NoTIP

NO TESTS IN PRODUCTION

Specifica tecnica - Data API

Versione	1.0.0
Data Modifica	2026-04-13
Utilizzo	Esterno

Abstract dei contenuti

Specifica tecnica del microservizio data-api: architettura logica, contratti *REST*$_G$, port applicativi, design di dettaglio dei componenti, gestione errori e strategia di test del servizio di consultazione delle misure cifrate.

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
1.0.0	2026-04-13	Leonardo Preo (responsabile)		Approvazione per ingresso in *baseline*$_G$ PB
0.2.3	2026-04-07	Matteo Mantoan	Francesco Marcon	Uniformazione titoli e intestazioni: solo prima lettera maiuscola
0.2.2	2026-04-01	Alessandro Contarini	Francesco Marcon	Aggiunta parti di stream *SSE*$_G$ e *NATS*$_G$
0.2.1	2026-04-01	Alessandro Mazzariol	Alessandro Contarini	Fix indentazione
0.2.0	2026-03-31	Alessandro Contarini	Alessandro Mazzariol	Stesura bozza completa del documento
0.1.0	2026-03-27	Alessandro Contarini	Alessandro Mazzariol	Sviluppo bozza del documento
0.0.1	2026-03-27	Alessandro Contarini	Alessandro Mazzariol	Creazione del documento prima bozza del documento

Indice

1. Introduzione ... 6
2. Dipendenze e configurazione ... 6
 2.1. Variabili d'ambiente ... 6
 2.2. Sequenza di avvio ... 6
3. Architettura logica ... 7
 3.1. Strati Architetturali ... 7
4. Design di dettaglio ... 8
 4.1. Moduli del microservizio ... 8
  4.1.1. MeasureModule ... 8
   4.1.1.1. MeasureController ... 8
   4.1.1.2. MeasureService ... 9
   4.1.1.3. MeasurePersistenceService ... 10
   4.1.1.4. StreamListenerService ... 11
   4.1.1.5. TelemetryStreamBridgeService ... 11
   4.1.1.6. CostNatsResponderService ... 12
  4.1.2. SensorModule ... 13
   4.1.2.1. SensorController ... 13
   4.1.2.2. SensorService ... 13
  4.1.3. Auth (*Tenant*$_G$ Access) ... 14
   4.1.3.1. TenantAccessGuard ... 14
   4.1.3.2. TenantId Decorator ... 14
   4.1.3.3. Interfacce ... 14
  4.1.4. MetricsModule ... 14
   4.1.4.1. MetricsService ... 14
   4.1.4.2. MetricsController ... 15
   4.1.4.3. MetricsInterceptor ... 15
  4.1.5. Database ... 15
 4.2. Entità ... 16
 4.3. Decisioni implementative ... 16
 4.4. Flussi di esecuzione ... 17
  4.4.1. Query paginata delle misure ... 17
  4.4.2. Export completo delle misure ... 17
  4.4.3. Streaming delle misure ... 17
  4.4.4. Elenco dei sensori disponibili ... 18
  4.4.5. Validazione *tenant*$_G$ (TenantAccessGuard) ... 18
  4.4.6. Bridge *NATS*$_G$ -> *SSE*$_G$ (TelemetryStreamBridgeService) ... 18
5. Test e verifica ... 19
6. Considerazioni finali ... 19

Indice delle figure

Figura 1 Architettura del microservizio 7

Indice delle tabelle

Tabella 1 Variabili d'ambiente richieste da data-api ... 6
Tabella 2 Sequenza di avvio del microservizio data-api ... 6
Tabella 3 Strati architetturali ... 7
Tabella 4 *Endpoint*$_G$ esposti da MeasureController ... 8
Tabella 5 Metodi privati di MeasureController ... 9
Tabella 6 Campi di MeasureService ... 9
Tabella 7 Costanti di MeasureService ... 9
Tabella 8 Metodi pubblici di MeasureService ... 9
Tabella 9 Metodi privati di MeasureService ... 9
Tabella 10 Campi di MeasurePersistenceService ... 10
Tabella 11 Metodi privati di MeasurePersistenceService ... 10
Tabella 12 Campi di StreamListenerService ... 11
Tabella 13 Metodi pubblici di StreamListenerService ... 11
Tabella 14 Metodi privati di StreamListenerService ... 11
Tabella 15 Campi di TelemetryStreamBridgeService ... 11
Tabella 16 Metodi pubblici di TelemetryStreamBridgeService ... 12
Tabella 17 Metodi privati di TelemetryStreamBridgeService ... 12

1. Introduzione

Questo documento illustra l'architettura interna e le scelte implementative del microservizio `data-api`. Sviluppato in *NestJS*$_G$, questo componente è responsabile dell'esposizione di funzionalità di consultazione delle misure cifrate raccolte dal sistema. Il servizio rende disponibili *endpoint*$_G$ HTTP per l'interrogazione paginata delle misure, l'esportazione completa di un intervallo temporale, la fruizione in streaming dei dati e l'elenco dei sensori osservati di recente.

