NoTIP

NO TESTS IN PRODUCTION

Specifica Tecnica

Versione	0.1.0
Data Modifica	2026-03-09
Utilizzo	Esterno

Abstract dei contenuti

Documento contenente la specifica tecnica del progetto, con particolare attenzione alle scelte tecnologiche e di design adottate

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
0.1.0	2026-03-09	Leonardo Preo	Valerio Solito	Prima stesura del documento
0.0.1	2026-03-09	Leonardo Preo	Valerio Solito	Creazione del documento

Indice

1. Introduzione

1.1. Scopo del Documento

Il presente documento costituisce la Specifica Tecnica del prodotto software e ha lo scopo di fornire una descrizione esaustiva e strutturata della sua architettura, delle componenti che la compongono, delle relative interazioni e della loro distribuzione nell'ambiente di esecuzione.

La Specifica Tecnica rappresenta il riferimento autorevole per le attività di progettazione e sviluppo del prodotto, assicurando la coerenza con il Proof of Concept (*PoC*$_G$) preliminare e introducendo raffinamenti architetturali volti ad accrescerne la solidità e la maturità. In particolare, il documento si prefigge i seguenti obiettivi:

- Definire l'**architettura logica** del prodotto, illustrandone le componenti principali, i rispettivi **ruoli funzionali** e le **interdipendenze**;
- Descrivere l'**architettura di deployment**, specificando le modalità di **distribuzione** e **orchestrazione** delle componenti nell'ambiente di esecuzione;
- Documentare i **design pattern architetturali** adottati, motivando le scelte progettuali in relazione alle tecnologie selezionate e ai requisiti del sistema
- Identificare gli **idiomi implementativi** impiegati a livello di dettaglio, con riferimento alle ottimizzazioni apportate alla **qualità** e alla **leggibilità** del codice;
- Fornire approfondimenti progettuali a supporto della comprensione, della **manutenibilità** e dell'**evoluzione futura** del prodotto.

1.2. Glossario

Per tutte le definizioni, acronimi e abbreviazioni utilizzati in questo documento, si faccia riferimento al Glossario v2.0.0, fornito come documento separato, che contiene tutte le spiegazioni necessarie per garantire una comprensione uniforme dei termini tecnici e dei concetti rilevanti per il progetto. Le parole che possiedono un riferimento nel Glossario saranno indicate nel modo che segue:

parola$_G$

1.3. Riferimenti

1.3.1. Riferimenti Normativi

- *Capitolato*$_G$ d'*appalto*$_G$ C7 - Sistema di acquisizione dati da sensori
  Ultimo accesso: 2026-03-09
- Norme di Progetto v1.1.0

1.3.2. Riferimenti Informativi

- Glossario v2.0.0
- PD1 - Regolamento del Progetto Didattico
  Ultimo Accesso: 2026-03-09
- C4 Model
  Ultimo Accesso: 2026-03-11
- Documentazione *Go*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *Typescript*$_G$
  Ultimo Accesso: 2026-03-11

- Documentazione *NestJS*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *Angular*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *NATS*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *Docker*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *Grafana*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *Prometheus*$_G$
  Ultimo Accesso: 2026-03-11
- Documentazione *TimescaleDB*$_G$
  Ultimo Accesso: 2026-03-11

2. Tecnologie

2.1. Linguaggi di Programmazione

Tecnologia	Versione	Descrizione
Go$_G$		*Go*$_G$ è un linguaggio compilato e staticamente tipizzato che combina semplicità sintattica e prestazioni di alto livello in contesti distribuiti. La sua gestione nativa della concorrenza, basata su goroutine e channel, consente di eseguire numerosi processi simultanei con un consumo minimo di risorse. Nel progetto viene impiegato per il Data Consumer, responsabile della sottoscrizione ai flussi telemetrici *JetStream*$_G$ e della scrittura su *TimescaleDB*$_G$, e per il Simulator *Backend*$_G$, in cui ogni *gateway*$_G$ simulato è gestito da una goroutine indipendente che emula in parallelo il comportamento dei sensori *BLE*$_G$.
Typescript$_G$		*TypeScript*$_G$ è un linguaggio di programmazione a tipizzazione statica che estende *JavaScript*$_G$ introducendo un sistema di tipi opzionale e strumenti di analisi del codice a tempo di compilazione. Favorisce la manutenibilità e la robustezza del codice in progetti di medio-grande scala. Nel progetto costituisce il linguaggio principale per tutti i *microservizi*$_G$ *backend*$_G$ sviluppati con *NestJS*$_G$ — Management API, Data API, Command API e *Provisioning*$_G$ Service — e per la Web Application *Angular*$_G$, incluso il Web Worker dedicato alla decifratura AES-256-GCM lato client tramite WebCrypto API.

