

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Università Politecnica delle Marche

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di un'applicazione mobile per il monitoraggio e visualizzazione di dati da sensori di una rete Internet-of-Everything nel contesto dell'Industria 5.0

Relatore:

Prof. Emanuele Storti

Candidato:

Filippo Ercoli

Anno Accademico: 2025-2026

Indice

Capitolo 1: Introduzione 6
Capitolo 2: Raccolta ed Analisi dei Requisiti 8
2.1 Requisiti 8
2.2 Attori 10
2.3 Gestione Utente 10
2.4 Gestione Attività 11
Capitolo 3: Framework 12
3.1 Servizi e API disponibili 13
Capitolo 4: Progettazione 18
4.1 Strumenti di progettazione utilizzati 18
4.2 Mockup 20
4.3 Pattern architetturale MVVM 23
4.4 Diagramma delle componenti 24
4.5 Meccanismo di interazione con le API 25
Capitolo 5: Implementazione 26
5.1 Struttura delle cartelle 26
5.2 Dettagli implementativi principali 27
5.3 Sistema di Autenticazione 31
5.4 Meccanismo di monitoring dei dati 33
5.5 Meccanismo di interrogazione dei dati storici 35
5.6 Sistema di aggiornamento 38
5.7 View 41
5.8 ViewModel 54
5.9 Utils e componenti riusabili 58
5.10 Miglioramenti 59
Conclusioni 60

Indice delle Immagini

1 Requisiti Funzionali e Non Funzionali 8
2 Attori del sistema 10
3 Caso d'Uso: Gestione Utente 10
4 Caso d'Uso: Gestione Attività 11
5 Architettura del framework di monitoraggio basata sul progetto HOMEY 12
6 Esempio di risposta JSON della chiamata POST /login. 14
7 Esempio di risposta JSON della chiamata GET /api/utils. 15
8 Esempio di output dello stream in tempo reale GET /monitoring. . . 16
9 Esempio di output della query storica GET /querying. 17
10 Esempio di risposta JSON dell'endpoint GitHub Releases. 19
11 Esempio di risposta JSON dell'endpoint GitHub Releases. 20
12 Mockup principali: autenticazione e configurazione. 23
13 Mockup: viste di monitoraggio e ricerca dati. 24
14 Mockup: output grafici in diverse tipologie di visualizzazione. . . . 25
15 Schema semplificato dell'interazione View–ViewModel–Model 26
16 Diagramma delle Componenti 27
17 Struttura delle cartelle in Flutter 29
18 Struttura della classe MachineInfo 31
19 Struttura della classe FieldInfo 32
20 Richiesta per il Login 34
21 Richiesta delle macchine associate all'utente 35
22 Streaming service 37
23 Funzione con retry automatico per la gestione degli errori 40
24 Recupero della versione remota tramite GitHub API 42

25 Dialog che notifica la disponibilità di una nuova versione 42
26 Finestra di avanzamento durante il download dell'APK 43
27 Chiamata nativa per installazione dell'APK tramite MethodChannel . 43
28 Schermata Login View 44
29 Controllo dei campi username e password 45
30 TextEditingController della schermata login 45
31 Inizializzazione dei RTViewModel in initState. 46
32 Toolbar con icona di modifica, ordinamento e switch per lo streaming. 46
33 Recupero del RTViewModel 48
34 Schermata RTView. 49
35 Schermata input della QueryView. 50
36 Gestione del booleano _bottomSheetOpen all'apertura. 51
37 Logica di chiusura del booleano _bottomSheetOpen. 51
38 Composizione del DropDownMenuItem per le macchine. 51
39 Composizione del DropDownMenuItem per i parametri. 51
40 Composizione del DropDownMenuItem per le aggregazioni. 52
41 Schermata QueryView. 54
42 Schermata EditView. 55
43 Condizione di visualizzazione grafici. 59

Capitolo 1: Introduzione

Negli ultimi anni, il concetto di Industria 4.0 caratterizzato dall'obiettivo di una produzione intelligente, flessibile e interconnessa, ha subito un'evoluzione, dando nascita all'industria 5.0. Mentre l'industria 4.0 si concentrava principalmente sull'automazione, sull'interconnessione tramite tecnologie IoT e sull'integrazione uomo-macchina, l'Industria 5.0 propone un sistema più avanzato e centrato sull'essere umano, mettendo in risalto la personalizzazione, sostenibilità e collaborazione intelligente tra persone, dispositivi e ambienti connessi. Secondo quanto riportato nel report AI e Fattore Umano a cura di AISS e Par-Tec (2024) [1], l'Industria 5.0 mira a integrare tecnologie avanzate come l'intelligenza artificiale in un'ottica realmente human-centric, in cui l'innovazione sia al servizio della persona, della creatività e del benessere collettivo.

In questo contesto, le applicazioni mobili giocano un ruolo fondamentale, in quanto strumenti di accesso immediato alle informazioni. Il presente lavoro illustra le funzionalità implementate in un'applicazione mobile per il monitoraggio e l'analisi di dati, con l'obiettivo di visualizzare in modo intuitivo informazioni sia in tempo reale che storiche. Inoltre, l'applicazione presenta la possibilità di personalizzare l'interfaccia grafica attraverso metodi di ordinamento delle macchine, la selezione dei colori per i grafici, la scelta della tipologia di grafico da visualizzare e la possibilità di attivare o disattivare il monitoraggio. Questo approccio riflette i principi dell'Industria 5.0, secondo cui l'interazione con i sistemi deve adattarsi ai bisogni dell'utente finale, rendendo le tecnologie più accessibili, inclusive e coinvolgenti.

L'applicazione, sviluppata in Dart tramite l'utilizzo del framework Flutter, è accessibile tramite dispositivi iOS e Android. Progettata per rispondere alle esigenze di utenti con differenti livelli di competenza tecnica, contribuisce così a una maggiore diffusione. Inoltre, le tecniche di implementazione utilizzate rendono l'applicazione flessibile e adattabile. In linea con la visione delineata da AISS e Par-Tec (2024) [1], essa si propone come uno strumento inclusivo, capace di favorire una fruizione consapevole e personalizzata della tecnologia anche da parte di utenti non esperti.

L'obiettivo di questo elaborato è presentare la progettazione e lo sviluppo di un'applicazione mobile per il monitoraggio di macchine e sensori. L'applicazione è il

risultato dell'attività svolta durante il tirocinio curricolare e si inserisce nell'ambito del progetto di ricerca PRIN 2022 HOMEY *"A Human-centric IoE-based Framework for Supporting the Transition Towards Industry 5.0"*.

Struttura del documento:

- Il **Capitolo 2** descrive la raccolta e l'analisi dei requisiti, i casi d'uso, gli attori coinvolti e le funzionalità richieste.
- Il **Capitolo 3** illustra il framework software e i servizi disponibili per l' autenticazione, monitoring e l'accesso ai dati storici.
- Il **Capitolo 4** è dedicato alla progettazione dell'applicazione, dai mockup iniziali ai diagrammi architetturali e ai modelli dati.
- Il **Capitolo 5** analizza l'implementazione concreta dell'app, la struttura delle cartelle, le schermate sviluppate, i ViewModel e le API.

Capitolo 2: Raccolta ed Analisi dei Requisiti

In questo capitolo verrà effettuata un'analisi completa di ciò che dovrà contenere l'applicazione, attraverso l'analisi dei requisiti. Nell'ambito dello sviluppo software la raccolta e l'analisi dei requisiti è un'attività fondamentale, poiché il fine principale è quello di riuscire a descrivere le funzionalità dell'applicazione [2].

2.1 Requisiti

Nel contesto dello sviluppo software, i requisiti servono per descrivere come il sistema deve comportarsi e cosa deve fare. Essi si dividono in due categorie principali (*Figura 1*):

- **Requisiti funzionali**: descrivono le funzionalità offerte dal sistema, ovvero i servizi messi a disposizione degli utenti dagli sviluppatori.
- **Requisiti non funzionali**: definiscono le caratteristiche qualitative del sistema, come l'affidabilità, le prestazioni, la sicurezza e l'usabilità.

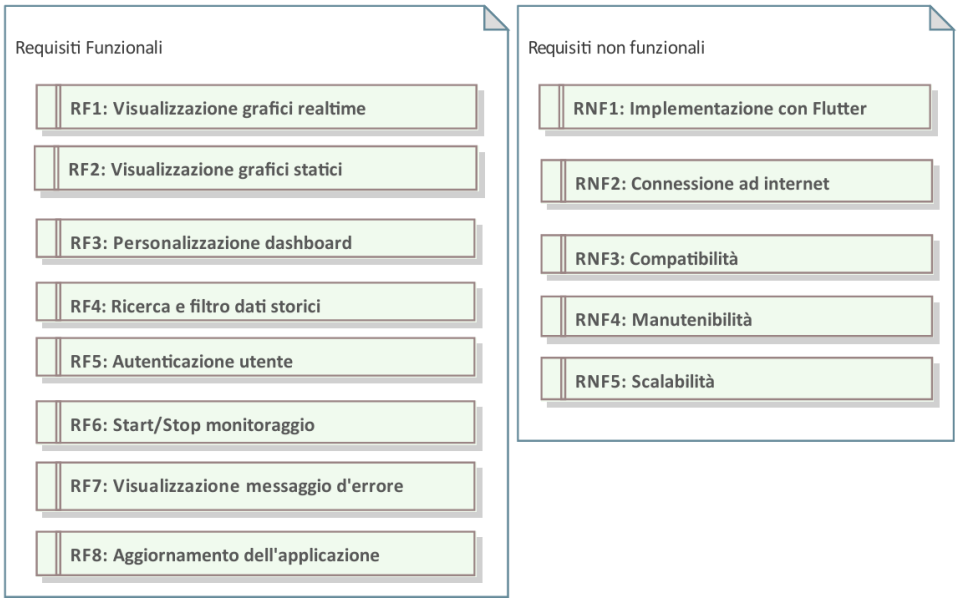


Figure 1: Requisiti Funzionali e Non Funzionali

- **Requisiti funzionali (RF):**
  - RF1: Il sistema consente la visualizzazione dei dati in tempo reale della componente di monitoring, attraverso dei grafici che si aggiorneranno all'arrivo di un nuovo dato.
  - RF2: Il sistema consente la visualizzazione dei dati storici attraverso dei grafici statici.
  - RF3: Il sistema permette all'utente di personalizzare le dashboard ad esempio il colore o visibilità dei grafici.

  - RF4: Attraverso l'interfaccia grafica, l'utente può inserire i criteri per interrogare il database.
  - RF5: Il sistema gestisce l'autenticazione dell'utente.
  - RF6: L'utente può attivare o disattivare il monitoraggio di una singola macchina, oppure di tutte le macchine associate, tramite un pulsante presente globalmente o per ciascuna macchina.
  - RF7: Nel caso in cui il sistema riceva un JSON contenente un errore, l'utente viene notificato mediante un messaggio appropriato.
  - RF8: Il sistema verifica la disponibilità di versioni più aggiornate dell' applicazione e ne consente l'installazione.

- **Requisiti non funzionali (RNF):**
  - RNF1: L'applicazione è implementata in linguaggio Dart utilizzando il framework Flutter.
  - RNF2: L'applicazione deve poter accedere a internet per effettuare il login, ricevere dati in tempo reale e per eseguire query storiche.
  - RNF3: L'applicazione deve essere compatibile sia con dispositivi iOS che Android.
  - RNF4: L'applicazione è sviluppata seguendo il pattern architetturale MVVM, al fine di garantire una struttura chiara e facilitare modifiche e manutenzione futura.
  - RNF5: L'applicazione deve essere scalabile, ovvero in grado di gestire un numero crescente di macchine e parametri senza degradare le prestazioni dell'interfaccia utente o del sistema di elaborazione.

2.2 Attori

Il sistema prevede principalmente una tipologia di attore (*Figura 2*) : l'utente, il quale può accedere a tutte le componenti dell'applicazione.

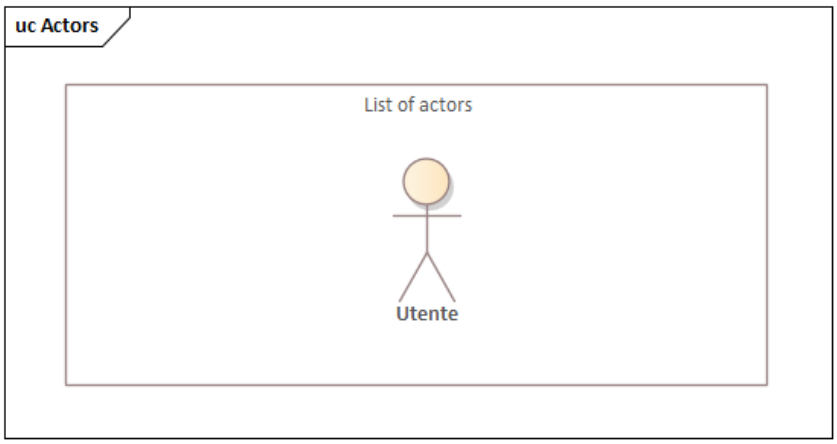


Figure 2: Attori del sistema

2.3 Gestione Utente

Nella figura caso d'uso Gestione Utente (*Figura 3*) è illustrato il sistema di autenticazione, basato sull'inserimento dei parametri username e password. Per avviare il processo, l'utente deve inserire in input tramite l'interfaccia grafica i due parametri descritti in precedenza. Successivamente, il server riceve la richiesta, ne verifica la validità e restituisce una risposta in formato JSON.

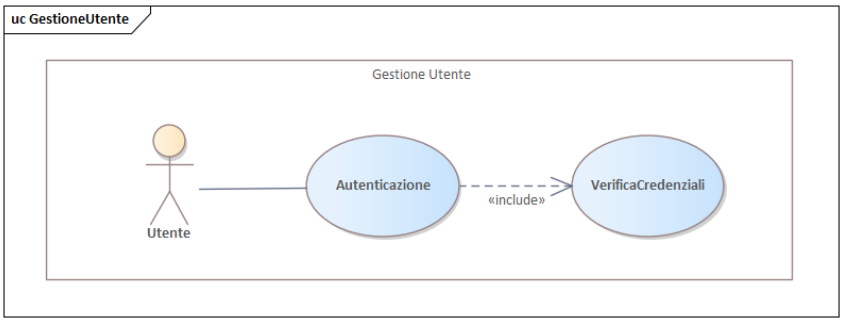


Figure 3: Caso d'Uso: Gestione Utente

2.4 Gestione Attività

Nella figura relativa al caso d'uso Gestione Attività (*Figura 4*) sono illustrate le funzionalità disponibili per l'utente.

Le azioni si possono dividere in due categorie: la sezione realtime e la sezione storica.

La componente realtime consente all'utente di visualizzare i dati provenienti dai sensori sotto forma di grafici lineari, qualora tali dati siano disponibili. Per ogni parametro viene generato un grafico dedicato, che l'utente può personalizzare modificandone la visibilità o anche modificare il colore della curva.

La componente storica, invece, si basa su un'interfaccia grafica che permette all'utente di interrogare il database. In caso di esito positivo, potranno essere generati tre tipologie di grafici: a torta, a barre e lineare. Quando è richiesta la visualizzazione simultanea di più tipologie di grafici, l'utente ha la possibilità di scegliere se mostrarli insieme o separatamente, tramite l'uso di appositi pulsanti.