L'obiettivo del componente è fornire un punto di accesso unificato ai dati telemetrici già acquisiti e persistiti, mantenendo separata la logica di esposizione API dalla logica di accesso alla persistenza. Il servizio restituisce verso i client esclusivamente *payload*$_G$ cifrati e *metadati*$_G$ tecnici associati alla misura, senza effettuare operazioni di decifratura.

Il progetto è strutturato in moduli *NestJS*$_G$ distinti per le funzionalità `measure` e `sensor`, con utilizzo di *DTO*$_G$ per i contratti esposti, model interni per la logica applicativa e servizi dedicati per orchestrazione, filtraggio e accesso ai dati.

2. Dipendenze e configurazione

2.1. Variabili d'ambiente

Tutte le variabili d'ambiente necessarie per il funzionamento del microservizio sono elencate di seguito, un'eventuale mancanza di una di queste variabili comporterà un errore all'avvio del microservizio:

Campo	Variabile d'ambiente	Default	Obbligatorio
`ApiPort`	DATA_API_PORT	`3000`	No
`DBHost`	MEASURES_DB_HOST	-	Sì
`DBPort`	MEASURES_DB_PORT	`5432`	No
`DBName`	MEASURES_DB_NAME	-	Sì
`DBUser`	MEASURES_DB_USER	-	Sì
`DBPassword`	MEASURES_DB_PASSWORD	-	Sì
`DBSSL`	DB_SSL	`false`	No
`NatsUrl`	NATS_URL	-	No
`NatsServers`	NATS_SERVERS	-	No
`MgmtApiUrl`	MGMT_API_URL	`http://management-api:3000`	Sì

Tabella 1: Variabili d'ambiente richieste da data-api

2.2. Sequenza di avvio

I passi bloccanti interrompono l'avvio del microservizio, pertanto è necessario assicurarsi che tutti i servizi esterni siano operativi prima di avviare `notip-data-api`. La sequenza di avvio è la seguente:

Step	Componente	Azione	Bloccante?
0	.env	Carica le variabili d'ambiente del servizio	Si
1	env.validation	Verifica la validità delle variabili d'ambiente	Si

2	bootstrap.*nestjs*G	Inizializza i moduli, controller e provider dell'applicazione *NestJS*G	Si
3	app.module	Inizializza TypeORM e registra i moduli Measure e Sensor	Si
4	auth.guard	Registra il TenantAccessGuard globale per la validazione del *tenant*G	Si
5	metrics.module	Registra MetricsService e MetricsInterceptor per il monitoraggio	Si
6	*main*G	Crea l'applicazione *NestJS*G e avvia il listener HTTP sulla porta configurata	Si

