KOMPOZER

Un e-commerce distribuito per la configurazione collaborativa di scaffalature modulari

Valerio Giannini

Kompo

Scaffalature modulari componibili.

L'utente compone una scaffalatura scegliendo componenti compatibili all'interno di una delle quattro famiglie di prodotto, vede preview e prezzo aggiornati ad ogni modifica, salva la configurazione e ordina.

Perché è distribuito

1. **Resource sharing**

   Un unico catalogo autoritativo condiviso da molti utenti e da cinque contesti diversi.

2. **Fault tolerance**

   Perdere una configurazione non è accettabile: è il core workflow.

3. **Evolvability**

   Il dominio si decompone in microservizi (bounded context) indipendenti.

4. **Scalability**

   Carico dominato da molte sessioni indipendenti e read-heavy.

Requisiti

Di seguito i 5 requisiti base che definiscono il design.

NFR1

Consistenza del catalogo condiviso

Prezzo e disponibilità propagati ad ogni contesto dipendente, applicati in modo idempotente.

NFR2

Consegna real-time

Notifiche e risposte del chatbot in push, senza polling del client.

NFR3

Fault tolerance del core workflow

Il contesto di configurazione tollera la perdita di un nodo del database.

NFR4

Recoverability

Lo stato di una sessione collaborativa sopravvive al crash del servizio.

