Università Politecnica delle Marche

Analizzatore di flussi musicali

Progetto di Programmazione Avanzata

Autore: **Lionel Djouaka Kelefack**
Docente: **Adriano Mancini**

19 novembre 2025

Indice

Capitolo 1

Introduzione

1.1 Scopo del progetto

Lo scopo di questo progetto è la realizzazione di un sistema denominato **Analizzatore di flussi musicali**, in grado di elaborare flussi di dati in formato JSON che simulano utenti ad ascoltare brani musicali. L'applicazione calcola in tempo reale diverse statistiche, come l'artista più ascoltato, la durata media di ascolto e i trend giornalieri di utilizzo e l'orario di punta. Il progetto ha l'obiettivo di integrare concetti avanzati di programmazione asincrona ed event-driven, facendo uso di pattern architetturali e strumenti professionali comunemente adottati nello sviluppo di sistemi distribuiti.

1.2 Contesto applicativo

Negli ultimi anni, le piattaforme di streaming musicale (ad esempio Spotify, Apple Music, Deezer) generano enormi quantità di dati legati all'attività degli utenti. Queste informazioni, se opportunamente analizzate, permettono di estrarre conoscenze utili per il marketing, la personalizzazione dei contenuti e la gestione dei diritti d'autore. L'obiettivo del progetto è riprodurre in modo semplificato questo scenario, simulando l'arrivo continuo di eventi di ascolto e applicando tecniche di elaborazione dei flussi per ottenere statistiche aggiornate in tempo reale.

1.3 Obiettivi principali

Il sistema si propone di raggiungere i seguenti obiettivi:

- Simulare un flusso continuo di eventi musicali (ascolti, durata, artista, utente);
- Elaborare i dati tramite pipeline asincrone, applicando strategie di filtraggio e aggregazione;

- Calcolare e aggiornare statistiche aggregate in tempo reale;
- Esporre i risultati attraverso un'interfaccia api REST;
- Garantire una struttura modulare e facilmente estendibile grazie all'uso di design pattern appropriati;
- Documentare e testare l'intero progetto tramite strumenti di sviluppo professionali.

1.4 Tecnologie utilizzate

Il progetto è stato sviluppato utilizzando un insieme di tecnologie moderne e open source:

- **Node.js** e **TypeScript** per la logica backend e la gestione asincrona degli eventi;
- **RxJS** per la costruzione di pipeline event-driven basate su stream reattivi;
- **Sequelize ORM** per l'interazione con il database relazionale;
- **Docker** e **Docker Compose** per la containerizzazione e l'esecuzione isolata dei servizi;
- **Jest** per i test unitari e d'integrazione;
- **Postman** per la validazione e la documentazione delle api;
- **GitHub** per il versionamento e la consegna del progetto.

1.5 Requisiti funzionali e non funzionali

Requisiti funzionali

- Il sistema deve ricevere e gestire flussi di eventi in formato JSON;
- Deve calcolare e aggiornare periodicamente le statistiche principali (artista più ascoltato, durata media, trend giornalieri e orario di punta);
- Deve consentire l'accesso ai dati elaborati tramite endpoint REST;
- Deve fornire risposte in formato JSON con tempi di latenza ridotti;
- Deve registrare gli eventi e le statistiche in un database relazionale (postgres).

Requisiti non funzionali

- **Scalabilità**: il sistema deve poter gestire un numero crescente di eventi;
- **Affidabilità**: deve garantire la coerenza dei dati anche in caso di errori o ritardi nei flussi;
- **Modularità**: ogni componente (stream, repository, API) deve essere indipendente e sostituibile;
- **Portabilità**: il sistema deve poter essere eseguito su qualsiasi piattaforma Docker;
- **Manutenibilità**: il codice deve essere documentato, testato e facilmente estendibile.

Capitolo 2

Analisi del Problema

Questo capitolo definisce il perimetro informativo del sistema, le caratteristiche dei dati in ingresso (flussi JSON), il flusso logico di trasformazione dall'ingestione alla produzione di statistiche e le metriche richieste dal progetto. L'obiettivo è fornire una base concettuale chiara per la progettazione dell'architettura e delle API.

2.1 Descrizione del dominio

Nel dominio dello *streaming musicale*, gli utenti interagiscono con una piattaforma che consente la riproduzione di brani (*tracks*) tratti da un catalogo. Ogni interazione genera uno o più eventi di ascolto: avvio, pausa, ripresa, stop, avanzamento, completamento. Questi eventi, emessi come messaggi JSON in tempo quasi reale, costituiscono la sorgente informativa del sistema.

Gli attori principali sono:

- **Utente**: soggetto che riproduce brani; è identificato da un `userId` e può avere attributi contestuali (es. paese, dispositivo, età).
- **Brano (Track)**: elemento del catalogo, con `trackId`, titolo, artista, album, genere.
- **Evento di ascolto**: occorrenza temporale che descrive lo stato dell'ascolto (`tempo`, `durata`) con riferimenti a utente e brano.

Il sistema deve aggregare tali eventi per calcolare *statistiche in tempo reale* e *trend giornalieri*, mantenendo coerenza rispetto a latenze di arrivo, duplicati e possibili disallineamenti temporali.

2.2 Tipologia dei dati (flussi JSON)

I dati arrivano come **flusso di eventi JSON** (*event stream*) a cadenza regolare/irregolare. Ogni record rappresenta uno stato dell'ascolto con le seguenti proprietà minime:

```
{
  "id": "1";        -> Identificatore dell'evento
  "userId": "u1",
  "trackId": "t001",
  "artist": "Ultimo",
  "duration": 832,
  "timestamp": "2025-01-01T00:00:00Z",
  "genre": "Dance",
  "country": "IT",
  "device": "mobile"
}
```

Caratteristiche salienti:

- **Temporalità**: presenza obbligatoria di `timestamp` in UTC (ISO 8601).
- **Identificativi**: `eventId` univoco (anti-duplicazione), `userId`, `trackId`.
- **Contesto opzionale**: `device`, `contry`, `genre` a supporto di filtri futuri.
- **Qualità dati**: possibili *out-of-order*, duplicati, eventi malformati; è necessaria validazione e deduplicazione.

La Tabella 2.1 riassume i campi principali.

Tabella 2.1: Campi principali dell'evento di ascolto

Campo	Tipo	Descrizione
id	number	Identificativo univoco dell'evento.
userId	string	Identificativo utente.
trackId	string	Identificativo brano.
artist	string	Nome artista associato al brano.
duration	number	Album di appartenenza (opzionale).
genre	string	Genere musicale (opzionale).
country	string	paese di ascolto (opzionale).
device	string	Dispositivo sorgente (opzionale).
timestamp	datetime	Istante di generazione dell'evento.

2.3 Flusso logico delle informazioni

Il flusso informativo dall'ingestione alla *serving layer* delle statistiche prevede le seguenti fasi:

1. **Ingestione**: acquisizione degli eventi JSON dallo stream simulato.

2. **Validazione & Normalizzazione**: controllo schema, parsing timestamp, coercizione tipi, scarto o quarantena record malformati.

3. **Arricchimento**: join con metadati di catalogo (artista, genere) se mancanti o non affidabili; armonizzazione dei nomi artista.

4. **Deduplicazione**: rimozione di duplicati tramite `id` e finestra temporale di tolleranza.

5. **Aggregazione in finestra**: calcolo incrementale su finestre temporali (ad es. *tumbling* 1 min, *sliding* 5 min) e *daily rollup*.

6. **Persistenza**: salvataggio di snapshot/indicatori per letture rapide via API.

7. **Esposizione**: pubblicazione delle statistiche tramite endpoint REST.

Per chiarezza, la **??** rappresenta il flusso (se disponibile un diagramma, inserirlo come immagine).

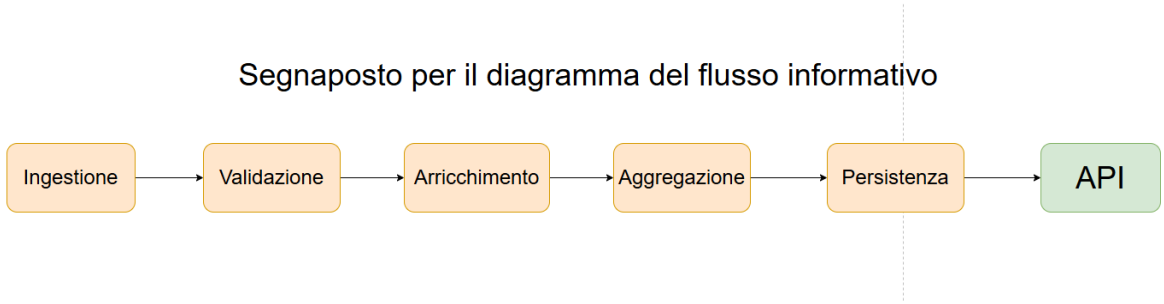

Figura 2.1: Flusso logico delle informazioni

Capitolo 3

Architettura del Sistema

Questo capitolo descrive l'architettura logica e fisica del sistema *Analizzatore di flussi musicali*, mettendo in evidenza i componenti principali, i pattern di progettazione adottati, i principi di programmazione asincrona ed event-driven e la containerizzazione con Docker. L'architettura è stata pensata per supportare flussi di eventi in ingresso, elaborazioni in tempo quasi-reale e pubblicazione di statistiche tramite API REST, mantenendo al contempo separazione delle responsabilità, testabilità e portabilità tra ambienti.

3.1 Visione generale

A livello macro, il sistema è composto da cinque blocchi funzionali:

1. **Producer di eventi (Simulatore)**: genera flussi di eventi d'ascolto in formato JSON (utente, brano, timestamp, durata, stato).
2. **Pipeline di elaborazione** (*normalizzazione* → *arricchimento* → *aggregazione*): applica trasformazioni e calcola metriche su finestre temporali.
3. **Servizi di Dominio** (pattern Strategy/Observer): incapsulano regole di calcolo e aggiornamento delle statistiche.
4. **Persistenza** (ORM): salva eventi e snapshot di statistiche in DB relazionale.
5. **API REST**: espone le statistiche in tempo reale e storiche a client esterni (strumenti di test, dashboard future).

La Figura 3.1 mostra la visione d'insieme (diagramma dei componenti).

