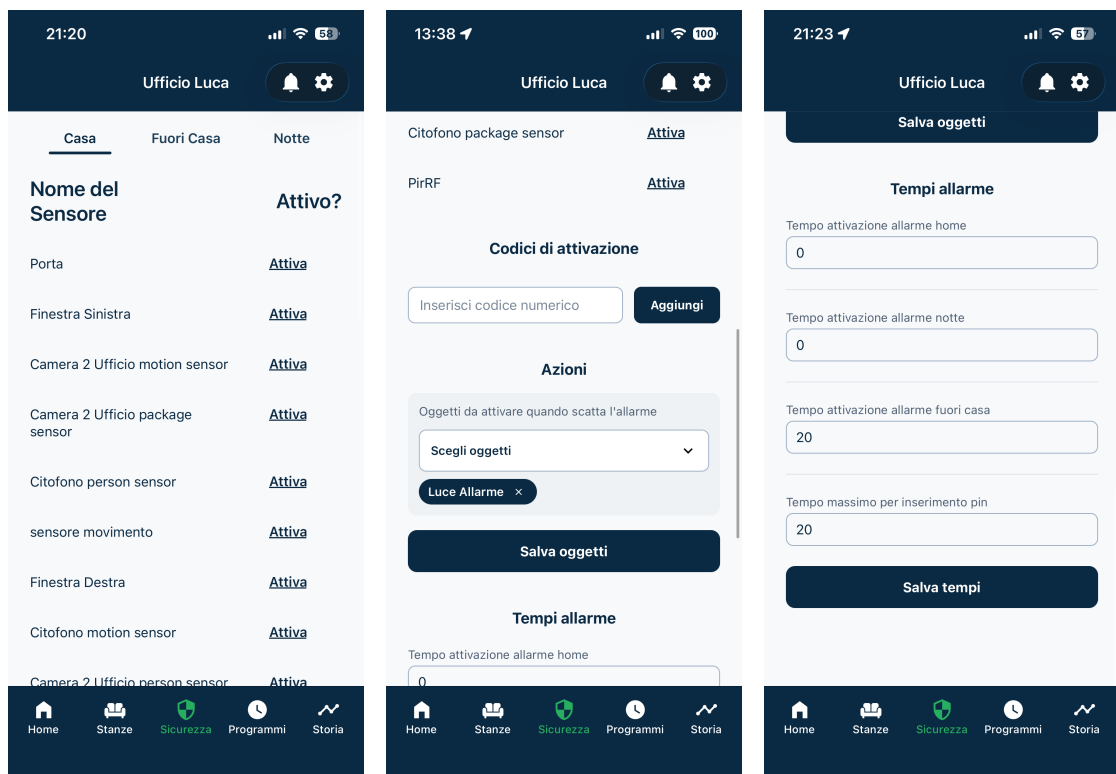


Figura 6.3: Sezione *Sicurezza*: attivazione dei sensori, configurazione dei codici e impostazione delle tempistiche dell'allarme

La Figura 6.3 mostra tre viste rappresentative della sezione dedicata alla sicurezza. La prima schermata evidenzia la gestione dei sensori suddivisa per modalità di inserimento dell'allarme, come *casa*, *fuori casa* e *notte*. In questo modo l'utente può abilitare o disabilitare selettivamente i diversi sensori in funzione dello scenario operativo, configurando il comportamento dell'impianto in maniera più flessibile rispetto a una semplice attivazione globale.

La seconda schermata mostra la configurazione dei codici di attivazione e delle azioni da eseguire quando l'allarme viene innescato. Oltre alla gestione dei codici numerici, l'interfaccia consente di associare oggetti o dispositivi da attivare automaticamente in caso di evento, come ad esempio luci o altri attuatori. Questa vista è significativa perché mostra come la sezione *Sicurezza* non si limiti alla sola visualizzazione dello stato, ma includa anche funzioni di configurazione persistente che modificano il comportamento del sistema.

La terza schermata documenta infine la configurazione delle tempistiche dell'allarme, quali i tempi di attivazione nelle diverse modalità e il tempo massimo per l'inserimento del codice PIN. Tale impostazione consente di adattare il comportamento del sistema alle esigenze dell'utente e al contesto reale di utilizzo, introducendo un ulteriore livello di personalizzazione. Nel complesso, questa sezione mostra bene il carattere ibrido dell'applicazione: accanto al controllo operativo immediato, il frontend supporta infatti anche la configurazione strutturata di parametri di sicurezza persistenti.

6.9.7 Programmi di temperatura e pianificazione stagionale

La sezione *Programmi* consente di definire, selezionare e modificare i programmi di temperatura associabili alle diverse stanze del sistema. Questa parte dell'applicazione supporta una logica di pianificazione persistente, nella quale il comportamento della climatizzazione non dipende soltanto da comandi manuali immediati, ma anche da profili temporali configurabili in funzione della stagione e delle abitudini d'uso dell'utente.

La Figura 6.4 mostra tre momenti rappresentativi della gestione dei programmi di temperatura. La prima schermata presenta l'elenco dei programmi disponibili per la stagione selezionata, in questo caso invernale, insieme alla visualizzazione sintetica degli andamenti giornalieri. L'interfaccia consente di creare nuovi programmi, selezionarne uno esistente e applicarlo alle stanze interessate, mantenendo separata la distinzione tra configurazione estiva e invernale.

La seconda schermata documenta la creazione iniziale di un nuovo programma tramite una procedura guidata semplificata. In questa fase l'utente inserisce alcune informazioni di base, come gli orari abituali di risveglio, uscita, rientro e riposo, in modo da ottenere rapidamente una prima struttura del programma. Questa modalità è rilevante perché riduce la complessità iniziale di configurazione