Tabella 2: Sequenza di avvio del microservizio data-api

3. Architettura logica

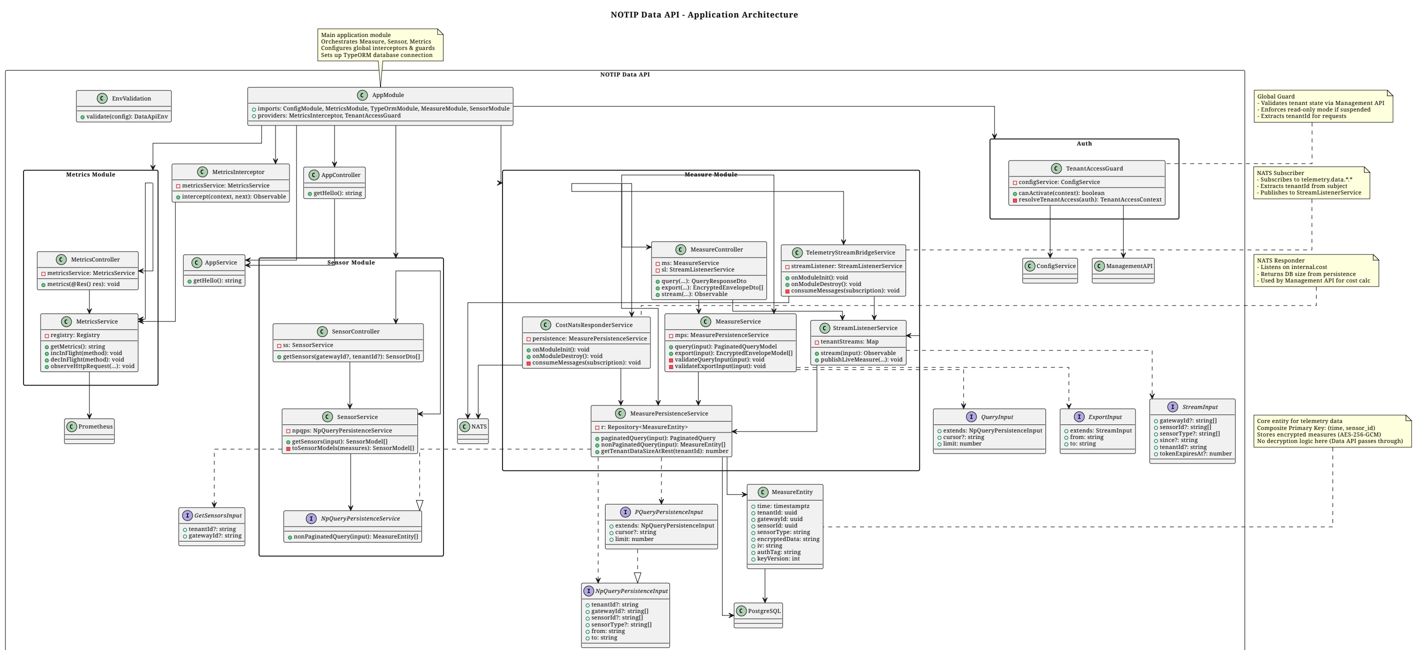

Figura 1: Architettura del microservizio

Il servizio adotta una Layered Architecture con organizzazione interna suddivisa per moduli logici. All'interno dei vari moduli è utilizzato prevalentemente il pattern Controller-Service-Persistence, che consente una chiara separazione delle responsabilità tra esposizione API, logica di business e accesso ai dati. I componenti collaborano tramite Dependency Injection e, dove opportuno, tramite interfacce e contratti applicativi. La presenza di Business Models, *DTO*G ed Entities ha portato all'introduzione di Mappers per la conversione dei dati tra i diversi livelli dell'applicazione.

3.1. Strati Architetturali

Strato	Package	Contenuto

Presentation	`src/app.controller.ts`
	`src/data-api/controller`
	`src/data-api/dto`G
		Gestione delle richieste HTTP, esposizione delle API *REST*G, validazione dei *payload*G,

	src/data-api/ openapi.decorators.ts	autenticazione e definizione dei contratti di ingresso/uscita dei dati.
Business	src/data-api/services src/data-api/models src/data-api/interfaces src/data-api/measure.mapper.ts	Logica applicativa e di dominio: validazione dei parametri di query, orchestrazione dei casi d'uso, trasformazione tra entity/model/*DTO*G, filtraggio dei dati e definizione delle interfacce tra componenti.
Persistence	src/data-api/entity src/data-api/services/ measure.persistence.service.ts	Accesso ai dati tramite TypeORM, definizione dell'entità `MeasureEntity`, costruzione delle query paginated e non-paginated su *PostgreSQL*G e incapsulamento delle operazioni di persistenza.

Tabella 3: Strati architetturali

4. Design di dettaglio

4.1. Moduli del microservizio

4.1.1. MeasureModule

Gestione delle funzionalità di interrogazione ed esportazione delle misure. Espone gli *endpoint*G dedicati, orchestra la logica applicativa tramite `MeasureService` e delega l'accesso ai dati a `MeasurePersistenceService` usando TypeORM sull'entità `MeasureEntity`, responsabile dell'esposizione degli *endpoint*G per la consultazione delle misure cifrate. Gestisce query paginate, export completo e streaming *SSE*G.

4.1.1.1. MeasureController

Controller *NestJS*G esposto sotto il prefisso `/measures`. Gestisce tre *endpoint*G principali.

Metodo	*Endpoint*G	Note
`query(...)`	`GET /measures/query`	Query paginata con cursor-based pagination; valida limit <= 999 e finestra <= 24h
`stream(...)`	SSEG `/measures/stream`	*Server-Sent Events*G; replay storico + live; termina con `token_expired` alla scadenza del *JWT*G
`export(...)`	`GET /measures/export`	Export completo senza paginazione; valida finestra <= 24h

Tabella 4: *Endpoint*G esposti da MeasureController

Metodi privati:

Funzione	Comportamento
`normalizeLimit(value)`	Parsing del parametro limit; default 999; scarta valori non interi
`normalizeArrayParam(value)`	Converte parametro singolo o array in `string[]`
`parseBearerToken(authorization)`	Estrae il token Bearer dall'header Authorization
`extractTokenExpiresAt(authorization)`	Decodifica il *payload*G *JWT*G e estrae il claim `exp` in millisecondi

Tabella 5: Metodi privati di MeasureController

4.1.1.2. MeasureService

Contiene la logica di business per query ed export. Coordina `MeasurePersistenceService`.

