NoTIP

NO TESTS IN PRODUCTION

Specifica tecnica - *Provisioning*$_G$ service

Versione	1.1.0
Data Modifica	2026-04-17
Utilizzo	Esterno

Abstract dei contenuti

Specifica tecnica del microservizio notip-*provisioning*$_G$-service: architettura esagonale (Ports and Adapters), *endpoint*$_G$ di *onboarding*$_G$ *gateway*$_G$, contratti *NATS*$_G$, gestione CA e metriche operative.

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
1.1.0	2026-04-17	Mario De Pasquale	Leonardo Preo	*Patch*$_G$ specifica post revisione PB
1.0.0	2026-04-13	Leonardo Preo (responsabile)		Approvazione per ingresso in *baseline*$_G$ PB
0.2.0	2026-04-07	Mario De Pasquale	Alessandro Mazzariol	Portata specifica up to date con le modifiche effettuate al codice
0.1.1	2026-04-07	Matteo Mantoan	Francesco Marcon	Uniformazione titoli e intestazioni: solo prima lettera maiuscola
0.1.0	2026-04-01	Mario De Pasquale	Francesco Marcon	Fix dati mancanti
0.0.1	2026-03-29	Mario De Pasquale	Alessandro Mazzariol	Creazione prima bozza del documento

Indice

1. Introduzione

Il servizio `notip-provisioning`$_G$`-service` è un microservizio *NestJS*$_G$ che espone un *endpoint*$_G$ HTTP per il *provisioning*$_G$ iniziale dei *gateway*$_G$ e un *endpoint*$_G$ HTTP per l'esposizione delle metriche *Prometheus*$_G$. Durante la richiesta di *onboarding*$_G$, il servizio valida le credenziali di fabbrica verso il Management API tramite *NATS*$_G$ Request-Reply, firma il *CSR*$_G$ del *gateway*$_G$ con la CA interna, genera una chiave *AES-256*$_G$ e completa il *provisioning*$_G$ persistendo il materiale chiave nel dominio management.

Il servizio adotta un'architettura esagonale (Ports and Adapters) con il core di business logic isolato da interfacce (`ports`), implementate da adapter infrastrutturali. Questa architettura mantiene separati:

- il core di business logic (`provisioning`$_G$`.service`, model di dominio);
- i driving port / primary adapter (*endpoint*$_G$ HTTP di *onboarding*$_G$ e metriche);
- i driven port / secondary adapter (*NATS*$_G$ client, CA signer, generatore chiavi, persistence su *filesystem*$_G$);
- configurazione e osservabilità (`config`, `metrics`).

2. Dipendenze e configurazione

2.1. Stack tecnologico

Il microservizio è implementato in *TypeScript*$_G$ su *Node.js*$_G$ con framework *NestJS*$_G$ e usa le dipendenze principali:

- *NestJS*$_G$ (`@nestjs`$_G$`/common`, `@nestjs`$_G$`/core`, `@nestjs`$_G$`/platform-express`) per modularità e dependency injection;
- `nats`$_G$ per chiamate Request-Reply ai subject interni di Management API;
- `node-forge` per parsing *CSR*$_G$ e firma certificati X.509;
- `prom-client` per metriche applicative *Prometheus*$_G$;
- `class-validator` e `class-transformer` per validazione *DTO*$_G$ di input.

2.2. Variabili d'ambiente

Il caricamento configurazione avviene in `loadConfig()` (`src/config/provisioning`$_G$`.config.ts`). Le variabili richieste mancanti causano errore all'avvio.