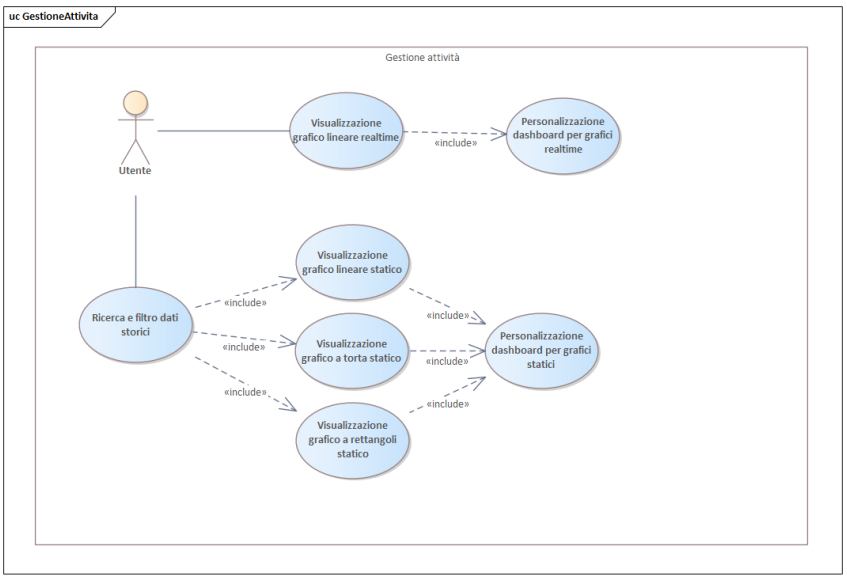


Figure 4: Caso d'Uso: Gestione Attività

Capitolo 3: Framework

Dopo aver analizzato i requisiti dell'applicazione, in questo capitolo ci si concentra sulle tecnologie impiegate per l'implementazione delle funzionalità individuate.

L'applicazione si basa su un framework di monitoraggio sviluppato per raccogliere, trasmettere e visualizzare dati provenienti da un ambiente industriale composto da macchine e sensori. In ambito informatico, un framework può essere definito come una struttura software riutilizzabile, costituita da strumenti e librerie che semplificano lo sviluppo di funzionalità complesse.

Nel caso specifico, il framework si inserisce nel progetto di ricerca PRIN 2022 HOMEY ("A Human-centric IoE-based Framework for Supporting the Transition Towards Industry 5.0"), e adotta metodologie di monitoraggio dei flussi di dati coerenti con gli approcci più recenti proposti in letteratura per l'Industria 5.0. In particolare, si fa riferimento al lavoro di Sciarroni, Esposito, Pierleoni e Storti, *Monitoring Data Streams in Industry 5.0: a Knowledge Graph Approach* [18], che descrive un'architettura basata su Knowledge Graph per la gestione e l'integrazione dei dati in contesti industriali avanzati. Lo schema riportato in *Figura 5* si ispira a tale modello. L'architettura prevede l'interazione con un insieme di API esposte da un server remoto, progettate secondo il paradigma REST (Representational State Transfer), che consente l'accesso alle risorse tramite operazioni HTTP standard quali `GET`, `POST` e `PUT`.

Lo scambio di dati tra client e server avviene in formato JSON, garantendo leggibilità e semplicità di integrazione con altri sistemi.

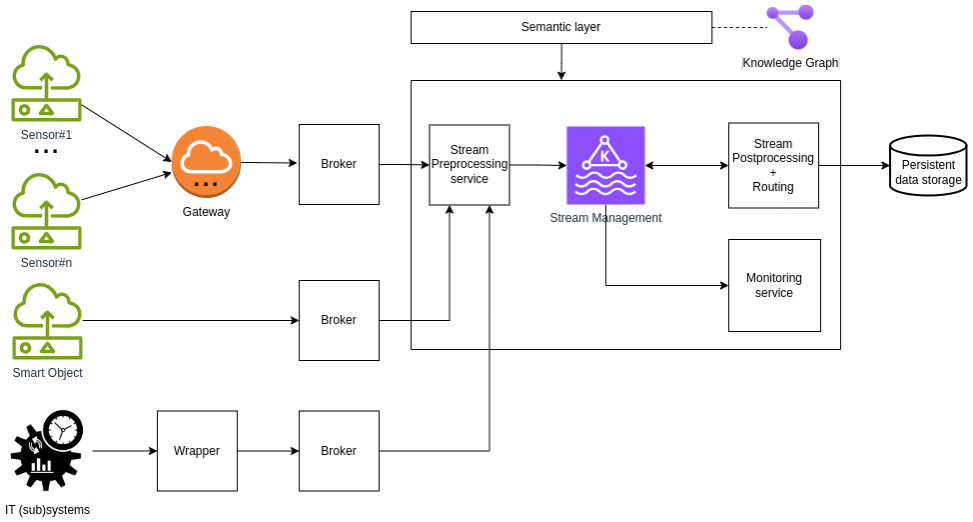


Figure 5: Architettura del framework di monitoraggio basata sul progetto HOMEY

Questo framework, basato su un'architettura a microservizi, permette l' implementazione modulare e scalabile delle principali funzionalità dell'applicazione. Nel seguito, ogni tecnologia utilizzata verrà descritta nel dettaglio.

3.1 Servizi e API disponibili

Il framework di monitoraggio espone un insieme di servizi REST che consentono all'applicazione mobile di gestire l'autenticazione dell'utente, il recupero delle informazioni sulle macchine associate, il monitoraggio in tempo reale dei dati, l'interrogazione storica e il controllo degli aggiornamenti software. Tutti i servizi utilizzano il formato JSON per lo scambio dei dati e seguono il paradigma REST, facendo uso dei metodi HTTP standard GET e POST.

Le figure riportate in questa sezione mostrano esempi di chiamata agli endpoint REST effettuate dall'applicazione mobile e le relative risposte JSON restituite dal server.

- **LoginService**: gestisce la fase di autenticazione dell'utente e il recupero delle informazioni iniziali necessarie all'utilizzo dell'applicazione.

  Il servizio espone le seguenti funzionalità:

  - **POST /login**: consente l'autenticazione dell'utente tramite l'invio delle credenziali di accesso. La richiesta prevede l'invio delle credenziali nel corpo del messaggio:
    - nome utente;
    - password.

    In caso di autenticazione avvenuta con successo, il server restituisce una risposta JSON contenente:
    - un access token, utilizzato per autorizzare le richieste successive verso i servizi protetti;
    - un refresh token, impiegato per ottenere un nuovo access token in caso di scadenza.

    La Figura 6 mostra un esempio di risposta restituita dall'endpoint di login.

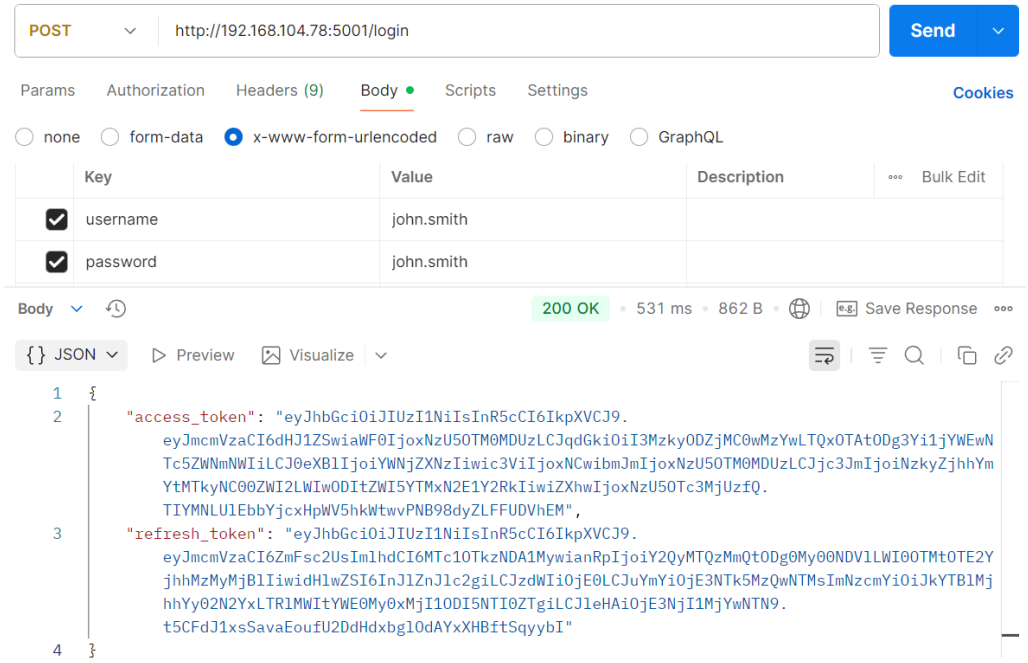


Figure 6: Esempio di risposta JSON della chiamata POST /login.

- **GET /api/utils**: consente di recuperare l'elenco delle macchine associate all'utente autenticato. La chiamata utilizza il token ottenuto in fase di login e permette di richiedere, oltre alla lista delle macchine, anche le informazioni descrittive associate.

  L'URL utilizzata dall'applicazione ha la seguente struttura:

  `/api/utils?username={username}&schema=true`

  dove:

  - `/api/utils` è il percorso dell'endpoint;
  - `username` è il nome utente associato alle macchine;
  - `schema` è impostato a `true`.

  La risposta JSON contiene:

  - la lista delle macchine associate all'utente;
  - per ciascuna macchina:
    - identificativo univoco della macchina (`source_id`);
    - posizione o sito di installazione;
    - smart object di riferimento;
    - topic di comunicazione;
    - elenco dei campi monitorati, ciascuno corredato da:
      - descrizione testuale del parametro;

- percorso logico del campo;
- proprietà fisica rappresentata;
- unità di misura.

La Figura 7 riporta un esempio dell'output completo restituito dall'endpoint.


```
[
  {
    "fields": [
      {
        "fieldDescription": "The temperature of the engine in Celsius",
        "fieldPath": "temp_motore",
        "property": "http://homey/example_graph/temp_motore",
        "uom_label": "degree Celsius",
        "uom_symbol": "°C"
      },
      {
        "fieldDescription": "The surface temperature of the machine in Celsius",
        "fieldPath": "temp_superficie",
        "property": "http://homey/example_graph/temp_superficie",
        "uom_label": "degree Celsius",
        "uom_symbol": "°C"
      },
      {
        "fieldDescription": "The level of produced noise in decibel",
        "fieldPath": "rumore",
        "property": "http://homey/example_graph/rumore",
        "uom_label": "Decibel",
        "uom_symbol": "dB"
      },
      {
        "fieldDescription": "The rotation speed of the engine in radians per second",
        "fieldPath": "vel_rotazione",
        "property": "http://homey/example_graph/vel_rotazione",
        "uom_label": "radian per second",
        "uom_symbol": "rad/s"
      },
      {
        "fieldDescription": "The level of vibrations in Hertz",
        "fieldPath": "vibrazioni",
        "property": "http://homey/example_graph/vibrazioni",
        "uom_label": "hertz",
        "uom_symbol": "Hz"
      }
    ],
    "site": "http://homey/example_graph/area_1",
    "smart_object": "http://homey/machinery_example/so_machinery_3",
    "source_id": "http://homey/machinery_example/sensor_factory_machinery_3",
    "topic": "factory_machinery_3"
  },
  {
```


Figure 7: Esempio di risposta JSON della chiamata GET /api/utils.

- **MachineStreamService**: è responsabile della gestione del monitoraggio in tempo reale dei dati provenienti dalle macchine.

  Il servizio mette a disposizione:

  – **GET /monitoring**: apre una connessione persistente con il server per ricevere aggiornamenti continui relativi a una specifica macchina, identificata tramite il proprio identificativo univoco.

  L'URL utilizzata per attivare lo stream è composta nel modo seguente:

```
/monitoring?source_id={identificativoMacchina}
```

dove:

- `/monitoring` è il percorso dell'endpoint di streaming;
- `source_id` individua univocamente la macchina da monitorare.

Il flusso restituito contiene eventi in tempo reale, ciascuno dei quali include un JSON con:

- i valori aggiornati dei parametri monitorati;
- il relativo riferimento temporale.

Un esempio di output dello stream è mostrato in Figura 8.


```
data: {"machine_id": 1, "temp_motore": 55.83, "temp_superficie": 27.15, "rumore": 91.24, "vibrazioni": 0.79,
"timestamp": 1759935088205, "ts_bridge": 1759935088206, "ts_prep_filter": 1759935088206}

data: {"machine_id": 1, "temp_motore": 56.81, "temp_superficie": 27.25, "rumore": 92.14, "vibrazioni": 0.74,
"timestamp": 1759935093211, "ts_bridge": 1759935093212, "ts_prep_filter": 1759935093212}

data: {"machine_id": 1, "temp_motore": 55.94, "temp_superficie": 26.95, "rumore": 92.61, "vibrazioni": 0.84,
"timestamp": 1759935098216, "ts_bridge": 1759935098218, "ts_prep_filter": 1759935098218}

data: {"machine_id": 1, "temp_motore": 55.1, "temp_superficie": 26.32, "rumore": 92.84, "vibrazioni": 0.85,
"timestamp": 1759935103220, "ts_bridge": 1759935103222, "ts_prep_filter": 1759935103222}

data: {"machine_id": 1, "temp_motore": 54.64, "temp_superficie": 25.59, "rumore": 92.66, "vibrazioni": 0.79,
"timestamp": 1759935108226, "ts_bridge": 1759935108227, "ts_prep_filter": 1759935108227}

data: {"machine_id": 1, "temp_motore": 54.82, "temp_superficie": 26.33, "rumore": 91.85, "vibrazioni": 0.79,
"timestamp": 1759935113232, "ts_bridge": 1759935113233, "ts_prep_filter": 1759935113233}

data: {"machine_id": 1, "temp_motore": 54.28, "temp_superficie": 26.14, "rumore": 92.8, "vibrazioni": 0.77,
"timestamp": 1759935118237, "ts_bridge": 1759935118238, "ts_prep_filter": 1759935118238}
```


Figure 8: Esempio di output dello stream in tempo reale GET /monitoring.

- **MachinePastService**: consente l'interrogazione del database storico per il recupero dei dati passati.

  Il servizio espone:

  - **GET /querying**: permette di ottenere dati storici relativi a una o più macchine.

  L'endpoint base è:

  ```
  /querying
  ```

  ed è completato dall'applicazione tramite una serie di parametri nella query string. La forma generale dell'URL è:

  ```
  /querying ?source_id={id} &param={nomeParametro}
  &aggr={tipoAggregazione} &start_time={istanteInizio}
  &stop_time={istanteFine} &limit={maxRisultati} &offset={offset}
  ```

dove:

- `source_id` identifica la macchina;
- `param` indica il parametro di interesse;
- `aggr` specifica un'eventuale modalità di aggregazione;
- `start_time` e `stop_time` definiscono l'intervallo temporale di riferimento;
- `limit` e `offset` permettono di gestire la paginazione dei risultati.