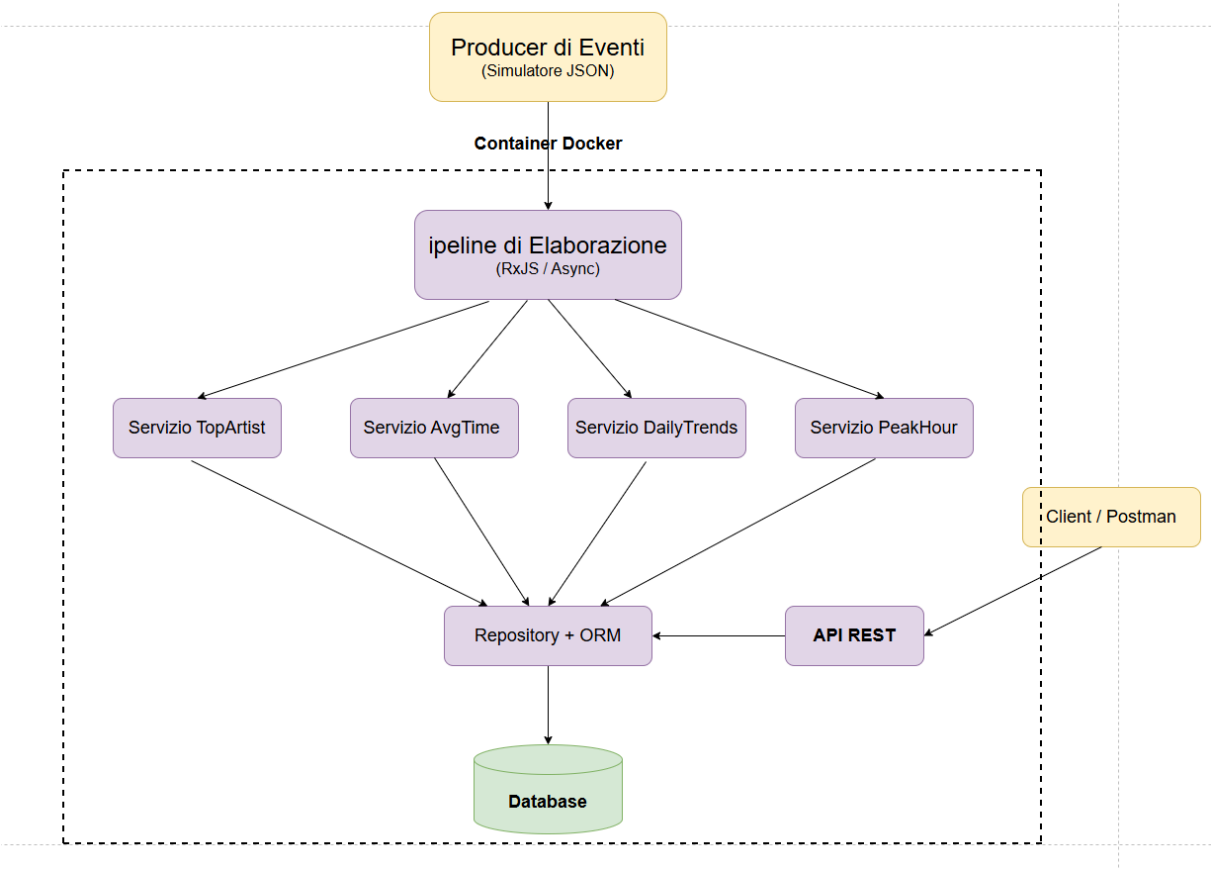

Figura 3.1: Diagramma compatto dei componenti principali del sistema *Analizzatore di flussi musicali*.

Criteri architetturali Le scelte sono guidate da: *coesione elevata* nei moduli, *accoppiamento debole* tra livelli, *estendibilità* delle strategie di calcolo, *osservabilità* (logging/-metriche) e *portabilità* (Docker).

3.2 Componenti principali

Producer di eventi

Genera eventi d'ascolto (stream JSON). Parametrizzabile per frequenza, distribuzione per artista/genere e durata di ascolto. Non è una fonte "reale", ma replica caratteristiche statistiche utili ai fini didattici.

Pipeline di elaborazione

Catena di operatori che:

- **Normalizzano** e validano gli eventi (filtri su campi mancanti o formati errati).
- **Arricchiscono** gli eventi (join con catalogo artisti).

- **Aggregano** su finestre (sliding/tumbling) per calcolare top artista, durata media, trend giornalieri e ore di punta.

Servizi di dominio

Contengono la logica delle metriche. Ogni servizio implementa un'interfaccia coerente (*calcola*, *aggiorna*...) e può essere sostituito o esteso tramite *Strategy* (cfr. §3.3).

Layer di persistenza (Repository/ORM)

I *Repository* astraggono l'accesso a DB e incapsulano le query. L'ORM (cfr. Cap. 4) supporta migrazioni, validazioni e mapping tra entità e tabelle.

API REST

Espongono gli endpoint per interrogare le statistiche (*top artist*, *avg listening time*, *daily trends*) e *peak hours*. Sono stateless e idempotenti per le operazioni di lettura.

Tabella 3.1: Componenti principali e responsabilità

Componente	**Responsabilità chiave**
Producer eventi	Generazione flussi, configurazione carico, formati JSON coerenti
Pipeline Rx	Validazione, arricchimento, aggregazioni su finestre, gestione back-pressure
Servizi di dominio	Regole di business, strategie di calcolo, aggiornamento osservatori
Repository/ORM	CRUD eventi/statistiche, indicizzazione, transazioni
API REST	Esposizione metriche, paginazione/filtri, gestione errori e codici di stato

3.3 Pattern di progettazione adottati

L'architettura applica tre pattern classici per mantenere separazione dei ruoli, estendibilità e testabilità.

Strategy

Problema: l'algoritmo di calcolo delle metriche può variare (es. media aritmetica vs. ponderata, diverse dimensioni di finestra).
Soluzione: incapsulare gli algoritmi in *strategie* intercambiabili che rispettano la stessa interfaccia.

Uso nel progetto: selezione runtime della strategia di aggregazione (configurazione o parametro di richiesta).
Benefici: estensione senza modificare il client; test indipendenti di ciascuna strategia.
Rischi: proliferazione di strategie simili; mitigare con linee guida e riuso di componenti comuni.

Observer

Problema: notificare più interessati (es. calcoli di metriche diverse) all'arrivo di nuovi dati.
Soluzione: gli *osservatori* si iscrivono allo stream; ogni evento/finestra emessa scatena *update* sui sottoscrittori.
Uso nel progetto: i servizi per *TopArtist*, *AvgTime*, *DailyTrends...* sono osservatori della pipeline.
Benefici: disaccoppiamento tra sorgente e consumatori; aggiunta di metriche senza toccare il producer.
Rischi: gestione di concorrenza e ordine; mitigare con scheduler/queue e regole di back-pressure.

Repository

Problema: evitare dipendenze dirette dal livello dominio verso l'ORM/DB.
Soluzione: un layer *Repository* fornisce metodi intenzionali (es. *salvaSnapshotTrend(giorno, dati)*).
Uso nel progetto: repository distinti per eventi, brani, statistiche.
Benefici: testabilità (mock), coerenza delle query, singolo punto di ottimizzazione.
Rischi: eccesso di astrazione; mitigare con interfacce essenziali e regole di naming.

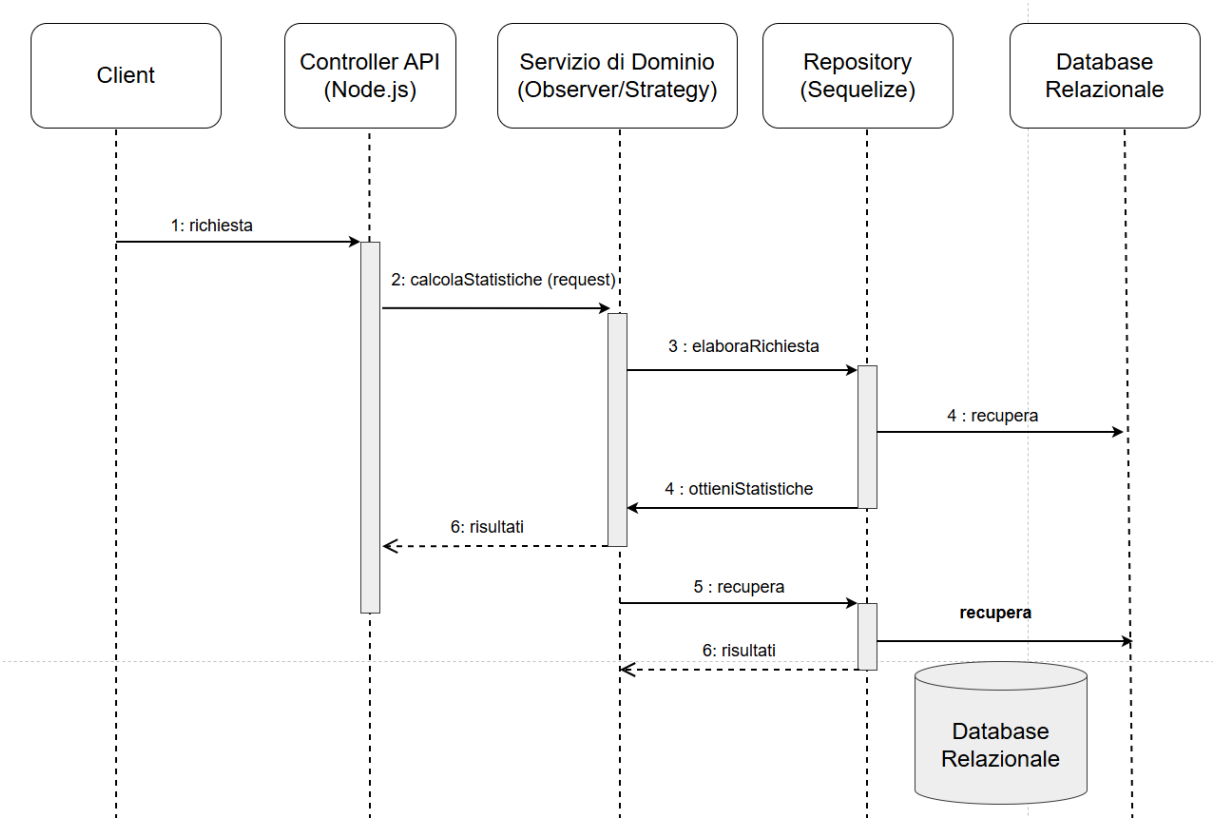

Figura 3.2: Diagramma di sequenze

3.4 Programmazione asincrona ed event-driven

L'elaborazione è modellata come *flusso di eventi* gestito da operatori reattivi. Principi chiave:

- **Asincronia non bloccante**: uso di primitive reattive e *scheduler* per isolare le sezioni I/O-bound (persistenza) e CPU-bound (aggregazioni).
- **Operatori di trasformazione**: `map`, `filter`, `reduce`, per aggregazioni su finestre temporali.
- **Gestione errori**: incanalare eccezioni su canali dedicati; log strutturati con contesto (id evento, artista, timestamp).
- **Idempotenza**: le operazioni di scrittura su snapshot mantengono chiavi naturali (finestra/metriche) per prevenire duplicati.

Finestre temporali Per le metriche si utilizzano due categorie:

1. **Finestre brevi** (es. 1–5 minuti) per il *quasi-real-time*.
2. **Finestre giornaliere** per trend e confronti inter-day.

