NoTIP

NO TESTS IN PRODUCTION

Specifica tecnica - Management API

Versione	1.1.0
Data Modifica	2026-04-17
Utilizzo	Esterno

Abstract dei contenuti

Specifica tecnica del microservizio notip-management-api: architettura interna, value object di dominio, definizione delle API, schema del database e metodologia di testing.

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
1.1.0	2026-04-17	Alessandro Mazzariol	Leonardo Preo	Correzione immagini diagrammi
1.0.0	2026-04-13	Leonardo Preo (responsabile)		Approvazione per ingresso in *baseline*$_G$ PB
0.2.1	2026-04-11	Alessandro Mazzariol	Alessandro Contarini	Correzione errori di battitura e refusi
0.2.0	2026-04-10	Alessandro Mazzariol	Alessandro Contarini	Stesura finale del documento
0.1.1	2026-04-07	Matteo Mantoan	Francesco Marcon	Uniformazione titoli e intestazioni: solo prima lettera maiuscola
0.1.0	2026-03-31	Alessandro Mazzariol	Francesco Marcon	Stesura finale del documento
0.0.1	2026-03-26	Alessandro Mazzariol	Alessandro Contarini	Creazione del documento prima bozza del documento

Indice

Indice delle figure

Indice delle tabelle

1. Introduzione

Questo documento illustra l'architettura interna e le scelte implementative del microservizio `notip-management-api`. Sviluppato in *NestJS*$_G$, questo componente è il fulcro del *backend*$_G$ e ha la duplice funzione di fare da tramite tra *frontend*$_G$ e servizi terzi come *Keycloak*$_G$ e di salvare in Database le informazioni ricevute dagli altri *microservizi*$_G$ attraverso *NATS*$_G$. Il microservizio espone molteplici *endpoint*$_G$ divisi per ruolo e funzione principale.

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
KeycloakRealm	KEYCLOAK_REALM	-	Si
KeycloakClientId	KEYCLOAK_MGMT_CLIENT_ID	-	Si
KeycloakClientSecret	KEYCLOAK_MGMT_CLIENT_SECRET	-	Si
KeycloakUrl	KEYCLOAK_URL	-	Si
KeycloakIssuerUrl	KEYCLOAK_ISSUER_URL	-	Si
NATSUrl	NATS_URL	-	Si
DBHost	MGMT_DB_HOST	—	Sì
DBPort	MGMT_DB_PORT	5432	No
DBName	MGMT_DB_NAME	—	Sì
DBUser	MGMT_DB_USER	—	Sì
DBPassword	MGMT_DB_PASSWORD	—	Sì
DBEncryptionKey	DB_ENCRYPTION_KEY	-	Si
ApiPort	MGMT_API_PORT	3000	No
NodeEnv	NODE_ENV	development	No
NatsServers	NATS_SERVERS	NATS_URL	No
NatsClientName	NATS_CLIENT_NAME	management-api	No
NatsDurablePrefix	NATS_DURABLE_PREFIX	management-api	No
NatsTlsCa	NATS_TLS_CA	-	No
NatsTlsCert	NATS_TLS_CERT	-	No
NatsTlsKey	NATS_TLS_KEY	-	No
NatsToken	NATS_TOKEN	-	No
NatsUser	NATS_USER	-	No

Tabella 1: Variabili d'ambiente del microservizio `notip-management-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-management-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	ConfigModule	Carica e valida la configurazione globale del microservizio	Si
3	app.module	Registra i moduli applicativi, EventEmitter e il provider globale di *audit*$_G$	Si
4	TypeORM	Inizializza la connessione *PostgreSQL*$_G$, tranne in ambiente di test	Si
5	auth.module	Registra i guard globali di autenticazione, ruoli, policy di accesso e blocco impersonazione	Si
6	*jwt*$_G$.strategy	Configura la strategia *JWT*$_G$ con *Keycloak*$_G$ e JWKS	Si
7	*main*$_G$	Crea l'applicazione *NestJS*$_G$, registra ValidationPipe, filtri globali e documentazione Swagger su `/docs`.	Si
8	http.server	Avvia il listener HTTP sulla porta configurata	Si

Tabella 2: Sequenza di avvio del microservizio `notip-management-api`

3. Architettura logica

Il microservizio adotta una Layered Architecture con organizzazione interna di tipo modulare. 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. Layout dei moduli

Essendo il microservizio troppo grande per essere contenuto in un unico diagramma, di seguito è riportata la struttura completa dei moduli interni al microservizio e delle cartelle di contorno:

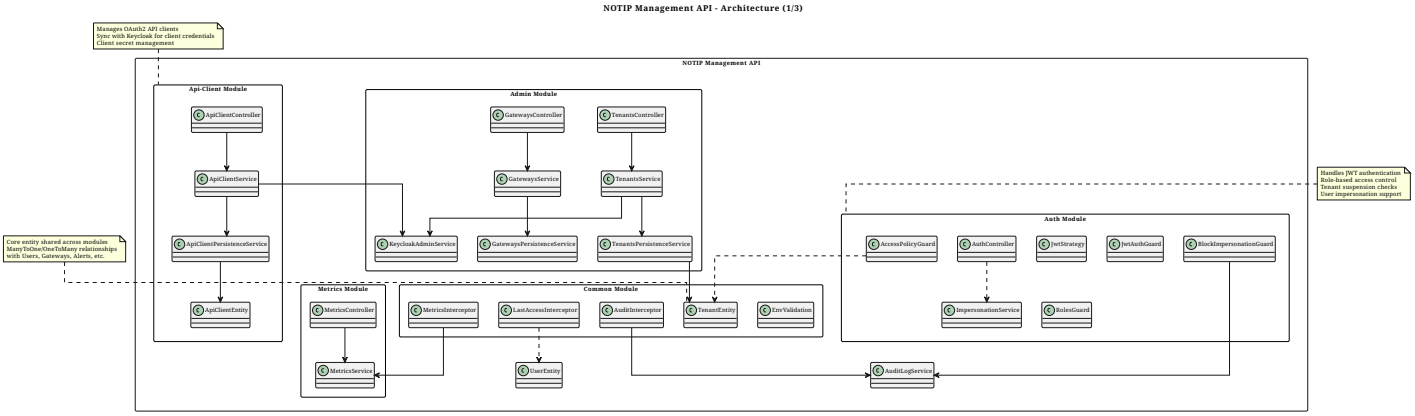

Figura 1: Architettura del microservizio (parte 1)

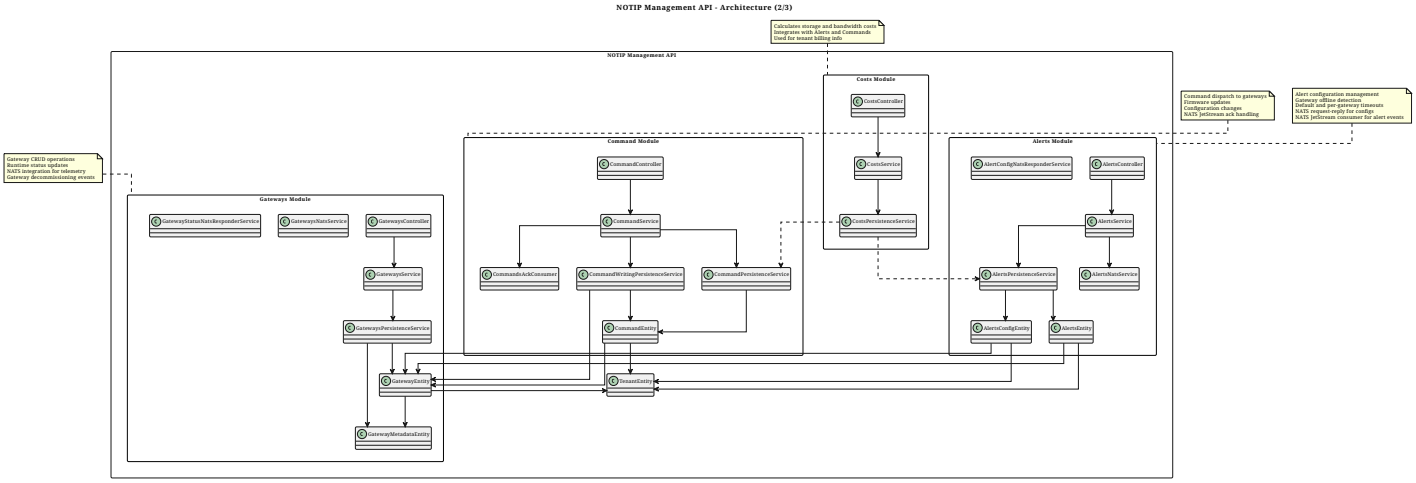

Figura 2: Architettura del microservizio (parte 2)

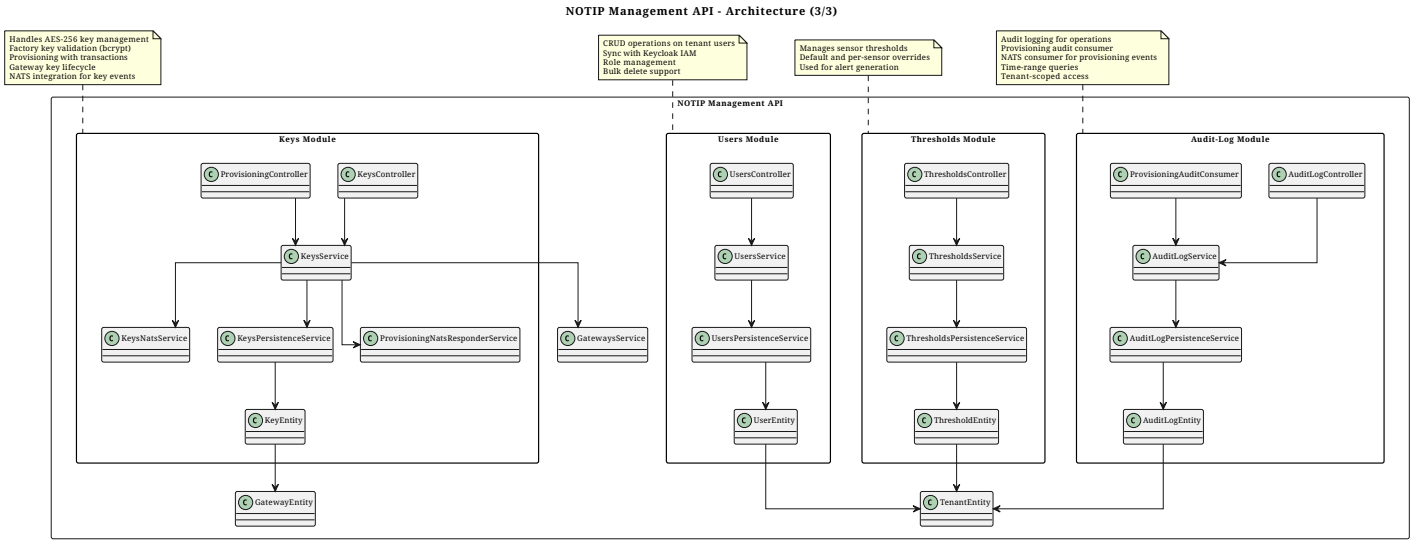

Figura 3: Architettura del microservizio (parte 3)

3.2. Strati architetturali

Di seguito è riportata la suddivisione in strati architetturali del microservizio, con l'indicazione delle cartelle e dei componenti principali:

Strato	Package	Componenti	Responsabilità

Presentation	`src/*/controller`
	`src/auth`
	`src/common/decorators`
	`src/auth/guards`
	`src/common/interceptors`
	`src/common/pipes`
	`src/common/filters`
	`src/*/dto`$_G$
		Controller, Guards, Interceptors, Filters, Decorators, *DTO*$_G$, Pipes	Gestione delle richieste HTTP, esposizione delle API *REST*$_G$, validazione dei *payload*$_G$, autenticazione e definizione dei contratti di ingresso/uscita dei dati. I Guards applicano le policy di accesso (*JWT*$_G$, ruoli, impersonazione). Gli Interceptors gestiscono metriche, *audit*$_G$ log e iniezione del *tenant*$_G$ context. I filtri globali gestiscono le eccezioni e le mappano verso risposte HTTP strutturate.

Business	`src/*/services`
	`src/*/models`
	`src/*/interfaces`
	`src/*/enums`
	`src/*/*.mapper.ts`
	`src/*/listeners`
	`src/*/encryption`
		Services, Models, Mappers, Listeners, Enum, Encryption utilities	Logica di business, regole di dominio, orchestrazione dei processi, definizione dei modelli di dominio e delle interfacce tra i servizi. Gestione delle richieste verso servizi esterni come *Keycloak*$_G$ (token exchange, CRUD utenti/ruoli/gruppi) e *NATS*$_G$ (publish, subscribe, request-reply). I Mappers convertono dati tra Entity, Model e *DTO*$_G$. I Listener reagiscono agli eventi interni del sistema (es. *gateway*$_G$ decommissioned). Le utilities di crittografia gestiscono *AES-256*$_G$*-GCM*$_G$ per il materiale delle chiavi.

Persistence	`src/*/entities`
	`src/*/services/*.persistence.ts`
	`src/database`
	`src/migrations`
		Entities, Persistence Services, *Repository*$_G$, Data Source, Migrations	Gestione dell'accesso ai dati tramite TypeORM, definizione delle entità e dei *repository*$_G$, gestione delle migrazioni del database, implementazione dei servizi di persistenza che interagiscono con il database *PostgreSQL*$_G$. Ogni modulo ha un proprio persistence service che

			astrae le operazioni CRUD sul *repository*$_G$ TypeORM. Il DataSource centralizza la configurazione della connessione e il supporto transazionale.
Infrastructure	`src/nats`$_G$ `src/common/ env.validation.ts src/metrics`	*NATS*$_G$ *JetStream*$_G$ Client, Config Validation, *Prometheus*$_G$ Metrics	Componenti infrastrutturali trasversali: connessione e interazione con *NATS*$_G$ *JetStream*$_G$ (pubblicazione comandi, consumo *ACK*$_G$, request-reply per *provisioning*$_G$ e alert), validazione della configurazione da variabili d'ambiente, esposizione di metriche *Prometheus*$_G$ (request count, duration, in-flight).

Tabella 3: Strati architetturali del microservizio `notip-management-api`

3.2.1. Pattern architetturali

Il microservizio adotta diversi pattern architetturali che garantiscono manutenibilità, testabilità e separazione delle responsabilità:

Pattern	Descrizione
Controller-Service-Persistence	Ogni modulo segue questa tripartizione: il **Controller** espone *endpoint*$_G$ HTTP, il **Service** contiene la logica di business e coordina i servizi esterni, il **Persistence Service** gestisce le operazioni CRUD su TypeORM. Questo pattern garantisce una chiara separazione tra i livelli dell'applicazione.
Dependency Injection	Tutti i componenti sono registrati come provider *NestJS*$_G$ e iniettati tramite constructor injection. Il grafo delle dipendenze è configurato nel composition root di ciascun modulo (`*.module.ts`).
Mapper	Le classi mapper statiche (`*Mapper`) convertono dati tra i diversi livelli: Entity → Model → *DTO*$_G$. Questo evita che i dettagli di persistenza (TypeORM decorators) o di presentazione (class-transformer) si propaghino tra i layer.
Event-driven	Il modulo Gateways emette eventi interni tramite `EventEmitter2` (es. `gateway`$_G$`.decommissioned`). I listener (`GatewaysListener`) li traducono in messaggi *NATS*$_G$. Questo disaccoppia la logica di dominio dal meccanismo di trasporto.
NATS$_G$ Request-Reply	Alcuni servizi espongono API interne tramite *NATS*$_G$ RR: `ProvisioningNatsResponderService` (factory key validation),

	`GatewayStatusNatsResponderService` (aggiornamento stato runtime), `AlertConfigNatsResponderService` (config listing). Questo permette la comunicazione asincrona tra *microservizi*$_G$.
Guard pattern	I Guards globali (`JwtAuthGuard`, `RolesGuard`, `AccessPolicyGuard`, `BlockImpersonationGuard`) applicano politiche di accesso su ogni richiesta. I decoratori (`@AdminOnly()`, `@TenantScoped()`, `@Roles()`, `@BlockImpersonation()`) configurano il comportamento per-*endpoint*$_G$ tramite *metadati*$_G$.

