UNIVERSITA' DEGLI STUDI DI TORINO

Dipartimento di Informatica

Corso di Laurea Magistrale in Informatica

Anno Accademico 2023/2024

Laboratorio di Modelli e Architetture Avanzati di Basi di Dati

Autore: Annalisa Sabatelli Matr. 866879

"Scuderia Online E-commerce"

Documentazione di progetto

1. Descrizione del progetto

Il progetto consiste nella realizzazione di un'applicazione per la gestione degli acquisti online di articoli per l'equitazione. L'applicativo è costituito da un'area pubblica sostanzialmente composta dalla landing page in cui è possibile visionare i prodotti commercializzati, le informazioni generali sull'azienda, i campi di login e/o la registrazione e un'area privata a cui l'utente accede dopo il login. Considerando il progetto nella sua totalità, a prescindere dal sottoinsieme di funzionalità effettivamente implementate e che verranno specificate in sezioni successive di questo documento, l'applicazione deve gestire due tipologie di utenti: un utente di amministrazione e un utente cliente. A seconda della tipologia di utente l'area privata offre funzionalità differenti. L'utente cliente può consultare e/o modificare le proprie informazioni di anagrafica, gestire il proprio carrello prodotti aggiungendo o

eliminando prodotti, procedere all'acquisto emettendo un ordine ed effettuando il pagamento, consultare lo storico di tutti gli ordini effettuati. L'utente di amministrazione ha la piena visibilità su tutti i dati e su tutte le funzionalità gestite dall'applicativo. Può consultare gli ordini emessi ordinandoli per data di emissione o per stato dell'ordine (in preparazione/in consegna/consegnato); può aggiungere nuovi prodotti da commercializzare; può consultare le informazioni degli utenti registrati; può consultare lo stato delle giacenze a magazzino per ciascun prodotto. L'utente di amministrazione, inoltre, ha la possibilità di consultare statistiche sugli acquisti in termini, per esempio, di quantità acquistate per ogni prodotto, importo medio di un ordine, totale fatturato settimanale o mensile.

Le tabelle seguenti contengono l'elenco delle funzionalità descritte sopra specificandone l'utente di riferimento. L'ultima colonna indica le funzionalità comprese nella porzione di sistema effettivamente implementato in questo lavoro.

	Funzionalità Utente Cliente	Stato implementazione
1	Login	✚
2	Registrazione	✖
3	Gestione anagrafica utente	✖
4	Gestione carrello acquisti	✚
5	Controllo stato ordini	✚
6	Gestione acquisti e pagamenti	✚

Tabella 1 : Funzionalità utente cliente

	Funzionalità Amministratore	Stato implementazione
1	Login	✖
2	Gestione anagrafica utenti	✖
3	Gestione carrello acquisti	✖
4	Gestione prodotti acquistabili	✖
5	Controllo stato giacenze magazzino	✖
6	Controllo stato ordini	✖
7	Controllo statistiche di vendita	✖

Tabella 2: Funzionalità utente di amministrazione

L'elaborato è completamente focalizzato sulle funzionalità dell'utente cliente.

2. Funzionalità e casi d'uso

Sulla base delle funzionalità individuate nel paragrafo precedente, si dettagliano i casi d'uso relativi alle funzionalità utente cliente. Si è deciso di non trattare nel dettaglio i casi d'uso relativi alla gestione di anagrafiche, in quanto essi comprendono fondamentalmente le attività di inserimento, aggiornamento e cancellazione tipiche della gestione di una qualsiasi anagrafica. Più interessante, invece, è la funzionalità di gestione del carrello acquisti e di gestione acquisti e pagamenti.

Informazioni Generali	
Nome	**Gestione carrello acquisti**
Portata	Sistema
Livello	Obiettivo utente
Attore Primario	Utente cliente
Parti interessate	
Pre-condizioni	L'attore si è autenticato e vuole aggiungere elementi al suo carrello acquisti
Post-condizioni	Il carrello acquisti è aggiornato con i nuovi oggetti selezionati