La scelta tra *tumbling* (non sovrapposte) e *sliding* (sovrapposte) dipende dalla granularità richiesta e dal carico sostenibile.

Osservabilità Metriche operative: throughput eventi/min, latenza media pipeline, tassi di errore, tempi risposta API. Questi indicatori guidano tuning e dimensionamento.

3.5 Architettura a container (Docker)

La distribuzione locale e la riproducibilità sono garantite tramite container. L'architettura minima prevede:

- **Servizio API**: applicazione Node.js/TypeScript.
- **Database**: DB relazionale (es. PostgreSQL).
- **Strumento admin (opz.)**: Adminer/pgAdmin per ispezione tabelle.

Rete e volumi I servizi condividono una rete interna dedicata; il DB monta un *volume* per la persistenza dei dati e delle migrazioni.

Configurazione Variabili d'ambiente minime: `DB_HOST`, `DB_PORT`, `DB_NAME`, `DB_USER`, `DB_PASSWORD`, `PORT`, `NODE_ENV`. Healthcheck applicativo per verificare:

1. raggiungibilità del DB;
2. disponibilità endpoint `/health`;
3. latenza entro soglia.

Ciclo di vita

1. **Build** immagini applicative.
2. **Up** dello stack (API, DB, admin).
3. **Migrazioni/seed** tramite ORM.
4. **Esecuzione** del simulatore e della pipeline.
5. **Test** (unit/integration) e invocazioni Postman.

Portabilità La containerizzazione consente di replicare l'ambiente su diverse macchine/OS in modo deterministico, riducendo le *configuration drift* e facilitando la valutazione.

Tabella 3.2: Servizi container e responsabilità

Servizio	**Responsabilità**
api	Espone API REST, orchestration pipeline, logging/metriche
db	Storage eventi/statistiche, indici e vincoli
admin (opz.)	Ispezione e manutenzione (query, verifiche)

Capitolo 4

Modello dei Dati

Questo capitolo definisce il modello informativo del sistema *Analizzatore di flussi musicali*, con particolare attenzione alle entità principali, alle relazioni e ai vincoli di integrità, nonché agli aspetti di configurazione ORM (Sequelize) coerenti con un'implementazione in TypeScript. Tutte le chiavi primarie sono intere auto-incrementali.

4.1 Schema concettuale (ER Diagram)

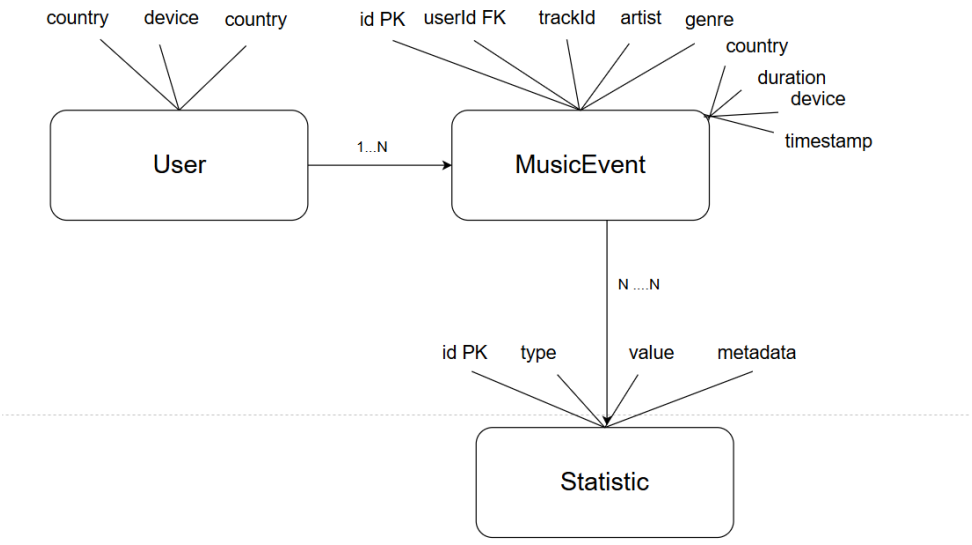

Figura 4.1: Schema ER compatto ad alto livello. Le tabelle statistiche derivano dagli eventi di ascolto.

Lo schema concettuale modella quattro nuclei informativi:

1. **Utente**: profilo minimo dell'ascoltatore.
2. **Evento di ascolto**: stream di eventi che legano un utente ad un brano in un certo istante, con durata.

3. **Statistiche**: snapshot e aggregati (top artista, durata media, trend giornalieri) calcolati su finestre temporali.

4.2 Diagrammi UML

In questa sezione vengono presentati i principali diagrammi UML utilizzati per descrivere il comportamento e la struttura del sistema *Analizzatore di flussi musicali*. In particolare, si riportano:

- eventuali diagrammi di sequenza, che illustrano il flusso dei messaggi per scenari significativi.
- Per una buona letture del diagrama UML, è disponibile nel documento "diagrams.pdf")

4.2.1 Diagramma delle classi

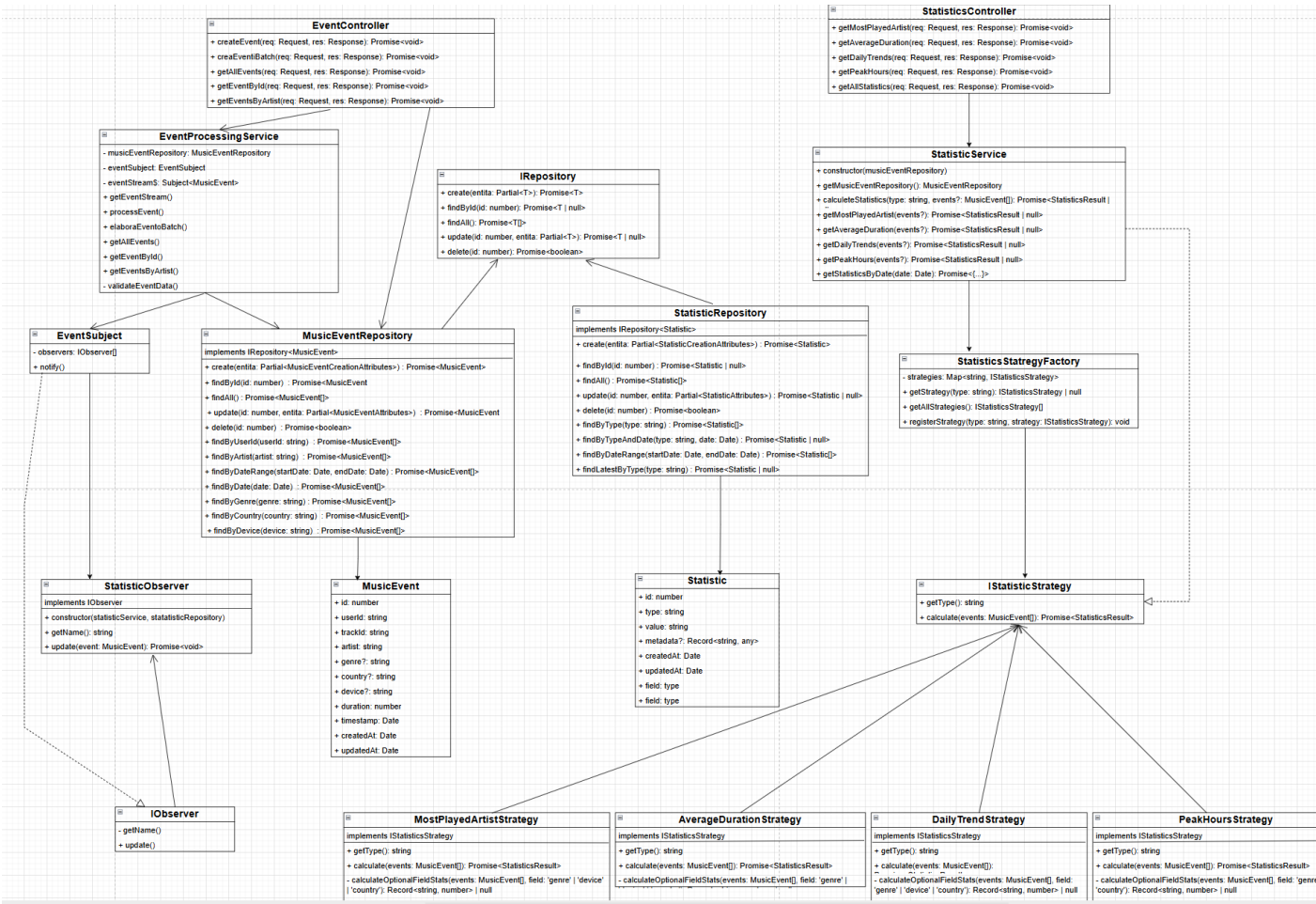

Figura 4.2: Diagramma delle classi ad alto livello del dominio applicativo.

Il diagramma delle classi descrive la struttura statica del dominio, mettendo in relazione:

- le entità principali (*Utente*, *EventoAscolto*, *Statistica*, ecc.);
- le associazioni tra gli oggetti del dominio e i componenti applicativi;
- eventuali interfacce e classi astratte utilizzate per implementare i design pattern.