Campi:

Campo	Tipo	Note
`mps`	`MeasurePersistenceService`	Iniettato via constructor

Tabella 6: Campi di MeasureService

Costanti:

Costante	Valore	Note
`MAX_QUERY_LIMIT`	`999`	Limite massimo di righe per pagina
`MAX_WINDOW_MS`	`86400000`	Finestra temporale massima: 24 ore

Tabella 7: Costanti di MeasureService

Metodi pubblici:

Metodo	Firma	Note
`query(input)`	`(input: QueryInput): Promise<PaginatedQueryModel>`	Valida input; chiama `mps.paginatedQuery()`; mappa risultato via `MeasureMapper`
`export(input)`	`(input: ExportInput): Promise<EncryptedEnvelopeModel[]>`	Valida input; chiama `mps.nonPaginatedQuery()`; mappa risultato via `MeasureMapper`

Tabella 8: Metodi pubblici di MeasureService

Metodi privati:

Metodo	Comportamento
`validateQueryInput(input)`	Verifica `limit <= MAX_QUERY_LIMIT` (code `QUERY_LIMIT_EXCEEDED`); chiama `validateWindow`
`validateExportInput(input)`	Chiama `validateWindow` con code `EXPORT_WINDOW_EXCEEDED`
`validateWindow(from, to, code)`	Parsa le date; se finestra > 24h solleva `BadRequestException` con il code fornito
`handleQueryError(error)`	Mappa errori: 400/BadRequest, 401/Unauthorized, 403/Forbidden

handleExportError(error)	Mappa errori: 400/BadRequest, 401/Unauthorized, 403/Forbidden

Tabella 9: Metodi privati di MeasureService

4.1.1.3. MeasurePersistenceService

Layer di accesso ai dati. Implementa NpQueryPersistenceService. Costruisce query TypeORM sull'entità MeasureEntity (tabella telemetry).

Campi:

Campo	Tipo	Note
repository$_G$	Repository$_G$<MeasureEntity>	*Repository$_G$* TypeORM iniettato

Tabella 10: Campi di MeasurePersistenceService

Metodi pubblici:

Metodo	Firma	Note
paginatedQuery(p)	(p: PQueryPersistenceInput): Promise<PaginatedQuery>	Query con filtri (tenantId, gatewayId IN, sensorId IN, sensorType IN, time range) e cursor-based pagination composta (time, sensorId). Fetch limit + 1 righe per determinare hasMore
nonPaginatedQuery(n)	(n: NpQueryPersistenceInput): Promise<MeasureEntity[]>	Query senza paginazione; stessi filtri; ordine time DESC
getTenantDataSizeAtRest(_tenantId)	(_tenantId: string): Promise<number>	Esegue SELECT pg_database_size (current_database()): :bigint; restituisce dimensione in byte

Metodi privati:

Metodo	Comportamento
applyScalarFilter(qb, column, param, value)	Aggiunge WHERE column = :param se value è definito
applyArrayFilter(qb, column, param, values)	Aggiunge WHERE column IN (:...param) se l'array ha elementi
parseCompositeCursor(cursor)	Splitta il cursor sull'ultimo |; restituisce { time, sensorId } o undefined
toCompositeCursor(time, sensorId)	Formatta come time|sensorId

Tabella 11: Metodi privati di MeasurePersistenceService

4.1.1.4. StreamListenerService

Gestisce lo streaming RxJS delle misure. Crea un Subject per *tenant*$_G$ e gestisce il replay storico e gli eventi live.

Campi:

Campo	Tipo	Note
tenantStreams	Map<string, Subject<EncryptedEnvelopeModel>>	Soggetti per *tenant*$_G$ (lazy-created)
mps	MeasurePersistenceService	Iniettato per replay storico

Tabella 12: Campi di StreamListenerService

Metodi pubblici:

Metodo	Firma	Note
stream(input)	(input: StreamInput): Observable<StreamEmission>	Concatena replay storico + live events; filtra per gatewayId/sensorId/sensorType; termina con token_expired
publishLiveMeasure(tenantId, event)	(tenantId: string, event: EncryptedEnvelopeModel): void	Pubblica una misura live sul Subject del *tenant*$_G$

Tabella 13: Metodi pubblici di StreamListenerService

Metodi privati:

Metodo	Comportamento
replayHistorical(input)	Se since è definito, query [since, now] ed emette ogni misura come evento data
listenToSource(input)	Sottoscrive al Subject del *tenant*$_G$ e wrappa come Observable
getTenantStream(tenantId?)	Crea lazy il Subject per il *tenant*$_G$ (default: 'anonymous')
matchesFilters(event, input)	Verifica se l'evento soddisfa i filtri gatewayId/sensorId/sensorType (OR dentro ogni array)

Tabella 14: Metodi privati di StreamListenerService

4.1.1.5. TelemetryStreamBridgeService

Bridge tra *NATS*$_G$ e lo streaming RxJS. Si sottoscrive a telemetry.data.*.* e pubblica le misure live sullo stream del *tenant*$_G$ corrispondente.