Scenario principale di successo

#	Attore	Sistema
1	Seleziona un oggetto tra quelli acquistabili	Visualizza le informazioni di dettaglio dell'oggetto selezionato
2	Indica quantità e/o taglia e richiede l'aggiunta al carrello	Registra le informazioni inserite
3	*Ripete il passo 1 finché non è soddisfatto*	
4	Opzionalmente visualizza il proprio carrello acquisti	Fornisce l'informazione richiesta
5	Opzionalmente, effettua logout dal sistema	Visualizza la pagina pubblica
	Termina caso d'uso	

Eccezione 3a

#	Attore	Sistema
3a.1	Indica quantità e/o taglia e richiede l'aggiunta al carrello	Il sistema segnala che la quantità richiesta eccede lo stock disponibile
3a.2	Effettua le correzioni necessarie	Visualizza l'informazione inserita
	Ripete il passo 1	

Eccezione 3b

#	Attore	Sistema
3b.1	Indica quantità e/o taglia e richiede l'aggiunta al carrello	Il sistema segnala che la quantità richiesta eccede lo stock disponibile
3b.2	Annulla l'operazione	Cancella eventuali dati inseriti e visualizza l'elenco prodotti acquistabili
	Ripete il passo 3	

Estensione 1a

#	Attore	Sistema
1a.1	Visualizza il carrello acquisti	Fornisce l'informazione richiesta
1a.2	Seleziona un oggetto da eliminare dal carrello e richiede l'eliminazione	Registra l'eliminazione
	Ripete il passo 1a.2 finché non è soddisfatto	
	Termina caso d'uso	

Eccezione 1a.1a

#	Attore	Sistema
1a.1a.1	Visualizza il carrello acquisti	Il sistema segnala che il carrello acquisti è vuoto
	Termina caso d'uso	

Estensione 2a

#	Attore	Sistema
2a.1	Visualizza il carrello acquisti	Fornisce l'informazione richiesta
2a.2	Richiede l'eliminazione del carrello	Registra l'eliminazione
	Termina caso d'uso	

Eccezione 2a.1a

#	Attore	Sistema
2a.1a.1	Visualizza il carrello acquisti	Il sistema segnala che il carrello acquisti è vuoto
	Termina caso d'uso	

Informazioni Generali	
Nome	**Gestione acquisti e pagamenti**
Portata	Sistema
Livello	Obiettivo utente
Attore Primario	Utente cliente
Parti interessate	Utente amministrativo
Pre-condizioni	L'attore si è autenticato e vuole acquistare gli oggetti presenti nel suo carrello
Post-condizioni	Gli oggetti acquistati sono eliminati dal carrello acquisti e viene creato un ordine

Scenario principale di successo

#	Attore	Sistema
1	Visualizza il carrello acquisti	Fornisce l'informazione richiesta
2	Seleziona gli oggetti da acquistare e ne richiede l'acquisto	Visualizza le informazioni di spedizione e pagamento
3	Inserisce l'indirizzo di spedizione	Visualizza l'informazione aggiornata
4	Sceglie il metodo di pagamento e conferma	Registra l'operazione e crea l'ordine
	Termina caso d'uso	

Eccezione 1a

#	Attore	Sistema
1a.1	Visualizza il carrello acquisti	Il sistema segnala che il carrello acquisti è vuoto
	Termina caso d'uso	

Eccezione 2a

#	Attore	Sistema
2a.1	Seleziona gli oggetti da acquistare e ne richiede l'acquisto	Il sistema segnala che la quantità richiesta eccede lo stock attualmente disponibile
2a.2	Effettua le correzioni necessarie	Visualizza l'informazione inserita
	Riprende al passo 3	

Eccezione 2b

#	Attore	Sistema
2b.1	Seleziona gli oggetti da acquistare e ne richiede l'acquisto	Il sistema segnala che il prodotto non è più disponibile
2b.2	Deseleziona il prodotto	Visualizza l'informazione aggiornata
	Riprende al passo 3	

Estensione 2b

#	Attore	Sistema
2a.1	Effettua il logout senza procedere all'acquisto	Visualizza l'area pubblica
	Termina caso d'uso	

3. Modellazione dati ed architettura database

3.1. Casi d'uso e caratteristiche dei dati

Sulla base delle funzionalità individuate nel paragrafo precedente, si possono definire i bisogni informativi e, di conseguenza, l'architettura effettiva del data store.

