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.
- **rdutils**: contiene la classe che gestisce la cache, e le varie operazioni su di essa. Oltre al gestore della cache c'è la classe che permette la memorizzazione delle tabelle del DB sulla 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

2.2 Customer

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

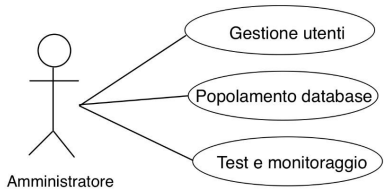


Figure 1: Use Cases per requisiti Amministratori

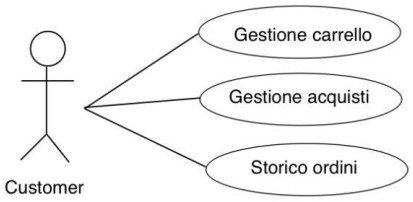


Figure 2: Use Cases per requisiti Customer

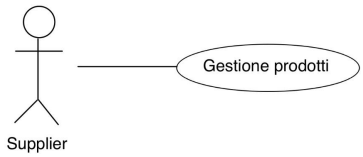


Figure 3: Use Cases per requisiti Supplier

2.3 Supplier

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

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

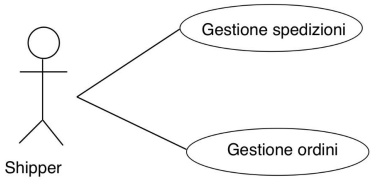


Figure 4: Use Cases per requisiti Shipper

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.

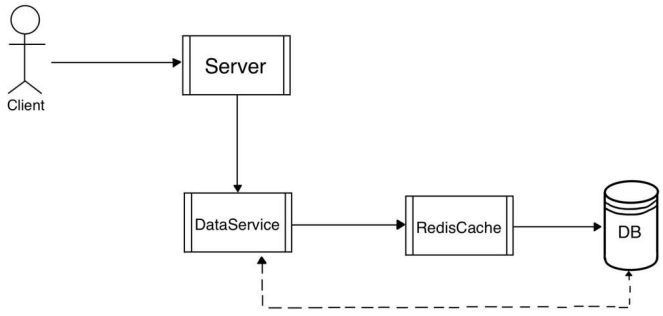
3.3 Testabilità delle funzionalità del sistema

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

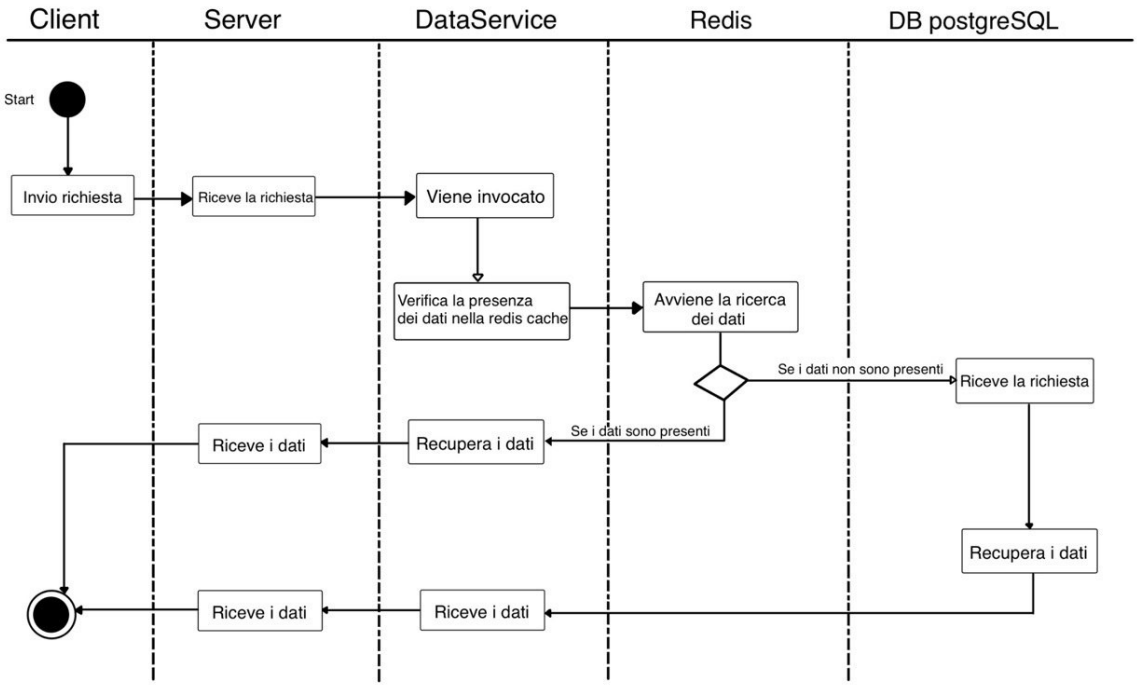
L'archietettura del sistema è composta nel seguente modo:

- **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.



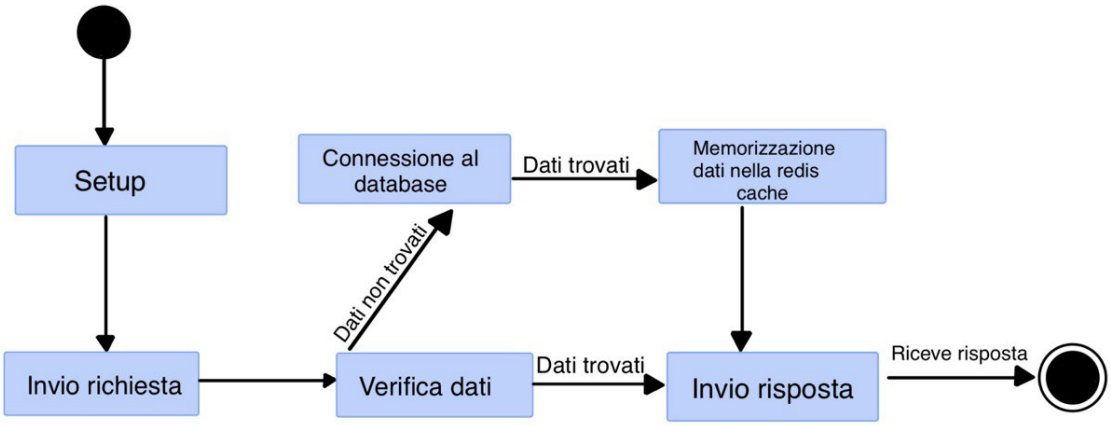
3.5 Activity diagram

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.