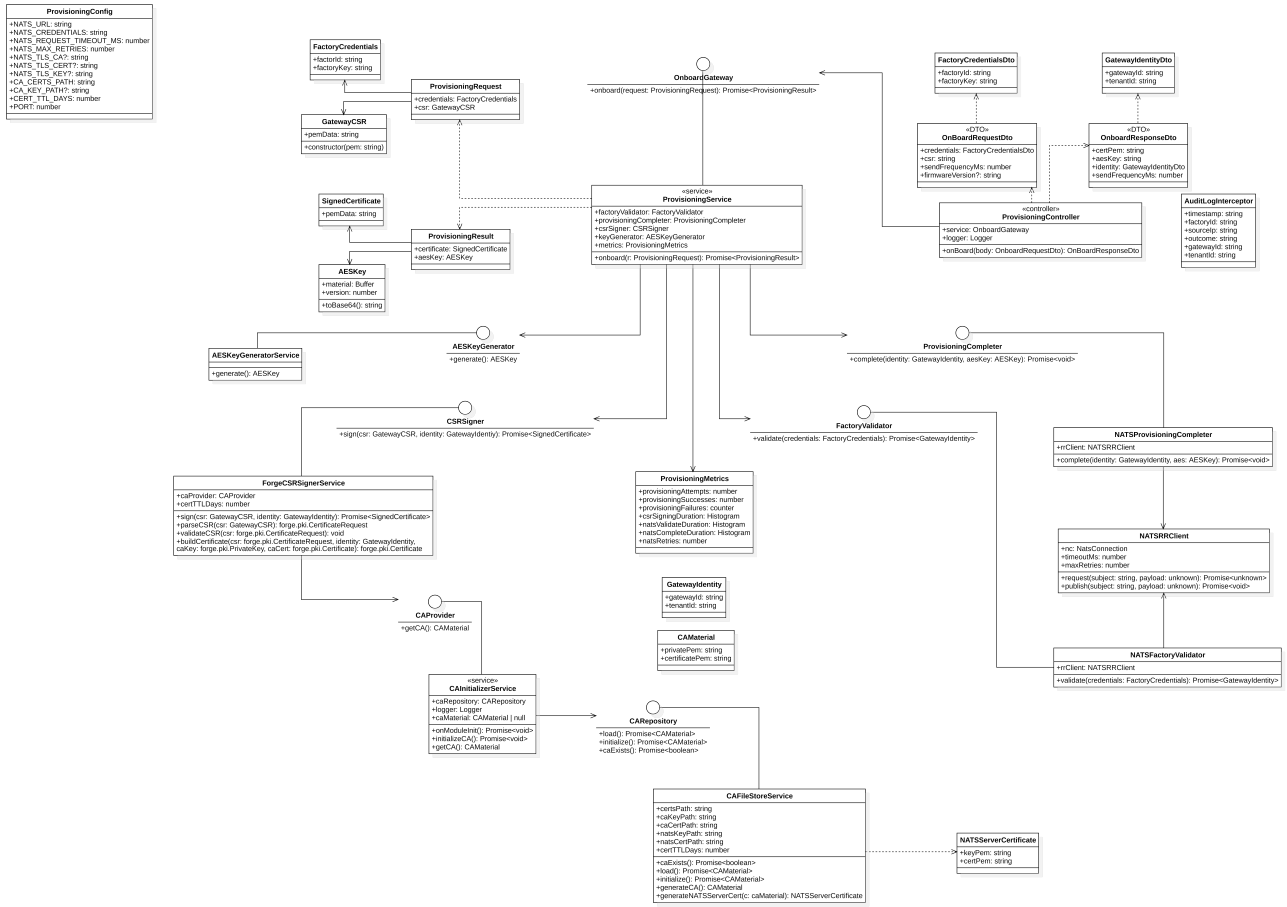
Campo	Variabile d'ambiente	Default	Obbligatorio
`NATS_URL`	NATS_URL	-	Si
`NATS_CREDENTIALS`	NATS_CREDENTIALS	-	Si
`NATS_TLS_CA`	NATS_TLS_CA	-	No
`NATS_TLS_CERT`	NATS_TLS_CERT	-	No
`NATS_TLS_KEY`	NATS_TLS_KEY	-	No
`NATS_REQUEST_TIMEOUT_MS`	NATS_REQUEST_TIMEOUT_MS	`5000`	No
`NATS_MAX_RETRIES`	NATS_MAX_RETRIES	`3`	No
`CA_CERTS_PATH`	CA_CERTS_PATH	`/certs`	No
`CA_KEY_PATH`	CA_KEY_PATH	`CA_CERTS_PATH/ca.key`	No
`CERT_TTL_DAYS`	CERT_TTL_DAYS	`90`	No