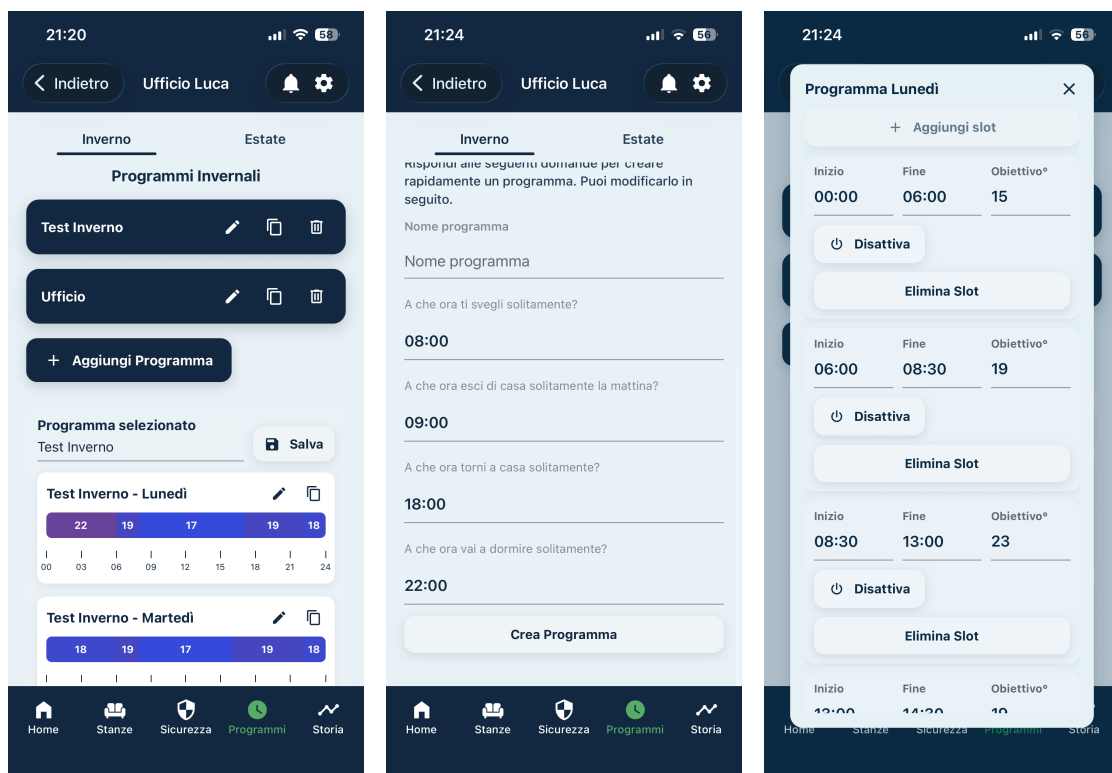


Figura 6.4: Sezione *Programmi*: elenco dei programmi stagionali, creazione guidata e modifica degli slot orari

e rende più accessibile una funzionalità che, se proposta direttamente in forma completamente manuale, risulterebbe più onerosa dal punto di vista dell'interazione.

La terza schermata mostra invece il livello di modifica fine del programma, attraverso l'editing dei singoli slot temporali di una giornata. Per ciascun intervallo è possibile specificare ora di inizio, ora di fine e temperatura obiettivo, oltre a disattivare o rimuovere gli slot esistenti. In questo modo il frontend combina due modalità complementari: una creazione rapida orientata alla semplicità e una configurazione dettagliata orientata alla personalizzazione. La sezione *Programmi* evidenzia quindi il ruolo del client non solo come strumento di controllo real-time, ma anche come interfaccia di pianificazione e gestione persistente del comportamento dell'impianto.

6.9.8 Automazioni dei dispositivi e schedulazione delle azioni

Accanto ai programmi di temperatura, l'applicazione supporta anche la definizione di automazioni per i dispositivi, ossia regole temporali che consentono di eseguire azioni programmate su attuatori e oggetti dell'impianto. Questa funzionalità estende il ruolo del frontend oltre il controllo manuale immediato, permettendo di configurare comportamenti ricorrenti nel tempo secondo una logica di pianificazione semplice ma flessibile.

La Figura 6.5 mostra tre momenti rappresentativi della gestione delle automazioni. La prima schermata presenta l'elenco dei programmi disponibili, nella quale l'utente può filtrare le automazioni in base allo stato o al tipo di azione, visualizzare i dettagli essenziali di ciascuna regola e accedere rapidamente alla creazione di una nuova schedulazione tramite il pulsante flottante dedicato.

La seconda schermata mostra il modale di creazione di una nuova automazione. In questa vista l'utente può definire il nome del programma, scegliere l'azione da eseguire, specificare l'orario e selezionare i giorni della settimana in cui la regola deve essere applicata. L'interfaccia raccoglie in un'unica finestra contestuale gli elementi essenziali della schedulazione, privilegiando una configurazione rapida e immediata.

La terza schermata documenta invece la fase di assegnazione dell'automazione ai dispositivi dell'impianto. In questo caso la UI consente di passare dalla sola definizione astratta della regola alla sua applicazione concreta su categorie come luci, prese o tapparelle. Questo passaggio è rilevante perché mostra che la funzionalità di automazione non si esaurisce nella creazione di un programma temporale, ma include anche la selezione esplicita dei dispositivi destinatari dell'azione pianificata.

Nel complesso, questa sezione evidenzia un'ulteriore dimensione del client Smartotum: oltre alla visualizzazione dello stato corrente e al controllo manuale dei dispositivi, il frontend consente infatti di definire comportamenti automatici persistenti

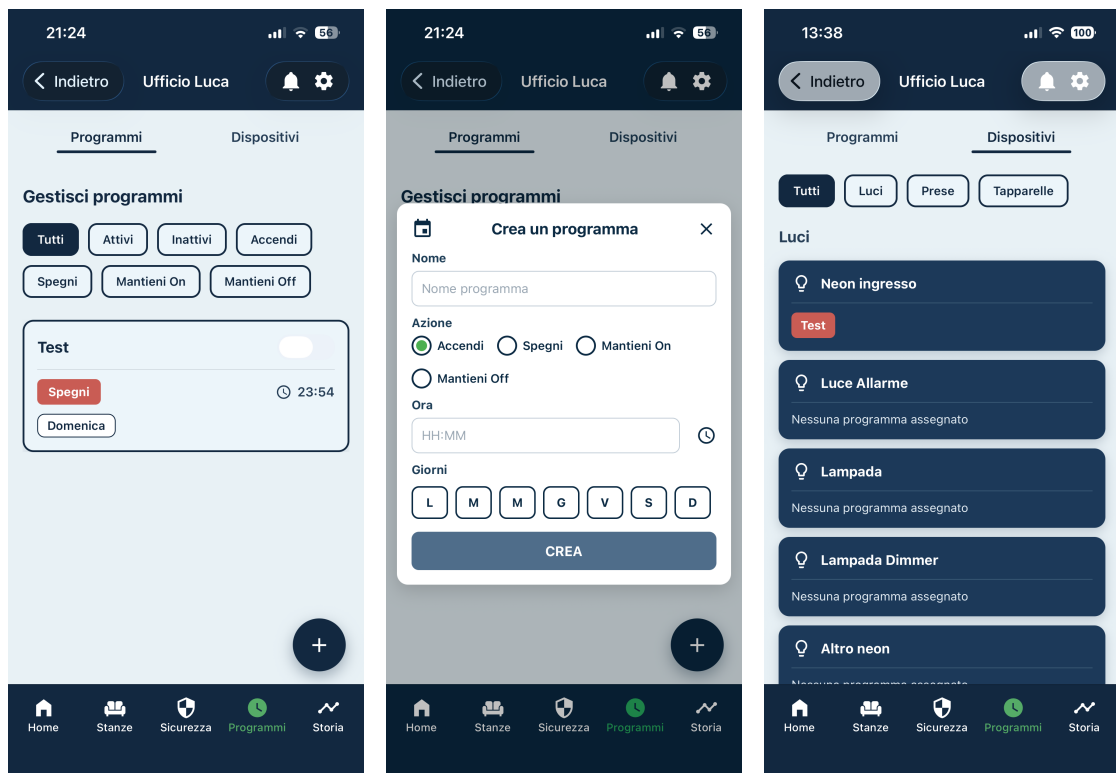


Figura 6.5: Sezione *Programmi*: gestione delle automazioni, creazione di una regola e assegnazione ai dispositivi

e di collegarli operativamente agli attuatori del sistema.

