Backend sito e-commerce

Romina Gryka (1993783) Matteo Marcelletti(1986290) Thomas Sharp(1986413)

Relazione per il progetto d'esame di
Ingegneria del software

Sapienza Università di Roma
Dipartimento di Informatica
Novembre 2024

1 Descrizione generale

L'obiettivo del progetto è quello di costruire e gestire un sistema di e-commerce, con funzionalità per la gestione di utenti,prodotti,ordini,carrelli e spedizioni.
Il sistema 'e organizzato nei seguenti moduli:

- **dbutils**: contiene la logica per interagire con il database PostgreSQL e la cache Redis. Si occupa di creare e gestire il database, le tabelle, e la gestione della cache.
- **server**: contiene funzioni utili in tutto il progetto,gestendo il flusso delle richieste e risposte.
- **test-generator**: contiene i test che verificano la corretta funzionalità del sistema, incluse le operazioni di database, la gestione della cache e del server. Questi test aiutano a garantire che il sistema funzioni in modo affidabile e senza errori.

2 User requirements

2.1 Amministratore

- Aggiungere, eliminare o modificare gli utenti
- Visualizzare informazioni degli utenti
- Popolare il database con dati di test
- Testing delle funzionalità
- Gestione degli errori

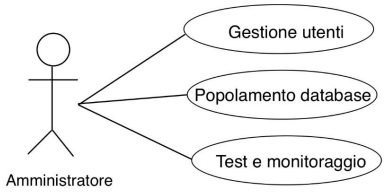


Figure 1: Use Cases per requisiti Amministratori

2.2 Customer

- Aggiungere prodotti al carrello
- Rimuovere prodotti dal carrello
- Confermare il contenuto del carrello e acquistarlo
- Visualizzare ordini passati

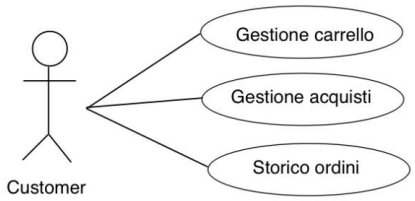


Figure 2: Use Cases per requisiti Customer

2.3 Supplier

- Associare prodotti al proprio utente
- Assegnare prezzi, quantità e descrizioni ai prodotti

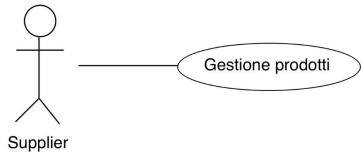


Figure 3: Use Cases per requisiti Supplier

2.4 Shipper

- Ottenere le spedizioni a proprio carico

- Aggiornare lo stato di una spedizione
- Collegare una spedizione a un ordine
- Assegnare le spedizioni da effettuare

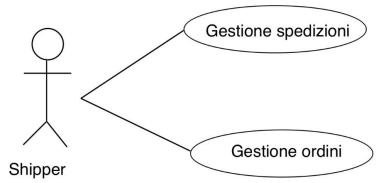


Figure 4: Use Cases per requisiti Shipper

3 System requirements

I requisiti del sitema vanno a descrivere le aspettative e le tecniche necessarie per garantire che il sistema funzioni correttamente, in modo tale da soddisfare le necessità degli utenti.

3.1 Gestione del database

Il sistema deve essere in grado di interagire efficacemente con il database, eseguire operazioni di lettura e scrittura, e gestire i dati degli utenti e degli oggetti coinvolti nel processo.

3.1.1 Connessione al database

Il sistema deve poter stabilire una connessione stabile e sicura a un database PostgreSQL.
La funzione getConnection (presente nel file pgutils.hpp) è utilizzata per creare una connessione al database.

3.1.2 Operazioni di scrittura nel database ed accesso ai dati

Il sistema deve supportare l'inserimento di dati nel database in modo efficiente e senza errori.
I dati devono essere gestiti in transazioni, in modo che, in caso di errore, l'integrità del database venga preservata.

Ogni operazione di scrittura, come l'inserimento di dati nelle tabelle users, orders, products, e shippings, è gestita attraverso l'uso della libreria pqxx::work. Il sistema deve essere in grado di recuperare e visualizzare dati dal database.

3.1.3 Gestione delle eccezioni

Il sistema deve essere in grado di gestire le eccezioni che potrebbero verificarsi durante l'esecuzione delle query al database.

