# Capitolo 1 – Introduzione a NATS e architetture a microservizi event-driven

Sommario

[1. Introduzione 1](#_Toc199457790)

[2. Perché un message broker? 3](#_Toc199457791)

[2.1 comunicazione sincrona vs asincrona 4](#_Toc199457792)

[2.2 I limiti delle chiamate HTTP dirette 4](#_Toc199457793)

[2.3 Vantaggi di resilienza, scalabilità e decoupling 5](#_Toc199457794)

[3 Esempi esplicativi “a bassa tecnologia” 6](#_Toc199457795)

[3.1 Il sistema di caselle postali in ufficio 6](#_Toc199457796)

[3.2 La bacheca in mensa universitaria 6](#_Toc199457797)

[4. Cos’è NATS 6](#_Toc199457798)

[4.1 Storia e filosofia “Simple & Performant” 7](#_Toc199457799)

[4.2 Architettura di base (server, cluster, leafnodes) 7](#_Toc199457800)

[4.3 Subjects, Channels e Wildcards 8](#_Toc199457801)

[5 Componenti chiave 8](#_Toc199457802)

[5.1 Publisher 8](#_Toc199457803)

[5.2 Subscriber 9](#_Toc199457804)

[5.3 NATS Server 9](#_Toc199457805)

[5.4 Subject: il “canale” logico 9](#_Toc199457806)

[5. Primi passi con Rust e NATS 10](#_Toc199457807)

[5.1 Esempio minimale 10](#_Toc199457808)

[Coding: struttura del progetto e file di base 11](#_Toc199457809)

## 1. Introduzione

Nel mondo delle architetture a microservizi, ogni componente è responsabile di un compito ben definito e, idealmente, non deve conoscere i dettagli interni dei suoi “vicini”. Ma come coordinare i servizi senza incorrere nei problemi di latenza, blocchi e tight-coupling tipici delle chiamate HTTP dirette? La risposta è: **un message broker**.

Un **tight-coupling** (accoppiamento stretto) in un’architettura a microservizi si verifica quando due (o più) servizi dipendono l’uno dall’altro in modo così diretto e “rigido” da diventare vulnerabili e difficili da evolvere e manutenere. Nel caso delle chiamate HTTP sincrone, i principali fattori di tight-coupling sono:

1. **Conoscenza esplicita dell’endpoint**

Il client deve conoscere con esattezza l’URL completo del servizio—host, porta e percorso—e basta anche una piccola modifica a questi parametri per interrompere immediatamente la comunicazione.

1. **Contract (schema) condiviso e rigido**

Client e server devono mettersi d’accordo fin dall’inizio sulla versione e sul formato del payload, che sia JSON o XML, perché qualsiasi modifica breaking alle proprietà dei dati costringe a un aggiornamento sincronizzato su entrambi i lati.

1. **Disponibilità e latenza**

Il servizio chiamante resta sospeso in attesa di una risposta e, se il servizio remoto è lento o non disponibile, l’intera catena di richieste subisce ritardi o arriva a un timeout.

1. **Error propagation e cascading failures**

Un errore o un crash in un servizio “a monte” si propaga immediatamente ai suoi client che tentano la chiamata HTTP, causando possibili interruzioni a catena.

1. **Versioning complesso**

Per supportare nuove feature o cambi di API senza danneggiare i client esistenti, bisogna gestire la coesistenza di più versioni (v1, v2, ecc.), complicando il deployment e il routing.

1. **Deployment dipendente**

Spesso, un cambiamento lato server impone di aggiornare contestualmente il client, rendendo il rilascio indipendente dei microservizi molto più laborioso.

Immagine che contiene testo, Viso umano, clipart, vestiti

Il contenuto generato dall'IA potrebbe non essere corretto.Un message broker come **NATS** agisce da “centralino”: riceve messaggi dai servizi che li **pubblicano** (publisher), li smista su canali logici detti **subjects**, e li consegna ai servizi che vi si **sottoscrivono** (subscriber). Questo modello asincrono e basato su eventi porta numerosi vantaggi:

* **Resilienza**: se un servizio è offline, i messaggi restano in coda finché non torna operativo.
* **Scalabilità**: basta aggiungere nuovi subscriber a uno stesso subject per bilanciare il carico.
* **Decoupling**: publisher e subscriber ignorano l’identità reciproca, comunicano esclusivamente via **subjects**.