Dalla lettura dei casi studi emergono le seguenti esigenze informative:

- Anagrafica cliente comprensiva dei dati di accesso;
- Anagrafica prodotti commercializzati;
- Dati su ordini emessi per ciascun cliente;
- Dati del carrello utente.

In uno scenario di questo tipo, sono presenti diverse condizioni per le quali è consigliabile l'utilizzo di un database relazionale. I dati sono ben strutturati ed organizzati e possono essere facilmente suddivisi in tabelle con colonne e righe (ad esempio, una tabella per i clienti, una per gli ordini, una per i prodotti, ecc.) e c'è una relazione ben definita tra di esse (ad esempio, un ordine è associato a un cliente). In sistemi di e-commerce e/o gestione degli inventari, inoltre, la consistenza del dato e la sua durabilità, specie se riferita ad ordini e pagamenti, sono delle caratteristiche cruciali, al pari dei sistemi bancari, ed è necessario supportare transazioni multiple che devono essere gestite in modo atomico, coerente, isolato e durevole (proprietà ACID).

Considerazioni diverse possono essere fatte relativamente ai dati del carrello utente. La gestione del carrello in un'applicazione di e-commerce deve garantire un'esperienza utente fluida e consistente. Quando si integra un sistema di carrello con un database, le scelte architetturali e le caratteristiche dipendono dal flusso di interazione con l'utente, dalle necessità di performance, di coerenza dei dati e di scalabilità. La descrizione del caso d'uso relativo alla gestione del carrello utente, esprime l'esigenza di una persistenza dei dati del carrello tra sessioni diverse, consentendo di tornare al carrello anche a distanza di tempo. In questo caso specifico, infatti, non è consentito a utenti non registrati di effettuare acquisti e, quindi, i dati relativi al carrello non devono essere legati ad un identificativo di sessione ma piuttosto ad un identificativo utente in modo da poter essere recuperati ogni volta che l'utente effettua un login. D'altra parte, a questa esigenza di persistenza, si aggiunge una forte esigenza di dinamismo, responsività (responsiveness) e disponibilità. Rispetto alla sostanziale staticità dei dati contenuti in un'anagrafica cliente o prodotto, le operazioni sul carrello di aggiunta, aggiornamento o rimozione devono essere rapide ed efficienti: quando un utente aggiunge o rimuove un prodotto, il sistema deve aggiornare rapidamente il carrello senza impattare negativamente sull'esperienza utente. E' importante che il carrello dell'utente sia sempre disponibile, anche in caso di guasti parziali del sistema, per evitare che gli utenti perdano i dati del loro carrello. La reattività e la disponibilità diventano prestazioni anche più importanti della consistenza. A quanto detto si

deve, inoltre, aggiungere un'esigenza di scalabilità. In un e-commerce, il traffico può aumentare improvvisamente, ad esempio durante offerte speciali o eventi stagionali, ed è auspicabile disporre di un database che possa scalare verso l'alto e verso il basso facilmente per gestire picchi di accesso e richieste simultanee da molti utenti.

Queste considerazioni giustificano l'esigenza di differenziare il tipo di database per la gestione delle anagrafiche da quello per la gestione del carrello utente affiancando, così, al database relazionale uno NoSQL che possa rispondere meglio alle esigenze presentate per la gestione del carrello acquisti.

L'utilizzo di più tipi di database in uno stesso applicativo a seconda delle esigenze specifiche di ciascuna parte prende il nome di approccio poliglotta. Questa strategia può offrire vari vantaggi ma anche alcuni svantaggi. Considerando che ciascun database, specie se NoSQL, è progettato per eccellere in determinati scenari, questo approccio consente l'ottimizzazione di casi d'uso specifici come nel caso di questo lavoro. Un database NoSQL, soprattutto se scelto coerentemente con le caratteristiche dei dati gestiti e delle interrogazioni, è spesso più scalabile orizzontalmente rispetto ai tradizionali database relazionali e permette una minore rigidità nella strutturazione del dato. D'altra parte, gestire diversi tipi di database in un'unica applicazione può introdurre una complessità significativa. L'utilizzo di più database può complicare la gestione della consistenza dei dati, soprattutto se i dati vengono replicati tra sistemi diversi. Ad esempio, in un sistema distribuito, garantire la coerenza tra database relazionali e NoSQL potrebbe risultare problematico. Integrare e sincronizzare dati tra diversi tipi di database può essere difficile. Le differenze nei modelli di dati, nelle prestazioni e nelle modalità di accesso potrebbero creare difficoltà, soprattutto in scenari complessi in cui le transazioni devono essere gestite tra database differenti. Un'attenta suddivisione dei dati tra i vari database può contenere queste problematiche. Nel caso in esame, i dati sono stati divisi tra database relazionale e NoSQL in modo che da evitare repliche e sovrapposizioni e quindi risolvere a priori i problemi di gestione di consistenza.