2.2. Framework per la Codifica

Tecnologia	Versione	Descrizione
NestJS$_G$		*NestJS*$_G$ è un framework *Node.js*$_G$ per la costruzione di applicazioni server-side modulari e scalabili, ispirato all'architettura di *Angular*$_G$ e basato su decoratori *TypeScript*$_G$. Fornisce un

		sistema di dependency injection, guard, interceptor e moduli che strutturano il codice in modo coerente e testabile. Nel progetto è adottato per tutti e quattro i *microservizi*$_G$ *backend*$_G$: la Management API per la gestione di *tenant*$_G$, *gateway*$_G$ e chiavi crittografiche; la Data API per l'esposizione dei dati telemetrici via *REST*$_G$ e SSE; la Command API per l'emissione di comandi ai *gateway*$_G$; il *Provisioning*$_G$ Service per la firma dei certificati *mTLS*$_G$ e la generazione delle chiavi AES.
Angular$_G$		*Angular*$_G$ è un framework *frontend*$_G$ sviluppato da Google per la realizzazione di *Single Page Application*$_G$ enterprise-grade. Adotta un'architettura basata su componenti, moduli e servizi, con supporto nativo per la programmazione reattiva tramite RxJS. Nel progetto viene utilizzato per la Web Application principale, che integra *keycloak*$_G$*-angular*$_G$ per la gestione del ciclo OIDC/PKCE e un Web Worker dedicato alla decifratura client-side dei payload telemetrici. *Angular*$_G$ è impiegato anche per la Simulation *Dashboard*$_G$, l'interfaccia di controllo del Simulator *Backend*$_G$.
TypeORM		TypeORM è un Object-Relational Mapper per *TypeScript*$_G$ e *Node.js*$_G$ che consente di definire le entità del dominio come classi tipizzate e di gestire le migrazioni del database in modo incrementale e versionato. Nel progetto è integrato nei *microservizi*$_G$ *NestJS*$_G$ per l'accesso al Management DB (*PostgreSQL*$_G$) e al Measures DB (*TimescaleDB*$_G$), garantendo coerenza tra il modello applicativo e lo schema relazionale sottostante.

2.3. Tecnologie per la Gestione di Dati Temporali

Tecnologia	Versione	Descrizione
PostgreSQL$_G$		*PostgreSQL*$_G$ è un sistema di gestione di database relazionale open-source, noto per la sua conformità agli standard SQL, l'estensibilità e l'affidabilità in ambienti di produzione. Nel progetto ospita il Management DB, che contiene le tabelle di dominio principali: *tenant*$_G$, *gateway*$_G$, chiavi di cifratura, comandi, configurazioni di alert e *audit*$_G$ log. Viene anche utilizzato come base per l'estensione *TimescaleDB*$_G$ nel Measures DB.
TimescaleDB$_G$		*TimescaleDB*$_G$ è un'estensione di *PostgreSQL*$_G$ ottimizzata per la gestione di serie temporali, che introduce il concetto di ipertabella con partizionamento automatico per intervalli temporali (chunk). Nel progetto gestisce la tabella telemetry_data del Measures DB, su cui convergono tutti i messaggi telemetrici cifrati pubblicati dai *gateway*$_G$. La tabella è configurata con un chunk interval giornaliero e compressione automatica dei dati dopo 7 giorni, ottimizzando le query per intervalli temporali e riducendo l'occupazione su disco nel lungo periodo.