In più, accoppiato a un’architettura Zero Trust, elimina la necessità di fidarsi implicitamente di altri servizi: ogni chiamata a un servizio di autorizzazione diventa un evento sul bus, garantendo verifiche in tempo reale.

Nei prossimi paragrafi vedremo come funziona NATS “sotto il cofano”, quali sono le sue componenti chiave e – grazie a due esempi semplici ma efficaci – capiremo subito il potere della messaggistica asincrona.

## 2. Perché un message broker?

Immagina che il tuo notes-service, per ogni singola richiesta di accesso alle note, debba contattare l’auth-service via HTTP per validare il token: basta un piccolo drop di rete o un timeout perché l’intera catena di elaborazione rallenti drasticamente. Man mano che i microservizi aumentano, quella semplice interdipendenza si trasforma in un groviglio di dipendenze rigide, difficile da gestire ed estendere.

Immagine che contiene testo, Carattere, linea, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.Con un broker, il notes-service invia sul canale relativo al subject TokenValidationRequest quello che viene chiamato “payload del token”, ossia l’insieme di dati racchiusi nel JSON Web Token (per esempio l’ID dell’utente, il timestamp di scadenza e i ruoli associati).

L’auth-service, che resta in ascolto su quel canale, recupera il messaggio, esamina queste informazioni e ne verifica l’autenticità.

In caso di esito positivo, pubblica il risultato su TokenValidationResponse; il notes-service intercetta quindi la risposta e può proseguire con l’elaborazione della richiesta senza mai sapere in quale linguaggio sia scritto l’auth-service né in quale server risieda.

Per rendere l’idea, immagina un ufficio con due reparti separati: per ogni documento (il payload) che vuoi far visionare al reparto sicurezza, lo lasci nella casella “Controllo accessi”; una volta ispezionato, il reparto sicurezza deposita nella casella “Accesso consentito” un promemoria con l’esito. Tu, tornando alla tua scrivania, prendi quel promemoria ed entri senza dover conoscere chi fosse esattamente la persona che ti ha controllato o dov’è posta fisicamente la stanza del reparto sicurezza.

Questo sistema di “cassette postali” (o “bacheche”) rende ogni servizio indipendente e più resiliente agli imprevisti di rete.

### 2.1 comunicazione sincrona vs asincrona

Per capire perché un message broker possa fare la differenza, vale la pena soffermarsi un attimo sulla distinzione fra comunicazione sincrona e asincrona. Quando due servizi interagiscono in modo sincrono, uno invia una richiesta e resta in attesa finché l’altro non risponde: è un po’ come fare una telefonata, in cui chi chiama deve pazientare al telefono finché dall’altra parte qualcuno non risponde. Se la telefonata cade o l’altra persona è occupata, la conversazione si interrompe e bisogna ritentare. Ogni ritardo o interruzione rallenta tutto il flusso di lavoro.

Nella modalità asincrona, al contrario, il mittente lascia un messaggio e prosegue subito per la sua strada, senza attendere una risposta immediata: è come lasciare un promemoria sulla bacheca dell’azienda, sapendo che il destinatario lo leggerà quando sarà disponibile. Questo approccio riduce i tempi di blocco — perché nessuno resta in attesa — e rende l’intera catena più tollerante ai guasti: se qualcuno non è online al momento giusto, il messaggio resta in coda fino a quando non potrà essere elaborato.

Un broker come NATS incarna proprio questo principio: gestisce il passaggio di consegne fra servizi in modo completamente asincrono, senza che nessuno debba sapere in anticipo chi risponderà, dove si trova o quanto tempo impiegherà. In questo modo si ottiene un sistema più reattivo e flessibile, capace di scalare e di adattarsi ai picchi di carico o ai guasti temporanei senza bloccare l’intero flusso di lavoro.

### 2.2 I limiti delle chiamate HTTP dirette

Prima di tutto, vale la pena ricordare che HTTP è un protocollo di tipo request–response basato su un modello client-server: il client invia una richiesta (formata da un metodo come GET, POST, PUT o DELETE, un URL che identifica la risorsa e una serie di header che trasportano metadati) e il server restituisce una risposta composta da un codice di stato, header e un eventuale body contenente il payload. Essendo progettato per essere **stateless**, ogni chiamata HTTP è autonoma e non mantiene alcun contesto tra una richiesta e l’altra, il che lo rende semplice e molto diffuso, ma introduce anche delle complicazioni per la logica applicativa.