La risposta consiste in una struttura JSON contenente i valori storici associati ai rispettivi riferimenti temporali. Un esempio di output specificando l'identificativo della macchina è riportato in Figura 9.


```
[
    {
        "source_id": "http://homey/machinery_example/sensor_factory_machinery_1",
        "machine_id": 1,
        "temp_motore": 56.84,
        "temp_superficie": 33.99,
        "rumore": 94.42,
        "vel_rotazione": 65,
        "vibrazioni": 0.32,
        "timestamp": 1734520940406
    },
    {
        "source_id": "http://homey/machinery_example/sensor_factory_machinery_1",
        "machine_id": 1,
        "temp_motore": 57.35,
        "temp_superficie": 34.17,
        "rumore": 94.38,
        "vel_rotazione": 65,
        "vibrazioni": 0.34,
        "timestamp": 1734520945413
    },
    {
        "source_id": "http://homey/machinery_example/sensor_factory_machinery_1",
        "machine_id": 1,
        "temp_motore": 56.72,
        "temp_superficie": 34.36,
        "rumore": 94.88,
        "vel_rotazione": 65,
        "vibrazioni": 0.37,
        "timestamp": 1734520950418
    },
    {
```


Figure 9: Esempio di output della query storica GET /querying.

- **UpdateService**: gestisce il controllo delle versioni disponibili e la procedura di aggiornamento dell'applicazione.

  Il servizio utilizza l'API ufficiale di GitHub e mette a disposizione:

  - **GET /releases/latest**: restituisce le informazioni relative all'ultima release disponibile dell'applicazione, includendo:
    - numero di versione;
    - data di pubblicazione;

- asset scaricabili associati alla release.

L'endpoint utilizzato per ottenere queste informazioni è:

`/releases/latest`

e, a partire dalla risposta JSON, viene ricavato anche il percorso di download del file di aggiornamento, tipicamente nel formato:

`/releases/download/{tag}/{nomeFile.apk}`

Le Figure 10 e 11 mostrano esempi della risposta JSON restituita dall'endpoint.

```
{
  "url":
"https://api.github.com/repos/FilErc/tirocinio-updates/releases/245152460",
  "assets_url":
"https://api.github.com/repos/FilErc/tirocinio-updates/releases/245152460/assets",
  "upload_url":
"https://uploads.github.com/repos/FilErc/tirocinio-updates/releases/245152460/
assets{?name,label}",
  "html_url": "https://github.com/FilErc/tirocinio-updates/releases/tag/v1.1.8",
  "id": 245152460,
  "author": {
    "login": "FilErc",
    "id": 170265072,
    "node_id": "U_kgDOCiYJ8A",
    "avatar_url": "https://avatars.githubusercontent.com/u/170265072?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/FilErc",
    "html_url": "https://github.com/FilErc",
    "followers_url": "https://api.github.com/users/FilErc/followers",
    "following_url": "https://api.github.com/users/FilErc/following{/other_user}",
    "gists_url": "https://api.github.com/users/FilErc/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/FilErc/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/FilErc/subscriptions",
    "organizations_url": "https://api.github.com/users/FilErc/orgs",
    "repos_url": "https://api.github.com/users/FilErc/repos",
    "events_url": "https://api.github.com/users/FilErc/events{/privacy}",
    "received_events_url": "https://api.github.com/users/FilErc/received_events",
    "type": "User",
    "user_view_type": "public",
    "site_admin": false
  },
  "node_id": "RE_kwDOPIBNYM4OnLrM",
  "tag_name": "v1.1.8",
  "target_commitish": "main",
  "name": "Tirocinio v1.1.8",
  "draft": false,
  "immutable": false,
  "prerelease": false,
  "created_at": "2025-07-06T22:58:15Z",
  "updated_at": "2025-09-05T18:34:42Z",
  "published_at": "2025-09-05T18:34:42Z",
  "assets": [
    {
      "url":
"https://api.github.com/repos/FilErc/tirocinio-updates/releases/assets/290099411",
      "id": 290099411,
      "node_id": "RA_kwDOPIBNYM4RSpDT",
      "name": "app-release.apk",
      "label": "",
      "uploader": {
        "login": "FilErc",
        "id": 170265072,
        "node_id": "U_kgDOCiYJ8A",
        "avatar_url": "https://avatars.githubusercontent.com/u/170265072?v=4",
        "gravatar_id": "",
        "url": "https://api.github.com/users/FilErc",
        "html_url": "https://github.com/FilErc",
        "followers_url": "https://api.github.com/users/FilErc/followers",
        "following_url":
```


Figure 10: Esempio di risposta JSON dell'endpoint GitHub Releases.

```
"https://api.github.com/users/FilErc/following{/other_user}",
        "gists_url": "https://api.github.com/users/FilErc/gists{/gist_id}",
        "starred_url":
"https://api.github.com/users/FilErc/starred{/owner}{/repo}",
        "subscriptions_url": "https://api.github.com/users/FilErc/subscriptions",
        "organizations_url": "https://api.github.com/users/FilErc/orgs",
        "repos_url": "https://api.github.com/users/FilErc/repos",
        "events_url": "https://api.github.com/users/FilErc/events{/privacy}",
        "received_events_url":
"https://api.github.com/users/FilErc/received_events",
        "type": "User",
        "user_view_type": "public",
        "site_admin": false
      },
      "content_type": "application/vnd.android.package-archive",
      "state": "uploaded",
      "size": 60998979,
      "digest":
"sha256:b75ef2b1d35401fdd278ae26fca35d2ac505d3763b1def2768ee409e4ea4f15e",
      "download_count": 5,
      "created_at": "2025-09-05T18:34:38Z",
      "updated_at": "2025-09-05T18:34:41Z",
      "browser_download_url":
"https://github.com/FilErc/tirocinio-updates/releases/download/v1.1.8/app-
release.apk"
    }
  ],
  "tarball_url":
"https://api.github.com/repos/FilErc/tirocinio-updates/tarball/v1.1.8",
  "zipball_url":
"https://api.github.com/repos/FilErc/tirocinio-updates/zipball/v1.1.8",
  "body": "UIs update"
}
```


Figure 11: Esempio di risposta JSON dell'endpoint GitHub Releases.

Tutti i servizi, ad eccezione della chiamata di login e del servizio di aggiornamento, richiedono un token di autorizzazione valido e implementano meccanismi di gestione degli errori, restituendo messaggi informativi in caso di risposte non valide o problemi di rete.

Capitolo 4: Progettazione

In questo capitolo vengono presentate le scelte progettuali che hanno guidato lo sviluppo dell'applicazione. In particolare, saranno descritti gli strumenti tecnologici e le librerie adottate in fase di progettazione e implementazione, i mockup dell'interfaccia, la struttura dei dati, il pattern architetturale utilizzato e i diagrammi delle componenti.

4.1 Strumenti di progettazione utilizzati

La progettazione del sistema si è basata sull'impiego di tecnologie consolidate e librerie affidabili, con l'obiettivo di garantire coerenza tra requisiti, architettura e interfaccia utente.

Flutter e Dart
Flutter, sviluppato da Google, è un Software Development Kit (SDK) open-source che consente di creare applicazioni multipiattaforma a partire da un unico codice sorgente, sfruttando un engine di rendering proprietario in grado di disegnare gli elementi grafici in modo coerente su Android, iOS, Web e desktop. Questo approccio consente prestazioni elevate e uniformità grafica indipendentemente dalla piattaforma. Il linguaggio Dart è stato progettato in modo complementare al framework, con compilatori ottimizzati e sintassi moderna: le applicazioni vengono generate una sola volta, riducendo tempi e costi di sviluppo, e offrono una forte integrazione con il dispositivo ospite [15].

Android Studio
Per l'ambiente di sviluppo è stato utilizzato Android Studio, IDE ufficiale per lo sviluppo Android, integrato con i plugin Flutter e Dart. Tale scelta facilita diverse attività fondamentali:

- Analisi del codice.
- Hot reload (aggiorna il codice senza perdere lo stato) e hot restart (riavvia l'app azzerando lo stato) durante lo sviluppo Flutter.
- Gestione di emulatori e dispositivi fisici tramite Device Manager.

Gestione dello stato con *provider*
Il package *provider* offre una soluzione leggera e immediata per la gestione dello stato in Flutter: separa chiaramente la logica applicativa dalla presentazione, migliorando la leggibilità. Nel modello MVVM adottato, *provider* facilita l'ascolto automatico delle modifiche di stato da parte dell'interfaccia grafica e ne permette una ricompilazione mirata e ottimizzata [17]. La documentazione ufficiale di Flutter lo raccomanda come punto di partenza per applicazioni non complesse, tramite ChangeNotifierProvider e widget ascoltatori come Consumer per distribuire i dati nel widget tree [16].

Controllo versione con GitHub
GitHub fornisce un sistema efficace di controllo versione distribuito che permette di tracciare le modifiche, gestire rami paralleli e sincronizzare il lavoro tra più sviluppatori. Rappresenta uno standard consolidato per la collaborazione su progetti software [9].

Comunicazione di rete con *http*
Per l'interfacciamento con le API REST del backend è stata impiegata la libreria *http*: offre una API basata su Future, semplice da utilizzare per operazioni GET/POST, gestione degli header e parsing delle risposte [12].

Persistenza locale con *shared_preferences*
shared_preferences è stata scelta per memorizzare in modo efficiente impostazioni utente e flag applicativi in chiave–valore, sfruttando le API native Android/iOS. È adatta a piccoli volumi di dati non sensibili; non è indicata per grandi quantità di informazioni.

Personalizzazione dell'interfaccia con *flutter_colorpicker*
La libreria consente di integrare picker di colore interattivi (HSV/HSL/RGB), permettendo all'utente di modificare visivamente i colori di grafici o widget, a beneficio dell'esperienza d'uso e della riconoscibilità dell'interfaccia.

Selezione temporale con *flutter_datetime_picker_plus*
Mette a disposizione widget per la selezione di date e orari, facilmente configurabili e localizzabili, utili per la definizione di filtri temporali nelle query storiche e per l'impostazione di parametri temporali nell'applicazione.

Accesso a metadati e filesystem: *package_info_plus* e *path_provider*
package_info_plus espone a runtime informazioni come versione e build dell'app, utili per diagnostica e aggiornamenti [10]. *path_provider* consente di accedere in modo sicuro a directory come cache, documenti e temporanei, abilitando funzionalità offline, esportazioni e download [13].

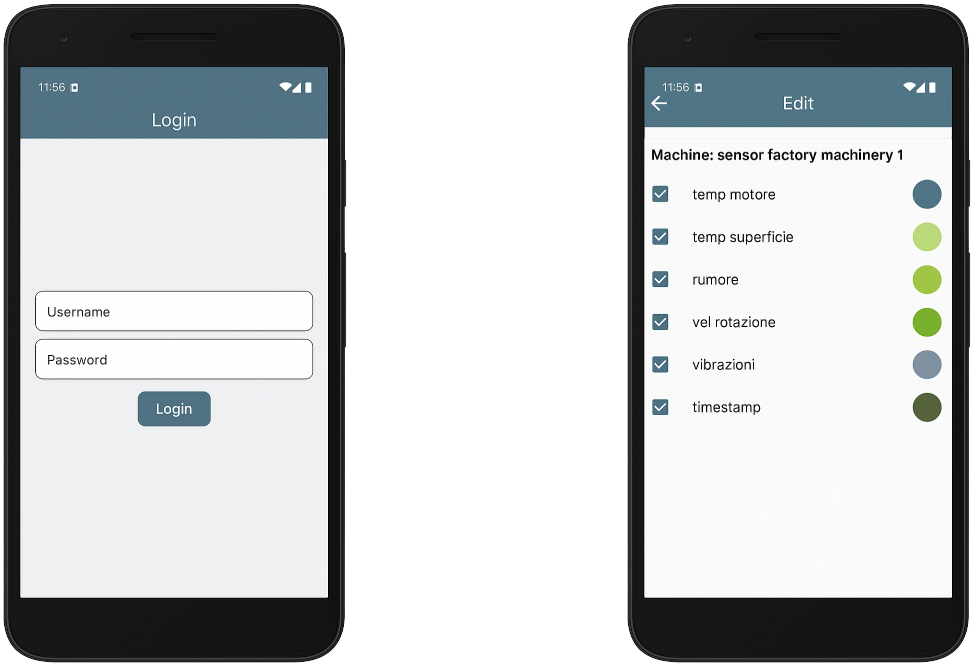
Gestione permessi con *permission_handler*
Centralizza le richieste di autorizzazioni di sistema come ad esempio l'autorizzazione per accedere alla fotocamera, consentendo la verifica dello stato, la richiesta esplicita all'utente e la gestione dei casi di negazione con reindirizzamento alle impostazioni del dispositivo [11].

Visualizzazione dati con *fl_chart*
La libreria *fl_chart* è stata selezionata per flessibilità e leggerezza. Supporta grafici a linee, barre, torta, dispersione e radar, ciascuno ampiamente configurabile; nei grafici a linee, per esempio, è possibile definire serie multiple, spessori personalizzati, gradienti, riempimenti e curve smussate. Sono inoltre disponibili interazioni touch con tooltip, evidenziazione dei punti, formattazione degli assi e legende personalizzabili, rendendola adatta sia a scenari in tempo reale sia ad analisi storiche [7].

4.2 Mockup

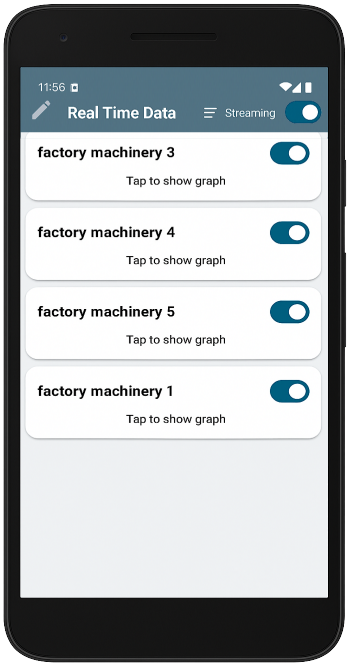
Per supportare la progettazione dell'interfaccia e garantire coerenza nell'esperienza d'uso, sono stati realizzati mockup ad alta fedeltà che hanno guidato la definizione di layout, gerarchia visiva e flussi di interazione. Le schermate sono presentate secondo l'ordine logico di utilizzo dell'applicazione: dall'autenticazione iniziale, alla visualizzazione e personalizzazione dei dati di monitoraggio in tempo reale, fino alle funzionalità di interrogazione storica e alla rappresentazione dei risultati tramite differenti tipologie di grafici.
Sintesi delle figure mostrate:

- *Figura 12a–12b* (a, b): *Mockup principali autenticazione e configurazione.*
- *Figura 13a–13d* (a, b, c, d): *Viste di monitoraggio e ricerca tipo 1.*
- *Figura 14a–14d* (a, b, c, d): *Ricerca tipo 2 e output grafici.*

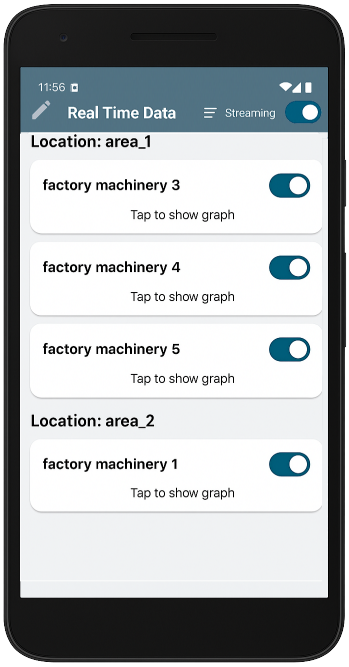


(a) Login Screen (b) Edit Screen

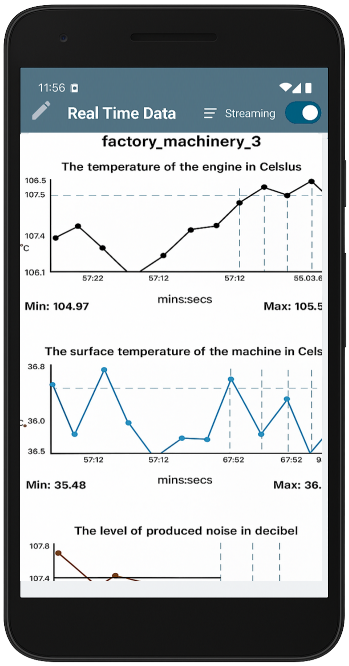
Figure 12: Mockup principali: autenticazione e configurazione.