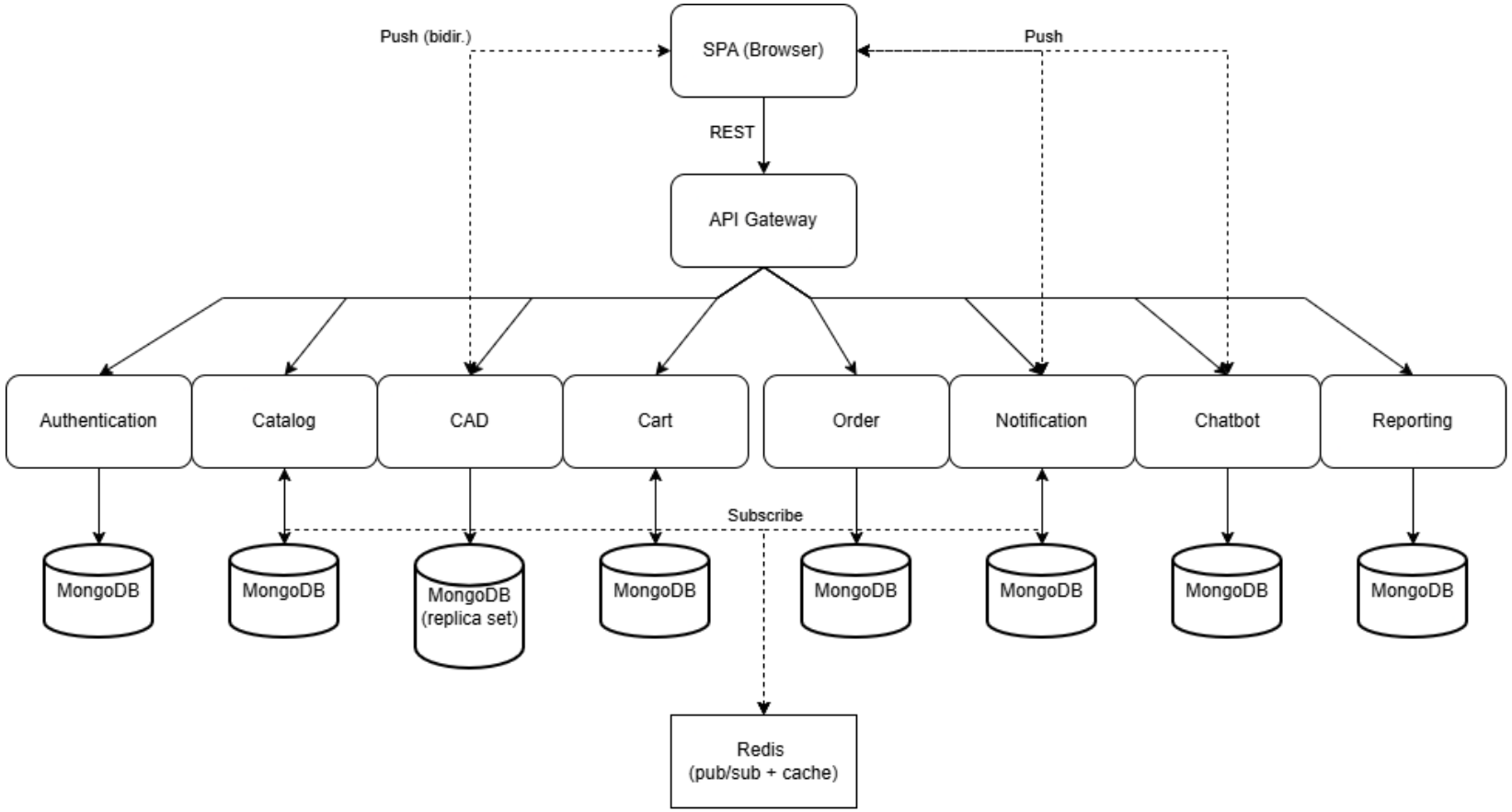
NFR5

Sicurezza e autorizzazione

Funzioni admin protette da ruolo; ogni utente accede solo alle proprie risorse.

Architettura

Microservizi dietro un unico gateway, un database per servizio.



La SPA non parla mai direttamente con un servizio di dominio: ogni chiamata attraversa il gateway.

API Gateway

Unico punto di ingresso: routing REST, proxy del canale WebSocket, verifica del token, normalizzazione degli header di identità,

8 servizi di dominio

Un servizio per ogni bounded context, completamente autonomo.

Database-per-service

Nessuna entità è condivisa oltre il confine del servizio: solo gli identificatori. Nessuna join cross-service.

Comunicazione

REST sincrono

HTTP/1.1 · JSON

- Default per comandi e query con un proprietario unico
- Timeout espliciti: 3 s CAD→catalog, 5 s gateway→servizi
- Nessun retry, tranne il chatbot verso l'LLM

Eventi asincroni

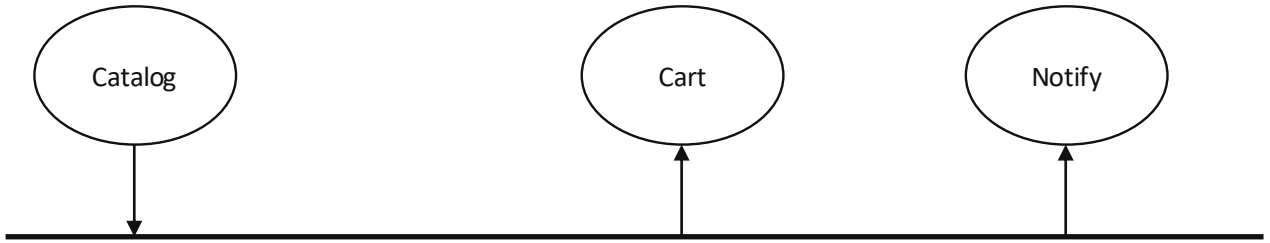
Redis pub/sub

- PRICE_CHANGED e AVAILABILITY_CHANGED da catalog a cart e notification
- At-most-once: chi è offline perde l'evento
- Consumo idempotente su eventId

Push real-time

Socket.IO · WebSocket

- Una room per utente e per sessione collaborativa
- Transport polling→websocket con fallback
- Bidirezionale solo per l'editor collaborativo



canale Redis · pub/sub, at-most-once

Limiti

Redis pub/sub non è una coda: se un consumer è giù quando l'evento è pubblicato, quell'evento è perso. L'idempotenza protegge dai duplicati, non dalle perdite. Con Redis Streams o un broker persistente il consumer riprenderebbe da dove si era fermato.

Concorrenza

Le letture concorrenti sono libere. Le scritture sullo stesso documento sono l'unico punto con race reali.

A Optimistic Concurrency Control (OCC) su intero documento

Componenti di catalogo · configurazione editata in solo

- Ogni documento porta un campo version
- La scrittura deve dichiarare la version letta
- findOneAndUpdate condizionale su { _id, version: expected }
- Mismatch → 409, mai sovrascrittura silenziosa

Previene il lost update classico.