3.1.4 Popolamento del Database con Dati di Test

Il sistema deve permettere di popolare il database con dati fittizi o casuali, in modo tale da poter testare il funzionamento delle funzionalità.

3.2 Gestione delle operazioni per gli utenti

3.2.1 Visualizzazione dei prodotti

Il sistema deve permettere agli utenti di visualizzare i prodotti disponibili, inclusi i dettagli come nome, prezzo e quantità disponibile. Le operazioni di visualizzazione dei prodotti sono gestite tramite query al database che recuperano le informazioni sui prodotti.

3.2.2 Aggiungere prodotti al carrello e creare ordini

Gli utenti devono poter aggiungere prodotti al carrello e procedere con l'ordine. Questo processo implica la gestione delle transazioni sul database, dove un ordine viene registrato e associato ai prodotti selezionati.

3.2.3 Gestione degli ordini e delle spedizioni

Gli utenti devono poter visualizzare lo stato dei propri ordini e le informazioni sulla spedizione. Il sistema deve consentire di recuperare e visualizzare lo stato degli ordini, aggiornare i dettagli di spedizione e gestire il flusso di informazioni.

3.3 Testabilità delle funzionalità del sistema

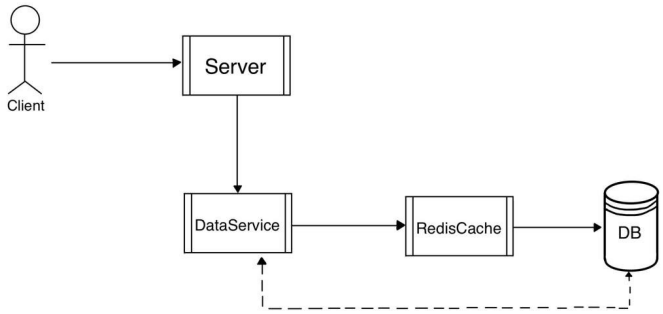
Gli utenti devono poter simulare scenari reali generando dati di test che popoleranno il database, come utenti, ordini, spedizioni, e prodotti.
La funzione populateDB(int n) è utilizzata per inserire un numero predefinito di record nel database, così da consentire test con un set di dati realistico.

3.4 Diagramma dell'architettura del sistema

L'archietettura del sistema è composta nel seguente modo:

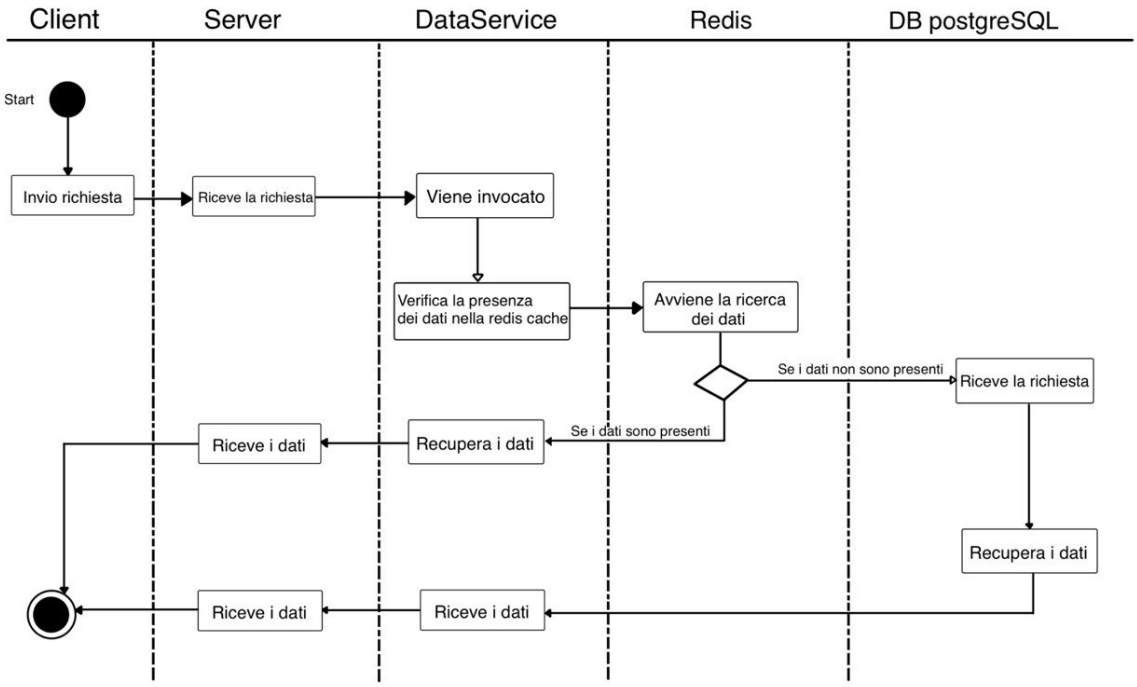
- **Server**:riceve richeste dai clienti, gestisce queste richieste e si occupa di comunicare con i servizi interni per soddisfarle.