Proprio questa caratteristica “senza stato” obbliga gli sviluppatori a ripassare in ogni singola chiamata tutti i dati necessari (token di autenticazione, ID di tracciamento, dati di sessione, ecc.), rendendo più ingombrante la gestione di flussi di lavoro distribuiti e costringendo a implementare meccanismi aggiuntivi di propagazione del contesto.

In un’architettura basata su chiamate HTTP sincrone, ogni servizio diventa fortemente dipendente dalla disponibilità degli altri: basta un piccolo ritardo o un’interruzione di rete su un singolo endpoint per bloccare l’intero flusso di elaborazione, peggiorando i tempi di risposta e l’esperienza complessiva.

Service A (client) Service B (server)

----------------------- ----------------------------------

Chiamata HTTP POST /process-data ➜ Ricezione richiesta

└─ Se fallisce → errore HTTP 500

←─ Attende risposta

Parallelamente, l’obbligo di conoscere con precisione URL, porte e contratti delle API — unitamente alla necessità di gestire versioni multiple per garantire retrocompatibilità — trasforma ogni modifica in un’operazione rischiosa e laboriosa, che richiede aggiornamenti coordinati su client e server. Infine, quando un servizio genera un errore o va offline, l’errore si propaga a catena lungo tutte le chiamate sincronizzate, creando colli di bottiglia difficili da isolare e risolvere.

Queste criticità — difficoltà nel mantenere lo stato, disponibilità, rigidità negli aggiornamenti e propagazione degli errori — riducono drasticamente scalabilità, manutenibilità e resilienza del sistema, spingendo verso modelli basati su comunicazione asincrona tramite message broker.

### 2.3 Vantaggi di resilienza, scalabilità e decoupling

L’adozione di un message broker come NATS trasforma radicalmente il modo in cui i microservizi interagiscono, offrendo benefici che diventano decisivi man mano che il sistema cresce in complessità e traffico.

Resilienza

In un modello asincrono, ogni messaggio viene consegnato attraverso il broker anziché direttamente da un servizio all’altro. Questo significa che, se un consumer è offline o sta subendo un picco di carico, i messaggi restano temporaneamente in coda nel broker anziché andare persi o generare errori immediati. Al ripristino del servizio o al ridimensionamento di nuove istanze, tutte le notifiche pendenti vengono consumate, garantendo che nessun evento vada perduto e che l’intero flusso di lavoro possa riprendere senza interventi manuali o complicate procedure di retry.

Scalabilità

Con NATS diventa banale bilanciare il carico di lavoro: basta aggiungere nuove istanze di subscriber in uno stesso “queue group” per spartirsi equamente i messaggi in arrivo. Non è necessario riprogettare il codice né gestire manualmente un layer di bilanciamento esterno: il broker si occupa di instradare ogni messaggio all’istanza libera successiva, permettendoti di reagire in tempo reale all’incremento del traffico semplicemente replicando il servizio.

Decoupling

Forse il vantaggio più eloquente di un message broker è il disaccoppiamento totale tra produttori e consumatori. I servizi non si chiamano più fra loro in modo sincrono ma scambiano informazioni via eventi, ignorando posizione, linguaggio di implementazione e contratto di rete degli interlocutori. Questo isolamento riduce drasticamente gli effetti collaterali: puoi sostituire, aggiornare o ridistribuire un servizio senza impattare direttamente gli altri, purché resti invariato il subject e il formato del messaggio. Il risultato è un ecosistema di microservizi molto più agile, in cui ogni componente evolve in autonomia e l’intero sistema resta coerente e robusto anche sotto cambiamenti continui.

## 3 Esempi esplicativi “a bassa tecnologia”

### 3.1 Il sistema di caselle postali in ufficio

Immagina un grande ufficio in cui ogni reparto – Amministrazione, Commerciale, Acquisti, Legale – dispone della propria cassetta postale affissa al muro. Quando il reparto Vendite ha bisogno di un’approvazione dalla Finanza, invece di chiamare o inviare una mail diretta, deposita il documento nella cassetta “Finanza”. I colleghi del reparto Finanza controllano periodicamente la loro cassetta, prelevano il documento e lo elaborano, lasciando poi il risultato nella cassetta “Vendite” o in un’ulteriore cassetta “Approvazioni”. In questo modo nessuno deve restare in attesa una risposta immediata, i reparti rimangono indipendenti e il flusso documentale è ordinato e tracciato, proprio come in un sistema a message broker.