Campi:

Campo	Tipo	Note
logger	Logger	Logger interno
connection	NatsConnection | null	Connessione *NATS*$_G$
subscription	Subscription | null	Sottoscrizione *NATS*$_G$
streamListener	StreamListenerService	Iniettato per pubblicare misure live

Tabella 15: Campi di TelemetryStreamBridgeService

Metodi pubblici:

Metodo	Firma	Note
onModuleInit()	(): Promise<void>	Salta in test mode; chiama connectAndSubscribe()
onModuleDestroy()	(): Promise<void>	Unsubscribe, drain e close della connessione *NATS*$_G$

Tabella 16: Metodi pubblici di TelemetryStreamBridgeService

Metodi privati:

Metodo	Comportamento
connectAndSubscribe()	Connette a *NATS*$_G$ con *TLS*$_G$/token/user-pass; sottoscrive telemetry.data.*.*; avvia consumeMessages()
consumeMessages(subscription)	Itera messaggi *NATS*$_G$; estrae tenantId dal subject; pars envelope; pubblica su streamListener
buildConnectionOptions()	Costruisce opzioni *NATS*$_G$: servers da env, *TLS*$_G$ *mTLS*$_G$, token o user/pass
extractTenantId(subject)	Estrae tenantId da telemetry.data.{tenantId}.{something} (parte 2)
parseEnvelope(data)	*JSON*$_G$-parsa i dati e valida tutti i campi richiesti; restituisce undefined se invalido

Tabella 17: Metodi privati di TelemetryStreamBridgeService

4.1.1.6. CostNatsResponderService

Responder *NATS*$_G$ per le richieste di costo. Risponde su internal.cost con la dimensione del database.

Campi:

Campo	Tipo	Note
logger	Logger	Logger interno
connection	NatsConnection | null	Connessione *NATS*$_G$
subscription	Subscription | null	Sottoscrizione *NATS*$_G$
persistence	MeasurePersistenceService	Iniettato per ottenere la dimensione DB

Metodi pubblici:

Metodo	Firma	Note
onModuleInit()	(): Promise<void>	Salta in test mode; chiama connectAndSubscribe()
onModuleDestroy()	(): Promise<void>	Unsubscribe, drain e close della connessione *NATS*$_G$

Metodi privati:

Metodo	Comportamento
`connectAndSubscribe()`	Connette a *NATS*$_G$; sottoscrive `internal.cost`; avvia `consumeMessages()`
`consumeMessages(subscription)`	Itera messaggi; estrae tenantId; chiama `persistence.getTenantDataSizeAtRest()`; risponde
`extractTenantId(data)`	*JSON*$_G$-parsa la richiesta e estrae `tenant_id`
`respondWithCost(message, dataSizeAtRest)`	Risponde su *NATS*$_G$ con `{ dataSizeAtRest }`

4.1.2. SensorModule

Gestione delle funzionalità di discovery e consultazione dei sensori disponibili. Espone gli *endpoint*$_G$ relativi ai sensori, utilizza `SensorService` per costruire la vista logica dei sensori a partire dalle misure persistite e riusa `MeasurePersistenceService` come dipendenza di persistenza. Riusa `MeasurePersistenceService` come dipendenza di persistenza tramite il token `NP_QUERY_PERSISTENCE`.

4.1.2.1. SensorController

Controller *NestJS*$_G$ esposto sotto il prefisso `/sensor`.

Metodo	***Endpoint***$_G$	Note
`getSensors(gatewayId?, tenantId?)`	`GET /sensor`	Restituisce i sensori attivi negli ultimi 10 minuti; filtro opzionale per gatewayId

4.1.2.2. SensorService

Contiene la logica di aggregazione dei sensori. Dipende dall'interfaccia `NpQueryPersistenceService`.

Campi:

Campo	Tipo	Note
`npqps`	`NpQueryPersistenceService`	Iniettato via token `NP_QUERY_PERSISTENCE`

Metodi pubblici:

Metodo	Firma	Note
`getSensors(input)`	`(input: GetSensorsInput): Promise<SensorModel[]>`	Query misure ultimi 10 minuti; deduplica per `gatewayId::sensorId::sensorType`; tiene il `lastSeen` più recente

Metodi privati:

Metodo	Comportamento
`toSensorModels(measures)`	Deduplica le misure in sensori univoci; traccia il timestamp `lastSeen` più recente per sensore

4.1.3. Auth (*Tenant*$_G$ Access)

Il modulo di autenticazione gestisce la validazione dello stato del *tenant*$_G$ tramite chiamate al Management API. Applica un guard globale su tutte le richieste.

4.1.3.1. TenantAccessGuard

Guard globale registrato come `APP_GUARD` in `AppModule`.

Campi:

Campo	Tipo	Note
`configService`	`ConfigService`	Iniettato per leggere `MGMT_API_URL`

Metodi pubblici:

Metodo	Firma	Note
`canActivate(context)`	`(context: ExecutionContext): Promise<boolean>`	Salta per path pubblici (`/`, `/metrics`) e OPTIONS; estrae *Bearer token*$_G$; chiama Management API; applica read-only mode

Metodi privati:

Metodo	Comportamento
`isPublicRequest(request)`	Verifica se il path è `/` o `/metrics`
`extractBearerToken(authorization)`	Estrae il token dopo `Bearer`
`resolveTenantAccess(authorization)`	Chiama `GET ${MGMT_API_URL}/auth/tenant`$_G$`-status`; mappa 401->Unauthorized, 403->Forbidden; valida *payload*$_G$

4.1.3.2. TenantId Decorator

Decoratore `@TenantId()` che estrae il `tenantId` dal `TenantAccessContext` della richiesta.