- **DataService**:interfaccia tra la cache e il database nel caso di una cache miss.Se i dati non sono nella cache,li recupera dal database,li memeorizza in redis e li restituisce al server.
- **Redis cache**:implementa il pattern Cache-Aside, verificando se i dati richiesti sono già presenti nella cache per accelerare l'accesso.In caso di cache miss, si connette al datababse.
- **PostgreSQL Database**:contiene le tabaelle strutturate e risponde alle query inviate da DataService in caso di necessità.



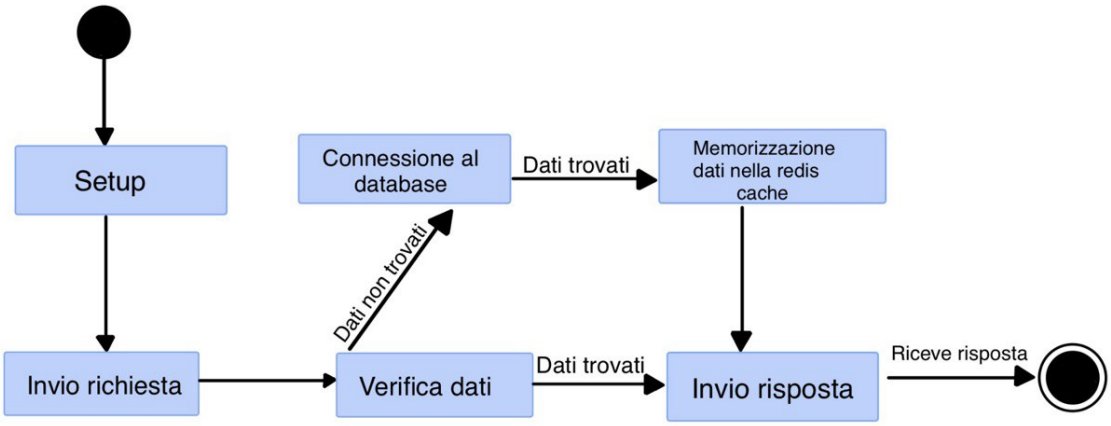
3.5 Activity diagram

Di seguito viene riportato l'activity diagram rappresentante un flusso di operazioni che l'applicazione esegue per soddisfare una richiesta da parte del client. Le operazioni includono verifiche, interrogazioni, memorizzazioni e restituzioni dei dati.