6.9.9 Cronologia e visualizzazione grafica dei sensori

La sezione *Storia* consente di consultare l'andamento temporale delle misure rilevate dai sensori, aggiungendo al controllo real-time una prospettiva storica e analitica. Questa funzionalità è particolarmente utile per osservare l'evoluzione di grandezze come energia, potenza o altri parametri ambientali, superando la sola visualizzazione dello stato istantaneo.

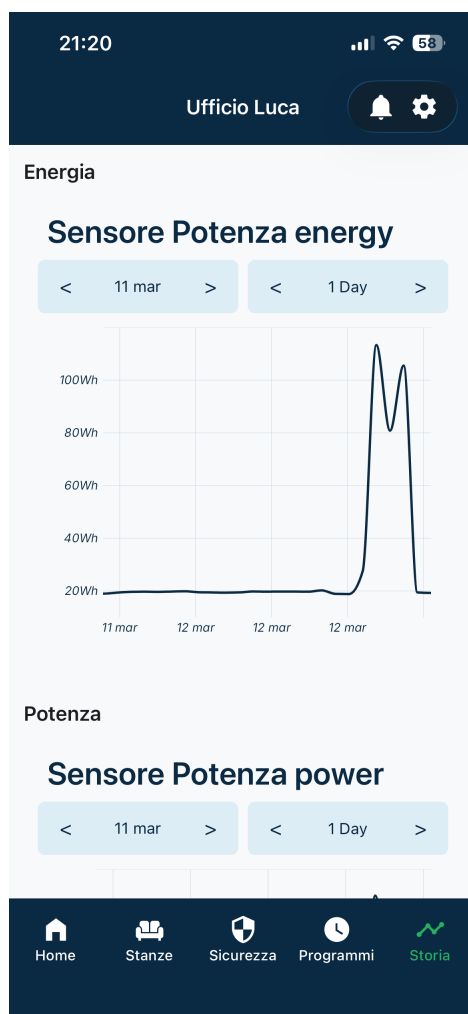


Figura 6.6: Visualizzazione storica dei dati dei sensori tramite grafici temporali

La Figura 6.6 mostra una schermata della sezione dedicata alla cronologia, nella quale i dati di sensori energetici vengono rappresentati attraverso grafici temporali. L'interfaccia consente di selezionare l'intervallo di osservazione e di navigare nel tempo, rendendo possibile il confronto tra valori e l'identificazione di variazioni significative nell'andamento delle misure.

Questa schermata completa in modo naturale le sezioni orientate al controllo immediato dei dispositivi. Mentre Home, Stanze e Sicurezza privilegiano il monitoraggio corrente e l'azione diretta, la sezione *Storia* offre uno strumento di lettura retrospettiva del comportamento dell'impianto. In tal modo, il client non si limita a fungere da interfaccia di comando, ma supporta anche attività di osservazione e interpretazione dei dati nel tempo.

6.9.10 Impostazioni applicative e selezione del sistema

La schermata delle impostazioni raccoglie le principali funzioni di configurazione trasversale dell'applicazione, separandole dalle aree operative di controllo domotico. In questa sezione l'utente può accedere a sottofunzioni come la gestione degli utenti, la configurazione di rete, gli aggiornamenti, le impostazioni dell'applicazione, la gestione dei videocitofoni, dei tastierini e il logout. Tra queste opzioni rientrano anche le *App Settings* e la gestione del sistema attivo, coerentemente con il supporto multi-sistema descritto nei capitoli precedenti.

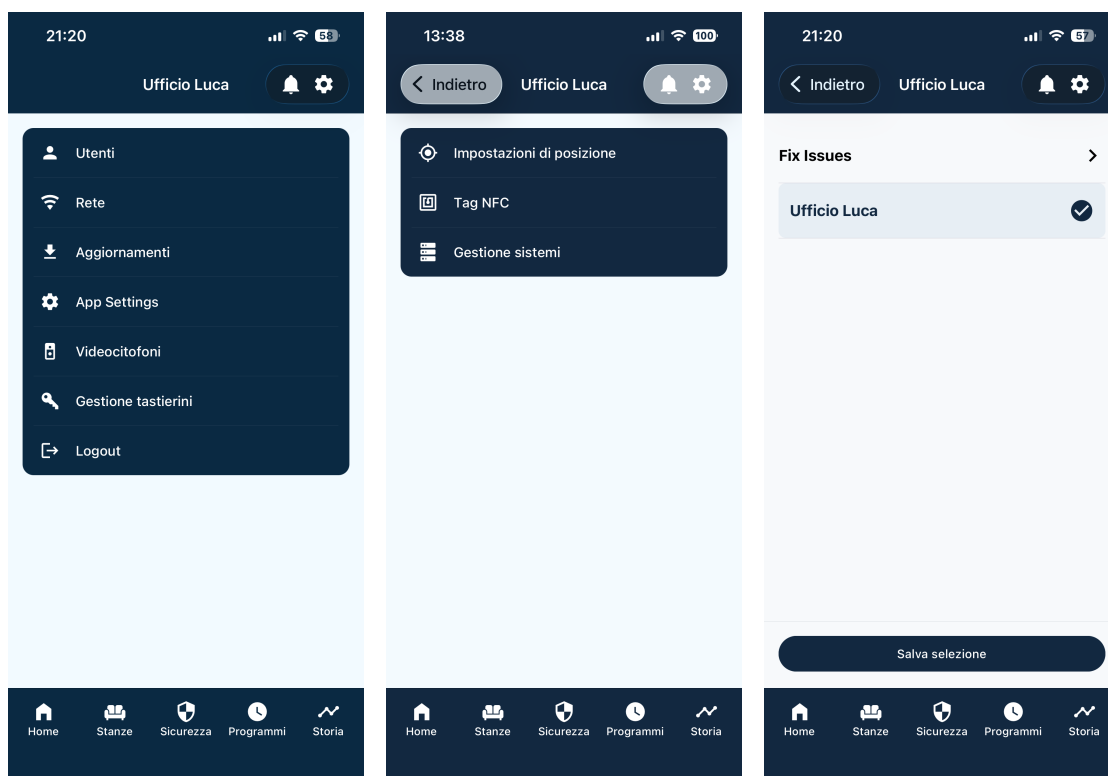


Figura 6.7: Sezione *Impostazioni*, dettaglio delle *App Settings* e selezione del sistema attivo

La Figura 6.7 mostra tre viste rappresentative dell'area di configurazione dell'applicazione. La prima schermata presenta il menu principale delle impostazioni, organizzato come punto di accesso alle funzionalità di amministrazione e configurazione trasversale. Questa vista evidenzia come il client non sia limitato al solo controllo dei dispositivi, ma includa anche aree dedicate alla manutenzione del contesto applicativo e alla gestione degli aspetti persistenti del sistema.