`PORT`	PORT	`3004`	No

2.3. Sequenza di avvio

Step	Componente	Azione	Bloccante
1	`ConfigModule`	Carica `CONFIG` globale dal processo e valida i campi richiesti	Si
2	`NATSRRClient.onModuleInit`	Apre connessione *NATS*$_G$ con URL e credenziali configurate	Si
3	`CAInitializerService.onModuleInit`	Carica CA dal volume o genera CA e certificati *NATS*$_G$ se assenti	Si
4	`main`$_G$`.ts`	Registra `ValidationPipe` globale con `whitelist: true`	Si
5	`app.listen(PORT)`	Espone listener HTTP sulla porta configurata (default 3004)	No

3. Architettura esagonale



3.1. Impostazione architetturale

Il servizio applica il pattern esagonale con una chiara separazione tra core di dominio e infrastruttura:

- il **core** contiene la business logic pura in `ProvisioningService` e i modelli di dominio senza dipendenze esterne;
- i **port** sono interfacce astratte (`OnboardGateway`, `FactoryValidator`$_G$, `CSRSigner`$_G$, `AESKeyGenerator`$_G$, `ProvisioningCompleter`$_G$) che definiscono i contratti tra il core e l'esterno;
- i **driving adapter** (lato cliente) includono il controller HTTP e gli interceptor di *audit*$_G$/ metriche;
- i **driven adapter** (lato fornitore) implementano i port per *NATS*$_G$, CA, crypto, e persistence su *filesystem*$_G$;
- le dipendenze alle framework e librerie esterne fluiscono **verso il core**, non dall'interno del core verso l'esterno.

3.2. Layout dei package

```
notip-provisioningG-service/
├── src/
│   ├── mainG.ts
│   ├── app.module.ts
│   ├── config/
│   │   ├── config.module.ts
│   │   └── provisioningG.config.ts
│   ├── provisioningG/
│   │   ├── provisioningG.module.ts
│   │   ├── provisioningG.controller.ts
│   │   ├── provisioningG.service.ts
│   │   ├── provisioningG-exception.filter.ts
│   │   ├── auditG-log.interceptor.ts
│   │   ├── dtoG/
│   │   ├── interfaces/
│   │   └── model/
│   ├── natsG/
│   │   ├── natsG.module.ts
│   │   ├── natsG-rr.client.ts
│   │   ├── natsG-factory-validator.service.ts
│   │   └── natsG-provisioningG-completer.service.ts
│   ├── ca/
│   │   ├── ca.module.ts
│   │   ├── ca-initializer.service.ts
│   │   ├── ca-file-store.service.ts
│   │   ├── forge-csrG-signer.service.ts
│   │   ├── interfaces/
│   │   └── model/
│   ├── crypto/
│   │   ├── crypto.module.ts
│   │   └── aes-key-generator.service.ts
│   └── metrics/
│       ├── metrics.module.ts
│       ├── metrics.controller.ts
│       ├── metrics.interceptor.ts
│       ├── metrics.service.ts
│       └── provisioningG.metrics.ts
└── test/
```