4.2.2 Diagramma di sequenza

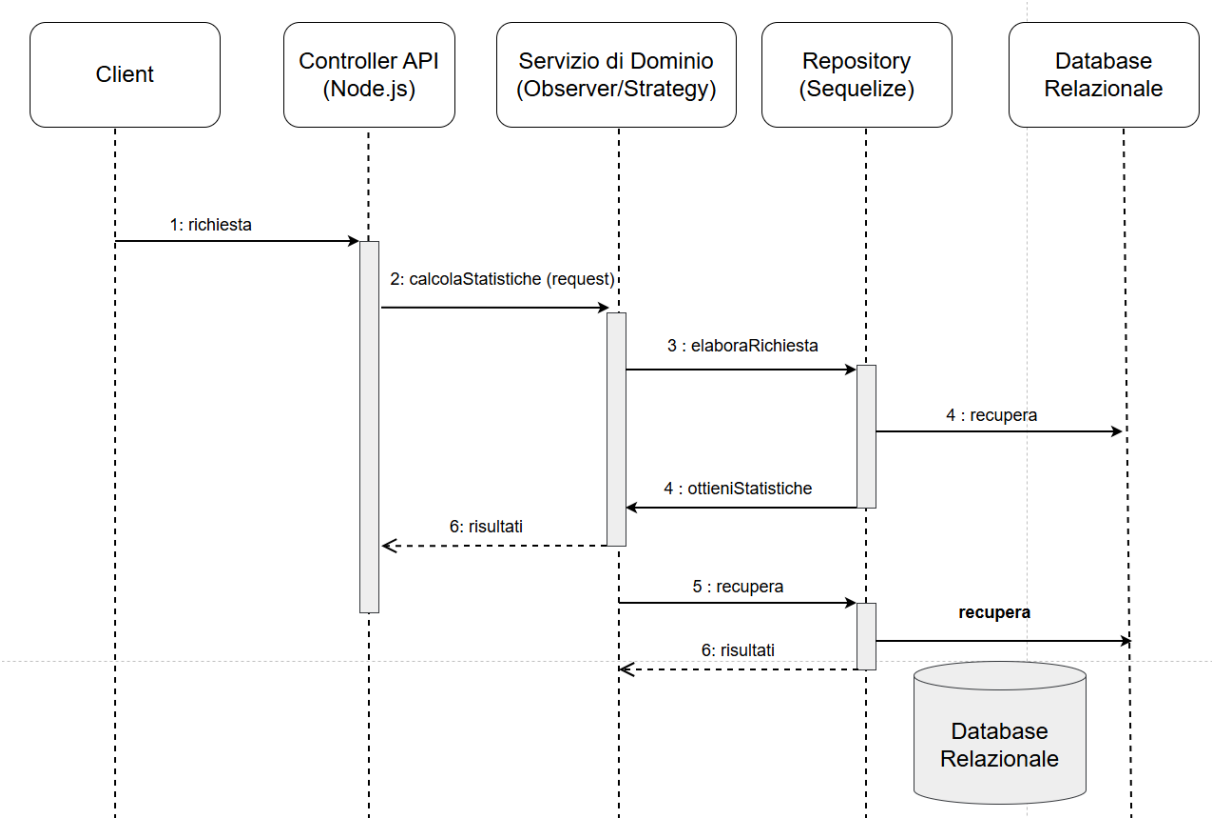

Figura 4.3: Diagramma di sequenza per la richiesta di statistiche da parte di un client.

Il diagramma di sequenza rappresenta il flusso di messaggi tra i vari componenti del sistema quando un client richiede, ad esempio, le statistiche dei trend giornalieri. Sono evidenziati:

- il ruolo del controller API;
- l'interazione con i servizi di dominio;
- l'accesso ai repository e al database;
- la restituzione del risultato al chiamante.

4.3 Struttura delle entità principali

Le tabelle di seguito descrivono i campi, con tipologie pensate per un dominio TypeScript e la loro mappatura tipica su DB relazionale. Si assumono colonne `createdAt`/`updatedAt` gestite automaticamente.

Utente

Campo	Tipo (TS)	Tipo (DB)	Note
userId	`string`	`AUTOINC PK`	Chiave primaria, auto-incrementale.
country	`string`	`VARCHAR(2)`	(opzionale, indicizzato per analisi).
device	`string`	`VARCHAR(40)`	es. `mobile`, `desktop`, `tablet`.
createdAt	`Date`	`TIMESTAMP`	Gestito dal DB/ORM.
updatedAt	`Date`	`TIMESTAMP`	Gestito dal DB/ORM.

Evento di ascolto

Campo	Tipo (TS)	Tipo (DB)	Note
id	`number`	`BIGINT` AUTOINC PK	PK intera per alto volume eventi.
userId	`string`	FK	FK → Utente(userId), indicizzato.
trackId	`string`	`VARCHAR(40)`	Non nullo.
artist	`string`	`VARCHAR(40)`	nome artista.
genere	`number`	`VARCHAR(40)`	genere della musica
country	`string`	`VARCHAR(40)`	paese.
duration	`string`	`VARCHAR(40)`	tempo di ascolto
device	`string`	`VARCHAR(40)`	dispositiva es. `Desktop, mobile....`

Statistiche (tabelle derivate)

StatTopArtist

Campo	Tipo (TS)	Tipo (DB)	Note
type	`string`	`VARCHAR(40)`	tipo statistica
value	`string`	`VARCHAR(120)`	nome artista.
metadata			detagli delle stastitiche(genere, max-Count..).

4.4 ORM e configurazione Sequelize

Questa sezione definisce le scelte di configurazione ORM, mantenendo la compatibilità con TypeScript.

Chiavi primarie e tipi

- **PK intere auto-incrementali**: `INTEGER` per la maggior parte delle entità; `BIGINT` per `EventoAscolto` (alto volume).
- Tipi TypeScript mappati: `number` per PK/FK e numerici; `string` per testi; `Date` per tempi.

Convenzioni di naming

- *snake_ case* per i nomi colonna (es. `user_id..`).
- Nomi tabella plurali e descrittivi (es. `utenti`, `brani`, `eventi_ascolto`, `stat_top_artist`).
- Abilitare `timestamps` (`createdAt`/`updatedAt`); opzionale `paranoid` se si necessita di `deletedAt`.

Vincoli e indici a livello ORM

- Definizione di `unique` su `username`, `(giorno, artista)`, `finestra_min`, `giorno` per le rispettive tabelle.
- Indici compositi su `eventi_ascolto`: `(utente_id)`.
- `validate` applicative (es. durata non negativa, `stato` ammesso) oltre ai vincoli DB.

Migrazioni e seed

- **Migrazioni iniziali**: creazione tabelle base, vincoli, indici.
- **Seed dati**: utenti e brani di esempio; eventi sintetici per testare aggregazioni (campioni per più giorni).
- **Rollback controllato**: migrazioni reversibili; ricostruzione tabelle statistiche da `eventi_ascolto`.

Strategia di aggregazione (persistenza statistiche)

- **Snapshot incrementali**: aggiornamento continuo di `StatTopArtist`, `StatAvgTime`, `StatDailyTrends` al flusso in ingresso.
- **Riconteggio batch**: job periodico per riallineare gli aggregati (tolleranza ad errori transitori).
- **Finestra temporale**: granularità configurabile (es. 1–5–15 min) e giornaliera per i trend.

Criteri prestazionali e crescita

- **Partizionamento logico** su `eventi_ascolto` per data (opzionale, lato DB).
- **Rotazione** e **archiviazione** degli eventi storici oltre una certa soglia.
- **Ottimizzazione indici**: revisione periodica in base ai pattern di query reali.

Capitolo 5

Progettazione del Sistema

Questo capitolo descrive l'impianto progettuale del sistema di analisi dei flussi musicali: il *flusso dei dati end-to-end* (Sez. 5.1), la *gestione degli eventi in streaming* (Sez. 5.2), la *pipeline di elaborazione* (Sez. 5.3), le strategie di *persistenza e repository* (Sez. 5.4) e la *gestione degli errori e resilienza* (Sez. 5.5). L'obiettivo è garantire accuratezza statistica, bassa latenza e robustezza operativa in contesti *event-driven*.

5.1 Flusso dei dati end-to-end

Il flusso end-to-end copre l'intero ciclo di vita dell'evento di ascolto: dalla generazione alla consultazione delle metriche via API.

Visione d'insieme

- **Ingestione:** eventi JSON di ascolto (utente, brano, timestamp).
- **Normalizzazione & Validazione:** conformità allo schema, unità temporali, deduplica.
- **Arricchimento:** join con metadati (artista, genere) e derivazione attributi (fascia oraria, giorno/settimana).
- **Aggregazione:** finestre temporali scorrevoli e giornaliere per le metriche richieste.
- **Persistenza:** memorizzazione eventi e snapshot delle statistiche.
- **Esposizione:** API REST per top artista, durata media, trend giornalieri.

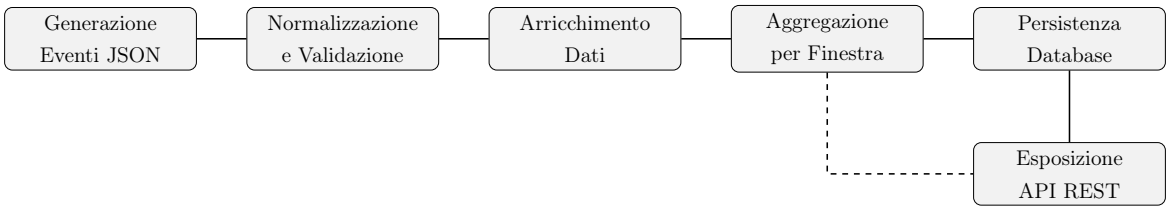

Figura 5.1: Diagramma di flusso dati end-to-end: dal JSON alla pubblicazione delle statistiche.

5.2 Gestione degli eventi in streaming

Sorgenti e schema eventi

Gli eventi provengono da un simulatore interno e rispettano uno schema logico (utente, brano, timestamp, durata, stato). Ogni evento include metadati per tracciabilità (origine, versione schema).

Controlli di qualità e normalizzazione

- **Validazione schema**: campi obbligatori, tipi, range temporali.
- **Normalizzazione**: unità di tempo in millisecondi, timezone unificata (UTC), trim stringhe.
- **Deduplica**: chiave hash su (utente, brano, inizio).

5.3 Pipeline di elaborazione (RxJS / Promise)

Obiettivi di progettazione

- **Composizione dichiarativa**: catena di operatori leggibile e testabile.
- **Condivisione risultati**: stream condivisi (*multicast*) per più osservatori (statistiche).

Operatori e fasi logiche

- **Filtraggio & mapping**: pulizia e proiezione dei campi rilevanti.
- **Enrichment**: join con catalogo (cache calda in memoria + fallback persistente).
- **Windowing**: *rolling* (es. 1–5 min) e *tumbling* giornaliero.
- **Aggregazioni**: conteggi per artista, medie di durata, top-k.

- **Materializzazione**: aggiornamento snapshot statistiche.

Concorrenti e scheduling

- **Concorrenti controllata**: limiti su operazioni di I/O (persistenza).
- **Condivisione risultati**: *share/replay* per evitare ricalcoli.
- **Policy di *retry*** per I/O transiente con backoff esponenziale.

5.4 Persistenza e repository

Principi

- **Separazione dei ruoli**: *Repository* incapsula l'accesso ai dati e nasconde l'ORM allo strato applicativo.
- **Schema duale**: storico eventi + tabelle di statistiche materializzate (*snapshot*).
- **Indice mirati**: per timestamp, artista, giorno, per ottimizzare query tipiche.

Politiche di scrittura

Tabella 5.1: Politiche di persistenza per eventi e statistiche.

Tipo dato	**Strategia**	**Note**
Eventi ascolto	Inserimento batch + deduplica	Idempotenza via chiave naturale
Statistiche rolling	*Upsert* su snapshot	Aggiornamenti frequenti, granularità fine
Trend giornalieri	Append + consolidamento a fine giornata	Ricalcolo controllato con watermark

Repository e contratti

- **EventRepository**: salva eventi validati, fornisce query per window riconteggio.
- **StatsRepository**: legge/scrive snapshot (top artista, durata media, trend).

Consistenza ed evoluzione

- **Consistenza eventuale**: le snapshot sono coerenti entro la tolleranza di ritardo.
- **Migrazioni controllate**: versionamento schema e migrazioni additive compatibili.
- **Caching**: layer in memoria per catalogo e letture metriche ad alta frequenza.