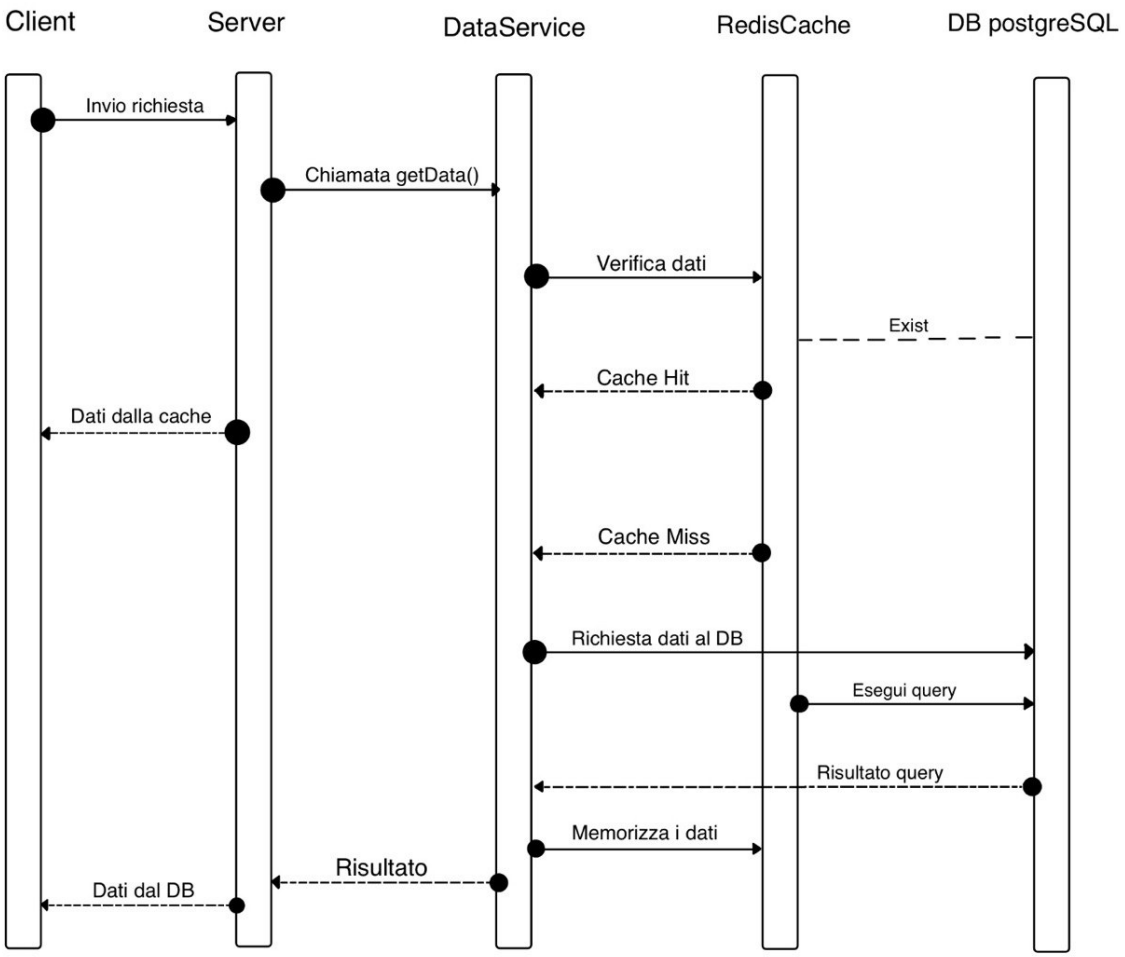
3.6 State diagram

Lo state diagram descrive come le richieste di dati vengono gestite attraverso vari stati, partendo dalla verifica nella cache, passando per l'eventuale interrogazione del database, fino alla memorizzazione dei dati nella cache per ottimizzare le risposte future.
Ogni stato del diagramma riflette un'azione concreta nel codice, come la chiamata a getData(), la verifica dei dati in RedisCache, l'esecuzione della query sul Database e la memorizzazione dei dati nella RedisCache.



3.7 Message Sequence Chart

Di seguito viene riportato il message sequence chart relativo all'invio di una richiesta da parte di un client,andando a visualizzare l'interazione tra i vari componenti del sistema in modo dettagliato ed ordinato, osservando l'elaborazione e l'invio della risposta da parte del server.



4 Implementazione

Il sistema si articola in varie fasi,come prima cosa il codice si connette ad un database PostgreSQL,dove vengono memorizzati i dati relativi agli utenti, prodotti, ordini, carrelli e spedizioni.
I dati degli utenti e delle entità sono generati in modo casuale attraverso delle funzioni (getRandomNames(), getRandomSurnames(), getRandomCities()) e allo stesso modo vengono creati ordini, prodotti e spedizioni.

Il sistema permette all'utente di navigare nel database, popolare i dati con numeri casuali o testare alcune funzionalità del sistema, ciò accade ad esmpio con la funzionalità testCustomer().

