

Documentazione Tecnica: Microservizi Asincroni con RabbitMQ e Docker

Team Sviluppo – De Cao Andrea

February 6, 2026

Contents

1 Obiettivo del Progetto	2
2 Architettura dei Microservizi	3
2.1 Schema logico aggiornato	3
2.2 Ruoli dei componenti	3
3 Prima Versione – Problemi Riscontrati	4
3.1 Comportamento	4
3.2 Analisi Tecnica	4
3.3 Codice Esempio	4
3.4 Comportamento visibile	4
4 Versione Corretta – Code Durabili e Messaggi Persistenti	5
4.1 Modifiche Implementate	5
4.2 Effetti delle modifiche	5
4.3 Comportamento visibile	5
4.4 Benefici	5
5 Confronto Tecnico Prima / Dopo	6
6 Conclusioni e Best Practice	7

1 Obiettivo del Progetto

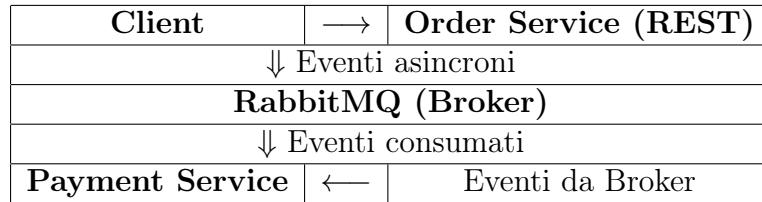
Il progetto implementa un sistema a **microservizi asincroni** per la gestione degli ordini e dei pagamenti di un e-commerce.

Gli obiettivi principali sono:

- Disaccoppiamento tra servizi
- Resilienza ai downtime dei microservizi
- Comunicazione asincrona tramite broker RabbitMQ
- REST sincrono per accettazione ordini
- Event-driven processing per pagamenti

2 Architettura dei Microservizi

2.1 Schema logico aggiornato



2.2 Ruoli dei componenti

- **Order Service:** espone endpoint REST /orders, pubblica eventi ORDER_CREATED.
- **Payment Service:** ascolta eventi dal broker e processa pagamenti.
- **RabbitMQ:** smista messaggi tra servizi, bufferizza eventi e garantisce resilienza.

3 Prima Versione – Problemi Riscontrati

3.1 Comportamento

- Publisher crea **code temporanee** (`exclusive: true`) per ogni listener.
- Se Payment Service è offline:
 - REST `/orders` → 202 Accepted.
 - Messaggi non memorizzati in RabbitMQ.
 - Payment Service non riceve eventi al riavvio.

3.2 Analisi Tecnica

- `exclusive: true` → coda esiste solo se il consumer è attivo.
- Nessun flag **persistent** sui messaggi → i messaggi vengono persi se il consumer non è presente.
- Risultato: disaccoppiamento parziale ma assenza di resilienza reale.

3.3 Codice Esempio

```
1 await channel.assertExchange(exchange, 'fanout', {durable: true});
2 channel.publish(exchange, '', Buffer.from('ORDER_CREATED'));
```

Listing 1: Publisher prima versione

```
1 const q = await channel.assertQueue('', { exclusive: true });
2 channel.bindQueue(q.queue, exchange, ''');
```

Listing 2: Listener prima versione

3.4 Comportamento visibile

- REST: 202 Accepted
- RabbitMQ Dashboard: nessuna coda visibile
- Logs Payment Service: nessun messaggio ricevuto

4 Versione Corretta – Code Durabili e Messaggi Persistenti

4.1 Modifiche Implementate

```

1 channel.publish(
2   exchange,
3   '',
4   Buffer.from('ORDER_CREATED'),
5   { persistent: true } // Messaggio persistente
6 );

```

Listing 3: Publisher versione corretta

```

1 const queueName = 'payment-queue';
2 const q = await channel.assertQueue(queueName, { durable: true });
3 channel.bindQueue(q.queue, exchange, '');

```

Listing 4: Listener versione corretta

4.2 Effetti delle modifiche

- Coda nominata `payment-queue`, durabile e persistente
- Messaggi persistenti → non persi se Payment Service offline
- Al riavvio Payment Service elabora tutti i messaggi accumulati

4.3 Comportamento visibile

- REST `/orders`: 202 Accepted (come prima)
- RabbitMQ Dashboard: `payment-queue` visibile con messaggi in coda
- Payment Service riceve eventi al riavvio

4.4 Benefici

- Resilienza reale
- Scalabilità
- Disaccoppiamento completo
- Debug e monitoraggio semplificati

5 Confronto Tecnico Prima / Dopo

Aspetto	Prima versione	Versione corretta
Coda	Temporanea (<code>exclusive: true</code>)	Durabile con nome fisso (<code>durable: true</code>)
Messaggi	Non persistenti	Persistenti (<code>persistent: true</code>)
Downtime Payment Service	Messaggi persi	Messaggi rimangono in coda e processati al riavvio
REST /orders	202 Accepted	202 Accepted
RabbitMQ Dashboard	Nessuna coda visibile	<code>payment-queue</code> visibile con messaggi accumulati

6 Conclusioni e Best Practice

- 202 Accepted indica solo l'accettazione della richiesta, non il completamento.
- L'uso di code durabili e messaggi persistenti è fondamentale in sistemi asincroni reali.
- RabbitMQ fornisce buffering, smistamento sicuro e monitoraggio dei messaggi.
- Questa architettura è pronta per scalabilità orizzontale e deployment con Docker/-Docker Compose.
- Pattern consigliato: REST sincrono per accettazione, Event-driven asincrono per elaborazione reale.