5.5 Gestione degli errori e resilienza

Classificazione degli errori

- **Transiente**: timeout I/O, picchi di carico (risolvibili con retry/backoff).
- **Permanente**: schema invalido, dati incoerenti (richiede correzione upstream).
- **Di dominio**: durata negativa, timestamp invertiti (scarto tracciato).

Policy di gestione

Tabella 5.2: Error handling: azioni e visibilità operativa.

Classe errore	**Azione**	**Osservabilità**
Transiente	Retry con backoff + jitter, circuito aperto su soglia	Log warn, metrica error_rate, alert su p95
Permanente	Scarto/rimedio, quarantena evento	Log error, dead-letter log, ticket manuale
Di dominio	Validazione/normalizzazione, scarto motivato	Log info+conteggio, report qualità dati

Osservabilità e salute

- **Logging strutturato**: correlation-id per tracciare una richiesta end-to-end.
- **Metriche**: throughput eventi/min, lag finestra, latenza p95 API, tasso scarti.
- **Healthcheck**: *liveness* (loop principale attivo) e *readiness* (DB raggiungibile, backlog sotto soglia).

Resilienza operativa

- **Graceful shutdown**: drenaggio buffer, flush snapshot, chiusura connessioni.
- **Ripartenza**: ripristino dallo snapshot e riconteggio mirato su finestra recente.

- **SLO/SLI**: p95 risposta API $\leq$ 150 ms; perdita eventi = 0% (deduplica/idempotenza); staleness snapshot $\leq$ 2 s (rolling breve).

Checklist di accettazione (capitolo)

- Flusso end-to-end definito con fasi E1–E6 e target di latenza.
- Regole per event-time, watermark e late data documentate.
- Pipeline con finestre, aggregazioni e condivisione risultati chiarita.
- Politiche di persistenza (*batch, upsert, append*) e repository mappate.
- Piano di *error handling*, osservabilità e healthcheck completo.

Sorgenti → Ingest → Pipeline → DB/Snapshot → API

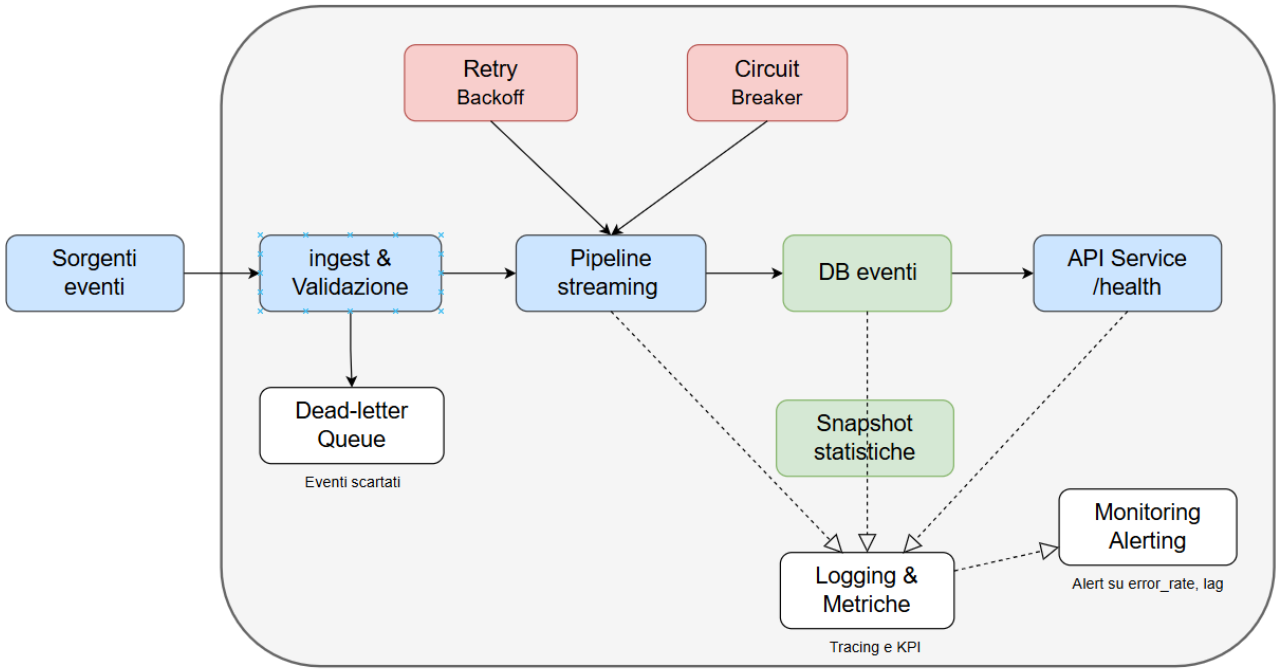

Figura 5.2: Schema logico di resilienza: retry/backoff, circuit breaker, dead-letter, logging e monitoring.

Capitolo 6

API REST

Questo capitolo descrive il contratto delle API REST esposte dal sistema di Analisi dei Flussi Musicali. Le API sono progettate per essere:

- **chiare e consistenti**: risorse e campi con naming uniforme in `lower-case-hyphenated`;
- **versionate**: prefisso `/api` per consentire evoluzioni retrocompatibili;
- **orientate ai dati**: formato di scambio `application/json; charset=utf-8`;
- **robuste**: gestione esplicita degli errori, codici di stato significativi, messaggi strutturati.

Salvo diversa indicazione, i tempi sono in UTC e le date in formato ISO 8601 (`YYYY-MM-DD`).

6.1 Struttura generale delle rotte

Base URL e versioning

- Base: `/api`
- Rotte eventi: `/api/events/...`
- Rotte statistiche: `/api/statistics/...`
- Salute servizio: `/api/health`

Formati e intestazioni

- Richieste: `Accept: application/json`
- Risposte: `Content-Type: application/json`

Convezioni di query e filtri

- `topN`: numero elementi da restituire (es. top artisti).

Codici di stato e errori (vedi §??):

- `2xx`: esito positivo (`200`, `201`, `204`)
- `4xx`: errore lato client (`400`, `404`, `429`)
- `5xx`: errore lato server (`500`)

6.2 Endpoints principali

6.2.1 `POST /api/events`

Descrizione Crea un evento nel data base.

Parametri query

Parametro	Tipo	Descrizione
`userId`	stringa	identificatore dell'utente.
`trackId`	stringa	identificatore della musica.
`artist`	stringa	nome dell'artista.
`duration`	number	durata di ascolto.
`genere`	stringa	genere della musica (Pop, Rock...).
`country`	stringa	paese di ascolto .
`device`	stringa	dispositivo di ascolto (optionale).

6.2.2 `POST /api/events`

Descrizione Crea molti eventi evento nel data base (i parametri sono configurati per postman) usando le collection.

```
{
  "userId": "{{userId}}",
  "trackId": "{{trackId}}",
  "artist": "{{artist}}",
  "duration": {{duration}},
  "timestamp": "{{timestamp}}",
  "genre": "{{genre}}",
  "country": "{{country}}",
  "device": "{{device}}"
}
```

6.2.3 GET /api/statistics/artista_piu_ascoltato

Descrizione Restituisce l'artista (o la classifica di artisti) più ascoltato nel periodo richiesto.

Risposta (200) Oggetto con metadati e lista ordinata di artisti.

Errori 400 parametri non validi; 500 errore interno.

6.2.4 GET /api/statistics/durata_media

Descrizione Restituisce la durata media di ascolto.

Risposta (200) Serie di punti temporali con media in secondi e un riepilogo globale.

Errori 400, 500.

6.2.5 GET /api/statistics/tendenza_giornaliera

Descrizione Restituisce trend giornalieri (conteggio ascolti, variazione percentuale, top artisti del giorno).

Risposta (200) Lista di giorni con metriche e top artisti.

Errori 400, 500.

6.2.6 GET /api/statistics/ore_di_punta

Descrizione Restituisce l'orario in cui gli utenti ascoltano di più un evento.

Risposta (200) Lista di giorni con metriche e top artisti.

Errori 400, 500.

6.2.7 GET /api/v1/health

Descrizione Indica lo stato di salute del servizio per scopi di monitoraggio e orchestrazione.

Risposta (200) Stato ok/degraded/down, versione, uptime, stato connessione DB.

Errori 500 in caso di dipendenze non disponibili.

Rotte disponiblili Eventuali rotte sono state implementate per aumentare il livello di filtraggio per artista, identificatore, range di data.....

- Rotte eventi: `GET /api/events/:id`
- Rotte eventi: `GET /api/events/artist/:artist`
- Rotte statistiche: `GET /api/statistics/artista_piu_suonato?date=2025-11-01`
- Rotte statistiche: `GET /api/statistics/durata_media?date=2025-11-01`
- Rotte statistiche: `GET /api/statistics/tendenza_giornaliera?startDate=2025-11-01 endDate=2025-11-10`
- Rotte statistiche: `GET /api/statistics/ore_di_punta?date=2025-11-01`
- Rotte statistiche: `GET /api/statistics/ore_di_punta?date=2025-11-01`
- Rotte statistiche: `GET /api/statistics/all?date=2025-11-01`

6.3 Esempi di request e response

6.3.1 Top Artist

Request

`GET /api/statistics/artista_piu_suonato`

Response (200)

```
{
{
    "success": true,
    "data": {
        "type": "artista\_pi \_suonato",
        "value": "Ultimo",
        "metadata": {
            "maxCount": 1121,
            "totalEvents": 20010,
            "percentage": "5.60",
            "genre": {
                "Dance": 6780,
                "Pop": 5299,
                "Indie": 3936,
                "Rock": 3385,
                "Trap": 609
            },
```

```
            "device": {
                "mobile": 4009,
                "desktop": 4000,
                "smart_speaker": 4000,
                "car": 4000,
                "tablet": 4000
            },
            "country": {
                "IT": 20009
            }
        }
    }
}
```


6.3.2 Durata media di ascolto

Request

`GET /api/statistics/durata_media`

Response (200)


```
{
    "success": true,
    "data": {
        "type": "durata\_media",
        "value": 222,
        "metadata": {
            "totalDuration": 4448982,
            "eventCount": 20010,
            "unit": "seconds",
            "genre": {
                "Dance": 6780,
                "Pop": 5299,
                "Indie": 3936,
                "Rock": 3385,
                "Trap": 609
            },
            "device": {
                "mobile": 4009,
                "desktop": 4000,
                "smart_speaker": 4000,
                "car": 4000,
                "tablet": 4000
```

```
            },
            "country": {
                "IT": 20009
            }
        }
    }
}
```