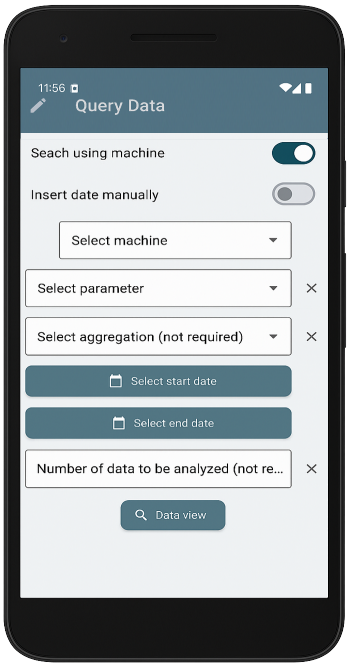
(a) Monitoring Screen



(b) Monitoring Screen Sorted

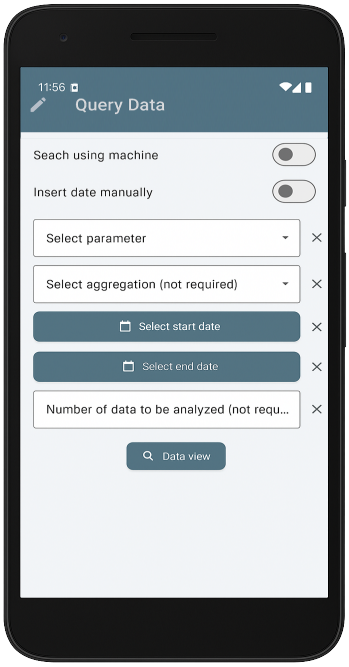


(c) Monitoring Graph Screen

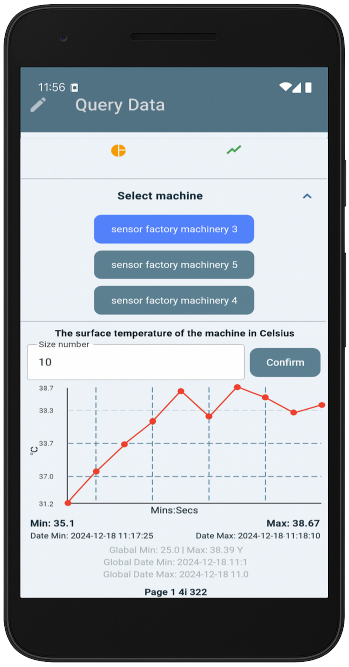


(d) Query Searching Type 1

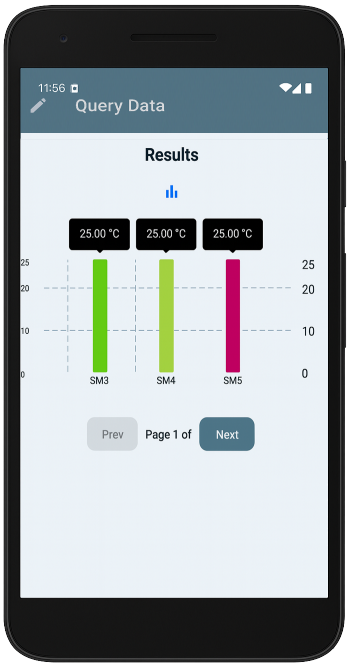
Figure 13: Mockup: viste di monitoraggio e ricerca dati.



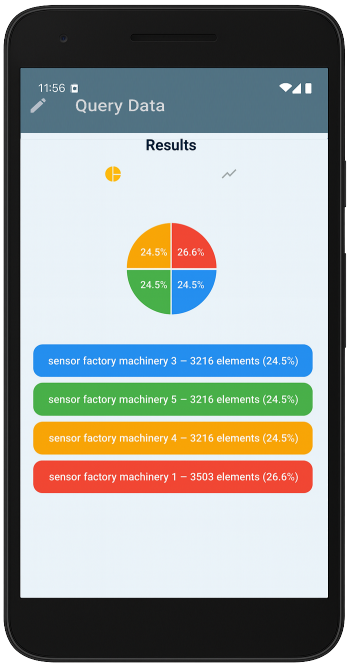
(a) Query Searching Type 2



(b) Query Output Linear



(c) Query Output Rectangle



(d) Query Output Pie

Figure 14: Mockup: output grafici in diverse tipologie di visualizzazione.

4.3 Pattern architetturale MVVM

L'applicazione adotta il pattern MVVM (Model–View–ViewModel) come architettura di riferimento (*Figura 15*), al fine di garantire una chiara separazione delle responsabilità tra componenti, incrementare la testabilità e favorire il riuso del codice. Come descritto nella documentazione ufficiale di Microsoft [17], tale pattern è stato ideato per disaccoppiare l'interfaccia utente dalla logica di business e di presentazione, consentendo un'evoluzione indipendente delle due parti e una più agevole manutenzione del software.

I componenti fondamentali dell'architettura sono:

- Model: rappresenta le entità e le regole di dominio, oltre alla logica di accesso e manipolazione dei dati ad esempio MachineInfo e FieldInfo.
- View: costituisce la parte visuale dell'applicazione, ovvero i widget Flutter responsabili della presentazione dei dati e dell'interazione con l'utente.
- ViewModel: contiene la logica applicativa, funge da mediatore tra View e Model, esponendo dati e comandi in una forma adatta all'interfaccia grafica.
- Service: gestisce l'accesso a risorse esterne (API, storage, database), isolando la logica di persistenza e comunicazione dal resto dell'applicazione.

Il ViewModel svolge il ruolo di adattatore, trasformando i dati grezzi provenienti dal Model in informazioni pronte per la visualizzazione, e ricevendo dalla View le azioni dell'utente per tradurle in operazioni sui dati. Questa intermediazione riduce l'accoppiamento diretto tra interfaccia e logica di business, favorendo la modularità del progetto [17].

Tra i principali vantaggi derivanti dall'adozione del pattern MVVM si evidenziano:

- Elevata testabilità delle componenti logiche, grazie alla possibilità di verificare ViewModel e Model in modo indipendente dalla UI.
- Manutenibilità e facilità di evoluzione, in quanto le modifiche all'interfaccia non richiedono interventi sulla logica e viceversa.
- Possibilità di lavoro parallelo tra sviluppatori e designer, che possono operare in autonomia sui rispettivi ambiti.
- Maggiore riuso di codice e componenti, soprattutto in contesti multi–piattaforma.



Figure 15: Schema semplificato dell'interazione View–ViewModel–Model

4.4 Diagramma delle componenti

Il diagramma delle componenti (*Figura 16*) rappresenta la struttura logica dell' applicazione, evidenziando i moduli principali, le dipendenze e i flussi di comunicazione. La rappresentazione segue una chiara separazione tra interfacce grafiche (View), logica di presentazione (ViewModel), servizi (Service) e modello dati (Model), in coerenza con l'architettura MVVM adottata.

Ogni rettangolo nel diagramma corrisponde a un file o a un insieme di file omogenei per funzione, mentre le frecce indicano le relazioni e le direzioni di interazione. In questo modo è possibile osservare:

- Quali file invocano metodi o accedono a dati esposti da altri.
- Come le View si interfacciano esclusivamente con i rispettivi ViewModel e non direttamente con il Model.
- Quali Service forniscono risorse esterne ai ViewModel.
- In che modo le entità di dominio (Model) fungono da struttura dati comune, condivisa da più componenti.

Questa rappresentazione facilita la comprensione delle responsabilità di ciascun file, rendendo immediatamente visibili eventuali accoppiamenti eccessivi o flussi di comunicazione non coerenti con il pattern architetturale.

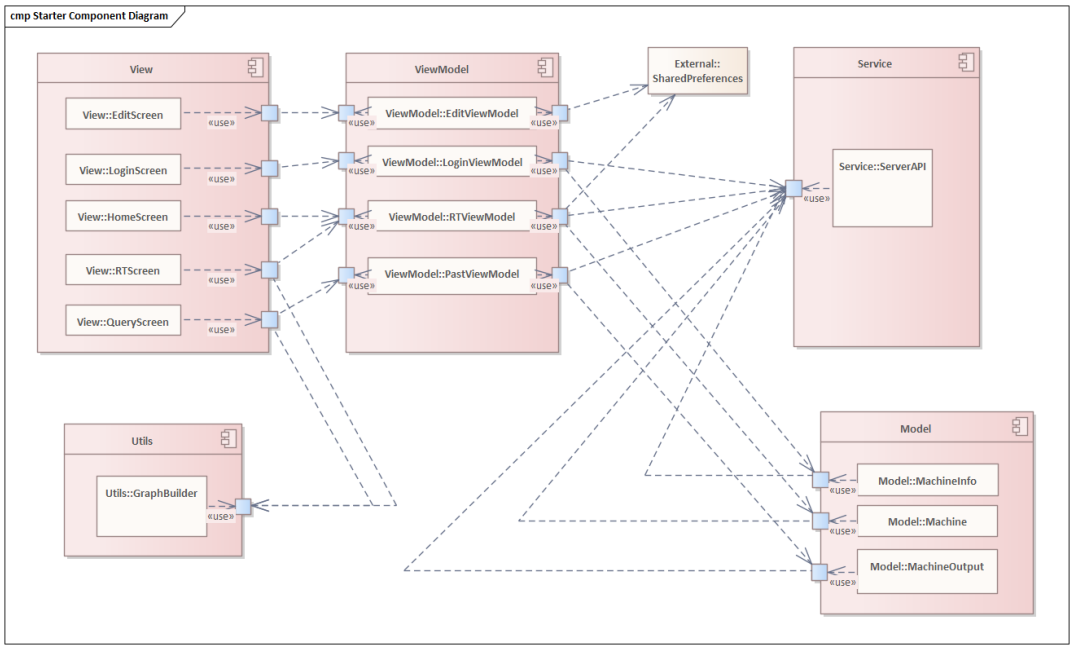


Figure 16: Diagramma delle Componenti

4.5 Meccanismo di interazione con le API

L'interazione tra l'applicazione mobile e il server si basa su un'architettura client–server di tipo REST. Dal punto di vista progettuale, la comunicazione è organizzata in modo da garantire una netta separazione delle responsabilità tra interfaccia utente, logica applicativa e servizi di rete.

Le View raccolgono gli input e presentano i dati, mentre i ViewModel gestiscono la logica e coordinano le richieste verso i Service, che costituiscono il livello dedicato alla comunicazione con il backend. I Service si occupano di inviare le richieste HTTP, allegare i token di autenticazione, gestire le risposte JSON e segnalare eventuali errori o ritardi di rete.

Il flusso di interazione inizia verificando l'esistenza di una nuova versione dell' applicazione, successivamente si procede con il metodo di autenticazione dell' utente, seguita dal recupero delle informazioni sulle macchine e dai dati di monitoraggio. Il sistema di streaming riceve aggiornamenti in tempo reale che vengono elaborati dai ViewModel e rappresentati graficamente. Per le interrogazioni storiche, il client invia richieste filtrate secondo parametri temporali o di aggregazione, gestendo automaticamente la paginazione dei risultati per garantire fluidità all'interfaccia.

Tutte le comunicazioni avvengono tramite protocollo sicuro HTTPS e prevedono controlli di validità, retry automatici in caso di errore temporaneo e messaggi informativi verso l'utente. Questa progettazione consente di mantenere un flusso dati stabile, modulare e facilmente estendibile, assicurando al tempo stesso affidabilità e chiarezza nella gestione delle informazioni.

Capitolo 5: Implementazione

5.1 Struttura delle cartelle

La struttura delle cartelle del progetto (*Figura 17*) segue fedelmente il pattern architetturale MVVM, con directory distinte per i model, viewmodel, view e service, favorendo la scalabilità e l'organizzazione del codice sorgente. Nel progetto è inoltre presente una directory utils, non direttamente prevista dal pattern MVVM ma utile come contenitore per funzionalità di supporto comuni alle varie componenti. In particolare, in questa cartella è collocato il file graph_builder.dart, dedicato alla generazione e configurazione dei grafici, utilizzato dalle diverse View per mantenere uno stile coerente e centralizzare la logica di costruzione delle visualizzazioni. Questo approccio permette di evitare duplicazioni di codice, semplificare la manutenzione e garantire uniformità nella presentazione dei dati.

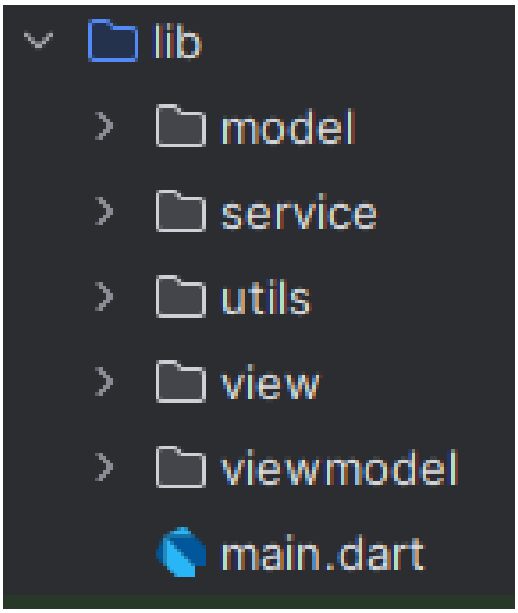


Figure 17: Struttura delle cartelle in Flutter

5.2 Dettagli implementativi principali

Modello dei dati dell'app

È stata privilegiata una progettazione flessibile, in modo da semplificare l'evoluzione e la manutenzione futura del codice. Infatti in quest'applicazione sono presenti solo due oggetti ben descritti, essi sono: FieldInfo e MachineInfo.