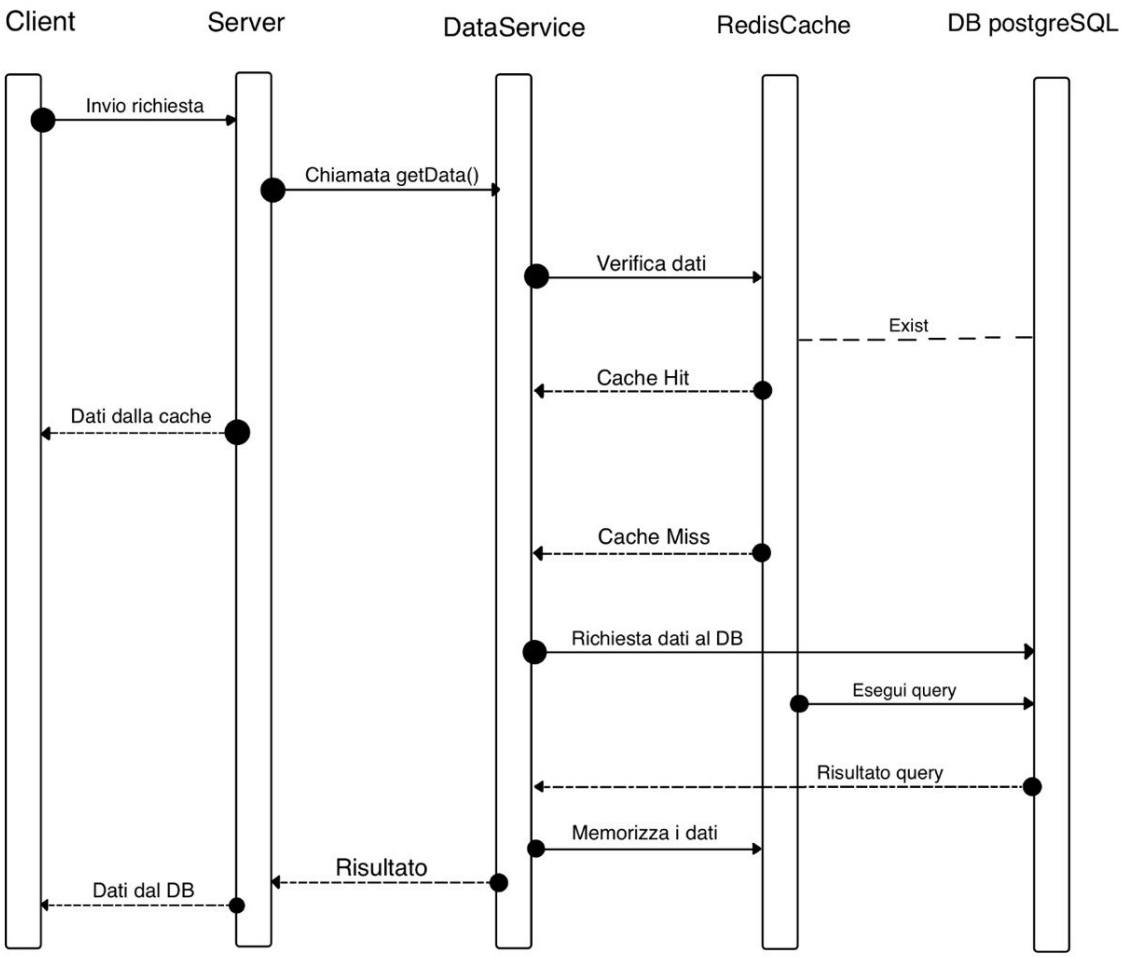
3.6 State diagram

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().

4.1 Database

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.

4.2 Redis

Dopo il popolamento del database avviene il primo utilizzo di Redis come sistema di cache. Essendo Redis una cache, è necessario scegliere come implementare la gestione dei dati in cache, adottando uno dei due principali approcci: *cache-aside* o *always cached*. Nel nostro caso, abbiamo deciso di implementare entrambi i metodi a seconda del dato o della tabella che dobbiamo memorizzare.

Ad esempio, successivamente al popolamento del database, la tabella dei prodotti viene caricata in Redis utilizzando l'approccio *always cached*, in quanto è quella a cui si accede più frequentemente.

4.2.1 Metodi di caching

I due metodi di caching hanno entrambi vantaggi e svantaggi che li rendono più o meno adatti a seconda del contesto.

Cache-aside In questo approccio, viene implementato un sistema di *hit or miss* sulla cache come conseguenza delle richieste provenienti dal client:

- In caso di *hit*, i dati richiesti vengono restituiti direttamente dalla cache, senza necessità di interpellare il database.

- In caso di *miss*, viene effettuata una connessione al database e una query per la ricerca dei dati. Se i dati sono presenti nel database, vengono caricati in cache e poi restituiti al client.

Always cached In questo approccio, una certa tabella può essere continuamente mantenuta in cache. I dati vengono recuperati direttamente e costantemente dalla cache, migliorando le prestazioni in lettura. Tuttavia, è necessario implementare un sistema di aggiornamento parallelo tra il database e Redis per garantire la coerenza dei dati. Questo implica che ogni modifica nel database debba essere riflessa immediatamente nella cache.

4.2.2 Rappresentazione dei dati in Redis

Redis mantiene i dati in un formato *Key:Value*. Essendo i dati che vogliamo memorizzare tuple delle tabelle del database, uno dei problemi principali da affrontare era trovare un modo semplice e veloce per memorizzare e recuperare queste tuple utilizzando i comandi di Redis, come `GET(Key)`.