Una funzione molto importatnte è **populateDB(int n)** che si occupa di popolare il database con dati generati casualmente, utilizzando diverse tabelle.
Lo scopo principale di questa funzione è simulare la creazione di dati reali per testare il sistema, e lo fa in più fasi, ognuna delle quali popola una sezione specifica del database.
La funzione inizia creando una connessione al database PostgreSQL tramite l'oggetto pqxx::connection, utilizzando il metodo getConnection,poi viene popolata la tabella users, che contiene le informazioni di base degli utenti e vengono generati n utenti.
La seconda parte della funzione riguarda la creazione di 3 diversi tipi di entità: customers, shippers, e suppliers.Dopo aver creato gli utenti, vengono aggiunti i prodotti al sistema per poi aggiungere degli ordini per i clienti.Successivamente, per ogni ordine viene associato uno o più prodotti e la funzione completa il processo inserendo le informazioni di spedizione per ogni ordine.

Oltre al database, viene utilizzato anche Redis,per gestire dati temporanei o per operazioni di caching.

4.1 Testing

Il file `test-generator.cpp` è stato sviluppato per simulare un ipotetico front-end che interagisce con il back-end del sistema. Questo componente è stato fondamentale per testare le funzionalità principali, simulando richieste realistiche, incluse situazioni di errore o richieste malevole.

La funzione principale, `testCustomer`, permette di eseguire test su diverse operazioni comuni svolte dai clienti. In particolare, include tre tipi di test configurabili tramite un vettore di flag `selected`:

- **addProductToCart**: verifica l'aggiunta di prodotti al carrello di un cliente selezionato casualmente. Viene creato un oggetto `Customer`, il cui carrello viene popolato con un massimo di 5 prodotti selezionati casualmente e con quantità variabili.
- **removeProductFromCart**: verifica la rimozione di prodotti dal carrello. Per ogni cliente selezionato casualmente, vengono eliminati fino a 5 prodotti con quantità casuali dal carrello esistente.
- **buyCart**: simula l'acquisto del carrello. Il cliente, selezionato casualmente, completa l'acquisto di tutti gli articoli presenti nel carrello.

Durante l'esecuzione, il sistema utilizza una combinazione di PostgreSQL e Redis per gestire i dati. La classe `DataService` e la cache Redis (`RedisCache`) sono state integrate per gestire le operazioni sul carrello e ottimizzare l'accesso

ai dati. Inoltre, la funzione consente di testare l'applicazione con diversi numeri di richieste (`n`) per valutare le performance in condizioni di carico variabile.

Questi test hanno lo scopo non solo di validare le funzionalità previste, ma anche di analizzare la robustezza del sistema nel gestire input non validi o comportamenti malevoli, garantendo così la sicurezza e l'affidabilità del back-end.

5 Risultati

Durante la fase sperimentale del progetto, abbiamo valutato le performance del sistema confrontando tre configurazioni principali: l'**assenza di Redis**, l'utilizzo di Redis in modalità **cache-aside**, e l'utilizzo di Redis in modalità **init cache**. I test sono stati eseguiti variando il numero di record generati dalla funzione populateDB(), che simula l'inserimento di dati nel database.

I risultati ottenuti mostrano come l'adozione di Redis offra un miglioramento significativo delle performance, specialmente quando il numero di dati cresce. In particolare:

- **Senza Redis**: il sistema ha tempi di risposta elevati dovuti all'accesso diretto al database per ogni richiesta, causando un sovraccarico proporzionale al numero di record.
- **Con Redis in modalità cache-aside**: i tempi di risposta migliorano sensibilmente grazie alla cache, con un overhead limitato alla prima richiesta per ogni dato non ancora in cache.
- **Con Redis in modalità init cache**: il sistema mostra il miglioramento più consistente. Sebbene ci sia un overhead iniziale per il caricamento di tutti i dati nella cache, questa configurazione garantisce tempi di accesso costanti e minimi durante l'esecuzione successiva.

I risultati dei test, sintetizzati nella tabella sottostante, evidenziano chiaramente il beneficio dell'uso di Redis, con un guadagno medio di oltre il 30% nei tempi di esecuzione rispetto alla configurazione senza cache:

n	Senza Redis (ms)	Cache-aside (ms)	Init Cache (ms)
1.000	487	312	259
10.000	5.078	3.125	2.904
100.000	49.863	30.492	28.376
1.000.000	503.127	318.948	279.453

Table 1: Performance comparativa con e senza Redis al variare del numero di dati.

Questi dati dimostrano come l'integrazione di una cache ben configurata non solo migliori le performance del sistema, ma offra anche flessibilità nella gestione del carico in base alle esigenze dell'applicazione.