Tabella 4: Pattern architetturali

3.2.2. Flusso di una richiesta

Una richiesta HTTP attraversa i seguenti strati prima di raggiungere la logica di dominio:

Step	Strato	Comportamento
1	Guards	`JwtAuthGuard` valida il token *JWT*$_G$. `AccessPolicyGuard` verifica la policy (`@AdminOnly/@TenantScoped`). `RolesGuard` controlla i ruoli richiesti. `BlockImpersonationGuard` blocca se in impersonazione su *endpoint*$_G$ protetti.
2	Interceptors	`MetricsInterceptor` registra la richiesta. `LastAccessInterceptor` aggiorna l'ultimo accesso dell'utente. `TenantInterceptor` inietta il *tenant*$_G$ context nella richiesta. `AuditInterceptor` registra l'evento di *audit*$_G$.
3	Pipes	Validazione automatica dei *DTO*$_G$ tramite `class-validator` e `class-transformer`. I decorator `@IsString()`, `@IsEnum()`, `@IsUUID()` garantiscono l'integrità dei dati in ingresso.
4	Controller	Il controller riceve la richiesta valida, delega al Service e mappa la risposta tramite il Mapper.
5	Service	Il Service applica le regole di business (validazione ownership, transazioni, chiamate a *Keycloak*$_G$/*NATS*$_G$).
6	Persistence	Il Persistence Service esegue le operazioni TypeORM sul database *PostgreSQL*$_G$.
7	Response	La risposta viene mappata dal Mapper al *DTO*$_G$ appropriato e restituita al client con il formato *JSON*$_G$ corretto.

Tabella 5: Flusso di una richiesta

4. Design di dettaglio

4.1. Moduli del microservizio

4.1.1. AdminModule

Funzionalità amministrative di livello di sistema, in particolare gestione dei *Tenant*$_G$ e di tutte le operazioni che richiedono privilegi elevati. Tra queste è inclusa la possibilità di impersonare un utente per conto del quale agire.

4.1.1.1. Admin Tenants

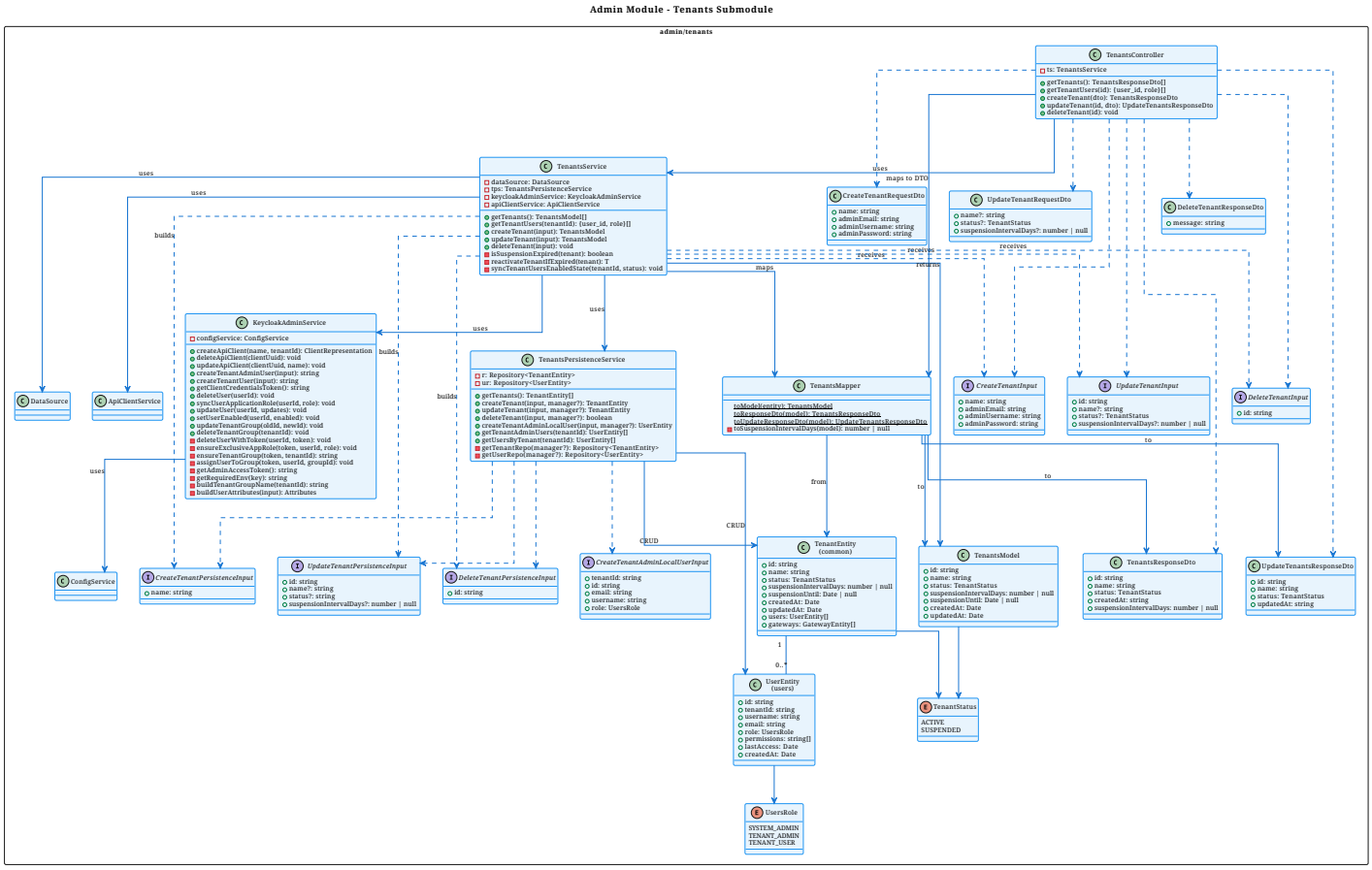

Figura 4: Diagramma del modulo AdminModule

Il sottosistema di gestione dei *Tenant*$_G$ si occupa di tutte le operazioni amministrative relative ai *tenant*$_G$ del sistema: creazione, modifica, sospensione, riattivazione ed eliminazione. È il punto di ingresso per la gestione del ciclo di vita dei *tenant*$_G$ e dei loro utenti.

4.1.1.1.1. TenantsController

Controller *NestJS*$_G$ esposto sotto il prefisso `/admin/tenants`. Protetto dal decoratore `@AdminOnly()`, espone cinque *endpoint*$_G$ *REST*$_G$. Delega ogni operazione a `TenantsService` e utilizza `TenantsMapper` per trasformare i risultati nei *DTO*$_G$ di risposta appropriati.

Metodo	*Endpoint*$_G$	Note
`getTenants()`	`GET /admin/tenants`	Restituisce la lista di tutti i *tenant*$_G$ con i relativi dettagli

`getTenantUsers(id)`	`GET /admin/tenants/:id/users`	Restituisce gli utenti associati a un *tenant*$_G$ specifico
`createTenant(dto`$_G$`)`	`POST /admin/tenants`	Crea un nuovo *tenant*$_G$ con il relativo utente amministratore
`updateTenant(id, dto`$_G$`)`	`PATCH`$_G$ `/admin/tenants/:id`	Aggiorna nome, stato o intervallo di sospensione di un *tenant*$_G$
`deleteTenant(id)`	`DELETE /admin/tenants/:id`	Elimina un *tenant*$_G$ e cascata tutti i dati associati

Tabella 6: *Endpoint*$_G$ del TenantsController

4.1.1.1.2. TenantsService

Contiene la logica di business principale. Coordina `TenantsPersistenceService`, `KeycloakAdminService` e `ApiClientService`.

Campi:

Campo	Tipo	Note
`tenantsPersistenceService`	`TenantsPersistenceService`	Iniettato via constructor
`keycloakAdminService`	`KeycloakAdminService`	Iniettato via constructor
`apiClientService`	`ApiClientService`	Iniettato via constructor

Tabella 7: Campi del TenantsService

Metodi pubblici:

Metodo	Firma	Note
`getTenants()`	`(): Promise<TenantsModel[]>`	Recupera tutti i *tenant*$_G$ dal database e li restituisce come array di modelli
`getTenantUsers(tenantId)`	`(tenantId: string): Promise<UserEntity[]>`	Recupera gli utenti di un *tenant*$_G$ specifico
`createTenant(input)`	`(input: CreateTenantInput): Promise<TenantsModel>`	Crea un *tenant*$_G$ in modo transazionale: entità DB, gruppo *Keycloak*$_G$, utente admin, client API. Rollback completo in caso di errore
`updateTenant(input)`	`(input: UpdateTenantInput): Promise<TenantsModel>`	Aggiorna un *tenant*$_G$ esistente; sincronizza il nome del gruppo su *Keycloak*$_G$ se il nome è cambiato
`deleteTenant(input)`	`(input: DeleteTenantInput): Promise<void>`	Elimina un *tenant*$_G$ e tutti i dati correlati: utenti,

		gateway$_G$, gruppo *Keycloak*$_G$, client API

Tabella 8: Metodi pubblici del TenantsService

Metodi privati:

Metodo	Comportamento
`isSuspensionExpired(tenant`$_G$`)`	Verifica se un *tenant*$_G$ sospeso ha superato il periodo di sospensione configurato confrontando `suspensionUntil` con la data corrente
`reactivateTenantIfExpired(tenant`$_G$`)`	Riattiva automaticamente un *tenant*$_G$ il cui periodo di sospensione è scaduto, impostando lo stato ad `ACTIVE`
`syncTenantUsersEnabledState(tenantId, status)`	Sincronizza lo stato enabled/disabled di tutti gli utenti del *tenant*$_G$ su *Keycloak*$_G$ in base allo stato del *tenant*$_G$

Tabella 9: Metodi privati del TenantsService

4.1.1.1.3. KeycloakAdminService

Servizio di integrazione con *Keycloak*$_G$ IAM. Utilizza `ConfigService` per recuperare le credenziali di accesso e comunica con le API *REST*$_G$ di *Keycloak*$_G$ tramite `fetch()`.

Campi:

Campo	Tipo	Note
`configService`	`ConfigService`	Iniettato; fornisce URL, realm, admin username/password

Tabella 10: Campi del KeycloakAdminService

Metodi pubblici:

Metodo	Firma	Note
`createApiClient(name, tenantId)`	`(name: string, tenantId: string): Promise<ClientRepresentation>`	Crea un client *OIDC*$_G$ su *Keycloak*$_G$ per un *tenant*$_G$
`deleteApiClient(clientUuid)`	`(clientUuid: string): Promise<void>`	Elimina un client *OIDC*$_G$ da *Keycloak*$_G$
`updateApiClient(clientUuid, name)`	`(clientUuid: string, name: string): Promise<void>`	Aggiorna il nome di un client *OIDC*$_G$
`createTenantAdminUser(input)`	`(input: CreateUserInput): Promise<string>`	Crea l'utente amministratore del *tenant*$_G$ su *Keycloak*$_G$; restituisce l'ID utente
`createTenantUser(input)`	`(input: CreateUserInput): Promise<string>`	Crea un utente generico del *tenant*$_G$ su *Keycloak*$_G$; restituisce l'ID utente

`getClientCredentialsToken()`	`(): Promise<string>`	Ottiene un token di accesso tramite client credentials grant
`deleteUser(userId)`	`(userId: string): Promise<void>`	Elimina un utente da *Keycloak*$_G$
`syncUserApplicationRole(userId, role)`	`(userId: string, role: string): Promise<void>`	Assegna o rimuove il ruolo applicativo a un utente
`updateUser(userId, updates)`	`(userId: string, updates: UserRepresentation): Promise<void>`	Aggiorna i dati di un utente su *Keycloak*$_G$
`setUserEnabled(userId, enabled)`	`(userId: string, enabled: boolean): Promise<void>`	Abilita o disabilita un utente su *Keycloak*$_G$
`updateTenantGroup(oldId, newId)`	`(oldId: string, newId: string): Promise<void>`	Rinomina il gruppo *tenant*$_G$ su *Keycloak*$_G$
`deleteTenantGroup(tenantId)`	`(tenantId: string): Promise<void>`	Elimina il gruppo *tenant*$_G$ da *Keycloak*$_G$

Tabella 11: Metodi pubblici del KeycloakAdminService

Metodi privati:

Metodo	Comportamento
`deleteUserWithToken(userId, token)`	Elimina un utente utilizzando un token di accesso specifico
`ensureExclusiveAppRole(token, userId, role)`	Garantisce che l'utente abbia un solo ruolo applicativo, rimuovendo ruoli conflittuali
`ensureTenantGroup(token, tenantId)`	Verifica o crea il gruppo *tenant*$_G$ su *Keycloak*$_G$; restituisce il gruppo ID
`assignUserToGroup(token, userId, groupId)`	Assegna un utente a un gruppo specifico su *Keycloak*$_G$
`getAdminAccessToken()`	Ottiene il token di accesso amministrativo tramite le credenziali configurate
`getRequiredEnv(key)`	Recupera una variabile d'ambiente obbligatoria; lancia se non presente
`buildTenantGroupName(tenantId)`	Costruisce il nome del gruppo *tenant*$_G$: `tenant`$_G$`-<tenantId>`
`buildUserAttributes(input)`	Costruisce la mappa degli attributi utente per *Keycloak*$_G$

Tabella 12: Metodi privati del KeycloakAdminService

4.1.1.1.4. TenantsPersistenceService

Layer di accesso ai dati. Utilizza due *repository*$_G$ separati gestiti tramite TypeORM: `tenantRepo` per `TenantEntity` e `userRepo` per `UserEntity`. Supporta le transazioni tramite `DataSource` (il parametro opzionale `manager` consente di partecipare a una transazione esterna).

Campi:

Campo	Tipo	Note
`tenantRepo`	`Repository`$_G$`<TenantEntity>`	*Repository*$_G$ TypeORM, schema `common`
`userRepo`	`Repository`$_G$`<UserEntity>`	*Repository*$_G$ TypeORM, schema `users`
`dataSource`	`DataSource`	Iniettato per supporto transazioni

Tabella 13: Campi del TenantsPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`getTenants()`	`(): Promise<TenantEntity[]>`	Recupera tutti i *tenant*$_G$
`createTenant(input, manager?)`	`(input, manager?): Promise<TenantEntity>`	Crea un *tenant*$_G$ con supporto transazionale
`updateTenant(input, manager?)`	`(input, manager?): Promise<TenantEntity>`	Aggiorna un *tenant*$_G$ esistente
`deleteTenant(input, manager?)`	`(input, manager?): Promise<boolean>`	Elimina un *tenant*$_G$; restituisce `true` se eliminato
`createTenantAdminLocalUser(input, manager?)`	`(input, manager?): Promise<UserEntity>`	Crea l'utente amministratore locale nel database
`getTenantAdminUsers(tenantId)`	`(tenantId: string): Promise<UserEntity[]>`	Recupera solo gli utenti admin di un *tenant*$_G$
`getUsersByTenant(tenantId)`	`(tenantId: string): Promise<UserEntity[]>`	Recupera tutti gli utenti di un *tenant*$_G$

Tabella 14: Metodi pubblici del TenantsPersistenceService

Metodi privati:

Metodo	Comportamento
`getTenantRepo(manager?)`	Restituisce il *repository*$_G$ *tenant*$_G$: se `manager` è fornito usa il suo *repository*$_G$, altrimenti il default
`getUserRepo(manager?)`	Restituisce il *repository*$_G$ user: se `manager` è fornito usa il suo *repository*$_G$, altrimenti il default

Tabella 15: Metodi privati del TenantsPersistenceService

4.1.1.1.5. TenantsMapper

Classe con metodi statici per la conversione tra i layer. Non ha dipendenze iniettate.