La soluzione adottata è stata rappresentare i dati come una stringa, facile da memorizzare e da leggere. In questo modo, la coppia *Key:Value* diventa:

`'NomeTabella'+'IDTupla' : 'Campo1_Campo2_...'`

dove:

- `'NomeTabella'+'IDTupla'` costituisce la *Key*, combinando il nome della tabella e l'ID della tupla.
- `'Campo1_Campo2_...'` costituisce il *Value*, contenente i campi della tupla, separati dal carattere _. L'ID non viene incluso nei campi, poiché è già presente nella *Key*.

Questa rappresentazione permette di recuperare velocemente i dati tramite la chiave e di leggere le informazioni in modo strutturato. La scelta di utilizzare una stringa semplifica la gestione e l'interoperabilità tra il database e Redis.

4.3 Sistema di login

Il sistema di login implementato nel nostro progetto è volutamente semplice e può essere definito un sistema "dummy". L'idea alla base è quella di associare un *token* univoco ad ogni utente che accede alla piattaforma, salvandolo in una cache basata su Redis per velocizzare l'accesso alle informazioni.

Il processo di login inizia controllando se l'utente ha già un token associato; in caso contrario, viene generato un nuovo token utilizzando la funzione `generateToken()` e memorizzato in Redis tramite la funzione `set()`.

La funzione di generazione del token crea una stringa alfanumerica casuale di lunghezza predefinita, sfruttando un generatore di numeri casuali.

Sebbene questo approccio sia utile per scopi dimostrativi, esso semplifica notevolmente la complessità di un reale sistema di login. In un'applicazione

concreta, un sistema di login robusto dovrebbe gestire direttamente le credenziali degli utenti, affrontando diverse problematiche legate alla sicurezza, tra cui:

- **Archiviazione sicura delle password:** è necessario memorizzare le password degli utenti in un formato non leggibile, utilizzando funzioni di hashing crittografiche sicure come SHA, bcrypt o MD5.
- **Utilizzo di salt unici:** ogni password deve essere associata a un salt univoco per prevenire attacchi basati su dizionari precomputati (ad esempio, le *rainbow tables*).
- **Aggiornamento degli algoritmi di hashing:** con il progresso tecnologico, alcuni algoritmi di hashing possono diventare obsoleti. È quindi necessario prevedere meccanismi per migrare gli hash delle password a metodi più sicuri nel tempo.
- **Protezione contro attacchi brute force:** occorre implementare sistemi per limitare il numero di tentativi di accesso consecutivi, come il blocco temporaneo degli account o l'uso di CAPTCHA.
- **Gestione delle vulnerabilità dei database:** nel caso in cui un database venga compromesso, è fondamentale che i dati rubati (password, hash e salt) siano inutilizzabili senza un'ulteriore fase di attacco molto complessa.

Considerando la complessità e i rischi associati alla gestione di credenziali e dati sensibili, in un progetto reale sarebbe preferibile affidarsi a servizi di autenticazione di terze parti, come quelli offerti da Google, Facebook o altri provider. Questi servizi utilizzano protocolli standard consolidati come OAuth 2.0 o OpenID Connect e delegano la gestione della sicurezza a sistemi altamente specializzati e continuamente aggiornati. Questo approccio, oltre a ridurre il rischio di vulnerabilità, migliora anche l'esperienza utente, permettendo un accesso rapido e sicuro attraverso autenticazione federata.

4.4 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.

4.4.1 Test Customer

La funzione `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.

4.4.2 Test Shipper

La funzione `testShipper` è progettata per testare diverse funzionalità legate ai trasportatori e alle spedizioni in un sistema di e-commerce.

- **assignUnassignedOrders**: simula il processo di assegnazione di ordini "non assegnati" a trasportatori disponibili,serve per verificare che il sistema sia in grado di trovare trasportatori disponibili e assegnargli gli ordini in attesa.
- **newShipping**: simula la creazione di una nuova spedizione per un ordine specifico,collegandolo ad un trasportatore.Testa la capacità del sistema di collegare ordini e trasportatori,verificando che tutto funzioni correttamente in scenari realistici.
- **trasportatore disponibile**: controlla se i trasportatori selezionati sono disponibili per accettare nuovi ordini,garantendo al sistema di distinguere tra trasportatori liberi e occupati,evitando di sovraccaricare chi è già impegnato.
- **shippingDelivered**: simula il processo di aggiornamento di una spedizione per indicare che è stata completata con successo,verificando che il sistema possa tracciare lo stato delle spedizioni e aggiornarlo in modo accurato.
- **getActiveShippings**: mostra l'elenco delle spedizioni che un trasportatore sta ancora gestendo e non ha ancora completato e controlla che il sistema tracci correttamente le spedizioni in corso per ogni trasportatore.
- **getShippings**: recupera la lista di tutte le spedizioni associate a un trasportatore,incluse quelle già completate. Serve per verificare che il sistema mantenga uno storico completo delle spedizioni gestite dai trasportatori.