3.2. Database relazionale

Si riporta di seguito lo schema del database relazionale utilizzato nell'applicativo di e-commerce "Scuderia Online E-commerce". In particolare, tra tutti i database relazionali open source è stato utilizzato PostgreSQL. Le funzionalità messe a disposizione dell'utente tramite l'interfaccia grafica di PgAdmin permettono di estrapolare facilmente una rappresentazione grafica della struttura del database.

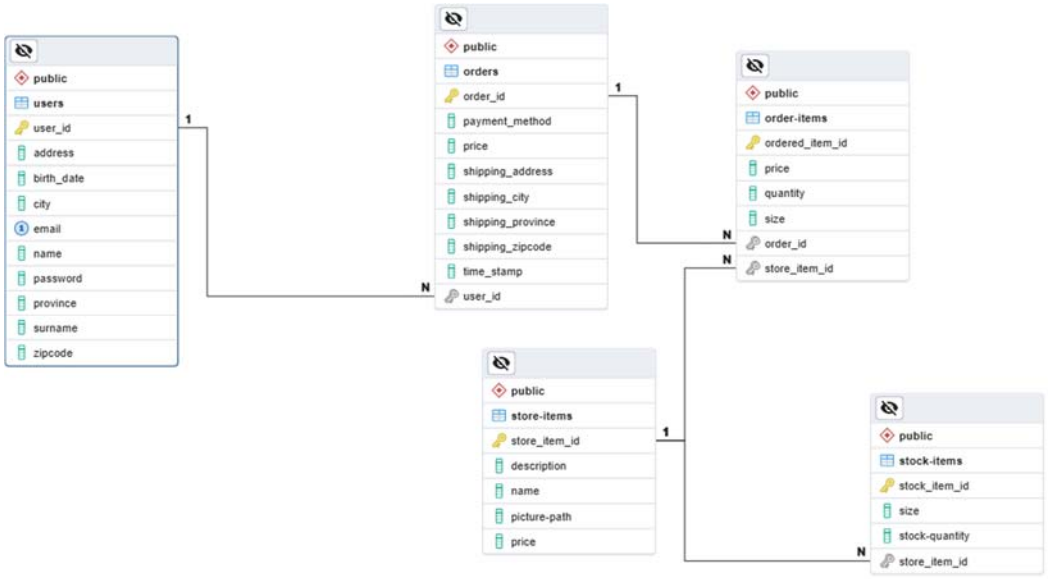

Figura 1: Scuderia Online E-commerce - Schema database relazionale

Si analizzano, ora, nel dettaglio le caratteristiche di tabelle e relazioni.

1. **Tabella USERS:** contiene i dati degli utenti registrati.

   Attributi:

   - user_id (PK): Identificatore univoco dell'utente;
   - Informazioni personali: name, surname, email, password, birth_date;
   - Indirizzo: address, city, province, zipcode.

   Relazioni:

   - Collegata a ORDERS tramite la relazione 1:N (un utente può avere più ordini).

NOTE: In una versione più estesa e completa dell'applicativo, potrebbe essere utile aggiungere una colonna per la data di registrazione (registration_date) ed una per gestire la cancellazione logica degli utenti.

2. **Tabella ORDERS:** contiene informazioni sugli ordini effettuati dagli utenti.

   Attributi:

   - order_id (PK): Identificatore univoco dell'ordine;
   - user_id (FK): Collegamento all'utente che ha effettuato l'ordine;
   - Dettagli dell'ordine: price, status, time_stamp (data e ora dell'ordine);
   - payment_method, shipping_address, shipping_city, shipping_zipcode, shipping_province.

   Relazioni:

   - Collegata a USERS tramite la relazione 1:N;
   - Collegata a ORDERED_ITEM tramite la relazione 1:N (un ordine può contenere più articoli).