La seconda schermata approfondisce il sottoinsieme delle *App Settings*, nel quale sono raccolte funzionalità applicative specifiche, come le impostazioni di posizione, la gestione dei tag NFC e la gestione dei sistemi. Questa articolazione mostra che l'area impostazioni non è monolitica, ma organizza le configurazioni per livelli successivi, distinguendo tra opzioni generali e sottofunzioni più specialistiche.

La terza schermata mostra la selezione del sistema attivo, accessibile all'interno delle impostazioni dell'app. Qui l'utente può scegliere esplicitamente l'impianto sul quale operare e confermare il contesto attivo. La presenza di questa vista è particolarmente rilevante in un'applicazione che supporta più sistemi associati allo stesso utente, poiché rende visibile e controllabile un aspetto che incide direttamente su autenticazione, stato real-time e operatività complessiva del client.

Nel complesso, l'accostamento delle tre schermate mette in evidenza una dimensione meno immediatamente visibile, ma centrale nell'esperienza d'uso dell'applicazione: oltre alle sezioni di comando e monitoraggio, il frontend fornisce infatti strumenti espliciti per amministrare il contesto applicativo, personalizzare il comportamento dell'app e selezionare in modo consapevole il sistema su cui operare.

6.9.11 Notifiche ed elenco degli eventi

La sezione delle notifiche raccoglie gli eventi generati dal sistema e consente all'utente di consultarli in forma ordinata e filtrabile. Questa vista completa le sezioni di controllo e configurazione dell'applicazione introducendo una dimensione orientata alla memoria operativa del sistema, utile per ricostruire eventi passati, verificare anomalie e mantenere traccia delle attività significative rilevate dall'impianto.

La Figura 6.8 mostra la schermata dedicata alle notifiche, nella quale gli eventi sono organizzati in una lista cronologica e possono essere filtrati secondo intervalli temporali predefiniti, come *oggi*, *ieri*, *ultimi 7 giorni* o eventi più vecchi. Ogni elemento della lista riporta una descrizione sintetica dell'evento e il relativo timestamp, consentendo all'utente di individuare rapidamente il tipo di anomalia o di segnalazione verificatasi.

Nel caso mostrato, la schermata documenta notifiche relative all'attivazione dell'allarme su diversi sensori, come porta e finestre. La presenza di filtri e paginazione evidenzia che la sezione non è pensata come semplice elenco statico, ma come strumento di consultazione progressiva degli eventi nel tempo. Questa

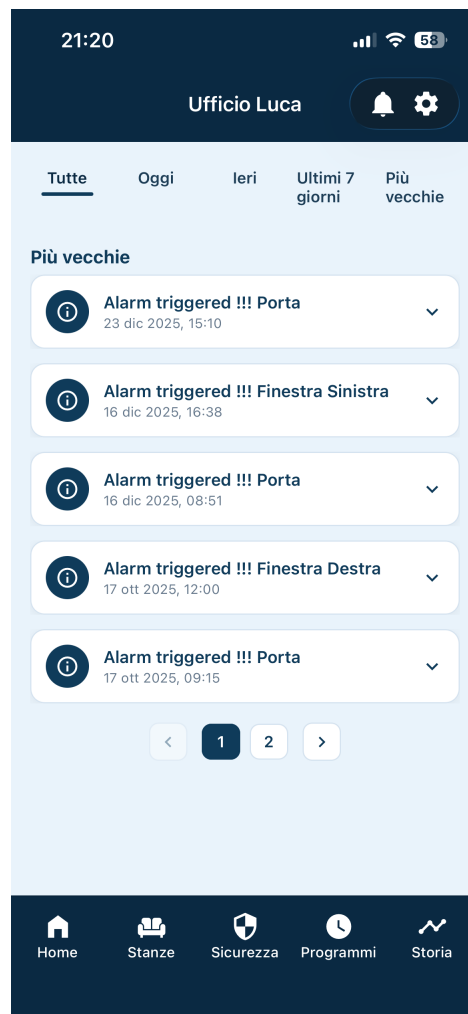


Figura 6.8: Sezione delle notifiche con filtri temporali e lista paginata degli eventi

funzionalità risulta particolarmente utile per correlare i comportamenti osservati nelle sezioni operative con accadimenti già avvenuti, rafforzando il ruolo del client anche come interfaccia di supervisione e analisi degli eventi del sistema.

6.10 Trade-off implementativi e considerazioni critiche

6.10.1 Centralizzazione vs modularità nel layer API

La centralizzazione delle chiamate in `API.ts` produce un vantaggio immediato in termini di uniformità della gestione `auth/refresh/errori`. Tuttavia, con la crescita del numero di endpoint e domini funzionali (Smartotum, Home Assistant REST, push, location, configurazioni), il modulo tende a diventare esteso e meno leggibile.

Il trade-off qui è classico:

- **pro:** policy comuni in un unico punto;
- **contro:** rischio di accoppiamento elevato e minore separazione per dominio.

Nel contesto della tesi, questa scelta è comunque giustificabile come compromesso pragmatico, soprattutto se il comportamento trasversale (`refresh`, `timeout`, `unauthorized`) è il requisito più critico da mantenere uniforme.

6.10.2 Complessità introdotta dal doppio livello di token e dal multi-sistema

La combinazione di:

- token BFF in token store nativo per sistema;
- metadati e chiavi applicative in encrypted storage;
- lista sistemi e metadati server in SQLite;
- supporto a più sistemi con switch e logout selettivo

fornisce una soluzione potente ma non banale. Il vantaggio è un supporto concreto alla continuità d'uso e ai casi d'uso multi-sistema; lo svantaggio è una maggiore complessità dei flussi di recovery e cleanup.

In particolare, switch di sistema, refresh concorrenti, riconnessione WS e attività native in background devono condividere una nozione coerente di “sessione attiva”, condizione che richiede una progettazione attenta dei confini tra stato React, storage e bridge nativi.

6.10.3 Robustezza del provider real-time e costo della complessità

Il `HomeAssistantProvider` integra connessione, heartbeat, backoff, gestione unauthorized, refresh e riconnessione. Questa ricchezza funzionale rende il provider un componente strategico ma anche intrinsecamente complesso.

Dal punto di vista progettuale, si tratta di un compromesso ragionevole: centralizzare la complessità del real-time in un singolo componente è preferibile rispetto a distribuirla nelle schermate. Tuttavia, la complessità interna del provider richiede particolare attenzione in termini di:

- leggibilità del codice;
- gestione degli stati intermedi;
- prevenzione di race condition;
- test e logging diagnostico.