4.4.3 Test Supplier

La funzione `testSupplier` permette di eseguire test su diverse operazioni comuni svolte dai fornitori. In particolare, include vari tipi di test configurabili tramite un vettore di flag `selected`, che consente di scegliere quali funzionalità specifiche testare:

- **addStock**: verifica l'aggiunta di una certa quantità allo stock di un prodotto. Viene generato un ID casuale per un produttore e per un prodotto, insieme a una quantità casuale compresa tra 0 e 1000. Successivamente, la funzione richiama `addStock` per aggiornare gli stock di quei prodotti per i relativi produttori.
- **addProduct**: verifica l'aggiunta di un prodotto a un certo produttore. Viene generato l'ID di un fornitore, insieme ai campi di un prodotto, per procedere con la creazione del prodotto e la sua associazione al fornitore selezionato.
- **setDiscontinuedProduct**: verifica che impostare un prodotto come discontinuo funzioni correttamente. Anche in questo caso, vengono generati un ID di produttore e un prodotto per testare la funzionalità.
- **getSoldProducts**: simula il recupero della lista dei prodotti venduti da un certo produttore, ordinati in ordine decrescente in base al timestamp di vendita del prodotto.

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.

4.5 Gestione dei monitor: registrazione degli errori

Per garantire una corretta gestione dei monitor e la tracciabilità degli errori nel nostro progetto, abbiamo implementato una funzione denominata `logError`, che si occupa di registrare eventuali messaggi di errore su un file di log dedicato. La funzione sfrutta il timestamp corrente per associare ogni errore a un momento preciso, garantendo così una chiara cronologia degli eventi. Il funzionamento della funzione può essere riassunto nei seguenti passi:

- **Ottenimento del timestamp:** viene utilizzata la libreria standard `ctime` per ottenere la data e l'ora corrente nel formato `YYYY-MM-DD HH:MM:SS`, convertendo i dati in una stringa utilizzabile per il log.

- **Scrittura nel file di log:** gli errori vengono registrati nel file `error.log`, situato nella directory dei monitor (`../monitors/`). La modalità `ios::app` garantisce che i nuovi errori vengano aggiunti in coda al file senza sovrascrivere i dati esistenti.

Questo approccio ci permette di centralizzare la gestione degli errori, offrendo un'unica fonte di verità per diagnosticare problemi durante l'esecuzione del sistema. Il file di log funge da monitor passivo, consentendo di analizzare retroattivamente il comportamento del sistema e identificare eventuali anomalie.

Riteniamo che questa implementazione rispecchi l'obiettivo principale della gestione dei monitor, ovvero fornire una visibilità chiara e continua sullo stato del sistema. Tuttavia, siamo consapevoli che in un'applicazione più complessa potrebbe essere necessario estendere il sistema con ulteriori funzionalità, come:

- **Rotazione dei log:** per evitare che il file di log cresca in modo eccessivo, si potrebbe implementare un sistema di rotazione automatica con backup periodici.
- **Monitoraggio attivo:** oltre alla registrazione passiva degli errori, si potrebbe integrare un sistema di notifiche per avvisare in tempo reale gli amministratori in caso di problemi critici.
- **Analisi e reportistica:** utilizzare strumenti esterni o librerie per analizzare i log e generare report utili al miglioramento continuo del sistema.

La funzione `logError` rappresenta quindi una soluzione efficace per il contesto del nostro progetto, ma resta aperta a futuri miglioramenti in caso di necessità di maggiore scalabilità o automazione.

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.