2.4. Tecnologie per la Comunicazione e Messaggistica

Tecnologia	Versione	Descrizione
NATS$_G$ *Jetstream*$_G$		*NATS*$_G$ *JetStream*$_G$ è un sistema di messaggistica ad alte prestazioni con supporto nativo alla persistenza dei messaggi, alla consegna garantita e al modello publish-subscribe. *JetStream*$_G$ estende il core *NATS*$_G$ aggiungendo stream durevoli, consumer con politiche di acknowledgement esplicito e retention configurabile. Nel progetto rappresenta il motore unico di comunicazione tra tutti i *microservizi*$_G$: gestisce l'ingestione della *telemetria*$_G$ *IoT*$_G$ via *mTLS*$_G$ sulla porta 4222, il pattern Request-Reply sincrono per la comunicazione interna tra servizi, la distribuzione degli alert infrastrutturali e i comandi verso i *gateway*$_G$, eliminando completamente qualsiasi canale HTTP service-to-service.
Server-Sent Events		Server-Sent Events (SSE) è un protocollo HTTP unidirezionale che permette al server di inviare eventi in streaming al client senza che quest'ultimo debba effettuare polling ripetuto. Nel progetto è il meccanismo adottato dal Data API per la distribuzione real-time dei payload telemetrici crittografati verso il *frontend*$_G$ *Angular*$_G$. Poiché il nativo EventSource del browser non supporta header custom, la comunicazione SSE è gestita tramite la libreria @microsoft/fetch-event-source, che consente l'invio del Bearer token *JWT*$_G$ a ogni connessione o riconnessione.

2.5. Tecnologie per Virtualizzazione e Deployment

Tecnologia	Versione	Descrizione
Docker$_G$		*Docker*$_G$ è una piattaforma di containerizzazione che consente di impacchettare applicazioni e relative dipendenze in unità isolate e portabili, eseguibili in modo coerente su qualsiasi ambiente. Nel progetto ogni microservizio, database e componente infrastrutturale è distribuito come container *Docker*$_G$ indipendente, garantendo isolamento a livello di processo e riproducibilità dell'ambiente tra sviluppo e produzione.
Docker Compose$_G$		*Docker Compose*$_G$ è uno strumento per la definizione e l'orchestrazione di applicazioni multi-container tramite file di configurazione dichiarativi. Nel progetto coordina l'avvio, la rete interna e il ciclo di vita di tutti i container della piattaforma NoTIP. Gestisce i volumi named — tra cui notip_ca_certs, che persiste la CA interna del *Provisioning*$_G$ Service — e i *Docker*$_G$ secrets utilizzati per l'iniezione sicura delle credenziali a runtime, senza che queste compaiano nel codice sorgente o nelle immagini.

Nginx		Nginx è un web server e reverse proxy ad alte prestazioni, largamente adottato per la gestione del traffico HTTP/*HTTPS*$_G$ in architetture a *microservizi*$_G$. Nel progetto funge da unico punto di ingresso esterno per il traffico HTTP, esponendo le porte 80 e 443. Applica gli header di sicurezza obbligatori (CSP, HSTS, X-Frame-Options), gestisce il passthrough degli stream SSE con le opportune configurazioni di buffering e applica *rate limiting*$_G$ sull'endpoint di *provisioning*$_G$ per mitigare tentativi di brute-force sulle factory key.

2.6. Tecnologie per Monitoraggio dei *Microservizi*$_G$

Tecnologia	Versione	Descrizione
Prometheus$_G$		*Prometheus*$_G$ è un sistema open-source di monitoraggio e alerting progettato per ambienti *cloud-native*$_G$ e architetture a *microservizi*$_G$. Adotta un modello di raccolta dati basato su scraping periodico degli endpoint /metrics esposti dai servizi, archiviando le misurazioni come serie temporali identificate da metriche e label. Nel progetto viene impiegato per la raccolta delle metriche operative dei *microservizi*$_G$ *NestJS*$_G$ e dei componenti infrastrutturali, consentendo di monitorare in modo continuo indicatori quali latenza delle richieste, throughput dei messaggi *NATS*$_G$ e stato dei container.
Grafana$_G$		*Grafana*$_G$ è una piattaforma open-source per la visualizzazione e l'esplorazione di dati provenienti da sorgenti eterogenee, tra cui *Prometheus*$_G$, database relazionali e sistemi di log. Permette di costruire *dashboard*$_G$ interattive con grafici, soglie di allerta e pannelli configurabili. Nel progetto è integrata come *frontend*$_G$ di osservabilità per le metriche raccolte da *Prometheus*$_G$, offrendo una visione centralizzata dello stato della piattaforma NoTIP: throughput della *telemetria*$_G$, latenza dei *microservizi*$_G$, stato dei *gateway*$_G$ e anomalie nei flussi di messaggistica *NATS*$_G$.