Metodi statici

Metodo	Comportamento
`toModel(entity)`	Converte `TenantEntity` in `TenantsModel`: mappa i campi uno-a-uno

`toResponseDto(model)`	Converte `TenantsModel` in `TenantsResponseDto`: adatta i nomi per la risposta HTTP
`toUpdateResponseDto(model)`	Converte `TenantsModel` in `UpdateTenantsResponseDto`: include `updatedAt`
`toSuspensionIntervalDays(model)`	Calcola i giorni di sospensione dal modello, gestendo il caso `null`

Tabella 16: Metodi statici del TenantsMapper

4.1.1.1.6. *DTO$_G$* e Input

CreateTenantRequestDto

Campo	Tipo	Note
`name`	`string`	Nome del *tenant$_G$*
`adminEmail`	`string`	Email dell'utente amministratore
`adminUsername`	`string`	Username dell'utente amministratore
`adminPassword`	`string`	Password iniziale dell'amministratore

Tabella 17: Campi del CreateTenantRequestDto

UpdateTenantRequestDto

Campo	Tipo	Note
`name`	`string | null`	Nuovo nome del *tenant$_G$*
`status`	`TenantStatus | null`	Nuovo stato del *tenant$_G$*
`suspensionIntervalDays`	`number | null`	Giorni prima della sospensione automatica

Tabella 18: Campi del UpdateTenantRequestDto

TenantsResponseDto

Campo	Tipo	Note
`id`	`string`	Identificativo univoco del *tenant$_G$*
`name`	`string`	Nome del *tenant$_G$*
`status`	`TenantStatus`	Stato corrente del *tenant$_G$*
`createdAt`	`string`	Data di creazione in formato *ISO 8601$_G$*
`suspensionIntervalDays`	`number`	Giorni prima della sospensione automatica

Tabella 19: Campi del TenantsResponseDto

UpdateTenantsResponseDto

Campo	Tipo	Note
`id`	`string`	Identificativo univoco del *tenant$_G$*

`name`	`string`	Nome del *tenant*$_G$
`status`	`TenantStatus`	Stato corrente del *tenant*$_G$
`updatedAt`	`string`	Data dell'ultimo aggiornamento in formato *ISO 8601*$_G$

Tabella 20: Campi del UpdateTenantsResponseDto

DeleteTenantResponseDto

Campo	Tipo	Note
`message`	`string`	Messaggio di conferma dell'eliminazione

Tabella 21: Campi del DeleteTenantResponseDto

CreateTenantInput

Campo	Tipo	Note
`name`	`string`	Nome del *tenant*$_G$
`adminEmail`	`string`	Email dell'utente amministratore
`adminUsername`	`string`	Username dell'utente amministratore
`adminPassword`	`string`	Password iniziale dell'amministratore

Tabella 22: Campi del CreateTenantInput

UpdateTenantInput

Campo	Tipo	Note
`id`	`string`	Identificativo del *tenant*$_G$ da aggiornare
`name`	`string | null`	Nuovo nome del *tenant*$_G$
`status`	`TenantStatus | null`	Nuovo stato del *tenant*$_G$
`suspensionIntervalDays`	`number | null`	Giorni prima della sospensione automatica

Tabella 23: Campi del UpdateTenantInput

DeleteTenantInput

Campo	Tipo	Note
`id`	`string`	Identificativo del *tenant*$_G$ da eliminare

Tabella 24: Campi del DeleteTenantInput

4.1.1.1.7. Enum

TenantStatus

Valore	Descrizione

ACTIVE	*Tenant*$_G$ operativo, tutti i servizi sono attivi
SUSPENDED	*Tenant*$_G$ sospeso, gli utenti non possono accedere ai servizi

Tabella 25: Valori dell'enum TenantStatus

UsersRole

Valore	Descrizione
SYSTEM_ADMIN	Amministratore di sistema, accesso completo a tutti i *tenant*$_G$
TENANT_ADMIN	Amministratore del *tenant*$_G$, gestisce utenti e configurazione del proprio *tenant*$_G$
TENANT_USER	Utente standard del *tenant*$_G$, accesso limitato alle funzionalità concesse

Tabella 26: Valori dell'enum UsersRole

4.1.1.1.8. Entità

TenantEntity (common)

Campo	Tipo	Note
id	uuid$_G$ v4	Identificativo univoco generato automaticamente
name	string	Nome del *tenant*$_G$
status	TenantStatus	Stato corrente del *tenant*$_G$
suspensionIntervalDays	number | null	Giorni di sospensione
suspensionUntil	Date | null	Data di sospensione programmata
createdAt	Date	Timestamp di creazione del record
updatedAt	Date	Timestamp dell'ultimo aggiornamento

Tabella 27: Campi del TenantEntity

Relazioni: OneToMany con `UserEntity` (tramite `tenantId`), OneToMany con `GatewayEntity` (tramite `tenantId`).

UserEntity (users)

Campo	Tipo	Note
id	uuid$_G$	Identificativo univoco dell'utente
tenantId	string	FK verso `TenantEntity.id`
username	string	Username dell'utente
email	string	Email dell'utente
role	UsersRole	Ruolo dell'utente nel sistema
permissions	string[] | null	Permessi specifici aggiuntivi
lastAccess	Date | null	Timestamp dell'ultimo accesso
createdAt	Date	Timestamp di creazione del record

Tabella 28: Campi del UserEntity

4.1.1.2. Admin Gateways

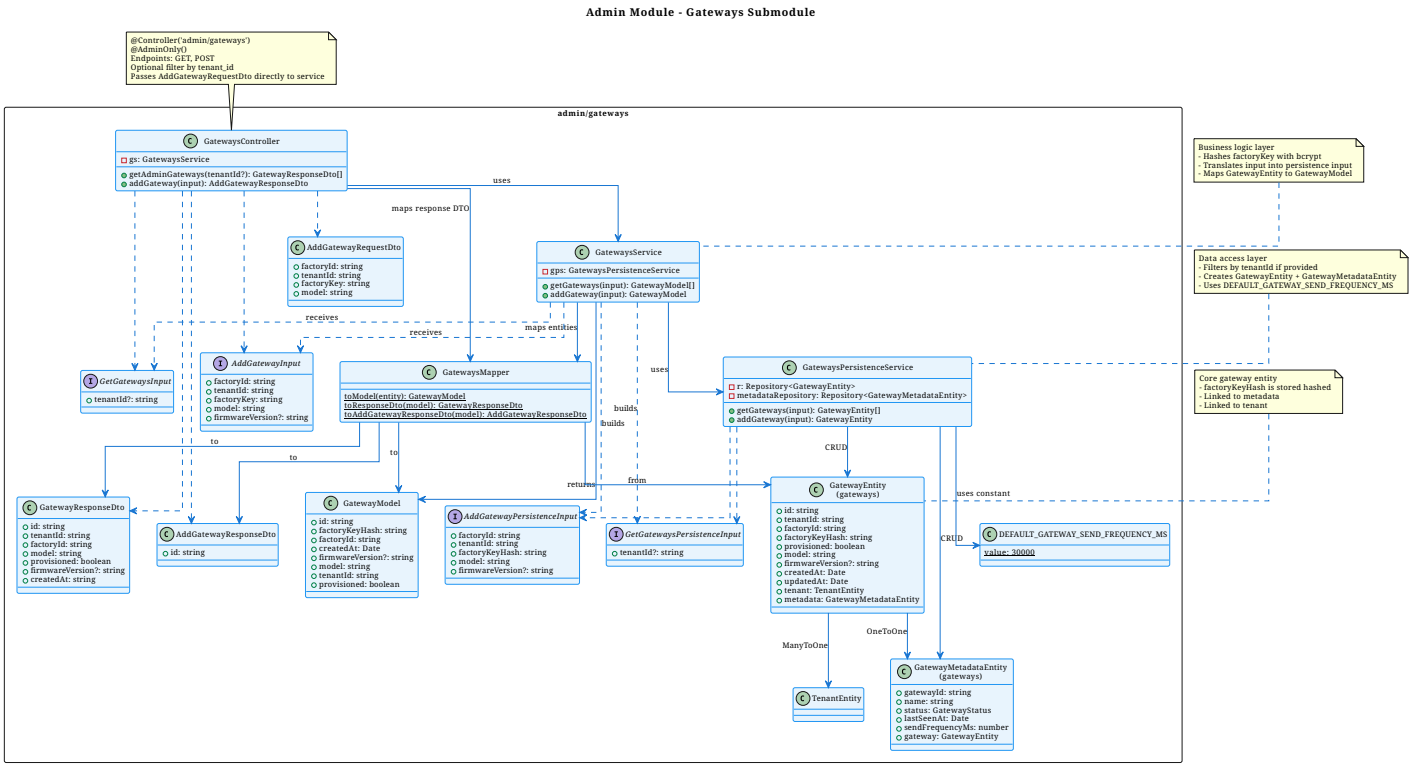

Figura 5: Diagramma del modulo Admin Gateways

Il sottosistema di gestione dei *Gateway*$_G$ amministrativi si occupa della registrazione e consultazione dei *gateway*$_G$ associati ai *tenant*$_G$. Espone *endpoint*$_G$ per elencare tutti i *gateway*$_G$ (con filtro opzionale per *tenant*$_G$) e per registrare nuovi *gateway*$_G$ a partire da un factory ID e una chiave di *provisioning*$_G$.

4.1.1.2.1. GatewaysController

Controller *NestJS*$_G$ esposto sotto il prefisso `/admin/gateways`. Protetto dal decoratore `@AdminOnly()`, espone due *endpoint*$_G$ *REST*$_G$. Delega ogni operazione a `GatewaysService` e utilizza `GatewaysMapper` per trasformare i risultati nei *DTO*$_G$ di risposta appropriati.

Metodo	*Endpoint*$_G$	Note
`getAdminGateways(tenantId?)`	`GET /admin/gateways`	Restituisce la lista di tutti i *gateway*$_G$, con filtro opzionale per tenantId
`addGateway(input)`	`POST /admin/gateways`	Registra un nuovo *gateway*$_G$ per un *tenant*$_G$ a partire da factoryId, tenantId, factoryKey e model

Tabella 29: Campi del GatewaysController

4.1.1.2.2. GatewaysService

Contiene la logica di business principale. Coordina `GatewaysPersistenceService` e utilizza il mapper per convertire tra *DTO*$_G$ e modelli di dominio.

Campi:

Campo	Tipo	Note
`gatewaysPersistenceService`	`GatewaysPersistenceService`	Iniettato via constructor

Tabella 30: Campi del GatewaysService

Metodi pubblici:

Metodo	Firma	Note
`getGateways(input)`	`(input: GetGatewaysInput): Promise<GatewayModel[]>`	Recupera tutti i *gateway*$_G$ dal database; se `tenantId` è fornito, filtra per *tenant*$_G$
`addGateway(input)`	`(input: AddGatewayInput): Promise<GatewayModel>`	Registra un nuovo *gateway*$_G$: hash della factoryKey con bcrypt, crea l'entità nel database con metadata di default

Tabella 31: Metodi pubblici del GatewaysService

4.1.1.2.3. GatewaysPersistenceService

Layer di accesso ai dati. Utilizza due *repository*$_G$ separati gestiti tramite TypeORM: `gatewayRepo` per `GatewayEntity` e `metadataRepo` per `GatewayMetadataEntity`. Supporta le transazioni tramite `DataSource`.

Campi:

Campo	Tipo	Note
`gatewayRepo`	`Repository`$_G$`<GatewayEntity>`	*Repository*$_G$ TypeORM, schema `gateways`
`metadataRepo`	`Repository`$_G$`<GatewayMetadataEntity>`	*Repository*$_G$ TypeORM, schema `gateways`
`dataSource`	`DataSource`	Iniettato per supporto transazioni

Tabella 32: Campi del GatewaysPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`getGateways(input)`	`(input: GetGatewaysInput): Promise<GatewayEntity[]>`	Recupera tutti i *gateway*$_G$; se `tenantId` è fornito, filtra con clausola WHERE
`addGateway(input)`	`(input: AddGatewayPersistenceInput): Promise<GatewayEntity>`	Crea il *gateway*$_G$ e il relativo metadata in modo transazionale

Tabella 33: Metodi pubblici del GatewaysPersistenceService

4.1.1.2.4. GatewaysMapper

Classe con metodi statici per la conversione tra i layer. Non ha dipendenze iniettate.

Metodi statici:

Metodo	Comportamento
`toModel(entity)`	Converte `GatewayEntity` in `GatewayModel`: mappa i campi uno-a-uno
`toResponseDto(model)`	Converte `GatewayModel` in `GatewayResponseDto`: adatta i nomi per la risposta HTTP
`toAddGatewayInput(dto`$_G$`)`	Converte `AddGatewayRequestDto` in `AddGatewayInput`: hash della factoryKey con bcrypt
`toAddGatewayResponseDto(model)`	Converte `GatewayModel` in `AddGatewayResponseDto`: restituisce solo l'`id`

Tabella 34: Metodi statici del GatewaysMapper

4.1.1.2.5. *DTO*$_G$ e Input

AddGatewayRequestDto

Campo	Tipo	Note
`factoryId`	`string`	Identificativo del factory di produzione
`tenantId`	`string`	Identificativo del *tenant*$_G$ proprietario
`factoryKey`	`string`	Chiave di *provisioning*$_G$ del *gateway*$_G$
`model`	`string`	Modello del *gateway*$_G$

Tabella 35: Campi dell'AddGatewayRequestDto

AddGatewayResponseDto

Campo	Tipo	Note
`id`	`string`	Identificativo univoco del *gateway*$_G$ creato

Tabella 36: Campi dell'AddGatewayResponseDto

GatewayResponseDto

Campo	Tipo	Note
`id`	`string`	Identificativo univoco del *gateway*$_G$
`tenantId`	`string`	Identificativo del *tenant*$_G$ proprietario
`factoryId`	`string`	Identificativo del factory di produzione
`model`	`string`	Modello del *gateway*$_G$
`provisioned`	`boolean`	Indica se il *gateway*$_G$ è stato provisionato
`firmwareVersion`	`string | undefined`	Versione del firmware installato

`createdAt`	`string`	Data di creazione in formato *ISO 8601*$_G$

Tabella 37: Campi del GatewayResponseDto

GetGatewaysInput

Campo	Tipo	Note
`tenantId`	`string | undefined`	Filtro opzionale per *tenant*$_G$

Tabella 38: Campi del GetGatewaysInput

AddGatewayInput

Campo	Tipo	Note
`factoryId`	`string`	Identificativo del factory di produzione
`tenantId`	`string`	Identificativo del *tenant*$_G$ proprietario
`factoryKey`	`string`	Chiave di *provisioning*$_G$ (in chiaro, verrà hashata)
`model`	`string`	Modello del *gateway*$_G$
`firmwareVersion`	`string | undefined`	Versione del firmware (opzionale)

Tabella 39: Campi dell'AddGatewayInput

AddGatewayPersistenceInput

Campo	Tipo	Note
`factoryId`	`string`	Identificativo del factory di produzione
`tenantId`	`string`	Identificativo del *tenant*$_G$ proprietario
`factoryKeyHash`	`string`	Hash bcrypt della factoryKey
`model`	`string`	Modello del *gateway*$_G$
`firmwareVersion`	`string | undefined`	Versione del firmware (opzionale)

Tabella 40: Campi dell'AddGatewayPersistenceInput

4.1.1.2.6. Enum

GatewayStatus

Valore	Descrizione
`GATEWAY_ONLINE`	*Gateway*$_G$ attivo e comunicante
`GATEWAY_OFFLINE`	*Gateway*$_G$ non raggiungibile
`GATEWAY_SUSPENDED`	*Gateway*$_G$ sospeso manualmente o automaticamente

Tabella 41: Valori dell'enum GatewayStatus

4.1.1.2.7. Entità

GatewayEntity (`gateways`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco generato automaticamente
`tenantId`	`string`	FK verso `TenantEntity.id`
`factoryId`	`string`	Identificativo del factory di produzione
`factoryKeyHash`	`string`	Hash bcrypt della chiave di *provisioning*$_G$; `select: false` per sicurezza
`provisioned`	`boolean`	Indica se il *gateway*$_G$ è stato provisionato
`model`	`string`	Modello del *gateway*$_G$
`firmwareVersion`	`string`	Versione del firmware installato
`createdAt`	`Date`	Timestamp di creazione del record
`updatedAt`	`Date`	Timestamp dell'ultimo aggiornamento

Tabella 42: Campi dell'GatewayEntity

Relazioni: ManyToOne con `TenantEntity` (tramite `tenantId`, cascade delete), OneToOne con `GatewayMetadataEntity` (cascade).