3. **Tabella STORE_ITEM:** Contiene i dati degli articoli disponibili nello store.

   Attributi:

   - store_item_id (PK): Identificatore univoco dell'articolo;
   - Informazioni: name, description, picture-path (per immagini), price.

   Relazioni:

   - Collegata a ORDERED_ITEM tramite una relazione 1:N (un articolo può essere ordinato in più ordini);
   - Collegata a STOCK-ITEMS tramite una relazione 1:N (un articolo può avere diverse varianti di stock).

4. **Tabella ORDERED_ITEM:** Tabella associativa per gestire gli articoli ordinati.

   Attributi:

   - ordered_item_id (PK): Identificatore univoco della riga;
   - order_id (FK): Collegamento all'ordine a cui appartiene l'articolo;
   - store_item_id (FK): Collegamento all'articolo ordinato;
   - Dettagli: price, quantity (quantità ordinata).

Relazioni:

- Collegata a ORDERS e STORE-ITEMS per gestire l'associazione tra ordini e articoli.

5. **Tabella STOCK-ITEMS:** Gestisce le varianti di magazzino per gli articoli.

   Attributi:

   - stock_item_id (PK): Identificatore univoco dello stock;
   - store_item_id (FK): Collegamento all'articolo nello store;
   - Dettagli dello stock: size (taglia o variante), stock-quantity (quantità disponibile).

   Relazioni:

   - Collegata a store-items tramite la relazione 1:N (un articolo può avere più varianti di stock).

Il dabase così progettato rispetta i principi della normalizzazione fino alla terza forma normale: i dati sono ben strutturati, non ci sono dipendenze parziali o transitive ed è stata evitata la ridondanza.

3.3. Database NoSQL

3.3.1. Analisi delle alternative disponibili

L'obiettivo di questa sezione è quello di definire nel dettaglio la struttura dati del carrello che si vuole gestire nell'applicazione e, sulla base delle considerazioni esposte nei paragrafi precedenti, individuare un database di tipo NoSQL che possa supportare correttamente la gestione di un carrello acquisti garantendo responsività, disponibilità e scalabilità.

Nel caso dell'applicazione specifica, si vuole che il carrello acquisti possa sopravvivere attraverso diverse sessioni utente. In pratica, un utente può effettuare login, inserire oggetti nel carrello, effettuare il logout senza finalizzare l'acquisto e avere comunque la possibilità di recuperare il contenuto del carrello acquisti al login successivo. È evidente che il carrello acquisti è legato all'identificativo utente e non all'identificativo di sessione. Il carrello acquisti conterrà una lista di item ciascuno con delle caratteristiche che possono essere variabili da item ad item. Per esempio, mentre per un capo di abbigliamento è importante

indicare la taglia, questa può non far parte degli attributi di altre categorie di oggetti. Una volta finalizzato l'acquisto, il carrello è trasformato in un ordine e viene rimosso dal database. Si potrebbe affermare che la persistenza del carrello è, per così dire, "momentanea" fino all'emissione dell'ordine. Un comportamento di questo tipo è molto più vicino ad un caching più che ad una persistenza.

La letteratura propone diverse soluzioni di database NoSQL per supportare un carrello acquisti di un e-commerce la cui scelta dipende dalle caratteristiche dell'applicazione e delle prestazioni che si vogliono assicurare. In questo elaborato, si è scelto di optare per un database key-value (chiave-valore) che possa assicurare elevate prestazioni in lettura e scrittura, una gestione semplice e veloce di sessioni temporanee o dati che non richiedono relazioni complesse e la possibilità di una scalabilità orizzontale per gestire un numero crescente di utenti.

La chiave è rappresentata dall'identificativo utente, mentre il valore è il vettore di item inseriti nel carrello.

Tra i database chiave-valore Redis e Riak sono probabilmente i più diffusi ed utilizzati.

La seguente tabella mette a confronto le caratteristiche principali di questi due database.