2.7. Librerie

Tecnologia	Versione	Descrizione
keycloak$_G$*-angular*$_G$		*keycloak*$_G$*-angular*$_G$ è una libreria *Angular*$_G$ che integra il client *JavaScript*$_G$ ufficiale di *Keycloak*$_G$, semplificando la gestione del ciclo di vita OIDC all'interno di un'applicazione *Angular*$_G$. Nel progetto gestisce il flusso Authorization Code con PKCE, il silent refresh tramite refresh token rotation — in sostituzione dell'approccio deprecato basato su iframe — e l'aggiunta automatica del Bearer token alle chiamate HTTP tramite un interceptor. I token non vengono mai scritti in localStorage o sessionStorage: l'access token è mantenuto in memoria e il refresh token è gestito tramite cookie HttpOnly di *Keycloak*$_G$.

@nestjs$_G$/passport + passport-*jwt*$_G$		*@nestjs*$_G$/passport e passport-*jwt*$_G$ sono librerie per l'implementazione di strategie di autenticazione nei *microservizi*$_G$ *NestJS*$_G$. Nel progetto abilitano la validazione *JWT*$_G$ in modalità completamente stateless: ogni richiesta viene autenticata verificando la firma del token contro l'endpoint JWKS di *Keycloak*$_G$, senza alcuna chiamata a runtime al server di identità. La JWKS è cachata all'avvio e aggiornata ogni 24 ore, o immediatamente in caso di kid sconosciuto.
jwks-rsa		jwks-rsa è una libreria *Node.js*$_G$ per il recupero e il caching delle chiavi pubbliche da un endpoint JWKS. Nel progetto è integrata con passport-*jwt*$_G$ per garantire che la rotazione delle chiavi di firma di *Keycloak*$_G$ non interrompa la validazione dei *JWT*$_G$ nei *microservizi*$_G$ *NestJS*$_G$, gestendo in modo trasparente l'aggiornamento del keystore locale.
@microsoft/fetch-event-source		@microsoft/fetch-event-source è una libreria che sostituisce il nativo EventSource del browser con un'implementazione basata su fetch, aggiungendo il supporto per header HTTP custom — tra cui Authorization: Bearer — necessari per l'autenticazione delle connessioni SSE. Nel progetto è il meccanismo client-side attraverso cui il *frontend*$_G$ *Angular*$_G$ apre e gestisce lo stream real-time verso il Data API, con supporto alla riconnessione automatica e al parametro since per il catch-up degli eventi persi.
WebCrypto API		WebCrypto API è un'API nativa del browser per operazioni crittografiche sicure, eseguita in un contesto isolato rispetto al thread principale. Nel progetto è utilizzata all'interno di un Web Worker *Angular*$_G$ dedicato per la decifratura AES-256-GCM dei payload telemetrici ricevuti via SSE. Le chiavi vengono importate con extractable: false, impedendo l'estrazione dei byte raw dopo l'import. L'oggetto CryptoKey risiede esclusivamente nel contesto del Worker e viene distrutto alla chiusura del tab, senza mai transitare su disco o storage persistente.
PapaParse		PapaParse è una libreria *JavaScript*$_G$ per il parsing e la serializzazione di file CSV, con supporto per dataset di grandi dimensioni tramite elaborazione in streaming. Nel progetto è impiegata nel flusso di export client-side: una volta che il Web Worker ha decifrato in batch i payload telemetrici richiesti, PapaParse li serializza in formato CSV e il *frontend*$_G$ forza il download direttamente nel browser, senza che i dati in chiaro transitino mai verso il server.

2.8. Tecnologie per Analisi Statica

Tecnologia	Versione	Descrizione

2.9. Tecnologie per Analisi Dinamica

Tecnologia	Versione	Descrizione

3. Architettura

3.1. Architettura Logica

3.2. Architettura di Deployment

L'architettura del sistema è basata su un **modello a *microservizi*$_G$**, in cui ogni componente funzionale viene eseguito come un'unità indipendente e isolata all'interno di un container *Docker*$_G$, garantendo portabilità tra gli ambienti di esecuzione e semplificando le procedure di **manutenzione** e **aggiornamento**. L'orchestrazione dell'intero sistema è affidata a *Docker Compose*$_G$, che coordina il ciclo di vita di tutti i container, gestisce la rete interna tra i servizi e provvede all'iniezione sicura delle credenziali tramite *Docker*$_G$ secrets.