GatewayMetadataEntity (`gateways`)

Campo	Tipo	Note
`gatewayId`	`uuid`$_G$	FK verso `GatewayEntity.id`, chiave primaria
`name`	`string`	Nome descrittivo del *gateway*$_G$; offuscato alla vista dell'admin
`status`	`GatewayStatus`	Stato corrente del *gateway*$_G$
`lastSeenAt`	`Date`	Timestamp dell'ultima comunicazione ricevuta
`sendFrequencyMs`	`number`	Frequenza di invio dati in millisecondi; default: 30000

Tabella 43: Campi del GatewayMetadataEntity

4.1.2. AuthModule

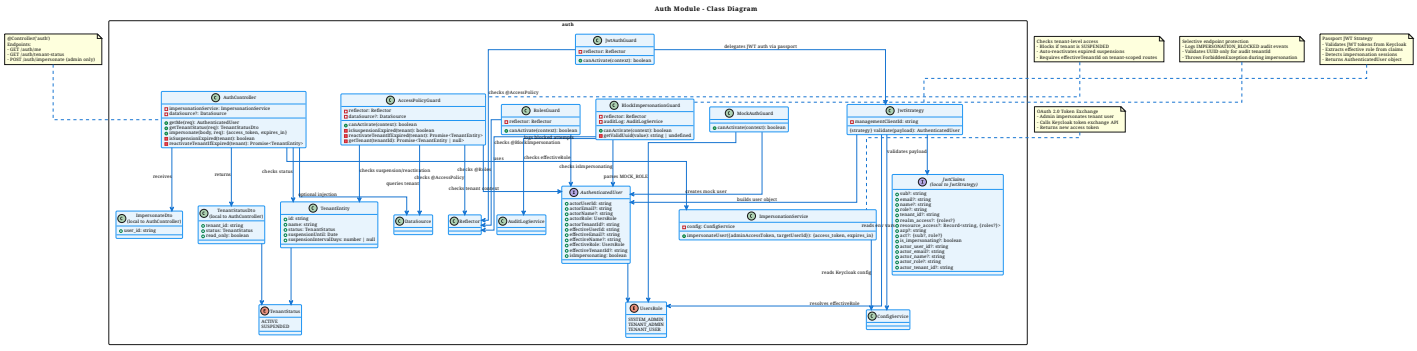

Figura 6: Diagramma del modulo Auth

Il modulo di autenticazione gestisce la validazione *JWT*$_G$ tramite *Keycloak*$_G$, il controllo degli accessi basato sui ruoli, la politica di accesso per *endpoint*$_G$ e il meccanismo di *impersonificazione*$_G$ utente.

4.1.2.1. AuthController

Controller *NestJS*$_G$ esposto sotto il prefisso `/auth`. Espone *endpoint*$_G$ per consultare i dati dell'utente autenticato, verificare lo stato del *tenant*$_G$ e generare token di *impersonificazione*$_G$.

Metodo	*Endpoint*$_G$	Note
`getMe(req)`	`GET /auth/me`	Restituisce i dati dell'utente autenticato come `AuthenticatedUser`; include informazioni su actor ed effective user
`getTenantStatus(req)`	`GET /auth/tenant`$_G$`-status`	Restituisce lo stato del *tenant*$_G$ corrente; riattiva automaticamente le sospensioni scadute
`impersonate(body, req)`	`POST /auth/impersonate`	Genera un token di *impersonificazione*$_G$ per un utente target; riservato ai `SYSTEM_ADMIN`; protetto da `@AdminOnly()`

Tabella 44: *Endpoint*$_G$ dell'AuthController

Metodi privati:

Metodo	Comportamento
`isSuspensionExpired(tenant`$_G$`)`	Verifica se `status === SUSPENDED` e `suspensionUntil` è nel passato
`reactivateTenantIfExpired(tenant`$_G$`)`	Se la sospensione è scaduta: imposta `status = ACTIVE`, `suspensionIntervalDays = null`, `suspensionUntil = null`

Tabella 45: Metodi privati dell'AuthController

4.1.2.2. ImpersonationService

Servizio per il token exchange con *Keycloak*$_G$. Consente a un amministratore di sistema di ottenere un token *JWT*$_G$ per impersonare un altro utente.

Campi:

Campo	Tipo	Note
`config`	`ConfigService`	Iniettato; fornisce credenziali *Keycloak*$_G$

Tabella 46: Campi dell'ImpersonationService

Metodi pubblici:

Metodo	Firma	Note
`impersonateUser(...)`	`({ adminAccessToken, targetUserId }): Promise<{ access_token, expires_in }>`	Esegue token exchange OAuth 2.0 con *Keycloak*$_G$; `grant_type: urn:ietf:params:oauth:grant-type:token-exchange`

Tabella 47: Metodi pubblici dell'ImpersonationService

4.1.2.3. JwtStrategy

Strategia Passport per la validazione *JWT*$_G$. Verifica i token *JWT*$_G$ provenienti da *Keycloak*$_G$ tramite JWKS.

Campi:

Campo	Tipo	Note
`configService`	`ConfigService`	Iniettato; fornisce URL, realm, issuer, client ID
`managementClientId`	`string`	Memorizzato per il matching dell'audience

Tabella 48: Campi del JwtStrategy

Metodi pubblici:

Metodo	Firma	Note
`validate(payload`$_G$`)`	`(payload`$_G$`: JwtClaims): AuthenticatedUser`	Estrae ruolo, *tenant*$_G$, informazioni di impersonazione dal *payload*$_G$ *JWT*$_G$; costruisce `AuthenticatedUser`

Tabella 49: Metodi pubblici del JwtStrategy

4.1.2.4. Guards

Il modulo registra quattro guard globali come `APP_GUARD`:

JwtAuthGuard — Estende `AuthGuard('jwt`$_G$`')`. Se la policy è `PUBLIC` (decoratore `@Public()`), bypassa la validazione *JWT*$_G$. Altrimenti delega a Passport per la validazione del token.

AccessPolicyGuard — Applica le policy di accesso: `ADMIN` richiede `SYSTEM_ADMIN`, `TENANT`$_G$ richiede un `effectiveTenantId`. Se il *tenant*$_G$ è `SUSPENDED`, solleva `ForbiddenException`. Riattiva automaticamente le sospensioni scadute.

RolesGuard — Verifica che `user.effectiveRole` sia presente nell'array di ruoli richiesto dal decoratore `@Roles()`.

BlockImpersonationGuard — Se il decoratore `@BlockImpersonation()` è impostato e l'utente sta impersonando, registra un evento *audit*$_G$ `IMPERSONATION_BLOCKED` e solleva `ForbiddenException`.

4.1.2.5. Interfacce

AuthenticatedUser

Campo	Tipo	Note
`actorUserId`	`string`	ID dell'utente che sta agendo (impersonante)
`actorEmail`	`string | undefined`	Email dell'attore
`actorName`	`string | undefined`	Nome dell'attore
`actorRole`	`UsersRole`	Ruolo dell'attore
`actorTenantId`	`string | undefined`	*Tenant*$_G$ ID dell'attore
`effectiveUserId`	`string`	ID dell'utente target (effettivo)
`effectiveEmail`	`string | undefined`	Email dell'utente effettivo
`effectiveName`	`string | undefined`	Nome dell'utente effettivo
`effectiveRole`	`UsersRole`	Ruolo dell'utente effettivo
`effectiveTenantId`	`string | undefined`	*Tenant*$_G$ ID dell'utente effettivo
`isImpersonating`	`boolean`	Indica se è attiva l'*impersonificazione*$_G$

Tabella 50: Campi dell'AuthenticatedUser

4.1.3. KeysModule

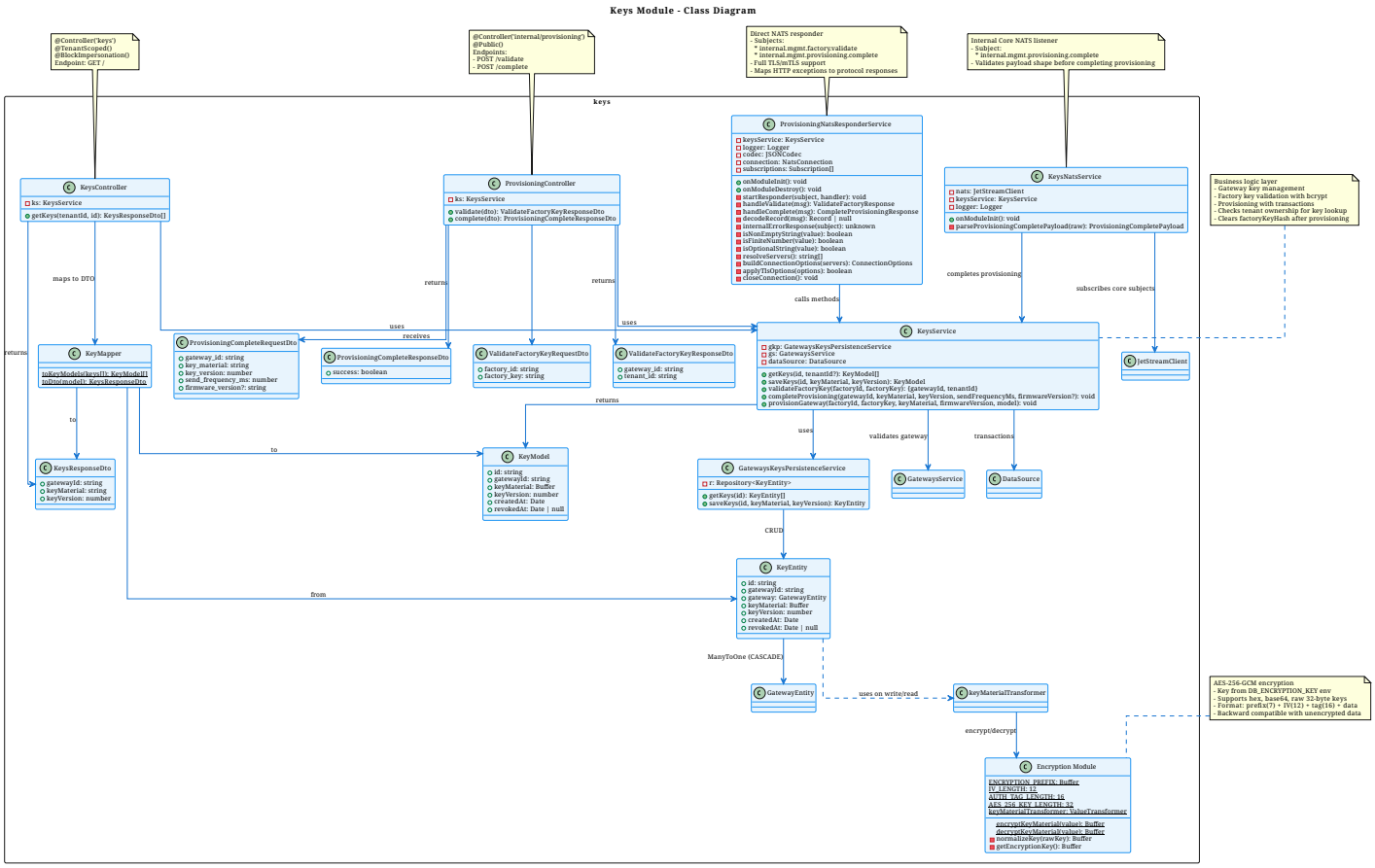

Figura 7: Diagramma del modulo KeysModule

Il modulo di gestione delle chiavi gestisce il *provisioning*$_G$ delle chiavi *AES-256*$_G$ per i *gateway*$_G$, la validazione delle chiavi di fabbrica e il completamento del *provisioning*$_G$. Implementa crittografia trasparente a livello di persistenza.

4.1.3.1. KeysController

Controller *NestJS*$_G$ esposto sotto il prefisso `/keys`. Protetto da `@TenantScoped()` e `@BlockImpersonation()`.

Metodo	*Endpoint*$_G$	Note
`getKeys(tenantId, id)`	`GET /keys?id=<gatewayId>`	Restituisce tutte le chiavi per un *gateway*$_G$ specifico; richiede `tenantId` dal contesto

Tabella 51: *Endpoint*$_G$ del KeysController

4.1.3.2. ProvisioningController

Controller pubblico per il *provisioning*$_G$ interno. Protetto da `@Public()`.

Metodo	*Endpoint*$_G$	Note
`validate(dto`$_G$`)`	`POST /internal/provisioning`$_G$`/validate`	Valida una chiave di fabbrica; restituisce `gateway_id` e `tenant_id` se valida

`complete(dto`$_G$`)`	`POST /internal/provisioning`$_G$`/complete`	Completa il *provisioning*$_G$ con materiale crittografico; restituisce `{ success: true }`

Tabella 52: *Endpoint*$_G$ del ProvisioningController

4.1.3.3. KeysService

Contiene la logica di business principale. Coordina `GatewaysKeysPersistenceService`, `GatewaysService` e `DataSource`.

Campi:

Campo	Tipo	Note
`gkp`	`GatewaysKeysPersistenceService`	Iniettato via constructor
`gs`	`GatewaysService`	Iniettato via constructor
`dataSource`	`DataSource`	Iniettato per supporto transazioni

Tabella 53: Campi del KeysService

Metodi pubblici:

Metodo	Firma	Note
`getKeys(id, tenantId?)`	`(id: string, tenantId?): Promise<KeyModel[]>`	Verifica ownership del *gateway*$_G$; solleva `ForbiddenException` se tenantId non corrisponde; restituisce le chiavi
`saveKeys(id, keyMaterial, keyVersion)`	`(id, keyMaterial, keyVersion): Promise<KeyModel>`	Persiste una nuova chiave per il *gateway*$_G$
`validateFactoryKey(factoryId, factoryKey)`	`(factoryId, factoryKey): Promise<{ gatewayId, tenantId }>`	Cerca *gateway*$_G$ per factoryId; verifica hash con bcrypt; solleva `UnauthorizedException` o `ConflictException`
`completeProvisioning(...)`	`(gatewayId, keyMaterial, keyVersion, sendFrequencyMs, firmwareVersion?): Promise<void>`	Transazione: crea KeyEntity, aggiorna GatewayMetadata, imposta provisioned=true, factoryKeyHash=null
`provisionGateway(...)`	`(factoryId, factoryKey, keyMaterial, firmwareVersion, model): Promise<void>`	Valida factory key, calcola nextVersion, salva nuova KeyEntity, aggiorna *gateway*$_G$

Tabella 54: Metodi pubblici del KeysService

4.1.3.4. GatewaysKeysPersistenceService

Layer di accesso ai dati per le chiavi.