`MachineInfo`: rappresenta un'entità macchina o sensore, contenente informazioni identificative, di localizzazione e di struttura dei dati (*Figura 18*). Gli attributi principali sono:

- sourceId: identifica univocamente la macchina o il sensore.
- location: indica la posizione geografica in cui è collocata la macchina o il sensore.
- smartObject: indica l'entità fisica o virtuale con ID univoco, dotata di sensori/attuatori in grado di raccogliere e trasmettere dati.
- topic: identifica il canale logico di comunicazione nei sistemi publish/subscribe su cui vengono inviati e ricevuti i dati.
- fields: lista di oggetti FieldInfo, ciascuno dei quali descrive un parametro o una variabile monitorata dalla macchina.

Attraverso il costruttore fromJson consente di creare un'istanza della classe a partire da un oggetto Map<String, dynamic>, in formato JSON. All'interno, il metodo ausiliario extractLast viene utilizzato per estrarre l'ultimo elemento di una stringa separata da "/".

La variabile fieldList viene popolata applicando il metodo map alla lista JSON dei campi fields: per ogni elemento della lista viene invocato FieldInfo.fromJson(item), trasformando ogni mappa in un oggetto FieldInfo, e infine convertendo il risultato in una lista concreta tramite toList(). Questo approccio sfrutta la programmazione funzionale in Dart, permettendo una trasformazione diretta e compatta dei dati serializzati in oggetti tipizzati.

```dart
class MachineInfo {
  final String sourceId;
  final String location;
  final String smartObject;
  final String topic;
  final List<FieldInfo> fields;

  MachineInfo({
    required this.sourceId,
    required this.location,
    required this.smartObject,
    required this.topic,
    required this.fields,
  });

  factory MachineInfo.fromJson(Map<String, dynamic> json) {
    String extractLast(String uri) => uri.split('/').last;

    var fieldList = (json['fields'] as List<dynamic>)
        .map((item) => FieldInfo.fromJson(item))
        .toList();

    return MachineInfo(
      sourceId: json['source_id'],
      location: extractLast(json['site']),
      smartObject: extractLast(json['smart_object']),
      topic: json['topic'],
      fields: fieldList,
    );
  }
}
```

Figure 18: Struttura della classe MachineInfo

`FieldInfo`: descrive i dettagli relativi a un singolo campo appartenente a un oggetto MachineInfo (*Figura 19*). Gli attributi sono:

- fieldDescription: descrizione testuale del parametro monitorato.
- fieldPath: percorso logico per identificare univocamente il campo.
- property: indica quale concetto o caratteristica reale rappresenta il valore contenuto nel field.
- uomLabel: etichetta testuale dell'unità di misura.
- uomSymbol: simbolo associato all'unità di misura (per esempio "°C", "m/s", "%").

```
class FieldInfo {
  final String fieldDescription;
  final String fieldPath;
  final String property;
  final String uomLabel;
  final String uomSymbol;

  FieldInfo({
    required this.fieldDescription,
    required this.fieldPath,
    required this.property,
    required this.uomLabel,
    required this.uomSymbol,
  });

  factory FieldInfo.fromJson(Map<String, dynamic> json) {
    return FieldInfo(
      fieldDescription: json['fieldDescription'],
      fieldPath: json['fieldPath'],
      property: json['property'],
      uomLabel: json['uom_label'],
      uomSymbol: json['uom_symbol'],
    );
  }
}
```

Figure 19: Struttura della classe FieldInfo

Strutture dinamiche: per garantire la massima adattabilità a modifiche lato backend, gli oggetti intercettati dal sistema di monitoring e querying sono gestiti come Map<String, dynamic> e List<dynamic>. Questo approccio consente di evitare vincoli strutturali stringenti e di adattarsi rapidamente a eventuali variazioni nel formato delle risposte API, senza dover intervenire sul codice ogni volta che la struttura dei dati subisce modifiche.

5.3 Sistema di Autenticazione

L'autenticazione avviene tramite una chiamata POST all'endpoint /login (*Figura 20*). Durante la chiamata, si utilizzano dati sensibili. L'utilizzo del metodo POST consente di inviare informazioni (username e password) nel corpo della richiesta, evitando così di esporle direttamente nell'URL. Il login viene considerato riuscito quando l'oggetto loginResponse restituisce il valore 200. Il valore è definito analizzando l'istanza dell'oggetto nel parametro status.code. Se le credenziali sono corrette, il server restituisce un token che consente di validare le operazioni dell'utente durante l'utilizzo dell'applicazione.

```
final loginResponse = await http.post(
  Uri.parse(loginUrl),
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
  },
  body: {
    'username': username,
    'password': password,
  },
);

if (loginResponse.statusCode != 200) {
  throw Exception('Login failed: ${loginResponse.statusCode}');
}
```

Figure 20: Richiesta per il Login

Successivamente, viene effettuata una richiesta GET all'endpoint /api/utils per recuperare le macchine associate all'utente.

In questo caso il metodo GET può essere utilizzato poiché si tratta di un' operazione in sola lettura, al fine di recuperare le informazioni essenziali per l'utilizzo dell'applicazione tra cui: sourceId, location, smartObject, topic e l'elenco dei relativi campi (fieldDescription, fieldPath, property, uomLabel, uomSymbol).

La codifica della risposta JSON avviene tramite il metodo fromJson della classe MachineInfo, il quale istanzia un oggetto del modello popolando automaticamente i campi corrispondenti con i valori restituiti dall'API (*Figura 21*).

```
final tokens = jsonDecode(loginResponse.body);
final accessToken = tokens['access_token'];
final refreshToken = tokens['refresh_token'];

final uri = Uri.parse('$machineBaseUrl?username=$username&schema=true');

final machineResponse = await http.get(uri);

if (machineResponse.statusCode != 200) {
  throw Exception('Fetch machine failed: ${machineResponse.statusCode}');
}

final machineList = (jsonDecode(machineResponse.body) as List)
    .map((json) => MachineInfo.fromJson(json))
    .toList();

return {
  'access_token': accessToken,
  'refresh_token': refreshToken,
  'machines': machineList,
};
```

Figure 21: Richiesta delle macchine associate all'utente

5.4 Meccanismo di monitoring dei dati

Il servizio di monitoraggio necessita della lista di macchine restituite dalla sessione di login, se non presenti, non verrà avviato alcun monitoring. Il MachineStreamService si connette all'endpoint /monitoring (*Figura 22*) tramite il protocollo Server-Sent Events (SSE). La tecnologia SSE permette al server di inviare aggiornamenti real-time al client, a connessione HTTP già avviata, in maniera unidirezionale [5].

Ogni flusso, unico per macchina, è gestito tramite un StreamController [6] in modalità broadcast, che permette di avere più listener contemporaneamente sullo stesso stream, garantendo così che diversi componenti dell'app possano ricevere e visualizzare i dati in parallelo. I dati ricevuti vengono decodificati secondo il seguente metodo:

- LineSplitter: utilizzato per dividere lo stream UTF-8 in linee indipendenti, in modo da elaborare ciascun evento secondo il formato SSE [8].
- parsing JSON tramite jsonDecode(): per decodificare la stringa JSON estratta in una struttura dati, necessaria per il funzionamento del monitoring.

Il sistema prevede una logica di start/stop dello streaming basata sull'utilizzo di flag booleani, che consentono di attivare, disattivare e successivamente riattivare il monitoring. Un possibile esempio di utilizzo è il seguente: dopo il login, tutte le macchine risultano attive e ricevono i dati di monitoraggio in tempo reale. Qualora l'utente disattivi il flag relativo a una singola macchina, lo streaming associato viene interrotto, mentre le altre continuano a ricevere aggiornamenti. In un momento successivo, riattivando lo stesso flag, la macchina riprende la ricezione dei dati senza influenzare il funzionamento delle altre.

```
try {
  await for (var line in streamedResponse.stream
      .transform(utf8.decoder)
      .transform(const LineSplitter())) {

    if (!RTViewModel.streamingEnabled || !_running) {
      break;
    }

    line = line.trim();

    if (!line.startsWith('data:')) {
      continue;
    }

    final jsonString = line.replaceFirst('data:', '').trim();

    try {
      final json = jsonDecode(jsonString);
      if (json is Map<String, dynamic> && json.containsKey('error')) {
        throw Exception('Error stream: ${json['error']}');
      }
      _controller.add(Map<String, dynamic>.from(json));
    } catch (e) {
      _controller.addError('Error parsing stream: $e');
    }
  }

} catch (e) {
  _controller.addError('Error reading stream: $e');
} finally {
  client.close();
}
```

Figure 22: Streaming service

5.5 Meccanismo di Interrogazione dei dati Storici

Il servizio richiede la lista di macchine ottenuta al login; in assenza di tali informazioni, non sarà possibile effettuare operazioni. Il recupero dei dati storici è implementato tramite MachinePastService, che interroga l'endpoint /querying usando parametri quali:

- param e aggr: specifica del campo e del tipo di aggregazione richiesto;
- start_time e stop_time: intervallo temporale della ricerca;
- limit e offset: parametri per la paginazione dei risultati.

Per lo stesso motivo, trattandosi di una chiamata in sola lettura viene utilizzato il metodo GET. L'accesso è protetto tramite il token d'accesso.

Paginazione e condizione di arresto. Per ciascun source_id, viene effettuata un'iterazione su pagine consecutive di risultati, impostando offset e limit. Dopo ogni richiesta:

1. Se la dimensione del batch restituito è inferiore a limit, ciò significa che abbiamo terminato i record da analizzare e quindi l'iterazione si interrompe.
2. In caso contrario, offset viene incrementato della lunghezza del batch e l' iterazione prosegue.

I risultati delle diverse macchine vengono concatenati in un'unica lista.

Struttura della risposta e deserializzazione. La fase di decodifica ha due casistiche:

- Se la risposta è una lista JSON, viene convertita in List<Map<String,dynamic» e restituita, questo per favorire una struttura flessibile.
- Se la risposta è un oggetto JSON singolo, viene trattata come caso non utile ai fini della query storica e restituita una lista vuota.

Affidabilità e retry. Per garantire stabilità al sistema, è stato implementato un metodo di retry fino a tre tentativi in caso di errore di rete o risposta non valida (*Figura 23*):

- Gli errori lato server o di parsing generano un'eccezione che innesca un nuovo tentativo.
- Tra un tentativo e il successivo è previsto un breve intervallo di attesa.
- Al superamento del numero massimo di tentativi, l'eccezione viene rilanciata al chiamante.

Sintesi del flusso.

1. Per ogni macchina in machineId, inizializza offset=0.
2. Costruisce la URI con i parametri forniti (param, aggr, start_time, stop_time, limit, offset).
3. Invoca il metodo di interrogazione passando il token di accesso nell'header.
4. Accoda il batch ai risultati globali e aggiorna offset; interrompe quando la dimensione del batch è inferiore a limit.
5. Al termine, chiude il client HTTP e restituisce allResults, ovvero il contenitore di tutti i risultati.

Le richieste per le diverse macchine vengono eseguite in modo sequenziale, così da contenere il carico sul server e semplificare la gestione degli errori.

```dart
static Future<List<Map<String, dynamic>>> fetchData(
    Uri uri,
    http.Client client, {
      required String accessToken,
      int retries = 3,
    }) async {
  for (int i = 0; i < retries; i++) {
    try {
      final response = await client.get(
        uri,
        headers: {
          'Authorization': 'Bearer $accessToken',
        },
      );

      if (response.statusCode == 200) {
        final decoded = jsonDecode(response.body);
        if (decoded is List) {
          return decoded.cast<Map<String, dynamic>>();
        } else if (decoded is Map) {
          return [];
        } else {
          throw Exception('Unexpected response: ${response.body}');
        }
      } else {
        final errorJson = jsonDecode(response.body);
        throw Exception(
          'Error ${response.statusCode}: ${errorJson['message'] ?? response.body}',
        );
      }
    } catch (e) {
      if (i == retries - 1) rethrow;
      await Future.delayed(Duration(milliseconds: 500));
    }
  }
  return [];
}
```

Figure 23: Funzione con retry automatico per la gestione degli errori

5.6 Sistema di Aggiornamento

Il sistema di aggiornamento automatico dell'applicazione si basa sull'integrazione con il servizio GitHub Releases, sfruttando l'endpoint /releases/latest delle GitHub REST API [9] per verificare la presenza di nuove versioni.

L'uso di questa tecnologia offre un meccanismo centralizzato, sicuro e versionato per la distribuzione degli aggiornamenti, evitando la necessità di ricorrere al Play Store.

La logica di aggiornamento implementa le seguenti fasi:

1. **Recupero versione remota**: Tramite una richiesta HTTP GET verso `https://api.github.com/repos/<repo>/releases/latest` viene acquisita la versione più recente disponibile sul repository remoto (*Figura 24*).

2. **Recupero versione locale**: La versione installata è letta tramite il plugin *package_info_plus* [10].

3. **Confronto delle versioni**: L'applicazione confronta le due stringhe contenenti in una la versione dell'applicazione e nell'altra stringa la versione ricavata dal latest.

4. **Dialog di conferma**: Se è disponibile una nuova versione, viene mostrato un AlertDialog che notifica all'utente la disponibilità dell'aggiornamento e offre la possibilità di avviare il download (*Figura 25*).

5. **Gestione dei permessi**: L'installazione di un APK esterno richiede il permesso REQUEST_INSTALL_PACKAGES. La libreria *permission_handler* [11] è utilizzata per richiederlo e verificarne lo stato.

6. **Download dell'APK**: Il file viene scaricato tramite *http.Client* [12] inviando l'header Accept: application/octet-stream. La directory di destinazione viene ottenuta con *path_provider* [13].

7. **Visualizzazione avanzamento**: Durante il download, un AlertDialog con LinearProgressIndicator mostra lo stato di avanzamento in tempo reale (*Figura 26*).

8. **Installazione nativa**: Al termine del download, l'installazione dell'APK è avviata tramite un MethodChannel [14] verso il codice nativo Android, che esegue l'azione installApk con il percorso del file (*Figura 27*).

```dart
final info = await PackageInfo.fromPlatform();
final localVersion = info.version;

final resp = await http.get(Uri.parse(githubApiUrl));
if (resp.statusCode == 404) {
  //It means there are no releases released yet.
  return;
} else if (resp.statusCode != 200) {
  return;
}


final data = jsonDecode(resp.body);
final remoteVersion = (data['tag_name'] as String).replaceAll('v', '');
```

Figure 24: Recupero della versione remota tramite GitHub API

```dart
showDialog(
  context: ctx,
  builder: (dCtx) => AlertDialog(
    title: const Text('Update available'),
    content: Text(
        'Newest version: $remoteVersion. Would you like to update?'),
    actions: [
      TextButton(
        onPressed: () => Navigator.of(dCtx).pop(),
        child: const Text('Cancel'),
      ),  // TextButton
      TextButton(
        onPressed: () async {
          Navigator.of(dCtx).pop();
          await _requestPermissionsAndDownload(ctx, url);
        },
        child: const Text('Update'),
      ),  // TextButton
    ],
  ),  // AlertDialog
);
```

Figure 25: Dialog che notifica la disponibilità di una nuova versione

```
showDialog(
  context: context,
  barrierDismissible: false,
  builder: (dCtx) {
    return StatefulBuilder(builder: (ctx, setState) {
      setStateDialog = setState;
      return AlertDialog(
        title: const Text('Download Update'),
        content: Column(
          mainAxisSize: MainAxisSize.min,
          children: [
            LinearProgressIndicator(value: total > 0 ? progress : null),
            const SizedBox(height: 16),
            Text(total > 0
                ? ''
                : 'Downloading...'),  // Text
          ],
        ),  // Column
      );  // AlertDialog
    });  // StatefulBuilder
  },
);
```

Figure 26: Finestra di avanzamento durante il download dell'APK

```
const channel = MethodChannel('tirocinio.updater/install');
await channel.invokeMethod('installApk', {'filePath': filePath});
```

Figure 27: Chiamata nativa per installazione dell'APK tramite MethodChannel

5.7 View

LoginView

In questa view sono presenti due campi di input e un pulsante: il primo consente l'inserimento dell'username, il secondo della password, mentre il pulsante rappresenta l'azione di login (*Figura 28*) .