### 3.2 La bacheca in mensa universitaria

Ora pensa a una mensa universitaria dotata di una grande bacheca in corridoio. Gli studenti possono affiggere bigliettini di ogni tipo: offerte di stanze in affitto, richieste di coinquilini, annunci di gruppi di studio o inviti per una partita di calcetto. Chiunque, passando in mensa, può leggere i post che gli interessano quando ha tempo, senza coordinare orari o inviare messaggi uno-a-uno. Se un annuncio non viene più “letto”, basta toglierlo o lasciarlo scadere, senza bloccare nessuno. Questa bacheca funge da “bus” centralizzato, esattamente come NATS: un punto unico dove tutti pubblicano e da cui tutti si sottoscrivono, in modo asincrono e scalabile.

## 4. Cos’è NATS

NATS è un message broker ultraleggero pensato per consentire a componenti distribuiti — come microservizi, dispositivi IoT o applicazioni cloud-native — di comunicare in modo asincrono, scalabile e a bassa latenza.

Invece di mantenere code persistenti o complessi stati di sessione, il core di NATS adotta un modello “*fire-and-forget*” in cui ogni messaggio viene recapitato al massimo una volta (at-most-once): se un subscriber non è connesso o non riesce a elaborare tempestivamente il messaggio, questo può essere scartato, ma il broker non rallenta mai per inseguire un singolo client. Questo design minimalista permette a NATS di raggiungere throughput di milioni di messaggi al secondo, offrendo un’architettura estremamente reattiva.

Per rendere il sistema sempre disponibile, NATS si comporta come un “*always available dial tone*”: in caso di guasti o connessioni lente, il server preferisce scollegare il client problematico piuttosto che bloccare l’intero servizio, e i subscriber si riconnettono automaticamente ad altri nodi del cluster. L’effetto pratico è un’infrastruttura di messaggistica che sacrifica la consegna garantita di ogni singolo messaggio per favorire la velocità e la disponibilità complessiva.

**Spiegazione di “rinunciando a garanzie di consegna ‘esattamente una volta’”:**

Quando si parla di consegna “*esattamente una volta*”, si intende la capacità di un broker di assicurare che ogni messaggio inviato venga recapitato *una e una sola volta* ai destinatari, né perso né duplicato. Questi passaggi introducono latenza e richiedono un controllo di stato sofisticato, che contrasta con l’obiettivo di NATS di offrire un’infrastruttura ultraleggera. Per mantenere throughput e disponibilità ai massimi livelli, il **core** di NATS opta invece per il modello **“at-most-once”**, accettando che alcuni messaggi possano non arrivare se il destinatario è temporaneamente offline o sovraccarico. Se le applicazioni richiedono però garanzie più solide, ci si appoggia a **JetStream**, che estende NATS con la persistenza distribuita e le conferme di consegna (“at-least-once” o persino “exactly-once” tramite deduplicazione avanzata), senza sacrificare la filosofia “simple & performant” del broker di base.

### 4.1 Storia e filosofia “Simple & Performant”

NATS nasce nel 2010 come progetto open source per la piattaforma Cloud Foundry, con un’implementazione iniziale in Ruby. La vera svolta avviene nel 2012, quando Derek Collison riscrive il server in Go, ottenendo un’esecuzione ultraleggera e ottimizzata: tecniche di buffering intelligente, hashing veloce e parsing “zero-allocation” permettono di superare i 2 Mmsg/s già nelle prime release in Go, salendo in seguito fino a 6 Mmsg/s e oltre.

Il mantra di NATS è **“***simple & performant***”**: il broker è un singolo binario senza dipendenze esterne, utilizza un protocollo di testo facile da implementare e mantiene uno stato cliente estremamente transitorio. La scelta del modello *at-most-once* dimostra come NATS rinunci esplicitamente alle garanzie di consegna “esattamente una volta”: per fornire quest’ultima – garantire cioè che ogni messaggio arrivi né più né meno di una volta – servono meccanismi di persistenza, checkpointing e deduplicazione coordinata tra mittente, broker e destinatario, con un notevole overhead in termini di latenza e complessità.

NATS preferisce quindi offrire un piano di base rapidissimo e lasciare a chi sviluppa l’applicazione la possibilità di introdurre un livello di affidabilità maggiore, ad esempio con **JetStream**, il modulo che aggiunge persistenza distribuita, replay degli stream e pattern più robusti.