	REDIS	RIAK
Tipo di Database	Key-Value Store in-memory con persistenza opzionale	Key-Value Store distribuito e altamente disponibile
Architettura	L'architettura di Redis supporta due modalità principali per la gestione dei dati in ambienti distribuiti: repliche (Replication) e sharding (Redis Cluster). Il Redis Cluster non segue esattamente un'architettura peer-to-peer ma include una gerarchia master-slave su un sottoinsieme di nodi.	E' un sistema distribuito con un'architettura peer-to-peer, progettato per alta disponibilità
Persistenza dei Dati	Persistenza opzionale tramite snapshots RDB o AOF (Append-Only File)	Persistenza automatica, replica dei dati tra i nodi
Consistenza	Eventual consistency	Eventual consistency, ma configurabile per una consistenza più forte
Velocità	Molto veloce, in-memory, latenza bassissima	Più lenta rispetto a Redis, progettato per alte performance in ambienti distribuiti
Scalabilità	Scalabilità orizzontale con Redis Cluster per letture scrittura e con le Repliche per la sola lettura	Scalabilità orizzontale automatica
Gestione della Disponibilità	Alta disponibilità sia con le Repliche che con Redis Cluster	Alta disponibilità e tolleranza ai guasti, replica automatica
Supporto per Strutture Dati	Strutture dati avanzate: stringhe, hash, set, liste, sorted sets	Supporto per dati chiave-valore, non supporta strutture avanzate come Redis
Gestione Sessioni Utente	Perfetto per sessioni utente, grazie alla velocità e alla possibilità di impostare il TTL (se il Time To Live è impostato a -1 viene disabilitato il timer di scadenza)	Supportato, ma con maggiore complessità rispetto a Redis
Tolleranza ai Guasti	Supportata, ma dipende dalla configurazione del clustero di Redis Sentinel per le repliche	Alta tolleranza ai guasti
Latente e Performance	Latenza molto bassa grazie alla gestione in memoria	Latenza più alta rispetto a Redis, ottimizzato per la disponibilità
Casi d'Uso Tipici	Caching, gestione sessioni utente, carrelli di acquisto, analisi in tempo reale, caching	Sistemi distribuiti ad alta disponibilità, IoT, e-commerce

Tabella 3: Database NoSQL Key Value-Redis e Riak a confronto

Un altro confronto interessante nella selezione dei due database può essere quello relativo al loro posizionamento rispetto al teorema del CAP (Consistency-Availability e Partition tollerance).

Proprietà	REDIS(Replica/Cluster)	RIAK
Consistenza (C)	Consistenza eventuale si in Replica che in Cluster	Consistenza eventuale, con vector clocks per la risoluzione
Disponibilità (A)	Alta ma vulnerabile in Cluster con perdita di molti nodi	Altissima, anche in presenza di guasti o partizioni
Partition Tolerance (P)	Garantita	Garantita
Posizionamento CAP	**AP (Availability + Partition Tolerance)**	**AP (Availability + Partition Tolerance)**

Tabella 4:Database NoSQL Key Value-Redis e Riak a confronto rispetto al teorema del CAP

Dato che il carrello acquisti modellato in questo elaborato non contiene dati critici tali da richiedere una consistenza forte e considerando anche le limitate esigenze di scalabilità, si è preferito adottare REDIS in modo da garantire elevate prestazioni in termini di reattività e bassa latenza, una configurazione semplice senza precludere la possibilità eventuale di impostare architetture distribuite sia tramite le Repliche che lo Sharding.

La figura seguente descrive la struttura dati di massima del carrello acquisti.
L'esempio è preso dall'applicativo oggetto di questo lavoro.