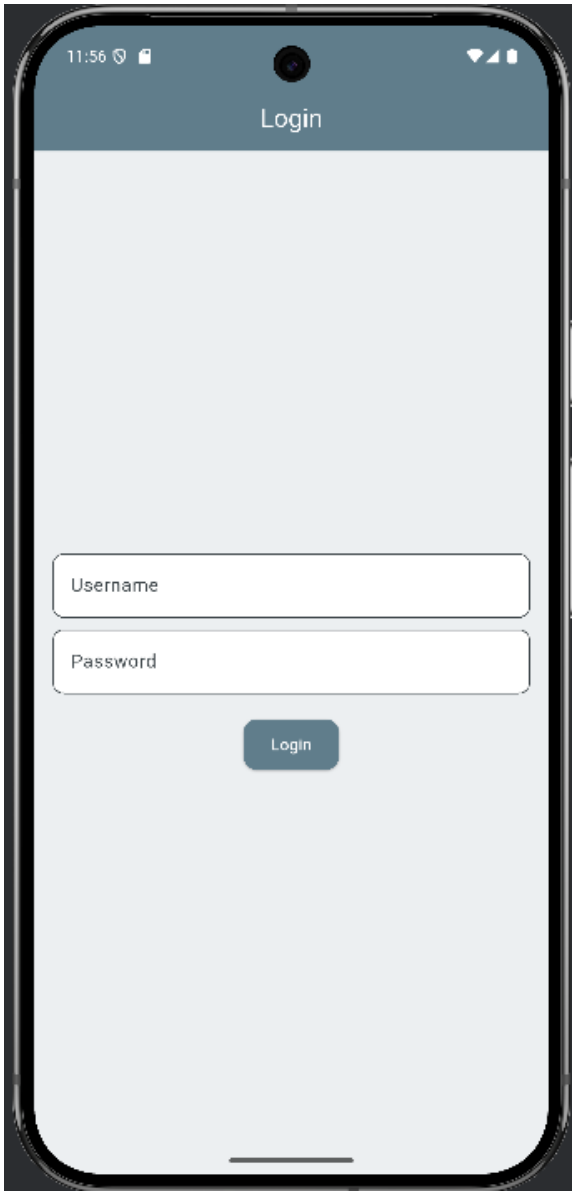


Figure 28: Schermata Login View

Nel file sono inoltre implementati i controlli di validazione (*Figura 29*) necessari affinché la richiesta di login sia accettata: entrambi i campi devono essere compilati. In caso contrario, viene mostrato un messaggio di errore a livello grafico.

```
if (username.isEmpty || password.isEmpty) {
  ScaffoldMessenger.of(context).showSnackBar(
    const SnackBar(content: Text("username and password are required")),
  );
  return;
}
```

Figure 29: Controllo dei campi username e password

I due blocchi di testo sono controllati dai TextEditingController (*Figura 30*). Loro svolgono un ruolo fondamentale nella gestione dei campi di input. In particolare, i due controlli

```
final TextEditingController _usernameController = TextEditingController();
final TextEditingController _passwordController = TextEditingController();
```

Figure 30: TextEditingController della schermata login

vengono utilizzati per associare, rispettivamente, un campo di testo dedicato all'inserimento dell'username e uno dedicato alla password.

L'uso di un TextEditingController consente di avere un collegamento diretto tra il widget grafico TextField e la logica dell'applicazione: non solo permette di leggere in qualsiasi momento ciò che l'utente ha digitato, ma anche di impostare o modificare il testo in maniera programmatica. In questo modo, il controller diventa un "contenitore" dinamico del contenuto inserito dall'utente.

Nel caso dell'_usernameController, l'applicazione potrà recuperare con semplicità il nome utente al momento del login, evitando di dover gestire manualmente il valore del campo di testo. Analogamente, il _passwordController permette di acquisire la password digitata, tipicamente nascosta tramite l'opzione obscureText, così da garantire la riservatezza dei dati inseriti.

Questi controlli, dichiarati come final, mantengono la stessa istanza per tutta la durata di vita dello stato del widget, assicurando stabilità e coerenza nei riferimenti ai campi. Tuttavia, è importante ricordare che devono essere correttamente rilasciati tramite il metodo dispose() all'interno dello State che li utilizza, per evitare perdite di memoria.

In sintesi, i due controller rappresentano lo strumento con cui l'interfaccia utente e la logica applicativa comunicano in modo immediato ed efficace, facilitando la gestione di dati sensibili come credenziali di accesso e rendendo l'interazione più fluida e sicura.

In caso di successo del login, i parametri token e machineList vengono passati alla HomeView. Prima di tale passaggio viene verificato che il token non sia null e che la machineList non sia vuota, poiché in tal caso l'applicazione risulterebbe inutilizzabile.

HomeView

La HomeView è la schermata principale dell'applicazione; riceve in ingresso il token di accesso e la lista delle macchine, e funge da contenitore delle viste secondarie. La dashboard consente quindi di accedere alle schermate dedicate a:

- Monitoraggio dei dati in tempo reale.
- Interrogazione del database per la consultazione dei dati storici.
- personalizzazione della visualizzazione dei grafici.

Oltre a fornire lo switching tra le pagine, la HomeView mette a disposizione un'unica funzionalità propria: il controllo globale dello streaming, ovvero permette di attivare o disattivare il flusso dei dati per tutte le macchine associate all'utente.

Durante l'initState, la HomeView crea un RTViewModel per ciascuna macchina presente nella lista autenticata. Ogni ViewModel (*Figura 31*) è inizializzato con l'accesToken e i parametri identificativi della macchina, ovvero: topic, location, smartObject, sourceId e fields. Inoltre il viewmodel si occupa di attivare lo stream dei dati e del loro aggiornamento verso l'interfaccia grafica.

```
void initState() {
  super.initState();
  _pageController = PageController(initialPage: 0);
  hasDataNotifier = ValueNotifier(false);

  RTViewModel.streamingEnabled = true;

  _viewModels = widget.machines.map((info) {
    final vm = RTViewModel(info.topic, info.location, info.smartObject, info.sourceId,
        info.fields, widget.accessToken);
    vm.onFirstData = () {
      if (!hasDataNotifier.value) {
        hasDataNotifier.value = true;
      }
    };
    return vm;
  }).toList();
}
```

Figure 31: Inizializzazione dei RTViewModel in initState.

La toolbar (Figura 32) superiore ospita: l'icona Modifica per accedere alla configurazione dei grafici e i controlli specifici della vista real-time ovvero l'ordinamento per location e lo switch per attivare/disattivare lo streaming.

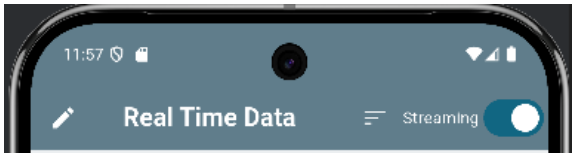


Figure 32: Toolbar con icona di modifica, ordinamento e switch per lo streaming.

L'icona Modifica è abilitata solo dopo la ricezione del primo pacchetto dati, grazie al ValueNotifier<bool> che abilita dinamicamente il pulsante. Il bottone streaming in questa view controlla in modo globale l'aggiornamento in tempo reale di tutti i RTViewModel, permettendo di sospendere o riattivare l'acquisizione con un singolo gesto.

Alla pressione dell'icona Modifica, la HomeView costruisce una mappa macchina e parametri disponibili e istanzia un EditViewModel, quindi apre l'EditView. Se non sono ancora arrivati dati da nessuna macchina, viene mostrato un messaggio d'errore non bloccante.

La seconda pagina del PageView ospita la QueryView, dove l'utente può interrogare i dati storici per macchina e parametri, riutilizzando le informazioni propagate dopo il login.

In sintesi, la HomeView funge da punto di accesso centrale alle funzionalità del sistema, uniformando in un'unica dashboard il monitoraggio real-time, la consultazione storica e la personalizzazione dell'esperienza utente.

RTView

Il widget RTView (Figura 34) utilizza l'istanza di RTViewModel fornita dalla HomeView, senza crearne una nuova ma gestendone il ciclo di vita operativo (Figura 33). In fase di inizializzazione il ViewModel viene recuperato dal contesto e se lo streaming globale risulta abilitato, viene avviato il flusso dati tramite il metodo startStream(). Un secondo controllo viene eseguito all'interno di didChangeDependencies per coprire i casi di reiniezione del Provider, sfruttando Future.microtask per evitare la lettura del contesto durante il frame di init.

```dart
class _RealTimeDataState extends State<RealTimeData> with AutomaticKeepAliveClientMixin {
  late RTViewModel vm;

  @override
  void initState() {
    super.initState();
    Future.microtask(() {
      vm = context.read<RTViewModel>();
      if (RTViewModel.streamingEnabled) {
        vm.startStream();
      }
    });
  }

  @override
  bool get wantKeepAlive => false;

  @override
  void didChangeDependencies() {
    super.didChangeDependencies();
    vm = context.read<RTViewModel>();
    if (RTViewModel.streamingEnabled && vm.machineList.isEmpty) {
      vm.startStream();
    }
  }
```

Figure 33: Recupero del RTViewModel

Durante la fase di build, il widget resta costantemente sincronizzato con il ViewModel attraverso context.watch<RTViewModel>(), reagendo automaticamente ai cambiamenti di stato come l'aggiornamento della lista dati, la variazione del flag di streaming o le preferenze utente. Al tap sulla card, se sono disponibili almeno due campioni, viene aperto un modal bottom sheet che permette la visualizzazione dei grafici. In questa fase vengono prima caricate le preferenze con loadPreferences() e successivamente, in base alle mappe:

- visibleChartMap: utilizzata per determinare se un grafico è da visualizzare o meno attraverso le preferenze dell'utente.
- chartColors: utilizzata per determinare il colore del grafico.
- specGetter: utilizzata per recuperare i dettagli di quel grafico come unità di misura e descrizione.

La lista dei grafici viene costruita dinamicamente la lista dei grafici tramite GraphBuilder.LinearChartRT. La toolbar della card include inoltre uno switch che consente di avviare o fermare lo stream del singolo RTViewModel quindi per ogni singola macchina visualizzata. Tale controllo locale rispetta sempre la proprietà globale definita da RTViewModel.streamingEnabled nella HomeView, garantendo coerenza tra livello locale e livello globale.

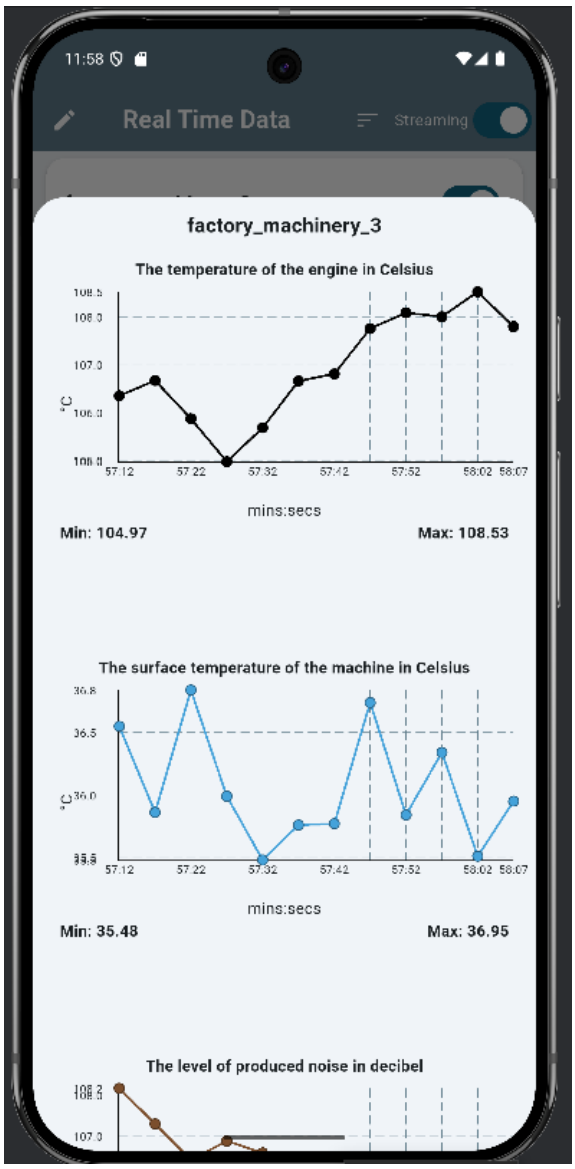


Figure 34: Schermata RTView.

QueryView

La schermata QueryView è un StatefulWidget dedicato all'interrogazione dei dati storici. Riceve in ingresso la lista di macchine autenticate e il token di accesso, crea e fornisce un PastViewModel tramite ChangeNotifierProvider, impostandone subito il token. L'interfaccia è composta da controlli di input (Figura 35) (etichette, menu a tendina, interruttori e campi testuali) che alimentano metodi di validazione e di aggiornamento dello stato nel ViewModel.

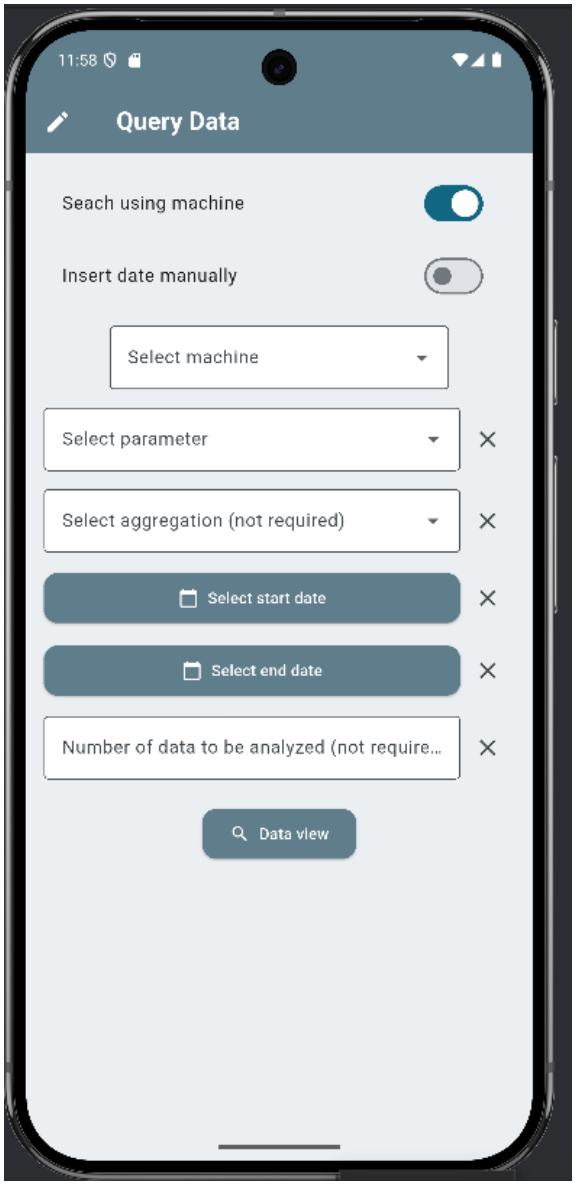


Figure 35: Schermata input della QueryView.