L'aggiornamento delle entità avviene inoltre in forma semplice (`setEntities` su payload ricevuto), soluzione efficace per mantenere il client chiaro, ma con potenziali limiti di efficienza se i payload di snapshot risultano frequenti o voluminosi.

6.10.4 Valutazione complessiva delle scelte lato client

Nel complesso, il frontend implementato mostra una direzione coerente con gli obiettivi della tesi:

- **codebase unica** per Android e iOS nella parte predominante dell'app;
- **perimetro nativo mirato** per secure storage e location/background;
- **gestione robusta della sessione** lato client, integrata con REST e WebSocket;
- **separazione ragionevole** tra UI, sessione, networking e real-time.

Le principali complessità residue (modulo API esteso, provider real-time articolato, coordinamento JS/native) non rappresentano una contraddizione rispetto alla scelta cross-platform, ma piuttosto la naturale conseguenza del dominio applicativo e dei requisiti di resilienza e continuità operativa.

Capitolo 7

Test e validazione

7.1 Obiettivi e criteri di validazione

La soluzione sviluppata combina un client mobile, un Backend for Frontend e l'integrazione con servizi esterni. Di conseguenza, le attività di verifica non possono limitarsi a testare singole funzioni isolate: risultano cruciali anche i flussi che attraversano più componenti (autenticazione, refresh e retry, connessione real-time, recovery da disconnessioni, invio eventi in background).

Gli obiettivi principali della validazione sono stati:

- verificare la correttezza dei flussi critici di sessione (login, refresh, logout) e la loro robustezza a token scaduti;
- verificare il canale WebSocket `/ha/stream` in condizioni normali e in condizioni degradate (assenza token, token invalido, disconnessione e riconnessione, rate limiting);
- verificare le principali integrazioni REST verso Home Assistant e Smartotum e la gestione coerente degli errori (timeout, validazione input);

Per raggiungere tali obiettivi è stata adottata una strategia mista:

1. **test automatici server-side** (unit e integration) per coprire le componenti core del BFF e le sue interfacce HTTP/WS;
2. **validazione funzionale ripetibile** tramite scenari end-to-end sui flussi che coinvolgono anche il client e/o funzionalità native in background.

7.2 Strumenti e perimetro dei test automatici

7.2.1 Framework e organizzazione della suite

La suite automatizzata è concentrata sul BFF ed è realizzata con **Jest**. La strategia di test è articolata in due livelli complementari:

Unit test. I test unitari verificano componenti isolati e funzioni core del BFF, con l'obiettivo di controllare la correttezza della logica interna senza dipendere da rete o servizi esterni. In questa categoria ricadono tipicamente: gestione della sessione, utilità HTTP (timeout/abort e parsing), funzioni crittografiche del vault e logiche di supporto del WebSocket manager.

Integration test. I test di integrazione verificano il comportamento del BFF tramite le sue interfacce pubbliche, simulando l'uso da parte del client e controllando che i flussi principali funzionino correttamente quando più componenti collaborano tra loro (routing, middleware di autenticazione, accesso a database di test, mapping degli errori). In particolare, gli integration test coprono:

- le API di **sessione** del BFF (`/auth/login`, `/auth/refresh`, `/auth/logout`);
- le chiamate esposte dal BFF verso **Smartotum** (`/smartotum/*`);
- le chiamate esposte dal BFF verso **Home Assistant** via REST (`/ha/*`);
- la connessione e il protocollo applicativo del canale **WebSocket** (`/ha/stream`).

7.2.2 Comandi e riproducibilità

La suite di test del BFF è eseguibile tramite script **npm** e utilizza una configurazione d'ambiente dedicata ai test. In sintesi:

- esecuzione completa della suite: `npm test`;
- esecuzione dei soli integration test: `npm run test:integration`;
- modalità watch e raccolta coverage: `npm run test:watch`, `npm run test:coverage`.

L'esecuzione dei test non richiede l'avvio manuale del server: la suite inizializza un ambiente isolato e invoca le route e/o il WebSocket in modo controllato, utilizzando configurazione dedicata e database di test.

7.3 Ambiente di test

7.3.1 Configurazione e variabili d'ambiente

I test del BFF utilizzano una configurazione d'ambiente dedicata (`.env.test`) che contiene:

- configurazione del database di test;
- parametri di base per gli upstream (URL e identificativi necessari ai flussi);
- chiavi e parametri necessari per il vault (cifratore a riposo);
- chiave condivisa per webhook push, dove applicabile.

Questa scelta consente di ripetere la suite in modo deterministico, evitando interferenze con l'ambiente di sviluppo.

7.3.2 Database di test e gestione dello stato

Il database usato per i test è un SQLite dedicato, separato da eventuali dati di sviluppo. Nei test di integrazione vengono eseguite operazioni di cleanup sulle tabelle rilevanti al termine dei casi di test, in modo da evitare dipendenze tra test e rendere ripetibile l'esecuzione anche in ordine diverso (ad esempio in watch mode).

7.3.3 Mock e stub degli upstream

Una parte importante della suite unit riguarda moduli che dipendono da servizi esterni. Per evitare dipendenze non deterministiche, nei test unitari sono presenti mock/stub di:

- layer HTTP (timeout/abort, parsing risposta);
- accesso al database via Prisma;
- librerie di connessione verso Home Assistant;
- funzioni di gestione token upstream.

Nei test di integrazione, i mock vengono usati in modo più mirato (ad esempio con `spyOn`) per simulare specifici failure mode senza sostituire completamente l'infrastruttura di test.

7.4 Copertura dei test automatici sul BFF

7.4.1 Unit test

I test unitari coprono le aree più sensibili del BFF:

- **gestione sessione** e lifecycle del token BFF (creazione, validazione, rotazione, revoca);
- **vault/cifratura** e funzioni crittografiche, fondamentali per la protezione a riposo dei token upstream;
- **DAO** di Smartotum e Home Assistant (validazioni, costruzione richieste, trattamento errori);
- **utility HTTP** (timeout, parsing e mapping degli errori);
- **WebSocket manager** (logica applicativa, rate limiting, gestione messaggi).

La scelta di concentrarsi su queste aree deriva dal fatto che un errore in tali componenti tende ad avere impatto sistemico: ad esempio un bug nella rotazione del refresh token o nella gestione del timeout può compromettere molte funzionalità, anche se le singole route sono corrette.

7.4.2 Integration test (HTTP e WebSocket)

I test di integrazione verificano il comportamento del BFF attraverso le sue interfacce pubbliche, coprendo i flussi principali che attraversano routing, middleware di autenticazione, accesso al database e gestione degli errori. Le aree coperte sono quattro:

Sessione BFF. Le API di sessione sono esposte sotto `/auth/*`. Verifica del ciclo di vita della sessione: creazione tramite login, rotazione tramite refresh e revoca tramite logout, includendo i casi limite più comuni (token mancanti o non validi, sessione revocata, scadenze).