6.3.3 Trend giornalieri

Request

`GET /api/statistics/tendenza_giornaliera`

Response (200)


```
{
    "success": true,
    "data": {
        "type": "tendenza\_giornaliera",
        "value": [
            {
                "date": "2025-01-01",
                "eventCount": 84,
                "totalDuration": 28118,
                "uniqueArtists": 18,
                "uniqueUsers": 75
            },
            {
                "date": "2025-01-02",
                "eventCount": 59,
                "totalDuration": 15277,
                "uniqueArtists": 18,
                "uniqueUsers": 59
            },
            {
                "date": "2025-01-03",
                "eventCount": 75,
                "totalDuration": 13959,
                "uniqueArtists": 18,
                "uniqueUsers": 75
            }
              ],
        "metadata": {
```

```
            "totalDays": 560,
            "dateRange": {
                "start": "2025-01-01",
                "end": "2026-07-22",
                "genre": {
                    "Dance": 6780,
                    "Pop": 5299,
                    "Indie": 3936,
                    "Rock": 3385,
                    "Trap": 609
                },
                "device": {
                    "mobile": 4009,
                    "desktop": 4000,
                    "smart_speaker": 4000,
                    "car": 4000,
                    "tablet": 4000
                },
                "country": {
                    "IT": 20009
                }
            }
        }
    }
}
```


6.3.4 Orario di punta

Request

`GET /api/statistics/ore_di_punta`

Response (200)


```
{
    "success": true,
    "data": {
        "type": "ore\_di\_punta",
        "value": {
            "allHours": [
                {
                    "hour": 0,
                    "eventCount": 2509,
                    "totalDuration": 559651,
```

```
                "averageDuration": 223,
                "uniqueUsers": 2500,
                "uniqueArtists": 9
            },
              {
                "hour": 2,
                "eventCount": 0,
                "totalDuration": 0,
                "averageDuration": 0,
                "uniqueUsers": 0,
                "uniqueArtists": 0
            },
            {
                "hour": 3,
                "eventCount": 2500,
                "totalDuration": 548866,
                "averageDuration": 220,
                "uniqueUsers": 2500,
                "uniqueArtists": 9
            },
                 ]
    },
    "metadata": {
        "totalEvents": 20010,
        "peakHour": 0,
        "peakHourEventCount": 2509,
        "analysisDate": "2025-11-18T21:56:43.725Z",
        "genre": {
            "Dance": 6780,
            "Pop": 5299,
            "Indie": 3936,
            "Rock": 3385,
            "Trap": 609
        },
        "device": {
            "mobile": 4009,
            "desktop": 4000,
            "smart_speaker": 4000,
            "car": 4000,
            "tablet": 4000
        },
```

```
            "country": {
                "IT": 20009
            }
        }
    }
}
```

6.3.5 Health

Request

`GET /api/health`

Response (200)

```
{
    "status": "ok",
    "timeStamp": "2025-11-18T21:55:18.833Z"
}
```

Capitolo 7

Test e Validazione

Questo capitolo descrive in modo dettagliato le attività di testing svolte sul progetto *Analizzatore di flussi musicali*, includendo test unitari, test di integrazione, test delle API e misurazioni della copertura. L'obiettivo è garantire l'affidabilità, la coerenza e la correttezza del sistema in ogni sua componente.

7.1 Testing unitario con Jest

Il testing unitario rappresenta la base della strategia di validazione del progetto. È stato svolto utilizzando il framework **Jest**, scelto per la sua semplicità, rapidità di esecuzione e integrazione nativa con TypeScript.

Ogni componente critica dell'applicazione è stata testata isolatamente tramite mocking delle dipendenze, seguendo i principi SOLID e garantendo un'elevata testabilità. La struttura dei file di test unitari rispecchia l'architettura del progetto.

NB: Ogni componente dell'applicazione può essere testato eseguendo il commando : **npm test – NomeComponente.test.ts** (es: npm test – EventController.test.ts).

- Il commando **npm test** esegue il test di ogni componente dell'applicazione

Componenti testate

- **Controllers**
  - EventController.test.ts
  - StatisticsController.test.ts
- **Observers**
  - EventSubject.test.ts

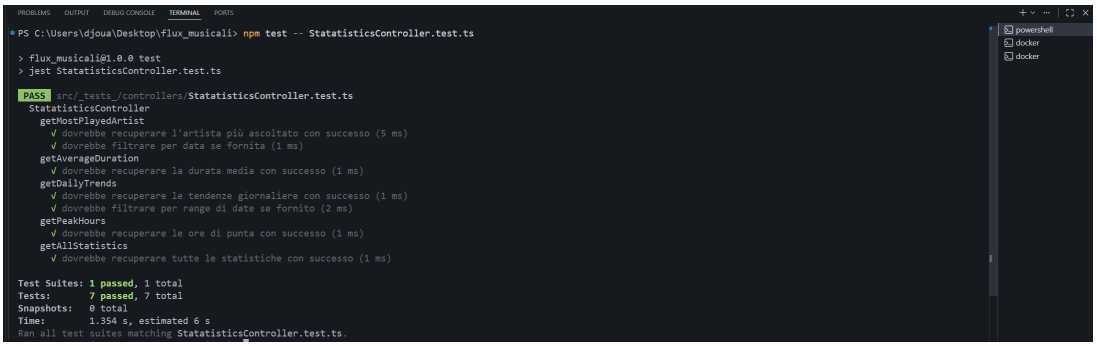

Figura 7.1: Test di un componente del componente StatisticController

- **Repository**
  - MusicsEventRepository.test.ts
- **Services**
  - EventProcessingService.test.ts
  - StatisticService.test.ts
- **Strategies** (Pattern Strategy)
  - AverageDurationStrategy.test.ts
  - DailyTrendStrategy.test.ts
  - MostPlayedArtistStrategy.test.ts
  - PeakHoursStrategy.test.ts
  - StatisticsStrategyFactory.test.ts

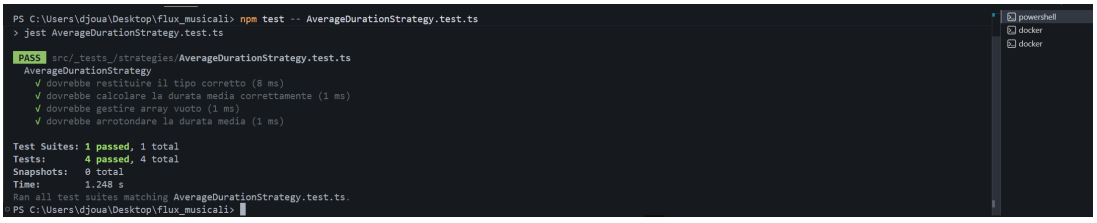

Figura 7.2: Test di un componente strategica

Ogni test unitario verifica:

- la correttezza delle funzioni principali;
- la gestione di input non validi;
- la coerenza dei risultati rispetto alle specifiche funzionali;
- il corretto uso dei pattern (Observer, Strategy, Repository).

7.2 Testing d'integrazione

Il testing d'integrazione ha come obiettivo verificare la corretta interazione tra i diversi moduli del sistema, in particolare tra:

- pipeline di elaborazione degli eventi;
- repository e database;
- controller e servizi applicativi;
- strategie statistiche applicate sullo stream di dati.

Il file principale di integrazione sviluppato è:

- **api.test.ts**

Figura 7.3: Test d'integrazione

Questo test verifica l'intero flusso:

- ingestione degli eventi;
- elaborazione tramite servizi;
- aggiornamento delle statistiche;
- risposta delle API ai client.

Le interazioni sono state validate tramite database mockato o tramite un database in-memory, assicurando un ambiente controllato e riproducibile.

7.3 Testing delle API con Postman

Per la validazione funzionale delle API REST è stata utilizzata una **Postman Collection** dedicata, eseguita sia manualmente sia tramite script di automazione.

Gli obiettivi principali dei test API sono stati:

- verificare la correttezza dei percorsi REST (status code, headers);

- controllare l'integrità delle risposte JSON;
- testare scenari reali (assenza di dati, eventi in arrivo, errori);
- valutare la coerenza delle statistiche generate.

Le rotte testate includono:

- `/api/statistics/artista_piu_suonato`

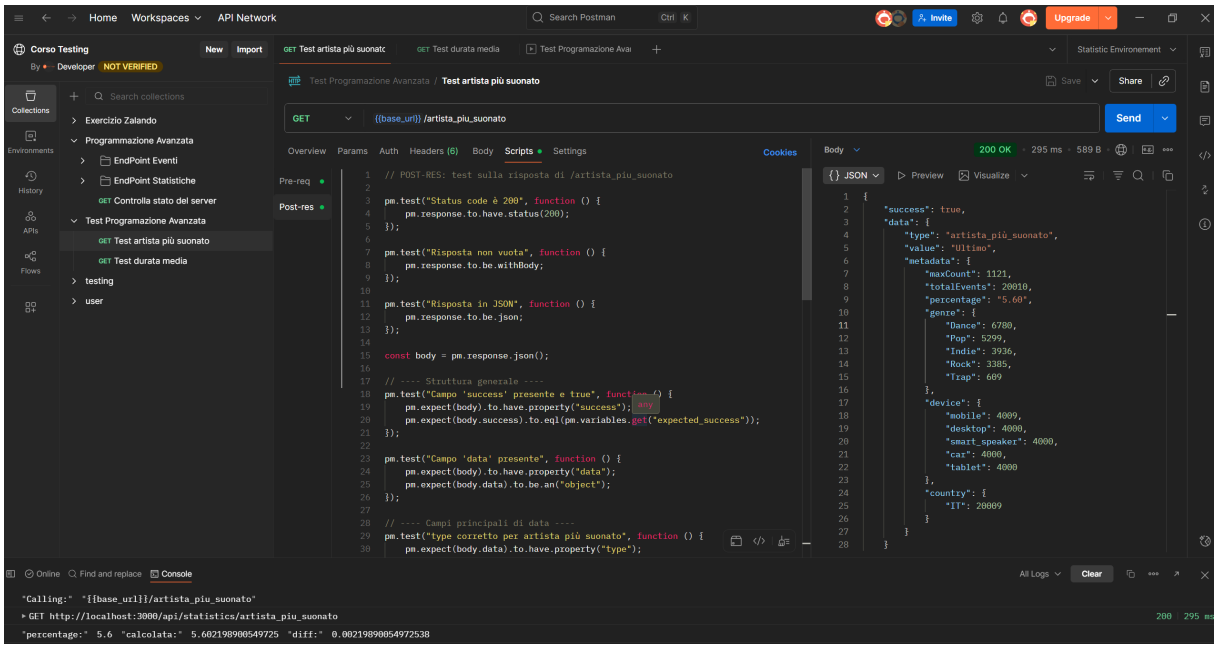

Figura 7.4: Test con postman per l'artista più suonato

- `GET /api/statistics/durata_media`

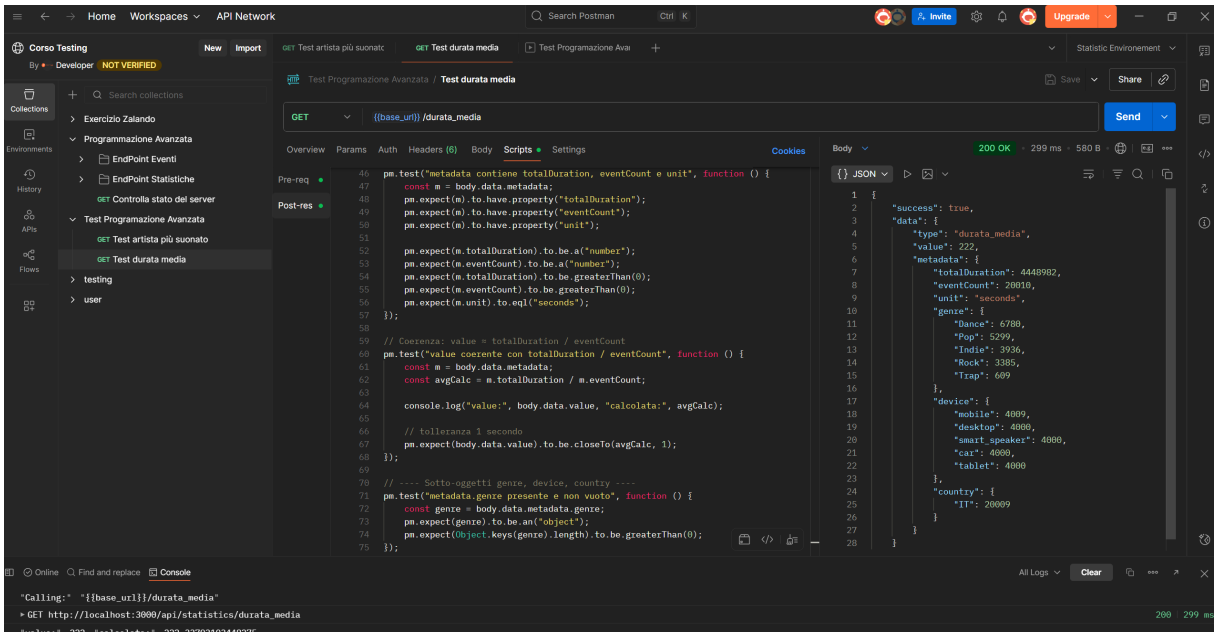

Figura 7.5: Test con postman sulla durata media

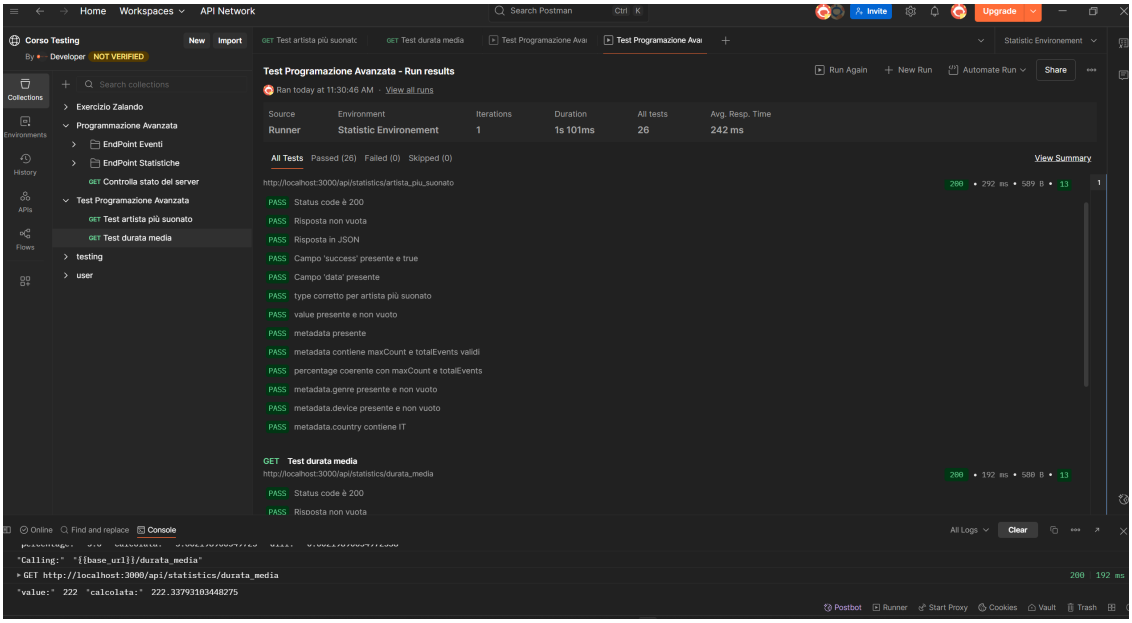

Figura 7.6: Test tramite collection

7.4 Misurazione della copertura dei test

La copertura dei test è stata misurata tramite le funzionalità integrate di Jest (`-coverage`). Sono stati raccolti i valori di:

- **Statements coverage**
- **Branches coverage**

- **Functions coverage**
- **Lines coverage**

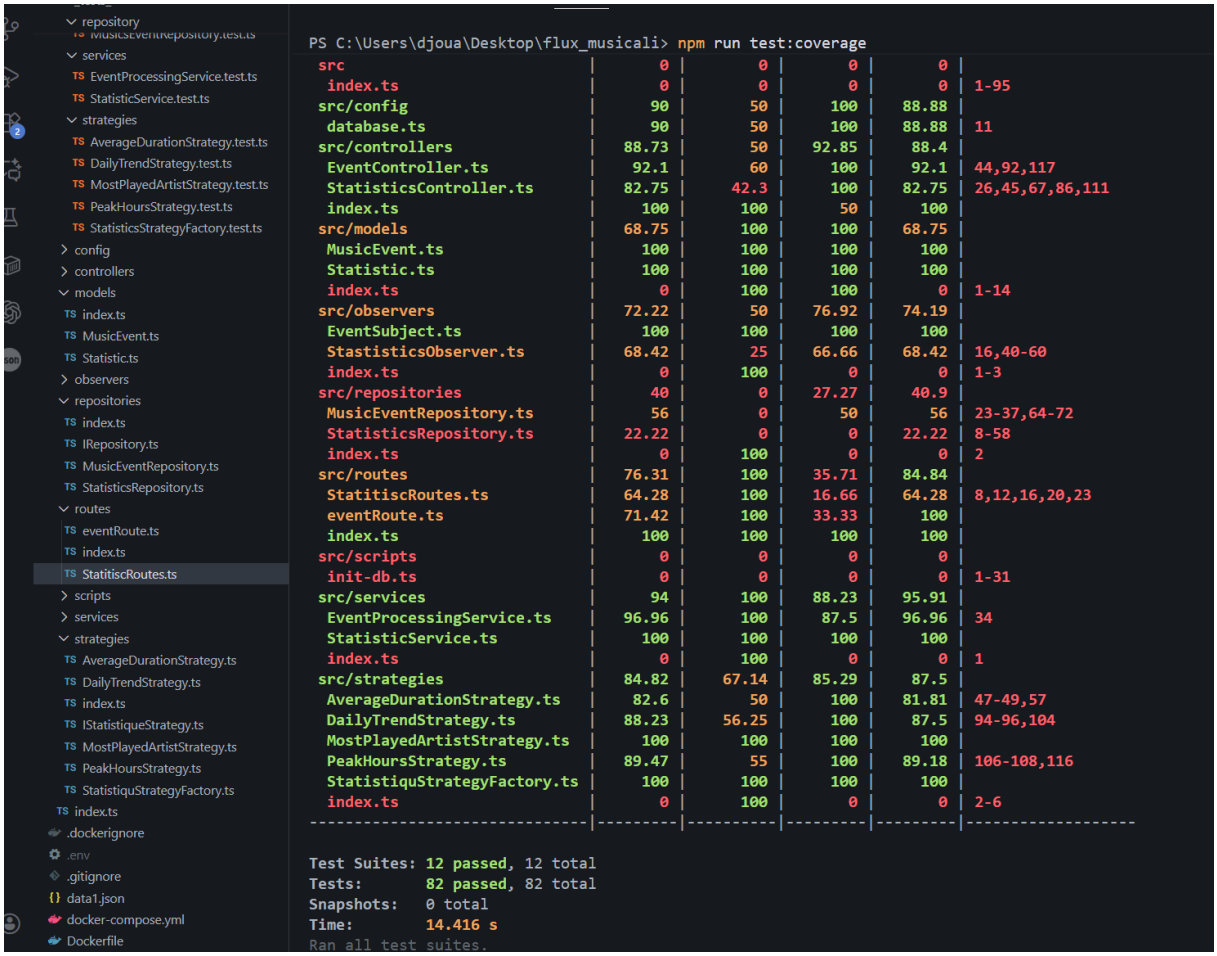

Figura 7.7: Test Coverage

La copertura dei test del progetto è stata misurata utilizzando il comando `npm run test:coverage`, che genera un report dettagliato per ogni file TypeScript presente nella codebase. Il sistema utilizza Jest come framework di testing, con la configurazione estesa per il calcolo della copertura su:

- **Statements** (istruzioni eseguite)
- **Branches** (ramificazioni condizionali)
- **Functions** (funzioni e metodi)
- **Lines** (righe di codice)

Dall'analisi del report emerge che sono stati eseguiti:

- **12 Test Suites**, tutte superate con successo;

- **82 Test totali**, tutti passati;
- **0 Snapshot**, poiché non impiegati nel progetto;
- **Tempo totale di esecuzione**: 14.416 s.

La copertura globale risulta elevata, con diversi moduli che raggiungono valori prossimi o pari al 100%. In particolare si osservano:

- Ottima copertura dei servizi come `EventProcessingService.ts` e `StatisticsService.ts`, con valori superiori al 95%;
- Copertura completa (100%) di vari elementi strutturali quali controller, routes e indici di modulo;
- Copertura leggermente inferiore nei repository (`MusicEventRepository.ts`, `StatisticsReposito` a causa di percorsi condizionali non sempre esercitati nei test;
- Alcune strategie (es. `DailyTrendStrategy.ts`, `MostPlayedArtistStrategy.ts`) mostrano coperture parziali su rami non critici, indicando possibili scenari non simulati.

Nel complesso, la qualità della suite di test risulta molto elevata: tutte le funzionalità principali sono coperte e validate, garantendo affidabilità nell'elaborazione dei flussi e nella generazione delle statistiche. Le parti con copertura minore possono essere ulteriormente migliorate introducendo casi di test aggiuntivi per coprire eccezioni, rami condizionali o input non standard.

7.5 Validazione dei risultati e metriche

Al termine del processo di testing, i risultati sono stati valutati secondo i seguenti criteri:

- **Correttezza funzionale**: le statistiche prodotte rispettano i requisiti richiesti.
- **Coerenza temporale**: le metriche aggiornate riflettono in modo coerente il flusso degli eventi.
- **Robustezza**: il sistema gestisce correttamente input anomali o incompleti.
- **Scalabilità**: nessun degrado significativo sotto carico simulato.
- **Affidabilità delle API**: tutte le risposte sono state validate tramite Postman e test di integrazione.