Per la gestione dello stato locale viene mantenuto il flag _bottomSheetOpen (*Figura 36* e *Figura 37*) , che indica se la modal bottom sheet dei risultati è attualmente aperta. Tale variabile viene impostata a true all'apertura e riportata a false alla chiusura; è utilizzata come guardia insieme a mounted per evitare aggiornamenti di stato o azioni quando il widget non è più attivo o la sheet è stata dismessa.

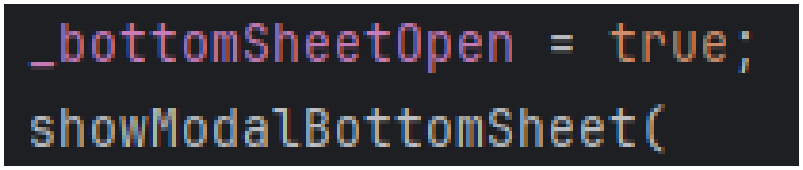


Figure 36: Gestione del booleano _bottomSheetOpen all'apertura.

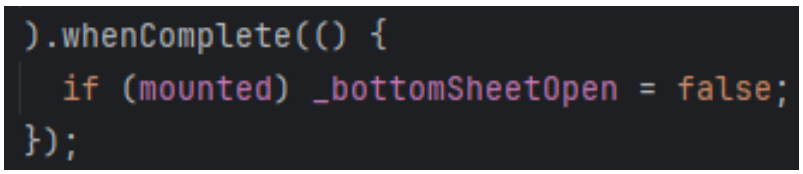


Figure 37: Logica di chiusura del booleano _bottomSheetOpen.

Le opzioni di selezione della macchina (*Figura 38*) (machineOptions) sono costruite mappando ciascun MachineInfo in un DropdownMenuItem: il valore è il sourceId, mentre l'etichetta mostrata deriva dal segmento finale del percorso, utilizzando un split sul valore '/', con gli underscore sostituiti da spazi. In questo modo l'identificativo viene reso leggibile senza alterarne il valore logico.

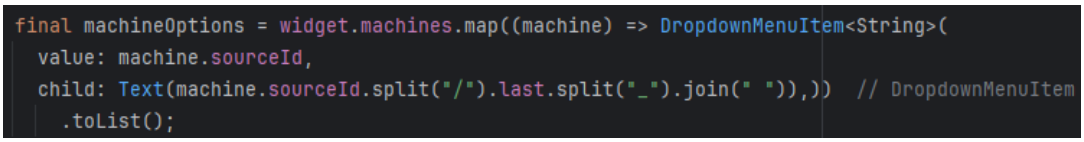


Figure 38: Composizione del DropDownMenuItem per le macchine.

In maniera analoga, le opzioni di parametro parameterOptions e di aggregazione aggregationOptions sono liste predefinite di DropdownMenuItem (*Figura 39 e Figura 40*).

- I parametri comprendono ad esempio temperature, noise_pollution, velocity e vibration.
- Le aggregazioni includono avg, max e min.

La selezione del parametro abilita opzionalmente la selezione dell'aggregazione; in assenza di parametro, l'aggregazione resta disattivata. L'effetto abilitazione/disabilitazione e la coerenza tra i due campi sono demandati a metodi del ViewModel.

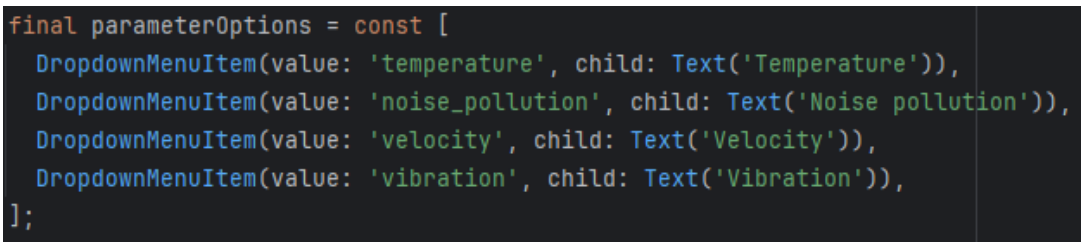


Figure 39: Composizione del DropDownMenuItem per i parametri.

```
final aggregationOptions = const [
  DropdownMenuItem(value: 'avg', child: Text('Average')),
  DropdownMenuItem(value: 'max', child: Text('Maximum')),
  DropdownMenuItem(value: 'min', child: Text('Minimum')),
];
```

Figure 40: Composizione del DropDownMenuItem per le aggregazioni.

La schermata osserva inoltre due proprietà esposte dal ViewModel per la gestione di errori e messaggi informativi:

- vm.errorMessage: se presente, mostra un AlertDialog di errore, azzerato alla chiusura per evitare ripetizioni. La natura dell'errore (validazione, rete, formattazione dati, ecc.) è gestita direttamente nel ViewModel.
- vm.objectSidedAlone: se presente, mostra un AlertDialog informativo con il messaggio "One data found" e i dettagli relativi a quell'oggetto.

Quando vm.groupedResults contiene risultati e vm.hasShownBottomSheet è false, la schermata apre automaticamente una modal bottom sheet (Figura 41) a schermo parziale per la visualizzazione. L'attributo vm.hasShownBottomSheet impedisce riaperture multiple non desiderate dopo la stessa interrogazione.

All'interno della bottom sheet sono definiti tre interruttori locali di visibilità (visiblePieChart, visibleBarChart, visibleLinearChart), inizialmente false, accompagnati dai rispettivi flag di caricamento (isLoadingPie, isLoadingBar, isLoadingLinear). L'attivazione delle icone comporta:

- inversione del flag di visibilità del grafico selezionato;
- impostazione del relativo flag isLoading a true;
- ripristino del flag a false al termine di una breve attesa, se il widget è ancora mounted e la sheet aperta.

Questa gestione avviene tramite StatefulBuilder e setModalState, confinando lo stato alla sola sheet.

Nel caso di grafico lineare, se vm.groupedResults contiene più di un gruppo, la sheet mostra un selettore di macchina. La variabile locale selectedSensorId mantiene l'identificativo selezionato; se null e c'è un solo gruppo disponibile, viene scelto automaticamente. La condizione selectedSensorId != null governa la costruzione dei grafici lineari, mentre isLoadingLinear gestisce l'indicatore di caricamento.

Analogamente, le sezioni Bar e Pie dipendono dai flag visibleBarChart e visiblePieChart. Se attivati mentre isLoadingBar o isLoadingPie sono true, viene mostrato un indicatore di caricamento; al termine i rispettivi flag tornano a false e le sezioni diventano visibili.

Le icone di attivazione dei grafici sono infine mostrate in base alle proprietà booleane vm.showBarChart, vm.showPieChart e vm.showLinearChart, che indicano quali visualizzazioni siano effettivamente disponibili per i risultati correnti.

La parte inferiore della schermata presenta il pulsante Data view, che innesca la validazione e, se superata, l'invio della richiesta:

- Se la ricerca è impostata "per macchina" (vm.searchChooser == true) e non è stata selezionata alcuna macchina, viene invocato vm.showMachineError() e il flusso si interrompe.
- Se la ricerca è "per parametro" (vm.searchChooser == false), viene assegnata a vm.selectedMachine la lista completa delle macchine in ingresso.
- Nella modalità "per parametro", se non è stato scelto alcun parametro, viene invocato vm.showParameterError() e l'azione sospesa.
- Se è attivo l'inserimento manuale delle date (vm.manualDateEntry == true), le validazioni su inizio e fine (vm.startDateValidationColor e vm.endDateVali dationColor) devono risultare corrette.

Solo se tutte le condizioni risultano soddisfatte viene invocato vm.submitQuery(). Durante l'elaborazione, il flag vm.isLoading governa la visualizzazione di un indicatore di progresso al posto del pulsante.

Molte delle condizioni citate (vm.hasShownBottomSheet, vm.groupedResults, vm.showChart, vm.errorMessage, vm.objectSidedAlone, vm.searchChooser, vm.selected Machine, vm.selectedParameter, vm.manualDateEntry, vm.startDateValidationColor, vm. endDateValidationColor, vm.isLoading) dipendono da logiche di business e validazione implementate nel PastViewModel. In questa sede esse sono trattate come condizioni di stato osservate dalla vista; i criteri precisi con cui vengono valorizzate saranno discussi nella sezione dedicata al ViewModel.



Figure 41: Schermata QueryView.

EditView

La schermata EditView è responsabile della personalizzazione della visualizzazione dei grafici associati a ciascuna macchina. Si presenta come uno StatelessWidget che usufruisce di un EditViewModel fornito a monte tramite Provider; l'istanza del ViewModel viene recuperata con context.watch<EditViewModel>() così che l'interfaccia grafica reagisca automaticamente alle variazioni di stato esposte dal modello.

All'avvio, la schermata costruisce un'interfaccia attraverso le macchine note del ViewModel (Figura 42).

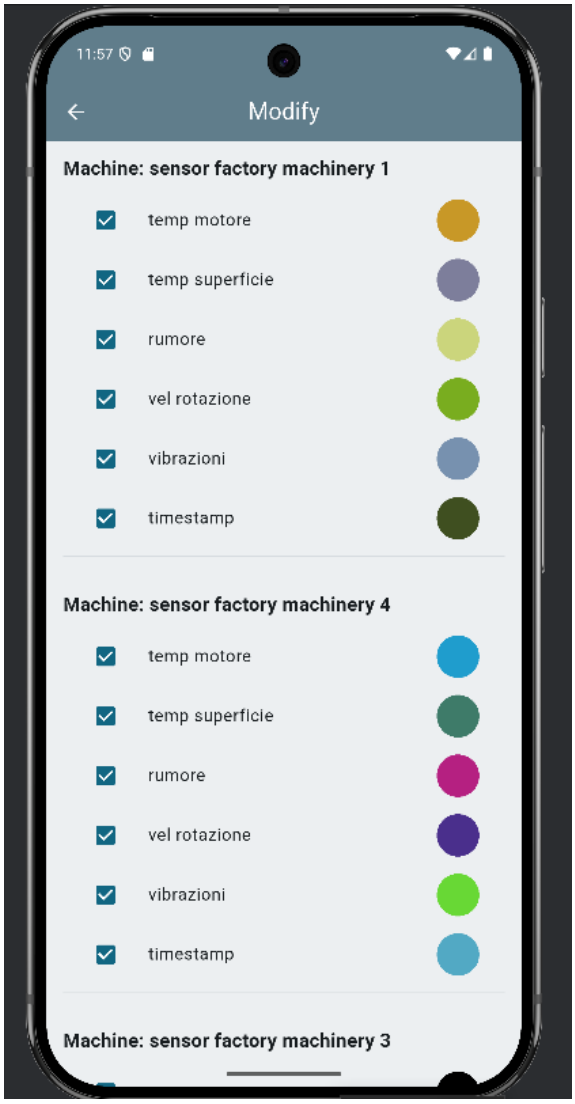


Figure 42: Schermata EditView.

Per ogni macchina viene calcolata un'etichetta leggibile a partire dal relativo identificativo tecnico: il segmento finale del percorso, ricavato attraverso uno split su '/' e sostituendo gli '_' con degli spazi. Tale etichetta è mostrata come intestazione di sezione e, subito sotto, viene presentato l'elenco dei parametri associati a quella macchina.

Per ciascun parametro, la riga è composta da:

- Una label testuale, che identifica il segnale da configurare.
- Un checkbox a sinistra, che indica e controlla la visibilità del grafico relativo a quel parametro; il suo stato iniziale è letto dal ViewModel e, quando l'utente interagisce, l'evento invoca il metodo di controllo toggleVisibility(machineId, param) del ViewModel, così da poter modificare la visibilità.
- Un indicatore di colore a destra (un CircleAvatar), mostrato solo se il parametro è attualmente visibile. Il colore corrente è ottenuto dal ViewModel e rappresenta il colore del grafico per quel parametro.

Il cerchio colore funge da ulteriore input: al tap, viene aperto un AlertDialog contenente un selettore colore creato dalla libreria *flutter_colorpicker*. All'interno del dialogo, l'utente può scegliere un nuovo colore per il parametro selezionato. Il selettore è inizializzato con il colore corrente e, per semplificare la configurazione, la componente di trasparenza è disabilitata. Il dialogo espone due azioni:

1. Cancel, che chiude la finestra senza apportare modifiche.
2. Save, che conferma la scelta e invoca sul ViewModel il metodo di controllo setChartColor(machineId, param, newColor), aggiornando lo stato applicativo e quindi l'interfaccia grafica.

L'organizzazione della schermata prevede una separazione per macchina: ogni gruppo mostra il titolo della macchina, l'elenco dei parametri configurabili e, a chiusura del blocco, una riga divisoria che ne marca la fine prima del gruppo successivo. L'insieme dei parametri per macchina è recuperato dal ViewModel. Lo stato di visibilità e il colore del singolo parametro sono anch'essi forniti dal ViewModel tramite metodi di lettura. In questo modo, tutti i controlli dell'interfaccia (label, checkbox, indicatore di colore e pulsanti del dialogo) agiscono come input ai metodi di controllo del ViewModel, che centralizza la logica di validazione e di persistenza delle preferenze utente.

La parte alta della schermata ospita un AppBar con il titolo Modify.

5.8 ViewModel

Tutti i ViewModel dell'applicazione estendono la classe ChangeNotifier, un componente del framework Flutter che consente di implementare il meccanismo di notifica e aggiornamento tra logica applicativa e interfaccia grafica. Ogni volta che lo stato interno di un ViewModel cambia, viene richiamato il metodo notifyListeners(), che provvede a informare automaticamente tutte le View registrate come listener. In questo modo, i widget interessati vengono ricompilati e la rappresentazione grafica resta coerente con i dati sottostanti. Questo approccio riduce la necessità di aggiornamenti manuali dell'interfaccia e favorisce una netta separazione tra logica e presentazione, garantendo coerenza e manutenibilità del codice.

LoginViewModel

La classe LoginViewModel rappresenta il ViewModel dedicato alla fase di autenticazione e al recupero delle macchine disponibili per l'utente. Essa mantiene una proprietà privata booleana _isLoading, esposta pubblicamente tramite il getter isLoading, che segnala quando un'operazione di login è in corso: durante la richiesta il flag è posto a true, mentre al termine (sia in caso di successo che di fallimento) ritorna a false, permettendo all'interfaccia grafica di mostrare un indicatore di progresso.