Il Cloud Layer è strutturato per ospitare i seguenti *microservizi*$_G$:

- **Management API**: il servizio centrale per la gestione dei *tenant*$_G$, dei *gateway*$_G$, degli utenti e delle chiavi crittografiche. Espone le API *REST*$_G$ consumate dalla Web Application e si interfaccia con *Keycloak*$_G$ per le operazioni di amministrazione dell'identità.
- **Data API**: il modulo responsabile dell'esposizione dei dati telemetrici verso i client autorizzati, tramite query storiche su *TimescaleDB*$_G$ e streaming real-time mediante Server-Sent Events.
- **Command API**: il servizio dedicato all'emissione di comandi verso i *gateway*$_G$ *IoT*$_G$, con verifica dell'ownership del *gateway*$_G$ e gestione dell'acknowledgement tramite *NATS*$_G$ *JetStream*$_G$.
- ***Provisioning*$_G$ Service**: il componente che opera come Certification Authority interna, firma le richieste di certificato dei *gateway*$_G$ e genera le chiavi AES-256 consegnate al momento del *provisioning*$_G$. La CA è persistita su un volume *Docker*$_G$ named dedicato (notip_ca_certs), condiviso con il broker *NATS*$_G$ per la verifica *mTLS*$_G$.
- **Data Consumer**: il modulo sviluppato in *Go*$_G$ responsabile della sottoscrizione ai flussi telemetrici su *NATS*$_G$ *JetStream*$_G$, della scrittura dei payload crittografati su *TimescaleDB*$_G$ e del monitoraggio della liveness dei *gateway*$_G$ tramite un meccanismo di *heartbeat*$_G$.
- **Simulator *Backend*$_G$**: il servizio in *Go*$_G$ che emula il comportamento dei *gateway*$_G$ *IoT*$_G$ fisici, in cui ogni *gateway*$_G$ simulato è gestito da una goroutine indipendente che pubblica *telemetria*$_G$ cifrata verso *NATS*$_G$. È accompagnato da una Simulation *Dashboard*$_G$ *Angular*$_G$ per il controllo della simulazione.
- **Web Application**: il *frontend*$_G$ *Angular*$_G$ che rappresenta il punto di decifratura dell'intera *pipeline*$_G$. Tramite un Web Worker dedicato e la WebCrypto API, decifra i payload AES-256-GCM ricevuti via SSE direttamente nel browser dell'utente, senza che i dati in chiaro transitino mai verso il *backend*$_G$.

Nginx funge da unico punto di ingresso esterno per il traffico HTTP/*HTTPS*$_G$, operando come reverse proxy verso i *microservizi*$_G$ interni e applicando gli header di sicurezza obbligatori. La porta 4222 è esposta esclusivamente per le connessioni *mTLS*$_G$ dei *gateway*$_G$ *IoT*$_G$ verso il broker *NATS*$_G$.

Tutta la comunicazione interna tra *microservizi*$_G$ è mediata da *NATS*$_G$ *JetStream*$_G$, che sostituisce completamente qualsiasi canale HTTP service-to-service. I servizi comunicano in modo sincrono tramite il pattern Request-Reply e in modo asincrono tramite stream *JetStream*$_G$ con persistenza configurabile, garantendo disaccoppiamento e resilienza ai riavvii temporanei dei singoli componenti.

La gestione dell'identità è centralizzata in *Keycloak*$_G$, deployato in modalità self-hosted, che emette *JWT*$_G$ contenenti il claim `tenant_id` iniettato tramite Protocol Mapper. Questo claim è il fondamento dell'isolamento multi-*tenant*$_G$: ogni microservizio lo estrae e lo applica come filtro implicito su ogni operazione, garantendo che ciascun *tenant*$_G$ acceda esclusivamente alle proprie risorse, a ogni livello dello stack — dall'API al database fino ai subject *NATS*$_G$.

3.3. Design Pattern

I design pattern architetturali adottati nel progetto sono stati selezionati in base alla loro capacità di soddisfare i requisiti funzionali e non funzionali del sistema, nonché di facilitare la manutenibilità, la scalabilità e l'evoluzione futura del prodotto.

Di seguito vengono descritti i principali pattern utilizzati, con una spiegazione dettagliata delle motivazioni alla base della loro scelta e del loro impiego all'interno dell'architettura complessiva del sistema.

3.3.1. pattern1

3.3.1.1. Descrizione

3.3.1.2. Motivi per la Scelta

3.3.1.3. Utilizzo nel Progetto

3.4. *Microservizi*$_G$ Sviluppati

4. Stato dei requisiti funzionali