Campi:

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

Tabella 55: Campi del GatewaysKeysPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`getKeys(id)`	`(id: string): Promise<KeyEntity[]>`	Trova tutte le chiavi per `gatewayId`
`saveKeys(id, keyMaterial, keyVersion)`	`(id, keyMaterial, keyVersion): Promise<KeyEntity>`	Crea e salva una nuova `KeyEntity`

Tabella 56: Metodi pubblici del GatewaysKeysPersistenceService

4.1.3.5. Crittografia del Materiale Chiave

Il file `key-material-encryption.ts` fornisce crittografia *AES-256*$_G$*-GCM*$_G$ trasparente per la colonna `keyMaterial` di `KeyEntity` tramite un `ValueTransformer` TypeORM.

Costanti:

Costante	Tipo	Valore
`ENCRYPTION_PREFIX`	`Buffer`$_G$	`Buffer`$_G$`.from('enc:v1:')` — 7 byte
`IV_LENGTH`	`number`	12 byte
`AUTH_TAG_LENGTH`	`number`	16 byte
`AES_256_KEY_LENGTH`	`number`	32 byte

Tabella 57: Costanti della crittografia

Funzioni esportate:

Funzione	Firma	Note
`encryptKeyMaterial(value)`	`(value: Buffer`$_G$`): Buffer`$_G$	Genera *IV*$_G$ casuale, cifra con *AES-256*$_G$*-GCM*$_G$, restituisce `[prefix | IV`$_G$` | authTag | encrypted]`
`decryptKeyMaterial(value)`	`(value: Buffer`$_G$`): Buffer`$_G$	Se inizia con prefix: estrae *IV*$_G$, authTag, decripta. Altrimenti (legacy): restituisce unchanged

Tabella 58: Funzioni della crittografia

keyMaterialTransformer — Oggetto `ValueTransformer` TypeORM con metodi `to` (cripta in scrittura) e `from` (decripta in lettura). Applicato alla colonna `keyMaterial` di `KeyEntity`.

Funzioni helper (non esportate):

Funzione	Comportamento

`normalizeKey(rawKey)`	Normalizza la chiave di crittografia da `DB_ENCRYPTION_KEY`: accetta hex (64 char), base64, o UTF-8 (32 byte)
`getEncryptionKey()`	Legge `DB_ENCRYPTION_KEY` dall'env e chiama `normalizeKey()` per produrre il *buffer*$_G$ a 32 byte

Tabella 59: Funzioni helper (non esportate)

4.1.3.6. *DTO*$_G$ e Input

ValidateFactoryKeyRequestDto — `factory_id: string`, `factory_key: string`.

ValidateFactoryKeyResponseDto — `gateway_id: string`, `tenant_id: string`.

ProvisioningCompleteRequestDto — `gateway_id: string`, `key_material: string`, `key_version: number`, `send_frequency_ms: number`, `firmware_version?: string`.

ProvisioningCompleteResponseDto — `success: boolean`.

KeysResponseDto — `gateway_id: string`, `key_material: string` (base64), `key_version: number`.

4.1.3.7. Entità

KeyEntity (tabella `keys`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco generato automaticamente
`gatewayId`	`uuid`$_G$	FK verso `GatewayEntity.id`; cascade delete
`keyMaterial`	`Buffer`$_G$	Materiale crittografato con *AES-256*$_G$*-GCM*$_G$; trasformatore automatico
`keyVersion`	`number`	Versione della chiave
`createdAt`	`Date`	Timestamp di creazione
`revokedAt`	`Date | null`	Timestamp di revoca (opzionale)

Tabella 60: Campi della KeyEntity

4.1.4. UsersModule

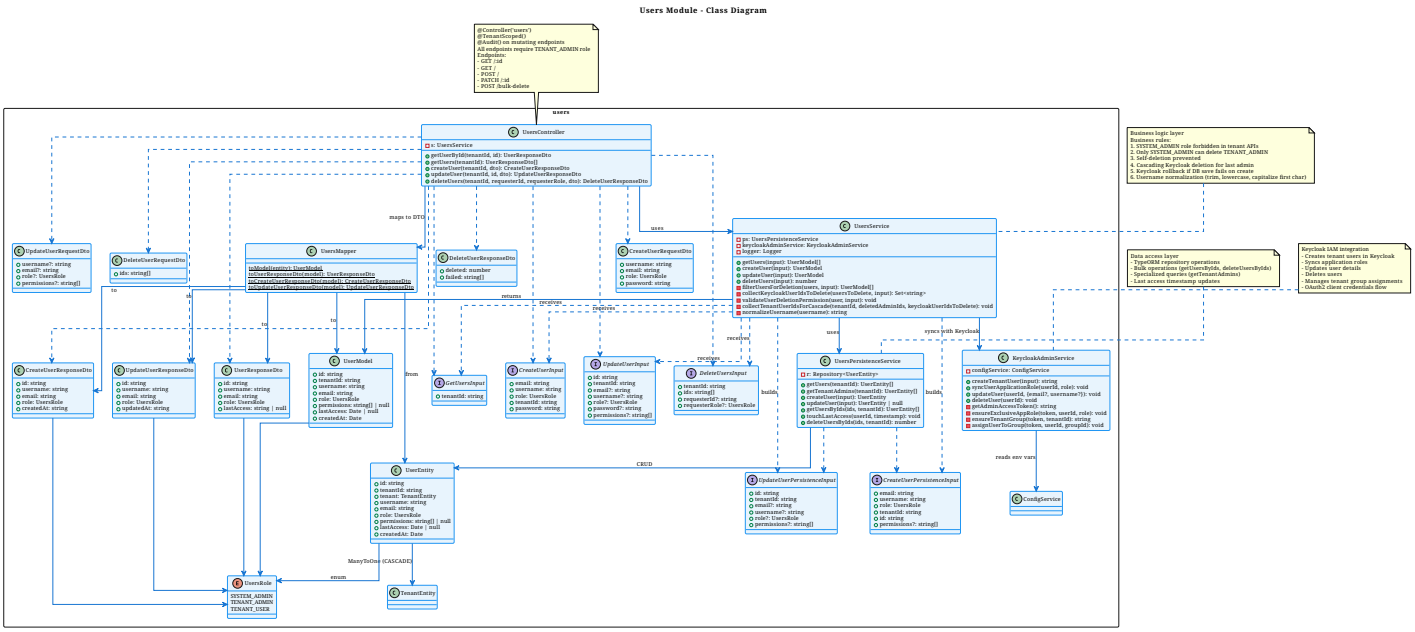

Figura 8: Diagramma del modulo UsersModule

Il modulo di gestione degli utenti gestisce le operazioni CRUD sugli utenti del *tenant*$_G$, inclusa la sincronizzazione con *Keycloak*$_G$ per la creazione, l'aggiornamento dei ruoli e l'eliminazione.

4.1.4.1. UsersController

Controller *NestJS*$_G$ esposto sotto il prefisso `/users`. Protetto da `@TenantScoped()`. Tutti gli *endpoint*$_G$ richiedono `TENANT_ADMIN`.

Metodo	*Endpoint*$_G$	Note
`getUsers(tenantId)`	`GET /users`	Restituisce la lista di tutti gli utenti del *tenant*$_G$
`getUserById(tenantId, id)`	`GET /users/:id`	Restituisce il dettaglio di un singolo utente
`createUser(tenantId, dto`$_G$`)`	`POST /users`	Crea un nuovo utente; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`updateUser(tenantId, id, dto`$_G$`)`	`PATCH`$_G$ `/users/:id`	Aggiorna un utente esistente; sincronizza ruolo e dati con *Keycloak*$_G$
`deleteUsers(tenantId, requesterId, requesterRole, dto`$_G$`)`	`POST /users/bulk-delete`	Eliminazione bulk di utenti; previene auto-eliminazione e protezione TENANT_ADMIN (deprecata)

Tabella 61: *Endpoint*$_G$ dello UsersController

4.1.4.2. UsersService

Contiene la logica di business. Coordina `UsersPersistenceService` e `KeycloakAdminService`.

Campi:

Campo	Tipo	Note
`ps`	`UsersPersistenceService`	Iniettato via constructor
`keycloakAdminService`	`KeycloakAdminService`	Iniettato via constructor
`logger`	`Logger`	Logger interno

Tabella 62: Campi dello UsersService

Metodi pubblici:

Metodo	Firma	Note
`getUsers(input)`	`(input: GetUsersInput): Promise<UserModel[]>`	Recupera gli utenti del *tenant*$_G$; solleva `NotFoundException` se la lista è vuota
`createUser(input)`	`(input: CreateUserInput): Promise<UserModel>`	Blocca `SYSTEM_ADMIN`; crea utente su *Keycloak*$_G$; salva nel DB; rollback *Keycloak*$_G$ se il DB fallisce
`updateUser(input)`	`(input: UpdateUserInput): Promise<UserModel>`	Blocca `SYSTEM_ADMIN`; aggiorna DB; sincronizza ruolo e dati con *Keycloak*$_G$ se cambiati
`deleteUsers(input)`	`(input: DeleteUsersInput): Promise<number>`	Filtra auto-eliminazione; blocca eliminazione TENANT_ADMIN da non-SYSTEM_ADMIN; cancella da *Keycloak*$_G$ e DB

Tabella 63: Metodi pubblici dello UsersService

Metodi privati:

Metodo	Comportamento
`normalizeUsername(username)`	Trim, lowercase, capitalizza prima lettera; restituisce stringa vuota se risultato vuoto

Tabella 64: Metodi privati del UsersService

4.1.4.3. UsersPersistenceService

Layer di accesso ai dati per gli utenti.

Campi:

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

Tabella 65: Campi dello UsersPersistenceService

Metodi pubblici:

Metodo	Firma	Note

`getUsers(tenantId)`	`(tenantId: string): Promise<UserEntity[]>`	Trova tutti gli utenti per `tenantId`
`getTenantAdmins(tenantId)`	`(tenantId: string): Promise<UserEntity[]>`	Trova gli amministratori del *tenant*$_G$
`createUser(input)`	`(input): Promise<UserEntity>`	Crea e salva un nuovo `UserEntity`
`updateUser(input)`	`(input): Promise<UserEntity | null>`	Trova per `id + tenantId`; aggiorna solo i campi forniti; restituisce `null` se non trovato
`getUsersByIds(ids, tenantId)`	`(ids, tenantId): Promise<UserEntity[]>`	Trova utenti per lista di ID con clausola `In(ids)`
`touchLastAccess(userId, timestamp?)`	`(userId, timestamp?): Promise<void>`	Aggiorna `lastAccess` al timestamp corrente o fornito
`deleteUsersByIds(ids, tenantId)`	`(ids, tenantId): Promise<number>`	Elimina utenti per lista di ID; restituisce il numero di righe eliminate

Tabella 66: Metodi pubblici dello UsersPersistenceService

4.1.4.4. *DTO*$_G$

CreateUserRequestDto — `username: string`, `email: string`, `role: UsersRole`, `password: string`.

CreateUserResponseDto — `id: string`, `username: string`, `email: string`, `role: UsersRole`, `created_at: string`.

UpdateUserRequestDto — `username?: string`, `email?: string`, `role?: UsersRole`, `permissions?: string[]`.

UpdateUserResponseDto — `id: string`, `username: string`, `email: string`, `role: UsersRole`, `updated_at: string`.

DeleteUserRequestDto — `ids: string[]` (*UUID*$_G$ v4).

DeleteUserResponseDto — `deleted: number`, `failed: string[]`.

UserResponseDto — `id: string`, `username: string`, `email: string`, `role: UsersRole`, `last_access: string | null`.

4.1.4.5. Entità

UserEntity (tabella `users`)

Campo	Tipo	Note
`id`	`uuid`$_G$	Chiave primaria
`tenantId`	`string`	FK verso `TenantEntity.id`; cascade delete
`username`	`string`	Username dell'utente
`email`	`string`	Email dell'utente

role	UsersRole	Ruolo; default: TENANT_USER
permissions	string[] | null	Permessi specifici in formato JSONB
lastAccess	Date | null	Ultimo accesso
createdAt	Date	Timestamp di creazione

Tabella 67: Campi della UserEntity

Relazione: ManyToOne con `TenantEntity` (tramite `tenantId`, cascade delete).

4.1.5. AlertsModule

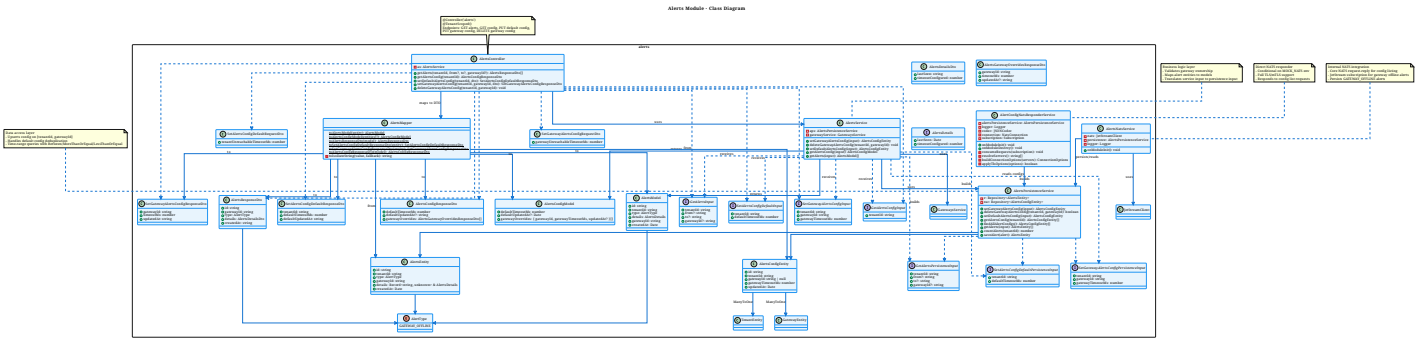

Figura 9: Diagramma del modulo AlertsModule

Il modulo di gestione degli alert gestisce la configurazione dei timeout di irraggiungibilità dei *gateway*$_G$ e la registrazione degli eventi di *gateway*$_G$ offline. Supporta configurazioni di default per *tenant*$_G$ e override per singolo *gateway*$_G$.

4.1.5.1. AlertsController

Controller *NestJS*$_G$ esposto sotto il prefisso `/alerts`. Protetto da `@TenantScoped()`.

Metodo	*Endpoint*$_G$	Note
`getAlerts(tenantId, from?, to?, gatewayId?)`	`GET /alerts`	Restituisce gli alert del *tenant*$_G$ con filtri opzionali per data e *gateway*$_G$
`getAlertsConfig(tenantId)`	`GET /alerts/config`	Restituisce la configurazione degli alert: timeout di default e override per *gateway*$_G$
`setDefaultAlertsConfig(tenantId, dto`$_G$`)`	`PUT /alerts/config/default`	Imposta il timeout di default per il *tenant*$_G$; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`setGatewayAlertsConfig(tenantId, gatewayId, dto`$_G$`)`	`PUT /alerts/config/ gateway`$_G$`/:gatewayId`	Imposta il timeout per un *gateway*$_G$ specifico; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`deleteGatewayAlertsConfig(tenantId, gatewayId)`	`DELETE /alerts/config/ gateway`$_G$`/:gatewayId`	Elimina la configurazione per un *gateway*$_G$; protetto da

		`@Audit`$_G$ e `@Roles(TENANT_ADMIN)`

Tabella 68: *Endpoint*$_G$ dell'AlertsController

4.1.5.2. AlertsService

Contiene la logica di business. Coordina `AlertsPersistenceService` e `GatewaysService`.