Oltre a ciò, il ViewModel gestisce tre variabili principali:

- _accessToken: il token di accesso ottenuto al login, usato per autenticare le chiamate alle API.
- _refreshToken: il token di aggiornamento, utile a richiedere un nuovo access token quando quello corrente scade.
- _machines: la lista di oggetti MachineInfo che rappresentano le macchine associate all'utente autenticato.

Il metodo principale è l'asincrono login che necessita dell'username e della password. Svolge le seguenti operazioni:

1. Imposta _isLoading = true e invoca notifyListeners() per aggiornare l'interfaccia.
2. Chiama il framework di login attraverso la chiamata a LoginService.loginAndFetchMachines, che restituisce access_token, refresh_token e la lista di macchine.
3. In caso di successo assegna i valori ricevuti alle variabili interne.
4. In caso di errore azzera token e macchine, tornando allo stato primario.
5. Al termine riporta _isLoading = false e notifica nuovamente i listener.

RTViewModel

La classe RTViewModel è il cuore della gestione dei dati in tempo reale. Viene istanziata per ciascuna macchina autenticata e ha il compito di ricevere i dati, aggiornarne i valori, mantenere le preferenze utente e notificare l'interfaccia grafica. Ogni istanza è costruita a partire dai parametri identificativi della macchina (topic, location, smartObject, sensorId, fields, accessToken). Il recupero dei dati avviene tramite il framework MachineStreamService, mentre la variabile statica streamingEnabled permette di abilitare o disabilitare globalmente lo streaming. Le informazioni sui campi disponibili sono accessibili tramite la mappa specGetter, che collega ogni fieldPath alla relativa descrizione.

I dati ricevuti sono salvati all'interno di una lista _machineList contenente coppie chiave–valore e relativo timestamp, è stato scelto di mantenere un massimo di 12 elementi per garantire una schermata di dati recenti. Parallelamente, la struttura _maxAndMinValues tiene traccia dei valori minimi e massimi registrati per ciascun parametro numerico.

Il ciclo di vita dello stream è gestito dai metodi startStream() e stopStream(): il primo avvia il flusso, controllando che lo streaming sia abilitato e che non vi siano già streaming attivi. Al primo pacchetto invoca eventualmente la callback onFirstData e carica le preferenze utente, così da evitare caricamenti inutile in casistiche in cui non si riceve nessun dato. Lo stato corrente è verificabile tramite streamActive.

Per la persistenza delle preferenze, il ViewModel utilizza SharedPreferences: loadVisibilityPreferences() legge un file JSON salvato sul dispositivo contenente le impostazioni relative al sensorId corrente, popolando le mappe _visibleChartMap (visibilità) e _chartColors (colori). Se non esistono preferenze, _initializeFromFirstData() inizializza le impostazioni a partire dai dati ricevuti, assegnando colori casuali.

PastViewModel

La classe PastViewModel contiene la logica per l'interrogazione dei dati storici. Il ViewModel riceve e mantiene il token di accesso necessario alle chiamate remote e attraverso variabili osservabili gestisce:

- Modalità di ricerca (searchChooser): per macchina singola oppure per parametro.
- Selezioni utente: macchine, parametro, aggregazione opzionale, limiti numerici, intervallo temporale.
- Flag di controllo: inserimento date manuali, stato di caricamento, invio effettuato.
- Messaggi di validazione o errore: errorMessage, machineValidationMessage, parameterValidationMessage, aggregationValidationMessage, e messaggi informativi come objectSidedAlone.
- Campi di supporto per la validazione delle date, con messaggi e colori di stato.

I risultati ottenuti dal MachinePastService vengono organizzati come segue:

- Raggruppati per sorgente in una mappa results.
- Conteggiati per sorgente.
- Ordinati temporalmente rispetto al parametro timestamp.
- Analizzati per la costruzione di estremi min/max per ogni campo numerico.

Se non sono presenti dati viene generato un errore, mentre se il risultato è unitario viene valorizzata la stringa objectSidedAlone per consentire alla vista di informare l'utente, ovvero dove l'output del framework è composto da un solo oggetto.

Il ciclo di vita di una query prevede l'azzeramento dei risultati precedenti, l' impostazione dello stato di caricamento e il reset dei messaggi di errore. La visualizzazione di una tipologia di un grafico è determinato da queste tre variabili booleane (*Figura 43*): showBarChart, showPieChart e showLinearChart. Il codice è strutturato in modo tale da comprendere in quale casistica è possibile visualizzare uno di essi:

- grafico a torta: quando non c'è aggregazione e la selezione comprende più di una macchina/sensore;
- grafico lineare: quando non è richiesta un'aggregazione, mostrando le serie temporali originali;
- grafico a barre: quando è presente un'aggregazione (media, massimo, minimo).

```
if (selectedMachine.length != 1 && selectedAggregation == null) {
  showPieChart = true;
}
if (selectedAggregation == null) {
  showLinearChart = true;
} else {
  showBarChart = true;
}
```

Figure 43: Condizione di visualizzazione grafici.

La validazione degli input è anch'essa centralizzata:

- in modalità "per macchina" è richiesta almeno una macchina;
- in modalità "per parametro" è richiesto almeno un parametro, altrimenti viene generato un errore;
- il limite opzionale limit rappresenta un parametro numerico aggiuntivo per la query;

- le date possono essere inserite manualmente (con formato yyyy-MM-dd HH:mm[:ss] e completamento automatico delle parti mancanti) oppure tramite selettore grafico. In entrambi i casi, il ViewModel aggiorna messaggi e colori di validazione per guidare l'utente.

Sono previsti metodi di reset per riportare i campi allo stato iniziale. Il ViewModel fornisce infine metodi di supporto che fungono da getter per descrizione dei campi e unità di misura, usati dai componenti grafici per mostrare tooltip, assi e didascalie coerenti.

EditViewModel

La classe EditViewModel si occupa della personalizzazione della visualizzazione dei grafici per ciascuna macchina, mantenendo e persistendo le preferenze utente relative a visibilità e colore dei parametri. In ingresso riceve una mappa machineParamsMap che definisce i parametri configurabili per ogni macchina. Le preferenze sono organizzate in una mappa annidata. All'istanziazione il ViewModel carica asincronamente le preferenze salvate; se non ne trova, crea da zero le impostazioni di default per tutti i parametri disponibili.

Il viewmodel si occupa di gestire:

- Il recupero delle informazioni: include la variabile visibleCharts ovvero la mappa completa delle preferenze. Il metodo isChartVisible utilizza l'identificativo della macchina e il suo parametro per verificare se il grafico associato al parametro condiviso è abilitato, e la funzione getChartColor utilizzando gli stessi parametri per ottenere il colore associato al grafico di quel parametro.

- Operazioni di modifica: comprendono la funzione toggleVisibility(macchina, parametro) per attivare o disattivare un parametro, e il metodo setChartColor(macchina, parametro, colore) per assegnare un nuovo colore al grafico corrispondente.

Ogni modifica aggiorna la struttura interna, salva le preferenze tramite SharedPreferences e invoca notifyListeners() per aggiornare l'interfaccia grafica.

La persistenza sfrutta un oggetto JSON salvato con la chiave visibleCharts. Voci relative a macchine non più presenti vengono scartate, e il ViewModel garantisce sempre che tutte le macchine e i parametri abbiano una preferenza valida altrimenti vengono assegnati colori casuali dove necessario.

5.9 Utils e componenti riutilizzabili

GraphBuilder

La classe GraphBuilder fornisce un insieme di metodi statici per la creazione di grafici riusabili, implementati mediante la libreria *fl_chart*. Non gestisce logica applicativa o stato duraturo, ma riceve dati pre-elaborati dalle Views e li trasforma in widget grafici pronti alla visualizzazione. I principali metodi includono:

- LinearChartRT: grafico lineare per dati real-time, con etichette temporali e tooltip dettagliati.
- ColumnChart: grafico a barre con supporto alla paginazione dei dati, tooltip e finestre di dettaglio.
- PieChartWidget: grafico a torta che mostra le percentuali relative a un insieme di categorie, accompagnato da legenda.
- LinearChartQuery: grafico lineare per dati storici con navigazione a pagine e possibilità di configurare l'intervallo di punti da visualizzare per pagina.

Ogni metodo si occupa esclusivamente della parte di presentazione, mentre le descrizioni dei campi e le unità di misura vengono ricavate con due metodologie diverse:

- Nel primo caso, ovvero per i grafici realtime, i dati vengono passati dalla view.
- Nel secondo caso, ovvero per i grafici che sfruttano i dati ricevuti dalle query, si utilizza il PastViewModel per ricavare tutti i dettagli.

Il file è stato collocato all'interno della cartella Utils al fine di semplificare eventuali modifiche future e di disporre di un contenitore centralizzato per tutti i grafici implementati. In tal modo, esso rappresenta una componente di supporto all'applicazione, progettata per essere facilmente riutilizzabile.

5.10 Miglioramenti

L'applicazione è già ben strutturata e pienamente funzionante rispetto al suo obiettivo principale, ovvero quello di osservare l'andamento e le variazioni nel tempo dei dati provenienti dalle macchine e dai sensori.

In quest'ottica, un possibile miglioramento consiste nell'introduzione di una funzionalità di comparazione tra due o più macchine e/o sensori, sia nella componente di monitoraggio in tempo reale sia in quella di consultazione storica. Ciò permetterebbe di evidenziare con maggiore immediatezza differenze, correlazioni e anomalie tra i diversi flussi informativi.

Un ulteriore sviluppo utile potrebbe essere l'aggiunta di una sezione dedicata ai log di sistema. Tale sezione, implementabile secondo diverse metodologie, avrebbe il fine di tracciare e archiviare in maniera strutturata i dati anomali rilevati dalle macchine, corredandoli con informazioni aggiuntive quali la provenienza e la data di registrazione. Questo consentirebbe una più efficace attività di diagnostica e analisi post-evento.

Infine, un miglioramento significativo potrebbe riguardare l'estensione dell' applicazione anche in versione Web. Considerata la duttilità del linguaggio Dart, risulterebbe relativamente semplice ospitare l'applicativo su piattaforme di hosting dedicate, rendendolo così ancora più facilmente accessibile e fruibile da qualsiasi dispositivo, non solo mobile.

Conclusioni

Quest'elaborato ha portato allo sviluppo di un'applicazione mobile per il monitoraggio e la visualizzazione dei dati sensoriali in ambito industriale, concepita come strumento flessibile e scalabile, in linea con i principi dell'Industria 5.0 e con il progetto PRIN 2022 HOMEY.

Nella fase di Raccolta ed Analisi dei Requisiti sono stati definiti attori e casi d'uso, delineando con chiarezza le funzionalità attese e le caratteristiche qualitative del sistema. Questa analisi ha guidato la successiva fase di progettazione, durante la quale sono stati stabiliti gli strumenti utilizzati, il pattern architetturale utilizzato, i flussi di interazione tra componenti e sono state definite le interfacce. Su tali basi si è innestata l'implementazione, che ha tradotto le decisioni progettuali in componenti concrete e riusabili, organizzate secondo una struttura coerente con il pattern utilizzato e supportate da utility condivise.

Il capitolo dedicato al Framework ha illustrato i servizi e le API disponibili, la gestione dell'autenticazione e i meccanismi per lo scambio dati, sia in streaming sia su interrogazione.

I risultati ottenuti dimostrano che la soluzione proposta soddisfa gli obiettivi prefissati: i dati in tempo reale sono visualizzati tramite grafici che si aggiornano dinamicamente; le interrogazioni storiche consentono differenti modalità di rappresentazione in funzione delle esigenze; l'utente può personalizzare visibilità e colori dei grafici; l'autenticazione e il controllo dello streaming sono gestiti in modo coerente con le scelte architetturali; il sistema di aggiornamento verifica la disponibilità di nuove versioni e ne abilita l'installazione. Tali funzionalità sono state ottenute mantenendo l'attenzione su modularità, riuso e chiarezza delle responsabilità.

L'applicazione, nella sua versione attuale, soddisfa pienamente lo scopo per cui è stata progettata. Nonostante ciò, sono stati delineati alcuni possibili sviluppi volti a potenziarne le funzionalità. Tra le proposte figura l'introduzione di un meccanismo di confronto tra più macchine o sensori, utile a mettere in risalto eventuali relazioni o discrepanze tra i dati raccolti. È stata inoltre suggerita la realizzazione di una sezione dedicata ai log di sistema, che permetta di registrare e analizzare in modo più strutturato gli eventi anomali. Infine, si è ipotizzata un'estensione della piattaforma anche in versione Web, così da renderla accessibile da qualsiasi dispositivo e non solo in ambiente mobile.

In sintesi, il progetto ha raggiunto tutti gli obiettivi iniziali, con un percorso metodologico coerente che ne ha garantito solidità e completezza. Il lavoro pone basi concrete per sviluppi futuri volti a migliorare la scalabilità e la centralità dell'utente nel sistema.

Bibliography

[1] Par-Tec S.p.A. – *AI e fattore umano – Verso l'Industria 5.0*. AISS (Associazione Italiana per la Sicurezza Informatica), 2024. https://www.par-tec.it/wp-content/uploads/2024/11/AI-e-FattoreUmano-AISS_Industria-5.0.pdf

[2] Wikipedia – *Analisi dei requisiti*. https://it.wikipedia.org/wiki/Analisi_dei_requisiti

[3] Wikipedia – *Framework*. https://it.wikipedia.org/wiki/Framework

[4] Red Hat – *What is a REST API?*. https://www.redhat.com/it/topics/api/what-is-a-rest-api

[5] Wikipedia – *Server-sent events*. https://en.wikipedia.org/wiki/Server-sent_events

[6] Flutter API – *Stream controller*. https://api.flutter.dev/flutter/dart-async/StreamController-class.html

[7] fl_chart package – Pub.dev. https://pub.dev/packages/fl_chart

[8] Dart API Reference – *LineSplitter class*. https://api.dart.dev/dart-convert/LineSplitter-class.html

[9] GitHub – *REST API: Get the latest release*. https://docs.github.com/en/rest/releases/releases#get-the-latest-release

[10] Flutter – *package_info_plus*. https://pub.dev/packages/package_info_plus

[11] Flutter – *permission_handler*. https://pub.dev/packages/permission_handler

[12] Flutter – *http*. https://pub.dev/packages/http

[13] Flutter – *path_provider*.
https://pub.dev/packages/path_provider

[14] Flutter Docs – *Platform channels*.
https://docs.flutter.dev/platform-integration/platform-channels

[15] FlutterApp.it – *Linguaggio Dart di Flutter*.
https://flutterapp.it/

[16] Flutter Docs – *Simple app state management*.
https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple

[17] Wikipedia – *Model–View–ViewModel*.
https://it.wikipedia.org/wiki/Model-view-viewmodel

[18] M. M. Sciarroni, M. Esposito, P. Pierleoni, and E. Storti, *"Monitoring Data Streams in Industry 5.0: a Knowledge Graph Approach,"* in *Proceedings of the 2024 IEEE 8th Forum on Research and Technologies for Society and Industry Innovation (RTSI)*, IEEE, 2024.
https://ieeexplore.ieee.org/document/10761601