3.3. Struttura esagonale

Componente	Sotto-componenti	Responsabilità

Core di business logic	`ProvisioningService`, modelli di dominio (`ProvisioningRequest`, `ProvisioningResult`, `GatewayIdentity`, `AESKey`, ecc.)	Definisce le regole di business e orchestrazione del *provisioning*$_G$ iniziale dei *gateway*$_G$. Indipendente da *NestJS*$_G$ e librerie esterne (dipende solo da port).
Port (Interfacce)	`OnboardGateway`, `FactoryValidator`$_G$, `CSRSigner`$_G$, `AESKeyGenerator`$_G$, `ProvisioningCompleter`$_G$, `CARepository`$_G$, `CAProvider`$_G$	Contratti astratti tra il core e gli adapter esterni. Definiscono input/output attesi senza dettagli implementativi.
Driving Adapter (Primary / Inbound)	`ProvisioningController`, `OnboardRequestDto`, `OnboardResponseDto`, `ProvisioningExceptionFilter`, `AuditLogInterceptor`, `MetricsController`	Espongono i port del core verso il mondo esterno. Ricevono richieste HTTP e le convertono in chiamate al core. Convertono risposte del core in HTTP e registrano *audit*$_G$/metriche.
Driven Adapter (Secondary / Outbound)	`NATSRRClient`, `NATSFactoryValidator`, `NATSProvisioningCompleter`, `ForgeCSRSignerService`, `AESKeyGeneratorService`, `CAFileStoreService`, `CAInitializerService`	Implementano i port del core per integrarsi con l'infrastruttura esterna (*NATS*$_G$, *PKI*$_G$, *filesystem*$_G$). Isolano il core dalle specifiche tecnologie.
Configurazione e osservabilità	`ConfigModule`, `MetricsService`, `MetricsInterceptor`, `ProvisioningMetrics`	Caricano configurazione all'avvio e forniscono metriche *Prometheus*$_G$ osservabili senza accoppiamento al core.

4. Definizione dei port

4.1. Driving port

POST /provision/onboard

DRIVING PORT

Endpoint$_G$ pubblico di *onboarding*$_G$ del *gateway*$_G$ esposto dal controller di *provisioning*$_G$.

Metodo	Responsabilità
`request`	`credentials.factoryId`, `credentials.factoryKey`, `csr`$_G$, `sendFrequencyMs >= 1`, `firmwareVersion?`
`response`	`certPem`, `aesKey`, `identity.gatewayId`, `identity.tenantId`, `sendFrequencyMs`
`status`	201 in caso di successo
`auth model`	Autenticazione con credenziali di fabbrica nel body, senza *JWT*$_G$

GET /metrics

DRIVING PORT

Endpoint$_G$ operativo per scraping *Prometheus*$_G$ esposto dal `MetricsController`.

Metodo	Responsabilità
`request`	`nessun payload`$_G$
`response`	`text/plain` *Prometheus*$_G$ exposition format
`status`	200 in caso di successo
`auth model`	Nessuna autenticazione applicativa implementata nel servizio

4.2. Driven port

FactoryValidator$_G$ **DRIVEN PORT**

Valida `factory_id` e `factory_key` tramite subject *NATS*$_G$ `internal.mgmt.factory.validate`.

Metodo	Responsabilità
`validate(credentials)`	Ritorna `GatewayIdentity` o solleva errore di dominio

CSRSigner$_G$ **DRIVEN PORT**

Firma il *CSR*$_G$ del *gateway*$_G$ usando la CA caricata in memoria.

Metodo	Responsabilità
`sign(csr`$_G$`, identity)`	Ritorna `SignedCertificate` o `MalformedCSRError`

AESKeyGenerator$_G$ **DRIVEN PORT**

Genera una chiave *AES-256*$_G$ casuale con *CSPRNG*$_G$ del sistema operativo.

Metodo	Responsabilità
`generate()`	Ritorna `AESKey` con `version = 1`

ProvisioningCompleter$_G$ **DRIVEN PORT**

Completa il *provisioning*$_G$ nel Management API tramite `internal.mgmt.provisioning`$_G$`.complete`.

Metodo	Responsabilità
`complete(identity, aesKey, sendFrequencyMs, firmwareVersion)`	Persistenza key material, key version, frequenza invio e versione firmware opzionale

***CARepository*$_G$ / *CAProvider*$_G$** **DRIVEN PORT**

Persistenza e accesso alla CA del servizio.