B Last-writer-wins per campo

Solo dentro una sessione CAD collaborativa aperta

- Ogni campo editabile ha il proprio Lamport clock
- Due partecipanti su campi diversi non entrano mai in conflitto
- Due edit sullo stesso campo sono ordinati, invece di far fallire uno dei due
- L'OCC a documento intero qui sarebbe insostenibile: tutti in conflitto con tutti

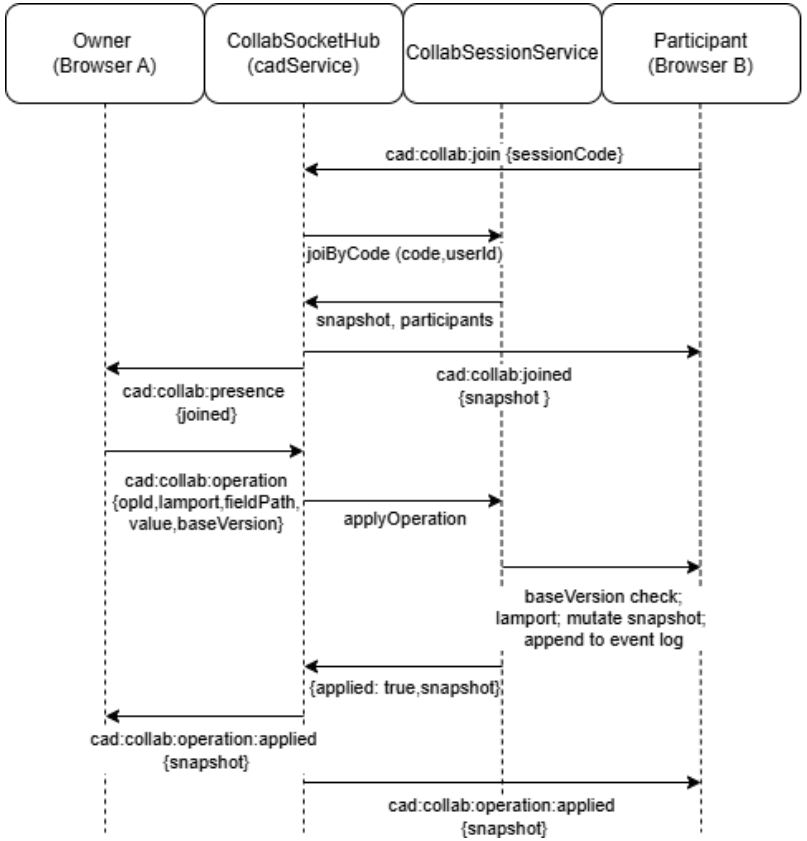
Ordinamento

Editing collaborativo senza con i Lamport clock.

Ogni operazione porta un opId, un timestamp di Lamport e la baseVersion su cui è stata calcolata. Il servizio è l'unica autorità: valida, risolve, muta lo snapshot condiviso e ritrasmette.

1. **Timestamp più alto vince**
   Confronto sul Lamport clock del singolo campo.
2. **Parità risolta sull'actor id**
   Tie-break deterministico, nessun coordinamento.
3. **Campi indipendenti convergono da soli**
   Due partecipanti su campi diversi non si vedono mai.
4. **opId duplicato → ack senza riapplicare**
   Le operazioni sono idempotenti: ritrasmissioni sicure.
5. **baseVersion vecchia → rifiuto**
   Il client si risincronizza sullo snapshot corrente.

Il risultato è indipendente dall'ordine di arrivo: tutte le repliche convergono.



Join con session code, poi operazioni field-level in broadcast sulla room.

Replicazione

Replica set cad

Perdere un membro lascia la maggioranza fra i due superstiti: le scritture continuano e, se a cadere è stato il primary, il replica set ne elegge automaticamente uno nuovo.

Configurazione

`writeConcern:` **`majority`**

durabile su una maggioranza prima dell'ack

`readConcern:` **`majority`**

nessuna lettura di dati non confermati

`readPreference:` **`primaryPreferred`**

fallback su un secondario se serve

Sotto partizione

- Il replica set CAD tollera una partizione che lasci raggiungibile la maggioranza
- Un MongoDB single-node non ha tolleranza: quel contesto è giù finché non torna
- Le rotte proxy del gateway non hanno fallback

Dove NON ho replicato, e perché

Ogni altro contesto usa un MongoDB a istanza singola con volume persistente: durabilità contro la perdita del disco, nessun failover contro la perdita del nodo. È una decisione di scope deliberata — nessuno di quei contesti è il core del prodotto, e concentrare l'investimento in fault tolerance dove serve davvero era preferibile a replicare tutto superficialmente.