Proxy Smartotum. Le API proxy sono esposte sotto `/smartotum/*`. Verifica che le route proxy verso Smartotum siano correttamente protette, che inoltrino le richieste con le credenziali appropriate e che gli errori upstream vengano mappati in risposte coerenti verso il client (inclusi timeout e payload di errore).

Home Assistant REST. Le API REST verso Home Assistant sono esposte sotto `/ha/*`. Verifica delle principali route REST, con attenzione a validazioni input, protezione via sessione BFF e trattamento uniforme delle anomalie (errori upstream, timeout).

WebSocket real-time. Il canale real-time è esposto su `/ha/stream`. Verifica del comportamento del WebSocket: handshake autenticato, rifiuto della connessione in assenza di token o con token invalido, gestione del protocollo base e close code nei casi di autorizzazione non valida.

7.5 Validazione funzionale tramite scenari ripetibili

Accanto alla suite automatica, è stata condotta una validazione funzionale basata su scenari ripetibili, costruiti per coprire i flussi critici dal punto di vista dell'utente e del comportamento del sistema in condizioni reali (rete instabile, token scaduti, recovery del canale real-time, attività in background). Gli scenari sono stati definiti in modo da poter essere eseguiti:

- tramite app mobile (dove coinvolgono UI o moduli nativi);
- tramite chiamate controllate (ad esempio `curl` o strumenti analoghi) quando è sufficiente verificare l'interfaccia del BFF;
- tramite osservazione del comportamento HTTP/WS e dei log del BFF.

7.5.1 Scenari su sessione e continuità operativa (A–C)

I primi scenari verificano la correttezza del ciclo di vita della sessione:

- **A – Login e creazione sessione:** verifica risposta del login con token e scadenze, e corretta persistenza lato app;
- **B – Refresh e retry REST:** verifica che una richiesta fallita per 401 inneschi un refresh (single-flight) e un secondo tentativo;
- **C – Logout e revoca:** verifica della revoca sessione e del rifiuto di richieste successive con token non più valido.

7.5.2 Scenari real-time su WebSocket (D–G)

Gli scenari D–G verificano il comportamento del canale `/ha/stream` sia in condizioni nominali sia in condizioni degradate:

- **D – Connessione, subscribe, snapshot e delta:** verifica del messaggio `ready`, della `subscribe` e della ricezione di aggiornamenti;
- **E – Recovery dopo disconnessione:** verifica della riconnessione automatica con `backoff` e del `resubscribe`;
- **F – Token scaduto e recovery:** verifica del trattamento di `unauthorized/close` code e della sequenza `refresh + reconnect`;
- **G – Rate limiting:** verifica del comportamento del server quando il client invia un numero eccessivo di messaggi in una finestra temporale.

Questi scenari sono importanti perché il real-time è uno dei requisiti distintivi del progetto e, in ambito mobile, è anche una delle principali fonti di problemi operativi (half-open connection, rete che cambia, sessioni scadute durante la connessione persistente).

7.5.3 Scenari su integrazione REST e gestione errori (H)

Lo scenario H è focalizzato su robustezza e corretto trattamento delle anomalie:

- **H – Timeout upstream:** verifica che un timeout o un abort verso un servizio esterno venga mappato in un errore coerente (codice e payload atteso), e che il client possa gestire in modo prevedibile l'esito dell'operazione.

7.5.4 Tabella riassuntiva degli scenari

La Tabella 7.1 riassume gli scenari validati e la componente primaria coinvolta.

Tabella 7.1: Scenari di validazione ripetibili: sintesi e componente primaria.

ID	Scopo	Componente primaria
A	Login e creazione sessione BFF	BFF (Auth) + client (persistenza token)
B	Refresh token e retry di richiesta protetta	Client (API layer) + BFF (refresh)
C	Logout e revoca sessione	BFF (revoke) + client (cleanup)
D	WS connect/subscribe + snapshot/delta	WS BFF + client WS
E	WS recovery dopo disconnessione	Client WS (reconnect/backoff)
F	WS unauthorized: refresh + reconnect	WS BFF + client (refresh/reconnect)
G	Rate limiting WS	WS BFF
H	Timeout upstream e mapping errore	BFF (error mapping)

7.6 Evidenze e osservabilità

Per rendere verificabili i risultati dei test e degli scenari, sono state utilizzate evidenze osservabili in modo diretto:

- codici e payload di errore coerenti (ad esempio `TIMEOUT`);
- comportamento atteso dei messaggi WebSocket (`ready`, `entities_update`, `unauthorized`);
- log applicativi lato server in casi di rate limit o failure mode;
- retry automatico in caso di 401 sia nel layer API del client sia nel bridge nativo per la location.

L'uso di messaggi e codici applicativi distinti semplifica la diagnosi: un errore di timeout, una revoca di sessione e una disconnessione di rete producono effetti osservabili diversi, facilitando la comprensione del problema e la sua riproduzione.

7.6.1 Discussione dei gap di copertura

La Figura 7.1 mostra valori complessivamente elevati per tutte le metriche considerate, con il 95.01% degli statement, l'88.18% dei branch, il 98.28% delle funzioni e

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	95.01	88.18	98.28	96.02	
server	96.82	86	95	99.09	
Errors.ts	94.73	90.47	100	100	22,44
Http.ts	86.66	60	66.66	92.3	21
index.ts	98.91	87.5	100	100	48,78,107
server/dao	91.29	88.47	100	94.67	
HomeAssistantDao.ts	100	100	100	100	
SmartotumDao.ts	87.63	79.47	100	92.36	... 46,466,522,542,564,575-576,585,590-591,606-607,682,730,750,770,792,814,841,872,888,903
server/lib	100	100	100	100	
Crypto.ts	100	100	100	100	
Db.ts	100	100	100	100	
Session.ts	100	100	100	100	
Vault.ts	100	100	100	100	
upstream.ts	100	100	100	100	
server/routes	99.52	98.36	100	99.41	
Auth.ts	100	100	100	100	
HomeAssistant.ts	97.29	94.33	100	96.84	86,128,145
Smartotum.ts	100	100	100	100	
server/validators	94.16	89.11	100	94.07	
Auth.ts	100	100	100	100	
HomeAssistant.ts	97.1	94.82	100	97.1	18,124
Smartotum.ts	91.82	85.79	100	91.66	7,15,29,51,72,91,111,135,167,178,204,225,284
server/wsConnection	92.69	77.35	94.91	93.33	
HaConnection.ts	91.88	75.98	95.23	92.91	... 28,136-137,157,176-178,200,257-258,295,309-310,372-373,418-422,425-426,466-467,479,521
Manager.ts	94.41	81.96	94.11	94.18	62-63,244-245,256-257,292-293,310-311
===== Coverage summary =====					
Statements	: 95.01% (2323/2445)				
Branches	: 88.18% (955/1083)				
Functions	: 98.28% (229/233)				
Lines	: 96.02% (2103/2190)				
=====					
Test Suites:	20 passed, 20 total				
Tests:	444 passed, 444 total				
Snapshots:	0 total				
Time:	24.38 s				

Figura 7.1: Coverage dei test

il 96.02% delle linee coperti dai test. Tali valori indicano che la logica principale del sistema risulta ampiamente esercitata dalla suite di test e che i percorsi nominali più rilevanti sono stati verificati in modo sistematico.