Metodo	Responsabilità
`CARepository`$_G$`.caExists/load/initialize`	Gestione stato e bootstrap materiale CA su volume
`CAProvider`$_G$`.getCA`	Esposizione sincrona di `CAMaterial` in memoria

5. Design di dettaglio

5.1. Flusso di *provisioning*$_G$ (*Use Case*$_G$ Onboard)

Il metodo `ProvisioningService.onboard(request)` esegue i seguenti step in sequenza:

Step	Operazione	Dettaglio
1	Validazione factory credentials	Chiama `FactoryValidator`$_G$`.validate`; riceve `GatewayIdentity` o errori 401/409/503.
2	Firma *CSR*$_G$	Chiama `CSRSigner`$_G$`.sign`; produce certificato foglia X.509 o errore 400.
3	Generazione chiave AES	Chiama `AESKeyGenerator`$_G$`.generate`; produce 32 byte random e `version = 1`.
4	Completamento *provisioning*$_G$	Chiama `ProvisioningCompleter`$_G$`.complete` con identity, chiave, `send_frequency_ms` e `firmware_version` opzionale.
5	Aggiornamento metriche	Incrementa counter success/failure e osserva durate delle operazioni critiche.
6	Mapping output	Restituisce `certPem`, `aesKey` (base64), `identity` e `sendFrequencyMs` con status 201.

5.2. Modello dati del dominio

Tipo	Campi	Note
`FactoryCredentials`	`factoryId: string`, `factoryKey: string`	Wrapper delle credenziali di fabbrica. `factoryKey` non deve essere loggata.
`GatewayCSR`	`pemData: string`	Valida header *PEM*$_G$ `-----BEGIN CERTIFICATE REQUEST-----` in costruzione.
`ProvisioningRequest`	`credentials`, `csr`$_G$, `sendFrequencyMs: number`, `firmwareVersion: string = ""`	Input applicativo del flusso *onboarding*$_G$. `sendFrequencyMs` deve essere un intero positivo maggiore o uguale a 1; `firmwareVersion` è normalizzato a stringa vuota quando omesso nel *DTO*$_G$.
`GatewayIdentity`	`gatewayId: string`, `tenantId: string`	Identità ricevuta dalla validazione Management API.

`AESKey`	`material: Buffer`$_G$`(32), version: number`	Espone `toBase64()` per serializzazione risposta/*NATS*$_G$.
`SignedCertificate`	`pemData: string`	Certificato foglia firmato dalla CA interna.
`ProvisioningResult`	`certificate`, `aesKey`, `identity`, `sendFrequencyMs`	Output applicativo consumato dal controller e dall'*audit*$_G$ interceptor.
`CAMaterial`	`privateKeyPem`, `certificatePem`	Materiale CA in memoria process-wide dopo bootstrap.
`NATSServerCertificate`	`keyPem`, `certPem`	Certificato server *NATS*$_G$ generato e salvato su volume in fase *init*$_G$ CA.

5.3. Gestione errori

Il filtro `ProvisioningExceptionFilter` converte gli errori di dominio in risposte HTTP stabili e preserva gli `HttpException` già costruiti da *NestJS*$_G$:

Errore	HTTP	Body
`HttpException` / validation error	status originale	body originale
`MalformedCSRError`	400	`{ "error": "MALFORMED_CSR" }`
`InvalidFactoryCredentialsError`	401	`{ "error": "INVALID_CREDENTIALS" }`
`GatewayAlreadyProvisionedError`	409	`{ "error": "ALREADY_PROVISIONED" }`
`ManagementAPIUnavailableError`	503	`{ "error": "SERVICE_UNAVAILABLE" }`
`ProvisioningDomainError` o altri	500	`{ "error": "INTERNAL_ERROR" }`

5.4. *Audit*$_G$ logging e regole di sicurezza

L'interceptor `AuditLogInterceptor` produce un record *JSON*$_G$ per ogni richiesta, con i campi:

- `timestamp`
- `factory_id`
- `source_ip` (header `x-forwarded-for` o `request.ip`)
- `outcome` (`success`, `invalid_credentials`, `already_provisioned`, `malformed_csr`, `service_unavailable`, `error`)
- `gateway_id` e `tenant_id` solo nei casi di successo

Nei casi di successo con `tenant_id` disponibile, l'interceptor pubblica anche un evento *audit*$_G$ su *NATS*$_G$ nel subject `log.audit`$_G$`.<tenant_id>`, con action `PROVISIONING_ONBOARD_<OUTCOME>` e dettagli contestuali (`factoryId`, `sourceIp`, `gatewayId`, `tenantId`).