Recovery

Checkpoint e replay: sopravvivere al crash del processo.

L'unica eccezione al «tutto in MongoDB»

Mentre una sessione è aperta, il suo stato autoritativo vive nella memoria dell'istanza cadService che la ospita: cambia molte volte al secondo e tutti i partecipanti devono vedere la stessa vista in-process. In questo caso MongoDB fa da checkpoint recuperabile, non da storage primario.

1. **Checkpoint ogni 10 s**

   Un timer di background salva lo snapshot di ogni sessione viva e i Lamport clock.

2. **Event log fra un checkpoint e l'altro**

   Ogni edit accettato è appeso a un log per sessione: al massimo 10 s possono mancare dallo storage durevole.

3. **Al riavvio: carica, scarta, ricostruisce**

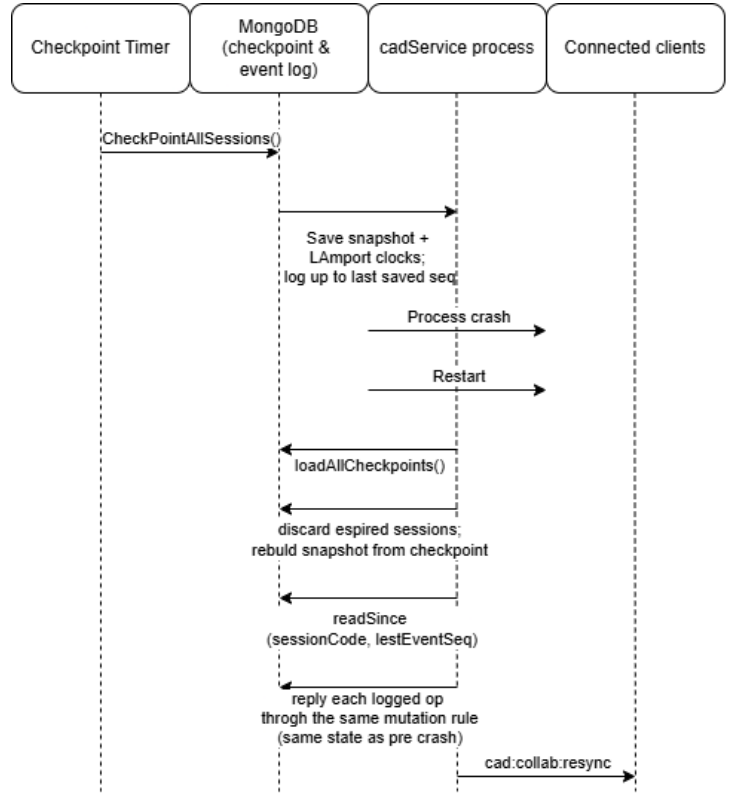
   Carica ogni checkpoint persistito, scarta quelli con TTL scaduto, ricostruisce la sessione in memoria.

4. **Replay con la stessa funzione di mutazione**

   Le voci di log successive sono riapplicate dall'identica funzione usata per gli edit live: nessun percorso di recovery che possa divergere.

5. **cad:collab:resync**

   I client connessi vengono avvisati e rileggono lo snapshot.



Checkpoint periodico più event log: il processo riavviato ricostruisce lo stato pre-crash.