L'analisi dei rami non coperti mostra tuttavia che la quota residua di branch coverage non riguarda in larga misura funzionalità di base assenti dai test, bensì soprattutto percorsi difensivi, condizioni di validazione, blocchi di `catch`, fallback e scenari non nominali. In particolare, una parte dei gap è concentrata in componenti come i validator, alcune funzioni di accesso ai servizi upstream e i percorsi di gestione degli errori HTTP, cioè in sezioni di codice che vengono attivate principalmente in presenza di input non validi, timeout, errori remoti o condizioni eccezionali.

Anche nei moduli con copertura dei branch relativamente più bassa, come quelli legati alla gestione delle connessioni WebSocket, i rami mancanti si collocano prevalentemente nei meccanismi di recovery, cleanup, riconnessione e gestione di failure, più che nel funzionamento ordinario del canale real-time. Questo significa che i test coprono in modo soddisfacente il comportamento atteso del sistema in condizioni operative normali, mentre restano meno esercitati alcuni percorsi di robustezza che si manifestano solo in condizioni degradate o anomalie di integrazione.

Di conseguenza, il valore dell'88.18% di branch coverage va interpretato in modo qualitativo oltre che quantitativo: la parte non coperta non si concentra sui flussi applicativi centrali, ma soprattutto su casi limite e comportamenti difensivi. Ciò non ne annulla la rilevanza, poiché tali rami contribuiscono alla resilienza complessiva del sistema, ma consente di affermare che la copertura raggiunta

rappresenta comunque un risultato solido rispetto agli obiettivi della tesi. Un possibile sviluppo futuro consiste nell'estendere ulteriormente i test verso scenari di failure controllata, in particolare per i meccanismi di riconnessione e gestione delle condizioni di errore nei sottosistemi real-time.

Capitolo 8

Conclusioni

La tesi ha affrontato la modernizzazione dell'applicazione mobile Smartotum partendo da una situazione di frammentazione tecnologica, in cui funzionalità e logiche erano distribuite tra codebase differenti (Android nativo, iOS nativo e componente web). Tale assetto comportava duplicazione della logica applicativa, maggior costo di manutenzione e rischio di divergenze funzionali tra piattaforme, criticità particolarmente rilevanti in un dominio smart home che richiede reattività, continuità operativa e gestione sicura delle credenziali.

Il lavoro ha portato alla definizione e realizzazione di una soluzione composta da due elementi principali: un client mobile cross-platform basato su React Native e un Backend for Frontend (BFF) in Express/TypeScript. La convergenza su una singola codebase mobile consente di ridurre la duplicazione e mantenere allineati i flussi fondamentali dell'app (autenticazione, selezione del sistema, navigazione e controllo). Al tempo stesso, è stato mantenuto un perimetro nativo mirato per quelle funzionalità che, per vincoli di piattaforma o requisiti di sicurezza, richiedono una gestione OS-level, in particolare la persistenza sicura dei token e l'invio della posizione in scenari di background.

L'introduzione del BFF rappresenta la scelta architetturale più significativa: il backend non viene utilizzato come semplice proxy, ma come livello dedicato al canale mobile che realizza un *boundary* stabile verso i servizi esterni (Smartotum API e Home Assistant). In questo modo il client interagisce con un contratto coerente e mobile-oriented, mentre il BFF centralizza responsabilità trasversali come autenticazione e gestione sessione, normalizzazione degli errori, timeout e mediazione dei protocolli. La separazione delle credenziali segue lo stesso principio: il client opera con token BFF a durata controllata conservati in storage nativo sicuro, mentre le credenziali verso i servizi esterni sono custodite lato server e protette a riposo. Questo impianto migliora la governabilità della soluzione e riduce l'esposizione di dati sensibili sul dispositivo.

Un ulteriore punto qualificante riguarda il real-time. La soluzione adotta un

canale WebSocket mediato dal BFF per lo streaming dello stato delle entità e l'inoltro dei comandi verso Home Assistant. Il real-time, in ambito mobile, è uno degli aspetti più delicati per via della rete variabile e del ciclo di vita dell'app; per questo motivo la progettazione include meccanismi di connessione autenticata, subscribe allo stream, aggiornamenti, gestione delle disconnessioni e strategie di recovery. Nel complesso, l'architettura ottenuta consente di conciliare l'esperienza reattiva richiesta dalla domotica con la necessità di mantenere controllabili sicurezza e integrazione.

La validazione è stata condotta combinando test automatici lato BFF (unit e integration) e scenari ripetibili sui flussi critici. I test unitari coprono le componenti più sensibili del backend, mentre gli integration test verificano il comportamento del BFF attraverso le sue interfacce pubbliche (API di sessione, chiamate verso Smartotum e Home Assistant, e connessione WebSocket). Gli scenari end-to-end consentono inoltre di osservare il comportamento complessivo nei casi che tipicamente generano instabilità su mobile, come token scaduti, retry e riconnessioni.

Nel bilancio complessivo, il contributo della tesi non consiste soltanto nell'adozione di React Native, ma nella combinazione coerente di scelte client-side e server-side che permettono di ridurre la frammentazione e rendere più sostenibile l'evoluzione del prodotto. La soluzione separa chiaramente responsabilità, centralizza nel BFF le policy trasversali e mantiene sul client un perimetro nativo ridotto ma sufficiente a rispettare i vincoli delle piattaforme.

8.1 Bilancio della convergenza architetturale

Un aspetto importante del lavoro svolto riguarda il bilancio concreto della riduzione della frammentazione iniziale. Come discusso nei capitoli introduttivi, il punto di partenza era costituito da una configurazione multi-codebase, nella quale funzionalità e responsabilità applicative risultavano distribuite tra una componente Android nativa in Kotlin, una componente iOS nativa in Swift e una componente web sviluppata con Vue. Un simile assetto comportava duplicazione della logica, maggior costo di manutenzione e rischio di divergenze funzionali tra piattaforme.

La soluzione realizzata in questa tesi ha consentito di ricondurre una parte significativa di tale complessità entro un'architettura più coerente. In particolare, la convergenza non consiste in una unificazione indiscriminata di tutti i componenti del sistema Smartotum, bensì nella ricollocazione delle responsabilità nei punti in cui possono essere gestite in modo più controllabile, manutenibile e sicuro. Il client React Native concentra i flussi applicativi mobili condivisi, mentre il BFF centralizza la mediazione verso i servizi esterni e le policy trasversali di sessione, autenticazione e gestione degli errori.