4.1.3.3. Interfacce

TenantAccessContext

Campo	Tipo	Note
`tenantId`	`string`	Identificativo del *tenant*$_G$
`status`	`'active' | 'suspended'`	Stato corrente del *tenant*$_G$
`readOnly`	`boolean`	Se true, blocca le modifiche

4.1.4. MetricsModule

Modulo per l'esposizione di metriche *Prometheus*$_G$. Registrato come globale in `AppModule`.

4.1.4.1. MetricsService

Servizio che gestisce il registry *Prometheus*$_G$ e le metriche custom.

Campi:

Campo	Tipo	Note
`registry`	`Registry`	Registry *Prometheus*$_G$ (private, readonly)
`httpRequestsTotal`	`Counter`	Contatore richieste totali
`httpRequestDurationSeconds`	`Histogram`	Istogramma durata richieste (11 bucket)
`httpRequestsInFlight`	`Gauge`	Gauge richieste in corso

Metodi pubblici:

Metodo	Firma	Note
`contentType` (getter)	`get contentType(): string`	Restituisce il content type del registry
`getMetrics()`	`(): Promise<string>`	Restituisce la stringa delle metriche
`incInFlight(method)`	`(method: string): void`	Incrementa il gauge in-flight
`decInFlight(method)`	`(method: string): void`	Decrementa il gauge in-flight
`observeHttpRequest(...)`	`(method, route, statusCode, durationSeconds): void`	Incrementa il counter e osserva l'istogramma
`resolveRouteLabel(req)`	`(req): string`	Restituisce `${baseUrl}${routePath}` o `'_unmatched'`

4.1.4.2. MetricsController

Controller che espone l'*endpoint*$_G$ `/metrics` per *Prometheus*$_G$.

Metodo	*Endpoint*$_G$	Note
`metrics(res)`	`GET /metrics`	Imposta `Content-Type` dal registry e invia la stringa delle metriche

4.1.4.3. MetricsInterceptor

Intercettatore globale registrato come `APP_INTERCEPTOR` in `AppModule`.

Comportamento:

- Salta i contesti non HTTP
- Incrementa il gauge in-flight all'ingresso
- Usa `finalize` per registrare durata, risolvere route, osservare la richiesta e decrementare il gauge

4.1.5. Database

Il file `data-source.ts` configura il TypeORM DataSource per le migration *CLI*$_G$:

- `type: 'postgres'`
- `entities: join(__dirname, '..', '**', '*.entity.{ts,js}')`
- `migrations: join(__dirname, '..', 'migrations', '*.ts')` e `*.js`

- Variabili: `MEASURES_DB_HOST`, `MEASURES_DB_PORT`, `MEASURES_DB_USER`, `MEASURES_DB_PASSWORD`, `MEASURES_DB_NAME`, `DB_SSL`

La cartella `migrations/` esiste ma contiene solo `.gitkeep` (nessuna migration al momento).

4.2. Entità

Entità	Campi
MeasureEntity	**time**: `timestamptz (PK)`, **tenantId**: `uuid`G, **gatewayId**: `uuid`G, **sensorId**: `uuid`G `(PK)`, **sensorType**: `varchar(255)`, **encryptedData**: `varchar(255)`, **iv**G: `varchar(255)`, **authTag**: `varchar(255)`, **keyVersion**: `integer`

Chiave primaria composita: `(time, sensor_id)`

4.3. Decisioni implementative

- **Interfacce tra le componenti dei moduli**

Sono state definite interfacce specifiche per la comunicazione tra i componenti dei moduli, in particolare per il passaggio dei parametri dei metodi tra **`Controller`** e **`Service`** e tra **`Service`** e **`PersistenceService`**. Questa scelta è stata motivata dalla volontà di mantenere una chiara separazione tra i livelli dell'applicazione, evitando di esporre direttamente i dati delle richieste fatte dal controller ai layer inferiori. Le interfacce consentono di definire contratti chiari e stabili tra i componenti, facilitando la manutenzione e l'evoluzione del codice, oltre a migliorare la testabilità isolata dei singoli componenti.

- **Introduzione di Mappers tra i layer**

L'utilizzo di **`Persistence Entities`**, **`Business Models`** e **`DTO`**G ha portato alla necessità di introdurre componenti di mapping dedicati per permettere una conversione corretta e centralizzata dei dati tra i diversi layer del modulo. I **`Mapper`** consentono di incapsulare la logica di trasformazione dei dati (inclusa la normalizzazione di timestamp, la conversione base64->hex per *IV*G/authTag/encryptedData), mantenendo i controller e i servizi focalizzati sulle rispettive responsabilità di esposizione API e logica applicativa.