Campi:

Campo	Tipo	Note
`aps`	`AlertsPersistenceService`	Iniettato via constructor
`gatewayService`	`GatewaysService`	Iniettato via constructor per validazione ownership

Tabella 69: Campi dell'AlertsService

Metodi pubblici:

Metodo	Firma	Note
`setGatewayAlertsConfig(input)`	`(input): Promise<AlertsConfigEntity>`	Verifica ownership *gateway*$_G$; solleva `NotFoundException` o `ForbiddenException`; upsert della configurazione
`deleteGatewayAlertsConfig(tenantId, gatewayId)`	`(tenantId, gatewayId): Promise<void>`	Verifica ownership; elimina configurazione; solleva `NotFoundException` se non eliminata
`setDefaultAlertsConfig(input)`	`(input): Promise<AlertsConfigEntity>`	Imposta o aggiorna il timeout di default per il *tenant*$_G$
`getAlertsConfig(input)`	`(input): Promise<AlertsConfigModel>`	Recupera tutte le configurazioni del *tenant*$_G$; mappa in modello con default + override
`getAlerts(input)`	`(input): Promise<AlertsModel[]>`	Recupera gli alert con filtri opzionali per data e *gateway*$_G$

Tabella 70: Metodi pubblici dell'AlertsService

4.1.5.3. AlertsPersistenceService

Layer di accesso ai dati per alert e configurazioni.

Campi:

Campo	Tipo	Note
`repository`$_G$	`Repository`$_G$`<AlertsEntity>`	*Repository*$_G$ per gli alert
`alertsConfigRepository`	`Repository`$_G$`<AlertsConfigEntity>`	*Repository*$_G$ per le configurazioni

Tabella 71: Campi dell'AlertsPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`setGatewayAlertsConfig(input)`	`(input): Promise<AlertsConfigEntity>`	Upsert su `['tenantId', 'gatewayId']`; restituisce l'entità aggiornata
`deleteGatewayAlertsConfig(tenantId, gatewayId)`	`(tenantId, gatewayId): Promise<boolean>`	Elimina configurazione; restituisce `true` se righe eliminate
`setDefaultAlertsConfig(input)`	`(input): Promise<AlertsConfigEntity>`	Trova l'ultimo default esistente; aggiorna o crea nuovo; elimina duplicati stale
`getAlertsConfig(tenantId)`	`(tenantId): Promise<AlertsConfigEntity[]>`	Trova tutte le configurazioni del *tenant*$_G$ con relazione *gateway*$_G$
`findAllAlertConfigs()`	`(): Promise<AlertsConfigEntity[]>`	Trova tutte le configurazioni (usato dal *NATS*$_G$ responder)
`getAlerts(input)`	`(input): Promise<AlertsEntity[]>`	Filtra per `tenantId`, `createdAt` (range), `gatewayId`; ordina per `createdAt DESC`
`countAlerts(tenantId)`	`(tenantId): Promise<number>`	Conta gli alert per un *tenant*$_G$
`saveAlert(alert)`	`(alert): Promise<AlertsEntity>`	Crea e salva un nuovo alert

Tabella 72: Metodi pubblici dell'AlertsPersistenceService

4.1.5.4. AlertsNatsService

Servizio *NATS*$_G$ per la ricezione degli alert. Implementa `OnModuleInit`. Si sottoscrive a due subject: `internal.mgmt.alert-configs.list` (request-reply per le configurazioni) e `alert.*.gw_offline` (*JetStream*$_G$ per gli alert *gateway*$_G$ offline).

4.1.5.5. *DTO*$_G$

AlertsResponseDto — `id: string`, `gateway_id: string`, `type: AlertType`, `details: { last_seen: string, timeout_configured: number }`, `created_at: string`.

AlertsConfigResponseDto — `default_timeout_ms: number`, `default_updated_at?: string`, `gateway_overrides: [{ gateway_id, timeout_ms, updated_at? }]`.

SetAlertsConfigDefaultRequestDto — `tenant_unreachable_timeout_ms: number`.

SetAlertsConfigDefaultResponseDto — `tenant_id: string`, `default_timeout_ms: number`, `default_updated_at: string`.

SetGatewayAlertsConfigRequestDto — `gateway_unreachable_timeout_ms: number`.

SetGatewayAlertsConfigResponseDto — `gateway_id: string`, `timeout_ms: number`, `updated_at: string`.

4.1.5.6. Enum

AlertType

Valore	Descrizione
`GATEWAY_OFFLINE`	*Gateway*$_G$ non raggiungibile entro il timeout configurato

Tabella 73: Valori dell'enum AlertType

4.1.5.7. Entità

AlertsEntity (tabella `alerts`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco
`tenantId`	`uuid`$_G$	FK verso il *tenant*$_G$
`type`	`AlertType`	Tipo di alert; default: `GATEWAY_OFFLINE`
`gatewayId`	`uuid`$_G$	ID del *gateway*$_G$ che ha generato l'alert
`details`	`jsonb`	Dettagli: `lastSeen`, `timeoutConfigured`
`createdAt`	`Date`	Timestamp di creazione

Tabella 74: Campi dell'AlertsEntity

AlertsConfigEntity (tabella `alert_configs`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco
`tenantId`	`string`	FK verso `TenantEntity`; cascade delete
`gatewayId`	`string | null`	FK verso `GatewayEntity`; `null` = configurazione di default
`gatewayTimeoutMs`	`number`	Timeout in millisecondi; default: 60000
`updatedAt`	`Date`	Timestamp dell'ultimo aggiornamento

Tabella 75: Campi dell'AlertsConfigEntity

Indice composito unico su `['tenantId', 'gatewayId']`.

4.1.6. CommandModule

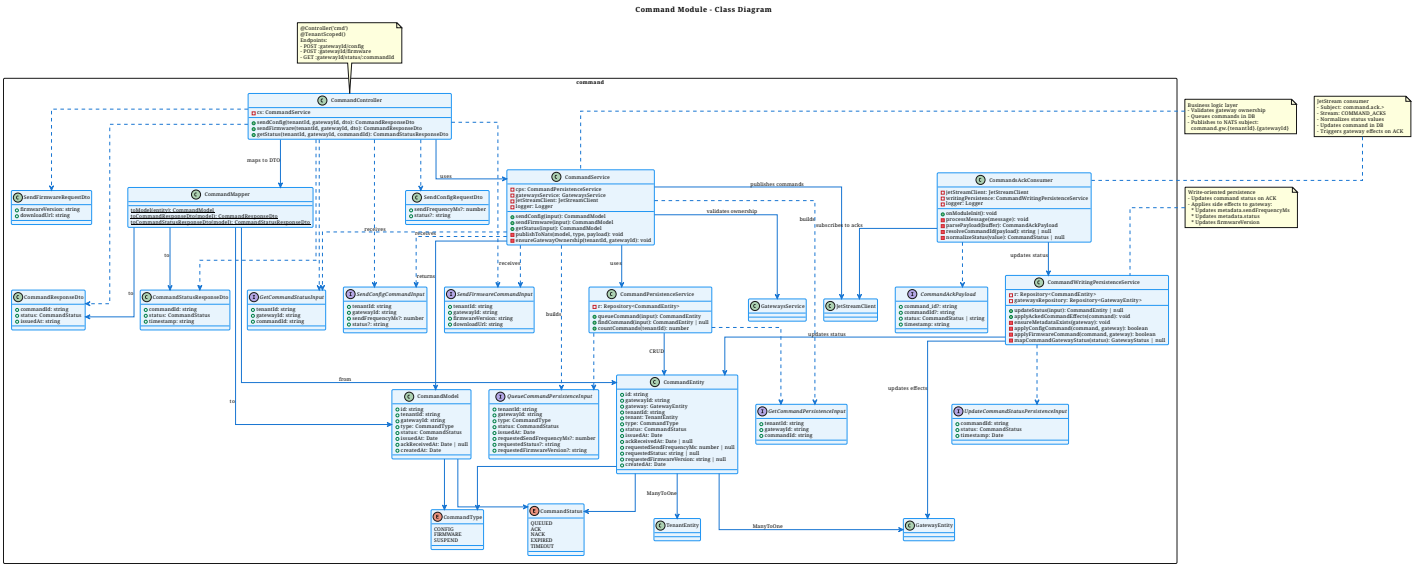

Figura 10: Diagramma del modulo CommandModule

Il modulo di gestione dei comandi gestisce l'invio di comandi ai *gateway*$_G$ (configurazione e firmware) e il tracciamento dello stato attraverso *ACK*$_G$ ricevuti via *NATS*$_G$ *JetStream*$_G$.

4.1.6.1. CommandController

Controller *NestJS*$_G$ esposto sotto il prefisso `/cmd`. Protetto da `@TenantScoped()`.

Metodo	*Endpoint*$_G$	Note
`sendConfig(tenantId, gatewayId, dto`$_G$`)`	`POST :gatewayId/config`	Invia un comando di configurazione; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`sendFirmware(tenantId, gatewayId, dto`$_G$`)`	`POST :gatewayId/firmware`	Invia un comando di aggiornamento firmware; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`getStatus(tenantId, gatewayId, commandId)`	`GET :gatewayId/status/:commandId`	Restituisce lo stato di esecuzione di un comando

Tabella 76: *Endpoint*$_G$ del CommandController

4.1.6.2. CommandService

Contiene la logica di business. Coordina `CommandPersistenceService`, `GatewaysService` e `JetStreamClient`.

Campi:

Campo	Tipo	Note
`cps`	`CommandPersistenceService`	Iniettato via constructor
`gatewaysService`	`GatewaysService`	Iniettato per validazione ownership
`jetStreamClient`	`JetStreamClient`	Iniettato per pubblicazione *NATS*$_G$
`logger`	`Logger`	Logger interno

Tabella 77: Campi del CommandService

Metodi pubblici:

Metodo	Firma	Note
`sendConfig(input)`	`(input): Promise<CommandModel>`	Valida input; verifica ownership; coda comando; pubblica su *NATS*$_G$ `command.gw.<tenantId>.<gatewayId>`
`sendFirmware(input)`	`(input): Promise<CommandModel>`	Verifica ownership; coda comando con tipo `FIRMWARE`; pubblica su *NATS*$_G$
`getStatus(input)`	`(input): Promise<CommandModel>`	Trova comando per ID; solleva `NotFoundException` se non trovato

Tabella 78: Metodi pubblici del CommandService

Metodi privati:

Metodo	Comportamento
`publishToNats(model, type, payload`$_G$`)`	Costruisce subject `command.gw.<tenantId>.<gatewayId>`; pubblica *JSON*$_G$ con `command_id`, `type`, `payload`$_G$, `issued_at`; non solleva errori in caso di fallimento
`ensureGatewayOwnership(tenantId, gatewayId)`	Delega a `gatewaysService.findById()`; solleva `NotFoundException` se il *gateway*$_G$ non appartiene al *tenant*$_G$

Tabella 79: Metodi privati del CommandService

4.1.6.3. CommandsAckConsumer

Consumer$_G$ *NATS*$_G$ *JetStream*$_G$ per gli *ACK*$_G$. Si sottoscrive a `command.ack`$_G$`.>`. Aggiorna lo stato del comando nel database e, se lo stato è `ACK`$_G$, applica gli effetti del comando sull'entità *gateway*$_G$.

Metodi privati:

Metodo	Comportamento
`processMessage(message)`	Parsa *payload*$_G$; risolve commandId (supporta `command_id` e `commandId`); normalizza stato; valida timestamp; aggiorna stato; se *ACK*$_G$ applica effetti; *ack*$_G$ del messaggio
`parsePayload(buffer`$_G$`)`	Decodifica *buffer*$_G$ UTF-8 e pars *JSON*$_G$
`resolveCommandId(payload`$_G$`)`	Controlla `command_id` poi `commandId`; restituisce primo valore valido o `null`
`normalizeStatus(value)`	Mappa stringa a `CommandStatus` enum tramite tabella di normalizzazione

Tabella 80: Metodi privati del CommandsAckConsumer

4.1.6.4. CommandWritingPersistenceService

Servizio di scrittura per i comandi. Gestisce l'aggiornamento dello stato e l'applicazione degli effetti sui *gateway*$_G$.

Metodi pubblici:

Metodo	Firma	Note
`updateStatus(input)`	`(input): Promise<CommandEntity | null>`	Trova comando per ID; aggiorna `status` e `ackReceivedAt`; restituisce `null` se non trovato
`applyAckedCommandEffects(command)`	`(command): Promise<void>`	Se $\texttt{ACK}_G$ e tipo `CONFIG`: aggiorna `sendFrequencyMs` e/o `status` su $gateway_G$ metadata. Se `FIRMWARE`: aggiorna `firmwareVersion` su $gateway_G$. Salva solo se ci sono modifiche

Tabella 81: Metodi pubblici del CommandWritingPersistenceService

Metodi privati:

Metodo	Comportamento
`mapCommandGatewayStatus(status)`	Mappa stringa a `GatewayStatus`: `online`/`gateway_online` ⟶ `GATEWAY_ONLINE`, `paused`/`suspended` ⟶ `GATEWAY_SUSPENDED`, `offline` ⟶ `GATEWAY_OFFLINE`

Tabella 82: Metodi privati del CommandWritingPersistenceService

4.1.6.5. CommandPersistenceService

Layer di accesso ai dati per i comandi (read/create).

Metodi pubblici:

Metodo	Firma	Note
`queueCommand(input)`	`(input): Promise<CommandEntity>`	Crea e salva un nuovo `CommandEntity` con stato `QUEUED`
`findCommand(input)`	`(input): Promise<CommandEntity | null>`	Trova comando per `id` + `tenantId` + `gatewayId`
`countCommands(tenantId)`	`(tenantId): Promise<number>`	Conta i comandi per un $tenant_G$

Tabella 83: Metodi del CommandPersistenceService

4.1.6.6. Enum

CommandStatus

Valore	Descrizione
`QUEUED`	Comando accodato in attesa di invio
$\texttt{ACK}_G$	Comando confermato dal $gateway_G$
`NACK`	Comando negato dal $gateway_G$
`EXPIRED`	Comando scaduto
`TIMEOUT`	Timeout nella ricezione dell'ACK_G

Tabella 84: Valori dell'enum CommandStatus

CommandType

Valore	Descrizione
`CONFIG`	Aggiornamento configurazione: frequenza di invio, stato operativo
`FIRMWARE`	Aggiornamento firmware
`SUSPEND`	Sospensione del *gateway*$_G$

Tabella 85: Valori dell'enum CommandType

4.1.6.7. Entità

CommandEntity (tabella `commands`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco
`gatewayId`	`uuid`$_G$	FK verso `GatewayEntity`; cascade delete
`tenantId`	`uuid`$_G$	FK verso `TenantEntity`; cascade delete
`type`	`CommandType`	Tipo di comando
`status`	`CommandStatus`	Stato corrente
`issuedAt`	`Date`	Timestamp di emissione
`ackReceivedAt`	`Date | null`	Timestamp di ricezione *ACK*$_G$
`requestedSendFrequencyMs`	`number | null`	Frequenza richiesta (per comandi CONFIG)
`requestedStatus`	`string | null`	Stato richiesto (per comandi CONFIG)
`requestedFirmwareVersion`	`string | null`	Versione firmware richiesta (per comandi FIRMWARE)
`createdAt`	`Date`	Timestamp di creazione

Tabella 86: Campi del CommandEntity

4.1.6.8. *DTO*$_G$

SendConfigRequestDto — `send_frequency_ms?: number` (min 0), `status?: string`.

SendFirmwareRequestDto — `firmware_version: string`, `download_url: string` (URL valido).

CommandResponseDto — `command_id: string`, `status: CommandStatus`, `issued_at: string`.

CommandStatusResponseDto — `command_id: string`, `status: CommandStatus`, `timestamp: string`.