### 4.2 Architettura di base (server, cluster, leafnodes)

Il cuore di NATS è il processo nats-server, che accetta connessioni TCP dai client su un singolo nodo. Per aumentare l’affidabilità, più istanze di nats-server vengono configurate in *cluster full-mesh*: ciascun nodo stabilisce connessioni di routing verso tutti gli altri, costruendo un grafo a un solo hop. In questo modo, un client connesso a uno qualsiasi dei nodi può inviare messaggi che verranno instradati ai subscriber interessati, anche se non collegati allo stesso server.

Oltre al cluster, NATS supporta il concetto di *leaf nodes* per estendere l’infrastruttura in scenari geo-distribuiti o isolati: ogni leaf node si collega a uno o più server del cluster principale, offrendo ai client locali un punto di accesso più vicino e continuando a funzionare anche in caso di temporanea disconnessione dal cluster. Quando la connessione viene ristabilita, il leaf node replica lo stato mancante e si riallinea automaticamente. Questa architettura tiered — server singolo, cluster e leaf nodes — garantisce sia la massima disponibilità che un bilanciamento del carico geografico, mantenendo al contempo la semplicità operativa del broker.

### 4.3 Subjects, Channels e Wildcards

La comunicazione in NATS è organizzata in *subjects*, ovvero stringhe separate da punti che fungono da nomi di canale logici (ad esempio orders.created o users.>) sui quali i publisher inoltrano messaggi e i subscriber si registrano per riceverli.

Per flessibilità, NATS introduce due wildcard:

* Il *token wildcard* \* sostituisce un solo livello di soggetto, per cui orders.\*.created corrisponde a orders.online.created o orders.instore.created.
* Il *full wildcard* > cattura tutti i livelli restanti: orders.> intercetta orders, orders.created, orders.created.email, e così via.

Grazie ai wildcard, è possibile realizzare pattern di Subscription molto potenti: ad esempio, un servizio di auditing può registrarsi con SUB > per ascoltare ogni singolo messaggio transitato sul bus, mentre un worker dedicato può usare SUB tasks.\* per elaborare qualsiasi attività di task scheduling. Inoltre, il meccanismo di *queue subscription* (pub/sub con bilanciamento) permette a più istanze di uno stesso servizio di formare un gruppo, ricevendo ciascuna un sottoinsieme dei messaggi in modo casuale, ideale per distribuire il carico di lavoro senza duplicare l’elaborazione.

## 5 Componenti chiave

In un’architettura basata su NATS, quattro elementi giocano un ruolo centrale: i publisher, i subscriber, il NATS Server e il subject (il “canale” logico). Di seguito li esploreremo facendo riferimento a esempi minimali.

### 5.1 Publisher

Il **publisher** è il servizio o il componente che **invia** messaggi nel sistema, scegliendo un *subject* sul quale pubblicare. In termini pratici, un publisher stabilisce una connessione TCP al NATS Server e invia comandi PUB secondo il protocollo testuale:

PUB amici.cli 5

Hello

In questo esempio, il publisher invia 5 byte di payload sul subject amici.cli. Non c’è alcun hand-shake o conferma di consegna: appena il comando PUB arriva al server, questo risponde con un semplice +OK (che perlopiù i client disabilitano, perché non fornisce garanzie reali di recapito) e il publisher può continuare senza alcuna attesa aggiuntiva.

### 5.2 Subscriber

Il **subscriber** è l’entità che **ascolta** uno (o più) subject e reagisce ai messaggi pubblicati. Per registrarsi, invia un comando SUB:

SUB amici.cli 1

+OK

Da quel momento, ogni messaggio su amici.cli genera dal server una riga:

MSG amici.cli1 5

Hello

dove compaiono il subject (amici.cli), l’ID della subscription (1) e la lunghezza del payload in byte (5), seguiti dal contenuto vero e proprio.