- **Utilizzo di TypeORM**

La scelta di utilizzare TypeORM come strumento di accesso alla persistenza è stata guidata dalla necessità di interagire con un database *PostgreSQL*G/*TimescaleDB*G in modo efficiente e strutturato. TypeORM offre un'astrazione di alto livello per la definizione delle entità, la costruzione delle query e la gestione delle connessioni al database. L'adozione di TypeORM ha permesso di implementare in modo rapido e robusto le operazioni di query paginata (con cursor-based pagination composta) e non paginata sulle misure.

- **Cursor-based Pagination Composta**

La paginazione utilizza un cursore composto `(time, sensorId)` per garantire ordinamento deterministico e consistenza anche in presenza di misure con lo stesso timestamp. Il cursore viene codificato come stringa `time|sensorId` e decodificato dal persistence service

per costruire la clausola `WHERE (time < cursorTime OR (time = cursorTime AND sensorId < cursorSensorId))`. Questo approccio evita duplicazioni o salti di record tra pagine consecutive.

- **Streaming con RxJS e *NATS*$_G$**

Lo streaming *SSE*$_G$ è implementato tramite RxJS: ogni *tenant*$_G$ ha un `Subject` dedicato che riceve eventi live da `TelemetryStreamBridgeService` (sottoscritto a *NATS*$_G$). Lo stream concatena un replay storico (opzionale, via query non paginata) con gli eventi live. Il stream termina automaticamente alla scadenza del *JWT*$_G$ (`tokenExpiresAt`) tramite `takeUntil`, emettendo un evento di errore `token_expired`. Questo design disaccoppia la fonte eventi (*NATS*$_G$) dalla consegna al client (*SSE*$_G$) permettendo filtri per-*tenant*$_G$ e per-sensore.

4.4. Flussi di esecuzione

Di seguito sono descritti i principali flussi di esecuzione del servizio `notip-data-api`, con particolare attenzione ai componenti applicativi coinvolti nell'elaborazione delle richieste e nell'accesso ai dati.

4.4.1. Query paginata delle misure

Il client invia una richiesta `GET` all'*endpoint*$_G$ `/measures/query`, specificando l'intervallo temporale di interesse ed eventuali filtri su `gatewayId`, `sensorId` e `sensorType`. Il `MeasureController` raccoglie i parametri di query, normalizza gli array e costruisce l'oggetto `QueryInput` per il `MeasureService`.

Il `MeasureService` valida i parametri ricevuti, verificando in particolare il limite massimo (`limit <= 999`) e la dimensione della finestra temporale (`<= 24h`). In caso di validazione positiva, la richiesta viene delegata al `MeasurePersistenceService`, che costruisce una query TypeORM sull'entità `MeasureEntity` con filtri e cursor-based pagination composta `(time, sensorId)`, recuperando `limit + 1` righe per determinare `hasMore`.

I dati ottenuti vengono poi trasformati tramite `MeasureMapper` in oggetti `QueryResponseDto`, comprensivi della lista delle misure (con normalizzazione timestamp e conversione base64->hex), dell'eventuale `nextCursor` e dell'informazione `hasMore`, quindi restituiti al client.

4.4.2. Export completo delle misure

Il flusso di export viene attivato tramite una richiesta `GET` all'*endpoint*$_G$ `/measures/export`. Il `MeasureController` estrae i parametri di filtro e li inoltra al `MeasureService`.

Il servizio applicativo verifica la correttezza dell'intervallo temporale richiesto (max 24h) e, se i parametri risultano validi, invoca il `MeasurePersistenceService` per eseguire una query non paginata sull'insieme delle misure persistite, ordinata per `time DESC`. I record recuperati vengono successivamente convertiti in `EncryptedEnvelopeDto` mediante `MeasureMapper`.

Il client riceve quindi l'elenco completo delle misure cifrate compatibili con i filtri indicati.

4.4.3. Streaming delle misure

Il servizio espone l'*endpoint*$_G$ `/measures/stream`, implementato come *Server-Sent Events*$_G$. Il `MeasureController` raccoglie gli eventuali filtri su *gateway*$_G$, sensore e tipo di sensore, anche in forma multi-valore, oltre al parametro opzionale `since`, quindi delega la gestione dello stream a `StreamListenerService`.

Il servizio espone un flusso continuo di eventi tramite l'*endpoint*$_G$ `/measures/stream`, implementato come *Server-Sent Events*$_G$. Il `MeasureController` raccoglie gli eventuali filtri e il parametro `since`, quindi delega la gestione dello stream a `StreamListenerService`.

Lo `StreamListenerService` genera un flusso osservabile che concatena:
1. **Replay storico**: se `since` è definito, esegue una query non paginata su `[since, now]` ed emette ogni misura come evento `data`.
2. **Eventi live**: sottoscrive al `Subject` del *tenant*$_G$ e filtra gli eventi per `gatewayId`/`sensorId`/`sensorType`.

Se il token *JWT*$_G$ è già scaduto (`tokenExpiresAt <= Date.now()`), emette immediatamente un evento di errore `token_expired`. Altrimenti, usa `takeUntil` per terminare lo stream alla scadenza del token.