4.1.7. AuditLogModule

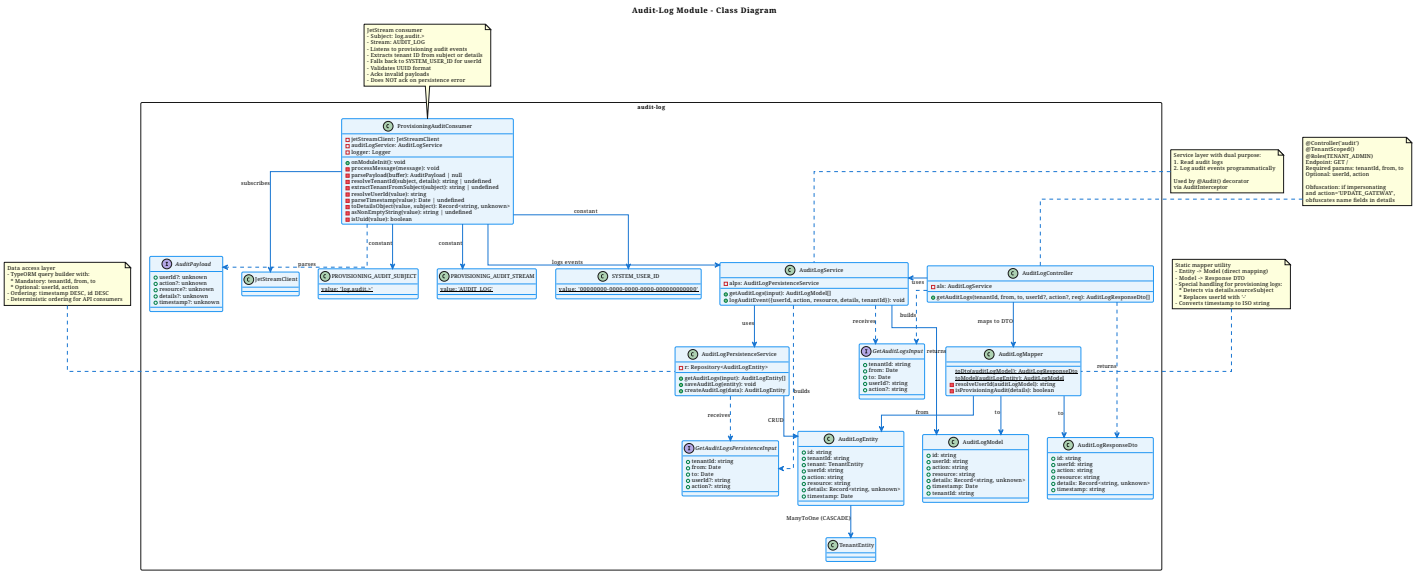

Figura 11: Diagramma del modulo AuditLogModule

Il modulo di *audit*$_G$ log gestisce la registrazione e la consultazione delle operazioni eseguite sul microservizio. Supporta la registrazione di eventi provenienti da altri *microservizi*$_G$ tramite *NATS*$_G$.

4.1.7.1. AuditLogController

Controller *NestJS*$_G$ esposto sotto il prefisso `/audit`$_G$. Protetto da `@TenantScoped()` e `@Roles(TENANT_ADMIN)`.

Metodo	*Endpoint*$_G$	Note
`getAuditLogs(tenantId, from, to, userId?, action?)`	`GET /audit`$_G$	Restituisce i log di *audit*$_G$ del *tenant*$_G$ con filtri per data, utente e azione; offusca i nomi dei *gateway*$_G$ durante l'*impersonificazione*$_G$

Tabella 87: *Endpoint*$_G$ dell'AuditLogController

4.1.7.2. AuditLogService

Servizio di registrazione e consultazione dei log di *audit*$_G$.

Campi:

Campo	Tipo	Note
`alps`	`AuditLogPersistenceService`	Iniettato via constructor

Tabella 88: Campi dell'AuditLogEntity

Metodi pubblici:

Metodo	Firma	Note
`getAuditLogs(input)`	`(input): Promise<AuditLogModel[]>`	Delega al persistence layer e mappa le entità in modelli

`logAuditEvent(...)`	`({ userId, action, resource, details, tenantId }): Promise<void>`	Crea e salva un nuovo evento di *audit*$_G$

Tabella 89: Metodi pubblici dell'AuditLogService

4.1.7.3. ProvisioningAuditConsumer

Consumer$_G$ *NATS*$_G$ *JetStream*$_G$ per gli eventi di *audit*$_G$ provenienti da altri *microservizi*$_G$. Si sottoscrive a `log.audit`$_G$`.>`. Parsa i *payload*$_G$, valida i campi richiesti, risolve *tenant*$_G$ ID e user ID, e registra l'evento.

Metodi privati:

Metodo	Comportamento
`processMessage(message)`	Parsa *payload*$_G$; valida action/resource/timestamp/tenantId; risolve userId; registra evento; *ack*$_G$ del messaggio
`resolveTenantId(subject, details)`	Estrae *tenant*$_G$ ID dal subject *NATS*$_G$ o dall'oggetto details; valida formato *UUID*$_G$
`resolveUserId(value)`	Restituisce *UUID*$_G$ string o `SYSTEM_USER_ID` (`00000000-0000-0000-0000-000000000000`)
`parseTimestamp(value)`	Parsa stringa in Date; restituisce `undefined` se non valida
`toDetailsObject(value, subject)`	Avvolge i dettagli; aggiunge `sourceSubject` se il subject esiste

Tabella 90: Metodi privati del ProvisioningAuditConsumer

4.1.7.4. AuditLogMapper

Metodi statici:

Metodo	Comportamento
`toModel(entity)`	Converte `AuditLogEntity` in `AuditLogModel`
`toDto(model)`	Converte in `AuditLogResponseDto`; offusca userId per *audit*$_G$ di *provisioning*$_G$ (restituisce -)

Tabella 91: Metodi statici dell'AuditLogMapper

Metodi privati:

Metodo	Comportamento
`resolveUserId(model)`	Restituisce - se è un *audit*$_G$ di *provisioning*$_G$ (`sourceSubject` inizia con `log.audit`$_G$`.`)
`isProvisioningAudit(details)`	Verifica se `sourceSubject` inizia con `log.audit`$_G$`.`

Tabella 92: Metodi privati dell'AuditLogController

4.1.7.5. Entità

AuditLogEntity (tabella `audits`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco

tenantId	uuid$_G$	FK verso TenantEntity; cascade delete
userId	uuid$_G$	ID dell'utente che ha eseguito l'azione
action	string	Tipo di azione eseguita
resource	string	Risorsa su cui è stata eseguita l'azione
details	jsonb	Dettagli aggiuntivi dell'evento
timestamp	Date	Timestamp dell'evento

Tabella 93: Campi dell'AuditLogEntity

4.1.8. ApiClientModule

Il modulo di gestione dei client API gestisce la creazione, consultazione ed eliminazione dei client *OIDC*$_G$ associati ai *tenant*$_G$, con sincronizzazione su *Keycloak*$_G$.

4.1.8.1. ApiClientController

Controller *NestJS*$_G$ esposto sotto il prefisso `/api-clients`. Protetto da `@TenantScoped()` e `@Roles(TENANT_ADMIN)`.

Metodo	*Endpoint*$_G$	Note
getApiClients(tenantId)	GET /api-clients	Restituisce tutti i client API del *tenant*$_G$
createApiClient(tenantId, input)	POST /api-clients	Crea un nuovo client API; protetto da @Audit$_G$
deleteApiClient(tenantId, id)	DELETE /api-clients/:id	Elimina un client API; protetto da @Audit$_G$

Tabella 94: *Endpoint*$_G$ dell'ApiClientController

4.1.8.2. ApiClientService

Campi:

Campo	Tipo	Note
acps	ApiClientPersistenceService	Iniettato via constructor
keycloakAdminService	KeycloakAdminService	Iniettato via constructor
logger	Logger	Logger interno

Tabella 95: Campi dell'ApiClientService

Metodi pubblici:

Metodo	Firma	Note
createApiClient(tenantId, name)	(tenantId, name): Promise<{ model, clientSecret }>	Verifica conflitto nome; crea client su *Keycloak*$_G$; salva nel DB; rollback *Keycloak*$_G$ se DB fallisce

`getApiClients(tenantId)`	`(tenantId): Promise<ApiClientModel[]>`	Recupera tutti i client API del *tenant*$_G$
`deleteApiClient(tenantId, id)`	`(tenantId, id): Promise<void>`	Elimina dal DB, poi da *Keycloak*$_G$ (warn se *Keycloak*$_G$ fallisce)
`deleteApiClientsForTenant(tenantId)`	`(tenantId): Promise<void>`	Elimina bulk di tutti i client API per un *tenant*$_G$

Tabella 96: Metodi pubblici dell'ApiClientService

4.1.8.3. ApiClientPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`createApiClient(id, tenantId, name, keycloakClientId)`	`(...): Promise<ApiClientEntity>`	Crea e salva un nuovo `ApiClientEntity`
`findByName(tenantId, name)`	`(tenantId, name): Promise<ApiClientEntity | null>`	Trova client per `tenantId + name`
`getApiClients(tenantId)`	`(tenantId): Promise<ApiClientEntity[]>`	Trova tutti i client per `tenantId`
`deleteApiClient(tenantId, id)`	`(tenantId, id): Promise<string | null>`	Trova entity, rimuove dal DB, restituisce il `keycloakClientId` o `null`

Tabella 97: Metodi pubblici dell'ApiClientPersistenceService

4.1.8.4. Entità

ApiClientEntity (tabella `api_clients`)

Campo	Tipo	Note
`id`	`string`	Chiave primaria (stringa, non *UUID*$_G$)
`tenantId`	`uuid`$_G$	FK verso `TenantEntity`; cascade delete
`name`	`string`	Nome del client; indice unico
`keycloakClientId`	`string`	Client ID su *Keycloak*$_G$; indice unico
`createdAt`	`Date`	Timestamp di creazione

Tabella 98: Campi dell'ApiClientEntity

4.1.9. ThresholdsModule

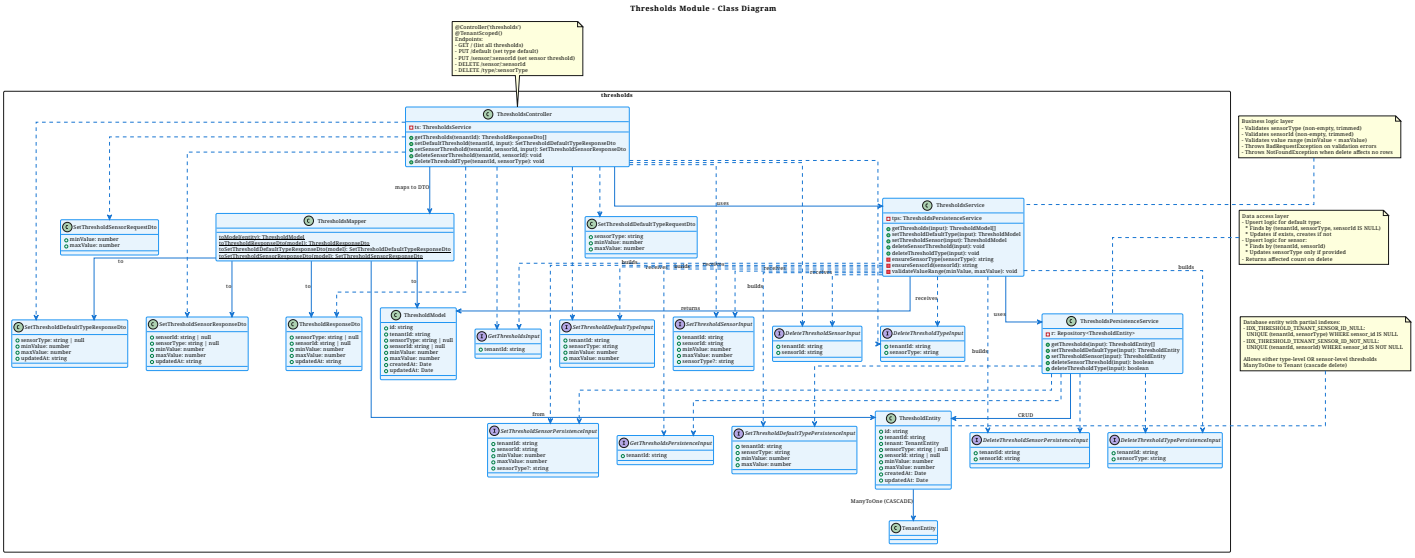

Figura 12: Diagramma del modulo ThresholdsModule

Il modulo di gestione delle soglie gestisce i valori minimi e massimi per i sensori, sia a livello di tipo di sensore (default per *tenant*$_G$) che a livello di singolo sensore.

4.1.9.1. ThresholdsController

Controller *NestJS*$_G$ esposto sotto il prefisso `/thresholds`. Protetto da `@TenantScoped()`.

Metodo	*Endpoint*$_G$	Note
`getThresholds(tenantId)`	`GET /thresholds`	Restituisce tutte le soglie del *tenant*$_G$; accessibile a `TENANT_USER` e `TENANT_ADMIN`
`setDefaultThreshold(tenantId, input)`	`PUT /thresholds/default`	Imposta la soglia di default per un tipo di sensore; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`setSensorThreshold(tenantId, sensorId, input)`	`PUT /thresholds/ sensor/:sensorId`	Imposta la soglia per un sensore specifico; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`deleteSensorThreshold(tenantId, sensorId)`	`DELETE /thresholds/ sensor/:sensorId`	Elimina la soglia per un sensore; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`
`deleteThresholdType(tenantId, sensorType)`	`DELETE /thresholds/ type/:sensorType`	Elimina la soglia di default per un tipo di sensore; protetto da `@Audit`$_G$ e `@Roles(TENANT_ADMIN)`

Tabella 99: *Endpoint*$_G$ del ThresholdsController

4.1.9.2. ThresholdsService

Campi:

Campo	Tipo	Note
`tps`	`ThresholdsPersistenceService`	Iniettato via constructor

Tabella 100: Campi del ThresholdsService

Metodi pubblici:

Metodo	Firma	Note
`getThresholds(input)`	`(input): Promise<ThresholdModel[]>`	Recupera tutte le soglie del $tenant_G$
`setThresholdDefaultType(input)`	`(input): Promise<ThresholdModel>`	Valida `sensorType` e range valori; delega al persistence layer
`setThresholdSensor(input)`	`(input): Promise<ThresholdModel>`	Valida `sensorId` e range valori; delega al persistence layer
`deleteSensorThreshold(input)`	`(input): Promise<void>`	Solleva `NotFoundException` se non trovato
`deleteThresholdType(input)`	`(input): Promise<void>`	Solleva `NotFoundException` se non trovato

Tabella 101: Metodi pubblici del ThresholdsService

Metodi privati:

Metodo	Comportamento
`ensureSensorType(sensorType)`	Trim e valida che non sia vuoto; solleva `BadRequestException`
`ensureSensorId(sensorId)`	Trim e valida che non sia vuoto; solleva `BadRequestException`
`validateValueRange(minValue, maxValue)`	Valida che i numeri siano validi e `minValue < maxValue`; solleva `BadRequestException`

Tabella 102: Metodi privati del ThresholdsService

4.1.9.3. ThresholdsPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`getThresholds(input)`	`(input): Promise<ThresholdEntity[]>`	Trova per `tenantId`; ordina per `updatedAt DESC`
`setThresholdDefaultType(input)`	`(input): Promise<ThresholdEntity>`	Upsert: trova per `tenantId + sensorType + sensorId IS NULL`; aggiorna o crea

`setThresholdSensor(input)`	`(input): Promise<ThresholdEntity>`	Upsert: trova per `tenantId + sensorId`; aggiorna o crea
`deleteSensorThreshold(input)`	`(input): Promise<boolean>`	Elimina per `tenantId + sensorId`; restituisce `true` se eliminato
`deleteThresholdType(input)`	`(input): Promise<boolean>`	Elimina per `tenantId + sensorType + sensorId IS NULL`; restituisce `true` se eliminato

Tabella 103: Metodi pubblici del ThresholdsPersistenceService

4.1.9.4. Entità

ThresholdEntity (tabella `thresholds`)

Campo	Tipo	Note
`id`	`uuid`$_G$ `v4`	Identificativo univoco
`tenantId`	`uuid`$_G$	FK verso `TenantEntity`; cascade delete
`sensorType`	`string | null`	Tipo di sensore (per soglie di default)
`sensorId`	`uuid`$_G$ `| null`	ID del sensore specifico (per override)
`minValue`	`number`	Valore minimo (float)
`maxValue`	`number`	Valore massimo (float)
`createdAt`	`Date`	Timestamp di creazione
`updatedAt`	`Date`	Timestamp dell'ultimo aggiornamento

Tabella 104: Campi del ThresholdEntity

Indici unici composti: uno su `['tenantId', 'sensorType']` dove `sensor_id IS NULL`, uno su `['tenantId', 'sensorId']` dove `sensor_id IS NOT NULL`.