Deployment

Docker Compose

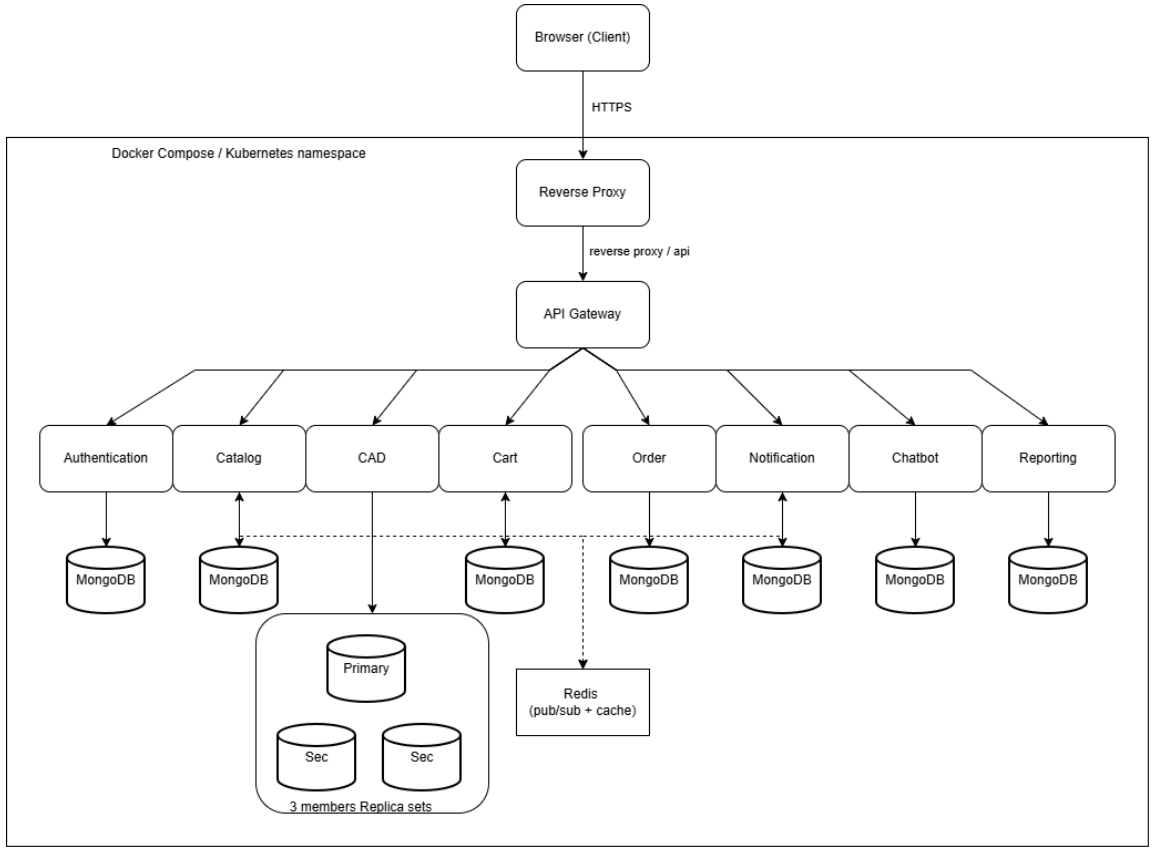
- Ambiente di sviluppo e base per la suite end-to-end
- Volume nominato e health check per ogni MongoDB
- Replica set CAD inizializzato da un job one-shot

Kubernetes · Minikube

- Manifest numerati per ordine di dipendenza, un solo kustomization.yaml
- Gateway e frontend sono gli unici NodePort; tutto il resto è ClusterIP
- Readiness probe per pod; nessuna liveness probe (limite noto)

Osservabilità

Grafana, Loki e Promtail: i log dei nove componenti backend si filtrano e si correlano da un'unica dashboard. Senza log aggregation, seguire un flusso che attraversa gateway, catalog, Redis e notification è impraticabile.



Ogni box diverso dal client è un container (Compose) o un pod (Kubernetes) dentro un'unica rete logica.

Trade-off

Semplificazioni deliberate

Nessuna cache di lettura

Redis è presente ma usato solo come event bus: ogni lettura va al database del servizio proprietario.

Una sola replica per Deployment

Il tier stateless è pronto a scalare orizzontalmente, ma i manifest sono fissati a replicas: 1.

Pub/sub at-most-once

Nessuna persistenza dei messaggi non consegnati; mitigato dall'idempotenza, non risolto.

Un confine piegato di proposito

Il notification service legge direttamente le collection di cart e CAD per risolvere «quali utenti hanno questo SKU». Sola lettura, ma è una violazione consapevole di database-per-service.

Prossimi passi, in ordine di impatto

1. **Pipeline di CI**

   Eseguire l'intera baseline a ogni push. È la singola aggiunta più utile.

2. **Suite di test sul frontend**

   Sostituire il walkthrough manuale sui percorsi core.

3. **Retry e liveness probe**

   Le chiamate fra servizi sono a tentativo unico; senza liveness probe Kubernetes non recupera un pod vivo ma bloccato.

4. **Scalare per davvero**

   Alzare i replica count, aggiungere un autoscaler, e risolvere l'affinità di sessione del cadService.

5. **TLS, validazione a schema, rate limiting**

   I tre punti di hardening già identificati.

DEMO