Tabella 8.1: Bilancio sintetico della convergenza architetturale

Aspetto	Situazione iniziale	Situazione dopo l'intervento
Client mobile	Due codebase native separate (Kotlin per Android e Swift per iOS)	Una codebase mobile condivisa basata su React Native
Flussi principali dell'app	Logiche replicate o da mantenere coerenti su più implementazioni	Logiche condivise per autenticazione, selezione del sistema, navigazione e controllo
Gestione della sessione	Maggiore rischio di duplicazione di refresh, retry e gestione errori	Comportamento più uniforme, con responsabilità distribuite in modo esplicito tra client e BFF
Integrazione con servizi esterni	Complessità esposta direttamente al client	Mediazione centralizzata nel BFF verso Smartotum API e Home Assistant
Credenziali e sicurezza	Maggiore esposizione della complessità di accesso lato dispositivo	Token BFF sul client e credenziali verso gli upstream custodite lato server
Perimetro nativo	Più esteso e intrecciato con la logica applicativa	Limitato a funzionalità OS-level mirate

Dal punto di vista architetturale, sono quindi stati unificati soprattutto i seguenti elementi:

- la codebase del client mobile;
- i flussi applicativi principali condivisi tra Android e iOS;
- la gestione della UI e della logica di presentazione;
- il punto di accesso ai servizi esterni per il canale mobile;
- una parte rilevante delle policy trasversali di autenticazione, sessione, timeout, normalizzazione degli errori e mediazione del canale real-time.

Allo stesso tempo, alcune responsabilità sono rimaste separate in modo intenzionale. Questa separazione residua non rappresenta un limite del lavoro, ma una conseguenza diretta dei vincoli del dominio e delle piattaforme. In particolare, restano distinti:

- i moduli nativi necessari per funzionalità strettamente dipendenti dal sistema operativo, come la persistenza sicura dei token e la gestione della location in background;
- il livello BFF, che costituisce un boundary architetturale esplicito tra il client mobile e i servizi esterni;
- i sistemi upstream, ossia Smartotum API e Home Assistant, che rimangono componenti autonomi con protocolli, vincoli e cicli di vita propri.

Ne consegue che il contributo della tesi può essere letto come una *razionalizzazione dei confini* del sistema. Ciò che poteva essere condiviso senza compromettere sicurezza, manutenibilità o aderenza ai vincoli di piattaforma è stato unificato; ciò che invece richiede isolamento tecnico o integrazione OS-level è stato mantenuto separato ma confinato entro interfacce più chiare. In questo senso, il risultato non è soltanto la migrazione verso React Native, ma la trasformazione di un'applicazione frammentata in una soluzione con responsabilità meglio distribuite, più prevedibile in evoluzione e più sostenibile nel medio periodo.

Sviluppi futuri

La direzione di sviluppo più rilevante consiste nel portare l'applicazione in **release** per consentirne l'adozione operativa. In una prima fase, l'app può essere pubblicata come soluzione che **affianca** quella esistente, permettendo:

- una migrazione graduale degli utenti e delle funzionalità;
- la raccolta di feedback e telemetria reale su stabilità, performance percepita e affidabilità del real-time;
- la verifica sul campo dei flussi più critici (sessione, refresh, riconnessione WebSocket, comportamento in background).

In una fase successiva, una volta consolidata stabilità e copertura funzionale, l'app potrà **sostituire** progressivamente la soluzione precedente, riducendo definitivamente i costi derivanti dalla gestione multi-codebase.

Accanto alla messa in release, ulteriori sviluppi utili includono: estendere la copertura di test lato client (in particolare su flussi di sessione e provider real-time), introdurre una pipeline di integrazione continua per l'esecuzione automatica dei test del BFF e controlli statici, e potenziare l'osservabilità (log strutturati, metriche minime su riconnessioni e timeouts, correlazione tra richieste REST e messaggi WebSocket). Questi interventi migliorerebbero la capacità di prevenire regressioni e diagnosticare anomalie in produzione, soprattutto in presenza di rete mobile instabile e dipendenze esterne.

Conclusione

Il lavoro svolto dimostra che la modernizzazione di un'app smart home richiede un approccio architetturale, non soltanto una scelta di framework. La combinazione di una codebase mobile unica, un Backend for Frontend dedicato e un canale real-time mediato consente di migliorare manutenibilità, coerenza e controllabilità della sicurezza. La naturale prosecuzione consiste nel portare la soluzione in release, inizialmente in affiancamento e poi in sostituzione dell'app precedente, completando la transizione verso un prodotto più sostenibile ed evolvibile nel tempo.

Bibliografia

- [1] International Organization for Standardization. *ISO/IEC 25010:2011. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO/IEC standard. 2011 (cit. alle pp. 7, 14).
- [2] Apple Inc. *Human Interface Guidelines*. Online documentation. Accessed 2026. 2026 (cit. a p. 8).
- [3] Google. *Android Developers Documentation*. Online documentation. Accessed 2026. 2026 (cit. a p. 8).
- [4] W3C. *Service Workers*. Technical specification. Accessed 2026. 2026 (cit. a p. 9).
- [5] Luis Corral, Alberto Sillitti e Giancarlo Succi. «Mobile Multiplatform Development: An Experiment for Performance Analysis». In: *Procedia Computer Science* 10 (2012) (cit. a p. 9).
- [6] Luis Corral, Andrea Janes e Tiziano Remencius. «Potential Advantages and Disadvantages of Multiplatform Development Frameworks – A Vision on Mobile Environments». In: *Procedia Computer Science* 10 (2012) (cit. a p. 9).
- [7] Hendrik Heitkötter, Sebastian Hanschke e Tim A. Majchrzak. «Comparing cross-platform development approaches for mobile applications». In: *Proceedings of the 8th International Conference on Web Information Systems and Technologies (WEBIST)*. 2012 (cit. alle pp. 10, 15).
- [8] Christoph Rieger e Bernhard Kuchen. «Towards an evaluation framework for cross-platform development tools». In: *Journal of Systems and Software* (2019) (cit. alle pp. 10, 14).
- [9] React Native Documentation. *About the New Architecture (JSI, Fabric, TurboModules)*. Online documentation. Accessed 2026. 2026 (cit. alle pp. 11, 12).
- [10] React Native Blog. *The New Architecture is here*. Online article. 2024 (cit. a p. 12).

- [11] React Native Documentation. *Turbo Native Modules: Introduction (Codegen, typed specs)*. Online documentation. Accessed 2026. 2026 (cit. a p. 12).
- [12] React Native Documentation. *Using Hermes*. Online documentation. Accessed 2026. 2026 (cit. a p. 12).
- [13] React Native Blog. *Hermes as the Default*. Online article. 2022 (cit. a p. 12).
- [14] Sam Newman. *Backends For Frontends (BFF)*. Architecture pattern reference. 2015 (cit. a p. 16).
- [15] Apple Developer Documentation. *Keychain Services*. Online documentation. Accessed 2026. 2026 (cit. a p. 17).
- [16] Android Developers Documentation. *EncryptedSharedPreferences (AndroidX Security)*. Online documentation. Accessed 2026. 2026 (cit. a p. 17).