Ogni evento compatibile viene trasformato nel formato *SSE*$_G$ e inviato al client. Questo flusso consente al client di ricevere aggiornamenti continui senza dover effettuare polling esplicito.

4.4.4. Elenco dei sensori disponibili

Il client può richiedere l'elenco dei sensori osservati di recente tramite l'*endpoint*$_G$ `GET /sensor`. Il `SensorController` costruisce l'input applicativo e lo inoltra al `SensorService`.

Il `SensorService` definisce automaticamente una finestra temporale degli ultimi dieci minuti e interroga il livello di persistenza attraverso l'interfaccia `NpQueryPersistenceService` (implementata da `MeasurePersistenceService`). Le misure recuperate vengono aggregate in memoria: per ogni chiave univoca `gatewayId::sensorId::sensorType` viene mantenuta la misura con il `time` più recente, che diventa il `lastSeen` del sensore.

Il risultato finale viene convertito in `SensorDto` e restituito come elenco dei sensori disponibili, eventualmente filtrato per `gatewayId`.

4.4.5. Validazione *tenant*$_G$ (TenantAccessGuard)

Ad ogni richiesta HTTP, il `TenantAccessGuard` globale intercetta il flusso:
1. Salta la validazione per path pubblici (`/`, `/metrics`) e richieste `OPTIONS`.
2. Estrae il *Bearer token*$_G$ dall'header `Authorization`.
3. Chiama `GET ${MGMT_API_URL}/auth/tenant`$_G$`-status` con il token.
4. Mappa la risposta: 401 -> `UnauthorizedException`, 403 -> `ForbiddenException`, altro errore -> `ServiceUnavailableException`.
5. Se il *tenant*$_G$ è in modalità `readOnly`, blocca tutti i metodi non-GET/HEAD/OPTIONS con `ForbiddenException`.

4.4.6. Bridge *NATS*$_G$ -> *SSE*$_G$ (TelemetryStreamBridgeService)

All'avvio del modulo, il `TelemetryStreamBridgeService`:
1. Si connette a *NATS*$_G$ con le opzioni configurate (*TLS*$_G$/token/user-pass).
2. Sottoscrive al subject wildcard `telemetry.data.*.*`.
3. Per ogni messaggio ricevuto:
   - Estrae il `tenantId` dal subject (`telemetry.data.{tenantId}.{something}`).
   - *JSON*$_G$-parsa l'envelope e valida i campi richiesti.
   - Pubblica la misura sul `Subject` del *tenant*$_G$ tramite `StreamListenerService.publishLiveMeasure()`.

I client *SSE*$_G$ connessi allo stream ricevono l'evento in tempo reale, filtrato per i parametri richiesti.

5. Test e verifica

Il progetto include:

- unit test sui servizi applicativi (MeasureService, SensorService, StreamListenerService);
- unit test sui controller (MeasureController, SensorController);
- unit test sul componente di persistenza (MeasurePersistenceService);
- unit test sulle metriche (MetricsService, MetricsInterceptor);
- unit test sulla validazione ambiente (env.validation);
- integration test applicativi con persistenza in memoria e stream mockato.

Le verifiche coperte dai test includono in particolare:

- corretto mapping di query ed export;
- rispetto del limite massimo di paginazione (`999`);
- rispetto del vincolo massimo di `24` ore sulla finestra temporale;
- cursor-based pagination composta `(time, sensorId)`;
- deduplicazione dei sensori e calcolo del `lastSeen`;
- corretto filtraggio dello stream per *gateway*$_G$/sensore/tipo;
- terminazione dello stream con `token_expired` alla scadenza del *JWT*$_G$;
- bridge *NATS*$_G$ -> StreamListenerService;
- corretto funzionamento del TenantAccessGuard (read-only mode);
- corretta propagazione delle eccezioni HTTP;
- metriche *Prometheus*$_G$ (counter, histogram, gauge).

Questa copertura consente di validare il comportamento funzionale principale del servizio, soprattutto per quanto riguarda i contratti esposti, la gestione dei casi di errore e l'integrazione con *NATS*$_G$.

6. Considerazioni finali

Il servizio `data-api` costituisce il punto di accesso applicativo ai dati telemetrici cifrati del sistema. La soluzione è costruita con una separazione chiara tra API, logica applicativa, mapping e accesso ai dati.

Le funzionalità principali includono:
- consultazione paginata delle misure con cursor-based pagination composta;
- export completo con validazione della finestra temporale;
- streaming *SSE*$_G$ con replay storico e eventi live da *NATS*$_G$;
- discovery dei sensori attivi con finestra automatica di 10 minuti;
- validazione dello stato del *tenant*$_G$ tramite guard globale;
- esposizione di metriche *Prometheus*$_G$;
- integrazione *NATS*$_G$ per *telemetria*$_G$ live e calcolo costi.

Il dominio di consultazione delle misure risulta ben definito sul piano contrattuale e della validazione applicativa, con un'architettura pronta a evolvere verso integrazioni più complete lato persistenza e streaming.