La validazione complessiva conferma che il sistema risponde in modo stabile e affidabile agli obiettivi del progetto, garantendo un'elaborazione accurata dei flussi musicali e statistiche coerenti nel tempo.

Capitolo 8

Containerizzazione e Deployment

8.1 Struttura Docker e Compose

La containerizzazione del progetto è realizzata tramite `Docker` e orchestrata tramite `Docker Compose`. L'obiettivo è garantire un ambiente di esecuzione isolato, riproducibile e facilmente distribuibile.

La soluzione prevede i seguenti contenitori principali:

- **API Service**: applicazione Node.js/TypeScript eseguita tramite un'immagine personalizzata.
- **Database**: istanza relazionale dedicata (PostgreSQL) per la persistenza dei dati.

La struttura logica del file `docker-compose.yml` è organizzata in:

1. definizione dei servizi coinvolti;
2. configurazione delle immagini e del build context;
3. gestione delle reti interne del progetto;
4. creazione dei volumi per la persistenza dati;
5. definizione delle dipendenze e dell'ordine di avvio.

8.2 Configurazione dei servizi (API, DB)

API Service

Il servizio API rappresenta il cuore applicativo:

- basato su un `Dockerfile` dedicato;
- espone la porta applicativa definita nelle variabili d'ambiente;

- comunica internamente con il database tramite rete Docker;
- utilizza un container leggero e ottimizzato per TypeScript/Node.js.

Database

Il database relazionale è configurato per garantire stabilità e persistenza:

- definizione di utente, password e nome del database;
- persistenza tramite volume dedicato;
- esecuzione su rete isolata in modo da evitare esposizione pubblica.

8.3 Variabili d'ambiente e rete interna

La configurazione delle variabili d'ambiente è centralizzata nel file `.env`. Le variabili fondamentali includono:

- `DB_HOST`
- `DB_USER`
- `DB_PASSWORD`
- `DB_NAME`
- `API_PORT`

La rete interna Docker consente:

- isolamento dei servizi dal mondo esterno;
- comunicazione sicura tra API e database;
- utilizzo di hostname definiti tramite `network aliases`.

8.4 Procedure di build e run

Le procedure di esecuzione sono progettate per garantire semplicità e riproducibilità del sistema.

Build

L'immagine delle API viene compilata tramite:

- processo di build definito nel `Dockerfile`;
- installazione delle dipendenze e compilazione TypeScript;
- creazione di un'immagine ottimizzata per la produzione.

Run

L'intero ecosistema viene avviato attraverso un singolo comando:

- avvio simultaneo dei servizi tramite `docker-compose`;
- creazione automatica della rete interna;
- esecuzione dei container con restart policies configurate.

8.5 Healthcheck e monitoraggio

Per garantire l'affidabilità del sistema, sono configurati meccanismi di controllo dello stato:

- **Healthcheck dell'API**: verifica periodica dell'endpoint di salute.
- **Healthcheck del Database**: controllo del servizio tramite comandi SQL di validazione.
- **Logging**: registrazione strutturata degli eventi applicativi.
- **Monitoraggio**: possibilità di estendere con strumenti come Grafana.

Tali strumenti consentono una supervisione continua e facilitano la diagnosi di problemi durante l'esecuzione del sistema.

Capitolo 9

Conclusioni

9.1 Sintesi del lavoro svolto

Il progetto ha realizzato un sistema completo per l'analisi in tempo reale di flussi musicali generati da utenti simulati. L'obiettivo principale era quello di progettare e implementare un'architettura moderna basata su principi di programmazione asincrona, gestione eventi e design pattern avanzati.

Il lavoro svolto ha permesso di:

- definire un modello dei dati coerente, capace di rappresentare utenti, brani, eventi di ascolto e statistiche aggregate;
- progettare un'architettura modulare attraverso i pattern *Strategy*, *Observer* e *Repository*;
- implementare una pipeline di elaborazione degli stream basata su `Promise` e/o `RxJS`;
- sviluppare un servizio API REST per l'esposizione delle principali metriche: artista più ascoltato, durata media, trend giornalieri, ore di punta;
- integrare `Sequelize` come ORM per la gestione della persistenza;
- containerizzare l'intero sistema tramite `Docker` e orchestrarlo tramite `Docker Compose`;
- validare il funzionamento attraverso test unitari e di integrazione con `Jest`, e test delle API tramite Postman;
- redigere documentazione tecnica completa, comprensiva di diagrammi UML, procedure di avvio e analisi architetturale.

L'insieme di queste attività garantisce un sistema stabile, coerente e facilmente estensibile.

9.2 Benefici del sistema

Il sistema sviluppato offre numerosi vantaggi sia da un punto di vista tecnico sia operativo.

Benefici tecnici

- **Scalabilità**: l'architettura event-driven permette di gestire un volume elevato di eventi senza compromettere le prestazioni.
- **Modularità**: l'utilizzo dei design pattern consente un'elevata separazione delle responsabilità e facilita la manutenzione del codice.
- **Affidabilità**: la containerizzazione garantisce ambienti riproducibili e controllati.
- **Testabilità**: la presenza di test unitari e di integrazione assicura robustezza e riduce il rischio di regressioni.

Benefici applicativi

- **Analisi in tempo reale**: il sistema calcola metriche aggiornate continuamente, utili per piattaforme musicali, analisi di mercato o sistemi di raccomandazione.
- **Centralizzazione dei dati**: le statistiche storiche e correnti sono accessibili tramite API dedicate.
- **Flessibilità**: la pipeline di elaborazione può essere facilmente adattata a nuovi tipi di eventi o nuove metriche.

9.3 Possibili evoluzioni

Nonostante il sistema sia pienamente funzionante, esistono numerosi margini di estensione futura che possono arricchire e ampliare le funzionalità.

Evoluzioni tecniche

- integrazione di un servizio di *message broker* come Kafka o RabbitMQ per l'ingestione massiva di eventi;
- introduzione di meccanismi di *caching* (Redis) per ottimizzare la lettura delle statistiche;
- aggiunta di strumenti di monitoraggio avanzato come Prometheus e Grafana.

Evoluzioni funzionali

- aggiunta di statistiche più complesse (ad esempio: distribuzione degli ascolti per fascia oraria, analisi per genere musicale);
- implementazione di un modulo di raccomandazione semplice basato sulle abitudini di ascolto;
- sviluppo di una dashboard grafica per la visualizzazione dei dati in tempo reale.

Evoluzioni architetturali

- trasformazione del progetto in un'architettura a microservizi;
- deploy su piattaforme cloud (AWS, Azure, GCP) con pipeline CI/CD;
- introduzione di un sistema di autenticazione per l'accesso alle API.

In conclusione, il progetto rappresenta una base solida e moderna per la gestione di flussi musicali in tempo reale e offre numerose possibilità di ampliamento sia dal punto di vista tecnologico che funzionale.

Capitolo 10

Appendici

10.1 Esempi di file JSON simulati

Questa sezione raccoglie alcuni esempi rappresentativi dei file JSON utilizzati per simulare i flussi di ascolto. Essi sono stati fondamentali per testare la pipeline di elaborazione e validare il comportamento del sistema.

Esempio 1: Evento di ascolto minimo


```
{
    "userId": "u123",
    "trackId": "t567",
    "artist": "Radiohead",
    "duration": 180,
    "timestamp": "2025-11-10T21:30:00Z",
    "genre": "Rock",
    "country": "IT",
    "device": "mobile"
}
```


Esempio 2: Evento con informazioni estese


```
{
     "id": 1,
     "userId": "u1",
     "trackId": "t001",
     "artist": "Ultimo",
     "genre": "Dance",
     "country": "IT",
```

```
    "device": "mobile",
    "duration": 832,
    "timestamp": "2025-01-01T00:00:00.000Z",
    "createdAt": "2025-11-17T09:52:33.388Z",
    "updatedAt": "2025-11-17T09:52:33.388Z"
}
```

Esempio 3: Batch di eventi

```
[
    {
        "userId": "u123",
        "trackId": "t567",
        "artist": "Radiohead",
        "duration": 180,
        "timestamp": "2025-11-10T21:30:00Z",
        "genre": "Rock",
        "country": "IT",
        "device": "mobile"
    },
    {
        "userId": "u123",
        "trackId": "t567",
        "artist": "Radiohead",
        "duration": 180,
        "timestamp": "2025-11-10T21:30:00Z",
        "genre": "Rock",
        "country": "IT",
        "device": "mobile"
    }
]
```

10.2 Script Postman di test

Gli script Postman sono stati impiegati per verificare la correttezza delle API REST esposte dal sistema.

Test 1: Verifica risposta endpoint /health

```
pm.test("Status code is 200", function () {
```

```
    pm.response.to.have.status(200);
});
```

Test 2: Struttura dati endpoint /stats/top-artist

```
pm.test("Top artist structure is valid", function () {
    const json = pm.response.json();
    pm.expect(json).to.have.property("artist");
    pm.expect(json).to.have.property("count");
});
```

Test 3: Controllo durata media

```
pm.test("Average listening time is numeric", function () {
    const json = pm.response.json();
    pm.expect(json.average).to.be.a("number");
});
```

10.3 Log di esecuzione e output campione

Questa sezione riporta frammenti reali di log generati durante l'esecuzione del sistema, utili per analisi, debugging e validazione.

Esempio di log sistema API

```
[INFO] 2025-11-14 10:21:33 - Server avviato sulla porta 3000
[INFO] 2025-11-14 10:21:34 - Connessione al database stabilita
```

Output campione delle statistiche

```
{
    "success": true,
    "data": {
        "type": "artista_più_suonato",
        "value": "Ultimo",
        "metadata": {
            "maxCount": 1125,
            "totalEvents": 20014,
            "percentage": "5.62",
            "genre": {
```

```
                "Dance": 6784,
                "Pop": 5299,
                "Indie": 3936,
                "Rock": 3385,
                "Trap": 609
            },
            "device": {
                "mobile": 4013,
                "desktop": 4000,
                "smart_speaker": 4000,
                "car": 4000,
                "tablet": 4000
            },
            "country": {
                "IT": 20013
            }
        }
    }
}
```


10.4 Bibliografia e riferimenti

La seguente bibliografia raccoglie le principali fonti teoriche e documentali utilizzate durante lo sviluppo del progetto.

Documentazione ufficiale

- Node.js Documentation: https://nodejs.org/docs
- TypeScript Handbook: https://www.typescriptlang.org/docs
- RxJS Documentation: https://rxjs.dev/guide/overview
- Sequelize ORM: https://sequelize.org
- Docker Documentation: https://docs.docker.com

Riferimenti aggiuntivi

- Postman Learning Center: https://learning.postman.com
- Twelve-Factor App Principles: https://12factor.net

- Docker Compose Reference: https://docs.docker.com/compose/