Campi sensibili esclusi dai log:

- `factory_key`
- `aeskey`/materiale AES
- `certificate` completo
- `CAMaterial.privateKeyPem`

5.5. Integrazione *NATS*$_G$

Il client condiviso `NATSRRClient` implementa:

- timeout per request (`NATS_REQUEST_TIMEOUT_MS`);

- numero tentativi totali pari a `NATS_MAX_RETRIES` (include il primo tentativo);
- retry con backoff esponenziale `2^(attempt-1)` secondi tra un tentativo e il successivo (es. con `NATS_MAX_RETRIES=3`: 1s, 2s);
- riconnessione automatica in caso di `NatsError`;
- incremento metrica `nats_retries_total` ad ogni retry;
- supporto retry sia per chiamate request-reply sia per publish.

Subject usati dal servizio:

- `internal.mgmt.factory.validate`
- `internal.mgmt.provisioning`$_G$`.complete`

5.6. Gestione CA e certificati

La componente CA usa la directory configurata in `CA_CERTS_PATH` (default `/certs`).

In assenza di materiale CA, `CAFileStoreService.initialize()` genera e persiste:

- `ca.key` (permessi 600)
- `ca.crt` (permessi 644)
- `nats`$_G$`.key` (permessi 600)
- `nats`$_G$`.crt` (permessi 644)

Parametri principali:

- CA root self-signed con validità 10 anni;
- certificato server *NATS*$_G$ con validità 1 anno;
- certificato foglia *gateway*$_G$ con *TTL*$_G$ `CERT_TTL_DAYS` (default 90).

6. Osservabilità e metriche

Il servizio registra metriche applicative tramite `ProvisioningMetrics` e metriche HTTP/process tramite `MetricsService` (`prom-client` default metrics, oltre alle metriche HTTP custom):

Metrica	Descrizione
`provisioning_attempts_total`	Numero richieste *onboarding*$_G$ ricevute
`provisioning_successes_total`	Numero *provisioning*$_G$ completati
`provisioning_failures_total{reason}`	Failure per categoria errore
`csr_signing_duration_ms`	Istogramma durata firma *CSR*$_G$
`nats_validate_duration_ms`	Istogramma durata chiamata validate
`nats_complete_duration_ms`	Istogramma durata chiamata complete
`nats_retries_total`	Numero totale retry *NATS*$_G$ (request-reply e publish)
`notip_provisioning_http_requests_total`	Counter HTTP per metodo, route e status code
`notip_provisioning_http_request_duration_seconds`	Istogramma durata delle richieste HTTP
`notip_provisioning_http_requests_in_flight`	Gauge delle richieste HTTP in corso per metodo

7. Strategia di test

La suite nella cartella `test/` copre i principali componenti:

- test unitari su servizi core (`provisioning`$_G$`.service`, `nats`$_G$`-rr.client`, `forge-csr`$_G$`-signer`, `ca-file-store`);
- test su boundary HTTP (`provisioning`$_G$`.controller`, `provisioning`$_G$`-exception.filter`, `audit`$_G$`-log.interceptor`);
- test e2e dedicati eseguibili con `npm run test:e2e`.

Script disponibili in progetto:

- `npm run test`
- `npm run test:cov`
- `npm run test:e2e`
- `npm run lint:check`
- `npm run typecheck`

8. Relazioni tra componenti

Da	Relazione
`ProvisioningController`	dipende da `OnboardGateway`
`ProvisioningService`	implementa `OnboardGateway`
`NATSFactoryValidator`	implementa `FactoryValidator`$_G$
`NATSProvisioningCompleter`	implementa `ProvisioningCompleter`$_G$
`ForgeCSRSignerService`	implementa `CSRSigner`$_G$
`AESKeyGeneratorService`	implementa `AESKeyGenerator`$_G$
`CAFileStoreService`	implementa `CARepository`$_G$
`CAInitializerService`	implementa `CAProvider`$_G$
`NATSRRClient`	supporta validator e completer con RR + retry
`AuditLogInterceptor`	avvolge *endpoint*$_G$ *onboarding*$_G$ con *audit*$_G$ outcome