### 5.3 NATS Server

Il **NATS Server** (processo nats-server o gnatsd) è il “centralino” che instrada tutti i messaggi tra publisher e subscriber:

1. **Singolo nodo**: gestisce decine di migliaia di connessioni e può processare milioni di messaggi al secondo.
2. **Clustering full-mesh**: più server si collegano fra loro, formando una rete in cui ogni nodo comunica direttamente con tutti gli altri. In questo modo, basta che un client sia connesso a uno qualunque dei nodi per poter pubblicare o sottoscrivere messaggi a prescindere dal punto di ingresso.
3. **Leaf nodes:** in scenari geo-distribuiti, un leaf node si collega a uno (o più) cluster principali, offrendo un punto di accesso locale e riallineandosi automaticamente quando la connessione torna disponibile.

### 5.4 Subject: il “canale” logico

Un **subject** in NATS è una stringa gerarchica (livelli separati da punti), che rappresenta il canale virtuale su cui transitano i messaggi. Ad esempio, users.created o payments.processed.us.

* Il subject è **solo testo**, non serve alcuna configurazione a server: basta usarlo.
* **Wildcard**
  + \* sostituisce un singolo livello (foo.\*.bar → foo.hello.bar).
  + > cattura tutti i livelli rimanenti (foo.> → foo.x, foo.x.y.z, ecc.).
* **Pattern dinamici**: un subscriber può registrarsi su SUB > per “ascoltare” ogni messaggio che transita sul broker, utile per logging centralizzato o operazioni di auditing.

In questo modo il modello Pub/Sub diventa estremamente flessibile: non si tratta di code fisiche come in altri sistemi, ma di **routing dinamico** basato sul nome del subject, che permette di introdurre nuovi servizi o modificare quelli esistenti senza interventi sul broker.

Con questi quattro elementi — **publisher**, **subscriber**, **NATS Server** e **subject** — si costruisce la spina dorsale di un sistema event-driven, dove i servizi comunicano in modo **decoupled**, **scalabile** e **performante**.

## 5. Primi passi con Rust e NATS

### 5.1 Esempio minimale

architettura di comunicazione tramite message broker NATS


Verifica su GitHub [**nats-docker-demo**](https://github.com/tecnodiario/vol2/tree/master/nats-docker-demo) un ambiente minimale per sperimentare NATS con un **publisher** Rust, un **subscriber** Rust e un client interattivo (natsio/nats-box).

## **Coding**: struttura del progetto e file di base

AB\_app/

├── docker-compose.yml # orchestrazione NATS + servizi

├── infra/ # provisioning e configurazioni infra

│ └── nats/ # config broker NATS (TLS, JetStream)

├── services/ # backend Rust microservices

│ ├── auth-service/ # validazione token via messaggistica

│ │ ├── src/

│ │ │ ├── handlers/ # entrypoint API HTTP e subscriber NATS

│ │ │ ├── models/ # DTO per TokenValidationRequest/Response

│ │ │ └── utils/ # parsing JWT, error handling, logging

│ │ ├── Cargo.toml # dipendenze e metadata Rust

│ │ └── Dockerfile # build dell’immagine container

│ └── notes-service/ # gestione note e autorizzazione

│ ├── src/

│ │ ├── handlers/ # endpoint /notes e publisher NATS

│ │ ├── models/ # entity Note, UserPermissions

│ │ └── utils/ # formattazione output, validazioni

│ ├── Cargo.toml

│ └── Dockerfile

├── frontend/ # client Flutter multi-platform

│ └── flutter\_app/ # progetto Flutter (lib, config, assets)

│ ├── lib/

│ │ ├── main.dart # entrypoint: MaterialApp / WidgetsFlutterBinding

│ │ ├── src/

│ │ │ ├── ui/ # widget, temi, stili condivisi

│ │ │ ├── navigation/ # gestione rotte e Navigator 2.0

│ │ │ ├── services/ # client NATS (grpc/JS bridge) e HTTP API

│ │ │ ├── models/ # classi per Note, UserData, errori

│ │ │ └── utils/ # helper (formattazioni, validazioni input)

│ ├── assets/ # icone, immagini, font

│ ├── web/ # configurazioni build e index.html per web

│ ├── windows/ # codice nativo desktop Windows

│ ├── macos/ # codice nativo desktop macOS

│ ├── linux/ # codice nativo desktop Linux

│ ├── pubspec.yaml # dipendenze Flutter e asset declaration

│ └── analysis\_options.yaml # regole di lint e best practice Dart

└── shared/ # moduli condivisi tra servizi

└── jwt-utils/ # libreria Rust per generazione e verifica JWT

├── src/

│ ├── lib.rs # entrypoint crate

│ └── claims.rs # definizione claim standard e custom

└── Cargo.toml