4.1.10. CostsModule

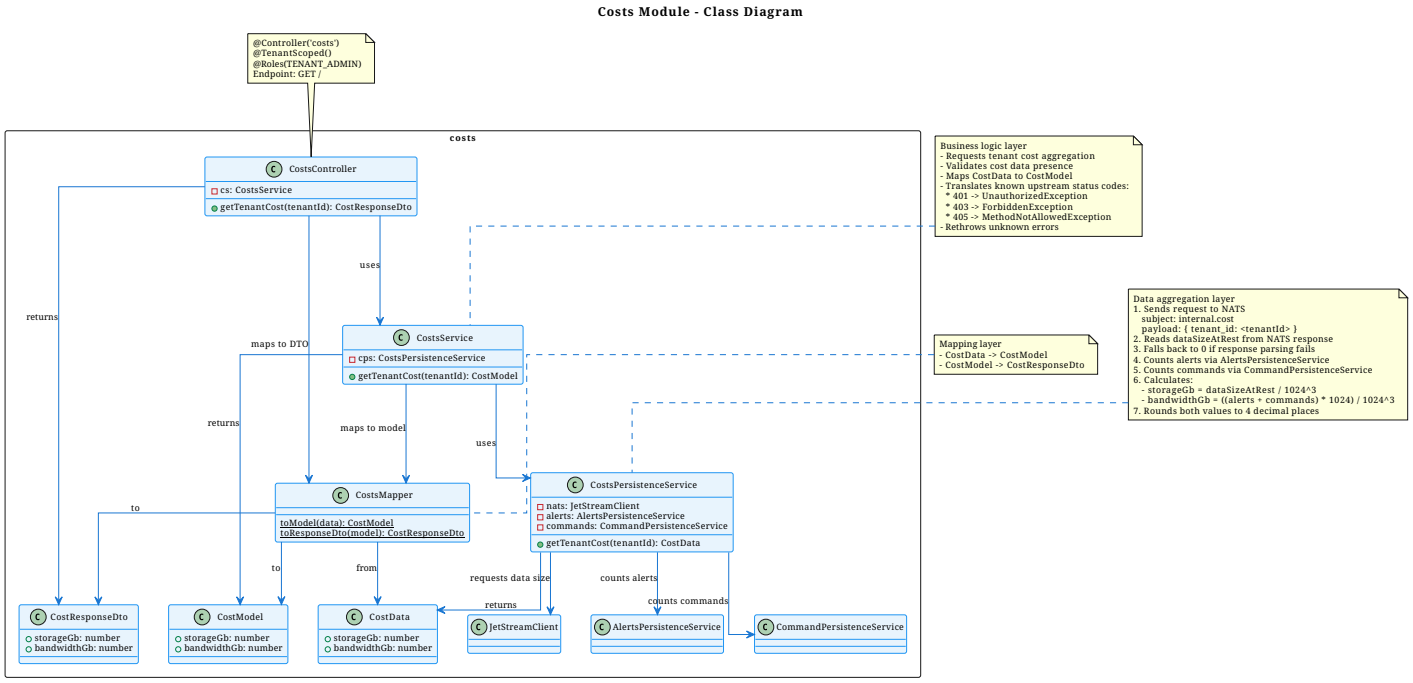

Figura 13: Diagramma del modulo CostsModule

Il modulo di gestione dei costi calcola lo storage e la banda utilizzati da un *tenant*$_G$, combinando dati da *NATS*$_G$, alert e comandi.

4.1.10.1. CostsController

Controller *NestJS*$_G$ esposto sotto il prefisso `/costs`. Protetto da `@TenantScoped()` e `@Roles(TENANT_ADMIN)`.

Metodo	*Endpoint*$_G$	Note
`getTenantCost(tenantId)`	`GET /costs`	Restituisce i costi stimati del *tenant*$_G$: storage e banda in GB

Tabella 105: *Endpoint*$_G$ del CostsController

4.1.10.2. CostsService

Campi:

Campo	Tipo	Note
`cps`	`CostsPersistenceService`	Iniettato via constructor

Tabella 106: Campi del CostsService

Metodi pubblici:

Metodo	Firma	Note
`getTenantCost(tenantId)`	`(tenantId): Promise<CostModel>`	Delega al persistence layer; solleva `NotFoundException` se nessun dato; mappa errori HTTP-like a eccezioni *NestJS*$_G$

Tabella 107: Metodi pubblici del CostsService

4.1.10.3. CostsPersistenceService

Campi:

Campo	Tipo	Note
`nats`$_G$	`JetStreamClient`	Iniettato per richiesta *NATS*$_G$
`alerts`	`AlertsPersistenceService`	Iniettato per conteggio alert
`commands`	`CommandPersistenceService`	Iniettato per conteggio comandi

Tabella 108: Campi del CostsPersistenceService

Metodi pubblici:

Metodo	Firma	Note
`getTenantCost(tenantId)`	`(tenantId): Promise<CostData>`	Invia richiesta *NATS*$_G$ a `internal.cost` con `{ tenant_id }`; calcola `storageGb = dataSizeAtRest / 1GB` e `bandwidthGb = (alertsCount + commandsCount) * 1KB / 1GB`; entrambi arrotondati a 4 decimali

Tabella 109: Metodi pubblici del CostsPersistenceService

4.1.10.4. *DTO*$_G$

CostResponseDto — `storage_gb: number`, `bandwidth_gb: number`.

4.1.10.5. Modelli

CostModel — `storageGb: number`, `bandwidthGb: number`.

CostData — `storageGb: number`, `bandwidthGb: number`.

4.2. Flussi di esecuzione

Di seguito sono descritti i principali flussi di esecuzione del microservizio, con particolare attenzione ai componenti coinvolti e alle interazioni con i servizi esterni.

4.2.1. Autenticazione

Il client ottiene un token *JWT*$_G$ da *Keycloak*$_G$ e lo include nelle richieste inviate al microservizio. Il *backend*$_G$ valida il token tramite la strategia *JWT*$_G$ configurata nel modulo di autenticazione, verificando issuer, audience e chiavi pubbliche del realm. In caso di validazione positiva, le informazioni contenute nel token vengono trasformate in un utente autenticato applicativo, includendo ruolo effettivo, *tenant*$_G$ di appartenenza ed eventuali informazioni di impersonazione. Tali informazioni vengono poi utilizzate dai guard e dalle policy di accesso per autorizzare o negare l'accesso agli *endpoint*$_G$.

4.2.2. Creazione di un *tenant*$_G$

Un utente con ruolo *system_admin* invia una richiesta di creazione di un *tenant*$_G$ tramite l'*endpoint*$_G$ amministrativo dedicato. Il controller delega l'operazione al service applicativo,

che avvia una transazione sul database locale e crea l'entità del *tenant*$_G$. Successivamente il *backend*$_G$ interagisce con *Keycloak*$_G$ per creare l'utente amministratore del *tenant*$_G$, assegnargli il ruolo applicativo previsto e associargli gli attributi necessari, incluso il riferimento al *tenant*$_G$. Una volta completata la creazione remota, il microservizio persiste anche il corrispondente utente locale nel database applicativo. In caso di errore durante il flusso, vengono applicati meccanismi di rollback per ridurre il rischio di inconsistenze tra database locale e *Keycloak*$_G$.

4.2.3. Creazione di un utente *tenant*$_G$

Un utente con ruolo *tenant_admin* invia una richiesta di creazione di un nuovo utente appartenente al proprio *tenant*$_G$. Il controller inoltra i dati al service applicativo, che coordina la creazione dell'utente sia nel database locale sia su *Keycloak*$_G$. Dopo la creazione remota, il *backend*$_G$ salva le informazioni dell'utente nel database applicativo, mantenendo allineati il sistema locale e il provider di identità. Il ruolo applicativo assegnato all'utente viene sincronizzato con i ruoli configurati in *Keycloak*$_G$.

4.2.4. Impersonazione di un utente

Un utente con ruolo *system_admin* invia una richiesta di impersonazione tramite l'*endpoint*$_G$ `/auth/impersonate`, specificando l'identificativo dell'utente target. Il microservizio estrae il token amministrativo della richiesta e delega a *Keycloak*$_G$ l'operazione di token exchange. *Keycloak*$_G$ restituisce un nuovo access token che rappresenta l'utente impersonato, mantenendo le informazioni necessarie a identificare anche l'attore originale. Il token risultante viene poi utilizzato nelle richieste successive verso il microservizio. Durante l'impersonazione, alcuni *endpoint*$_G$ risultano bloccati tramite guard dedicati, così da impedire specifiche operazioni sensibili mentre si opera nel contesto di un altro utente.

4.2.5. *Provisioning*$_G$ di un *gateway*$_G$

Il flusso di *provisioning*$_G$ è esposto tramite *endpoint*$_G$ interni dedicati. In una prima fase il sistema chiamante invia `factory_id` e `factory_key` all'*endpoint*$_G$ di validazione; il microservizio verifica che il *gateway*$_G$ esista, che non sia già provisioned e che la factory key fornita corrisponda al valore atteso. In caso di esito positivo, il *backend*$_G$ restituisce l'identificativo del *gateway*$_G$ e del *tenant*$_G$ associato. In una seconda fase viene inviata la richiesta di completamento del *provisioning*$_G$, contenente il materiale della chiave e la relativa versione. Il microservizio persiste la nuova chiave associandola al *gateway*$_G$, applica la cifratura del materiale sensibile in fase di salvataggio e aggiorna lo stato del *gateway*$_G$ come `provisioned`. A partire da questo momento il *gateway*$_G$ viene considerato attivo dal punto di vista applicativo.

4.2.6. Recupero delle chiavi di un *gateway*$_G$

Un utente autenticato del *tenant*$_G$ richiede le chiavi associate a uno specifico *gateway*$_G$. Il controller inoltra l'identificativo del *tenant*$_G$ e del *gateway*$_G$ al service applicativo, che verifica preliminarmente che il *gateway*$_G$ appartenga effettivamente al *tenant*$_G$ chiamante. Dopo il controllo di autorizzazione, il layer di persistenza recupera le chiavi dal database. Il materiale crittografico viene decifrato automaticamente durante la lettura tramite il transformer associato all'entità, e i dati vengono infine convertiti nel formato di risposta esposto dall'API.

4.2.7. Creazione di un API client

Un utente con ruolo *tenant_admin* richiede la creazione di un nuovo API client associato al *tenant*$_G$. Il microservizio delega a *Keycloak*$_G$ la creazione del client applicativo e del relativo

secret, quindi persiste nel database locale le informazioni necessarie a rappresentarlo nel dominio applicativo. La risposta restituisce sia i dati del nuovo client sia, nel momento della creazione, il secret necessario per l'utilizzo da parte del chiamante.

4.2.8. Invio di un comando a un *gateway*$_G$

Un utente con ruolo *tenant_admin* invia un comando verso un *gateway*$_G$, ad esempio per aggiornare la configurazione o avviare un aggiornamento firmware. Il controller raccoglie i dati della richiesta e li passa al service applicativo, che crea il record del comando nel database con stato iniziale coerente con il flusso di esecuzione. Successivamente il comando viene inoltrato al layer di integrazione con *NATS*$_G$/*JetStream*$_G$ per la pubblicazione verso il sistema destinatario. Lo stato del comando può poi essere interrogato tramite l'*endpoint*$_G$ dedicato, che restituisce le informazioni persistite sul relativo avanzamento.

4.2.9. Configurazione di alert e threshold

Gli utenti del *tenant*$_G$ possono consultare la configurazione degli alert e delle soglie, mentre le operazioni di modifica sono riservate agli utenti con privilegi amministrativi sul *tenant*$_G$. Le richieste vengono gestite dai rispettivi controller e inoltrate ai service applicativi, che verificano il contesto *tenant*$_G$ e applicano le regole di autorizzazione. Le configurazioni possono essere definite a livello generale di *tenant*$_G$ oppure, a seconda del caso d'uso, a livello specifico di *gateway*$_G$ o sensore. Le modifiche vengono persistite nel database e restituite al chiamante in forma coerente con il contratto API.

4.2.10. Consultazione dei log di *audit*$_G$

Un utente con ruolo *tenant_admin* può interrogare i log di *audit*$_G$ specificando almeno un intervallo temporale di interesse. Il controller valida la presenza dei parametri obbligatori e inoltra la richiesta al service dedicato, che recupera dal database gli eventi registrati per il *tenant*$_G$. I risultati vengono quindi trasformati nel formato di risposta previsto e restituiti al chiamante. Questo flusso consente di tracciare le operazioni rilevanti eseguite nel sistema da utenti reali o impersonati.

5. Metodologie di testing

Il microservizio prevede test di unità e test di integrazione intra-service, con l'obiettivo di verificare la correttezza della logica applicativa, dei controlli di accesso, delle trasformazioni dei dati e delle interazioni con il layer di persistenza. Le attività di test sono orientate alla validazione del comportamento dei componenti e non alla semplice verifica della tecnologia utilizzata.

5.1. Test di unità

I test di unità coprono i controller, i service applicativi, i guard, i mapper, i transformer e i componenti di supporto che implementano regole di business o di sicurezza. In particolare, devono essere verificati i seguenti aspetti:

- corretta delega dei controller ai service applicativi;
- corretta applicazione dei controlli di accesso basati su ruolo, *tenant*$_G$ e stato di impersonazione;
- corretta validazione e interpretazione delle informazioni estratte dal token *JWT*$_G$;
- corretta gestione dei principali casi d'uso applicativi, inclusi *tenant*$_G$, utenti, *gateway*$_G$, chiavi, client API, alert, threshold e comandi;

- corretta gestione delle condizioni di errore e delle eccezioni applicative;
- corretta trasformazione tra *DTO*$_G$, model ed entity;
- corretta cifratura e decifratura del materiale sensibile associato alle chiavi dei *gateway*$_G$;
- corretta validazione della configurazione applicativa caricata all'avvio.

5.2. Test di integrazione intra-service

I test di integrazione intra-service verificano il comportamento coordinato di più componenti interni al microservizio, con particolare attenzione alle interazioni tra logica applicativa, persistenza e configurazione. In particolare, devono essere coperti i seguenti aspetti:

- corretta inizializzazione del bootstrap applicativo in presenza di configurazione valida;
- corretto fallimento dell'avvio in presenza di configurazione mancante o invalida;
- corretto comportamento dei componenti di persistenza verso il database di test;
- corretta esecuzione delle transazioni nei flussi critici, come *provisioning*$_G$ dei *gateway*$_G$ e creazione dei *tenant*$_G$;
- corretta persistenza e recupero delle entity applicative;
- corretta integrazione del layer di autenticazione con il modello utente applicativo;
- corretta integrazione dei componenti che interagiscono con *Keycloak*$_G$ o con *NATS*$_G$, ove tali integrazioni siano simulate o testate tramite fixture dedicate.

5.3. Obiettivi di copertura funzionale

Le attività di testing devono garantire che il microservizio sia verificato almeno rispetto ai seguenti ambiti funzionali:

- autenticazione e autorizzazione;
- segregazione multi-*tenant*$_G$;
- gestione *tenant*$_G$ e utenti;
- gestione *gateway*$_G$ e *provisioning*$_G$;
- gestione delle chiavi e protezione del materiale sensibile;
- gestione dei client API;
- gestione alert e threshold;
- invio e tracciamento dei comandi;
- *audit*$_G$ delle operazioni rilevanti;
- corretto caricamento della configurazione e bootstrap del microservizio.