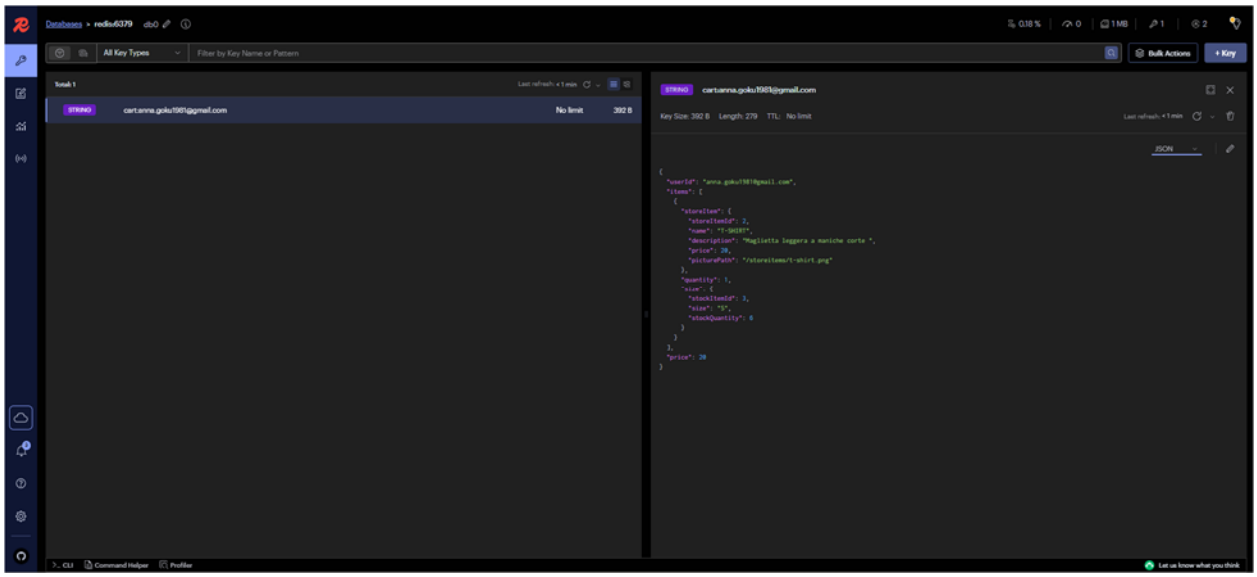

Figura 2:REDIS client- Struttura dati carrello acquisti

Per interagire con REDIS è stato utilizzato un suo client, in particolare REDIS Insight. La figura mostra alcuni parametri di Redis già accennati in precedenza. Il Time To Live (TTL) è impostato a "NO LIMIT" in modo da non attribuire una scadenza alle informazioni depositate in REDIS e garantire la persistenza.

Il path chiave-valore è del tipo:

chiave→ cart:<id_utente>

valore→Json che rappresenta il carrello

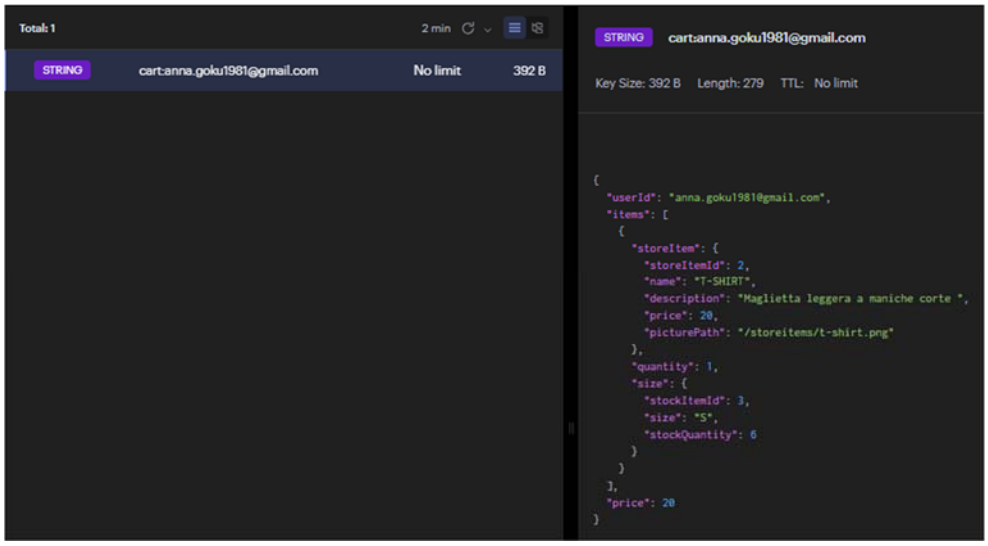

Figura 3:Redis client- Focus su path chiave-valore

La parte di value è composta da un Json che in sostanza è un vettore di "items" i cui elementi hanno a loro volta una struttura annidata su più livelli e un importo totale del carrello. Ogni elemento del vettore "items" è composto da "storeItem", "quantity" e "size".

Figura 4: Struttura dati carrello acquisti- Elementi del vettore "items"

A sua volta “storeItem” presenta un secondo livello di annidamento come mostrato in figura 4.

3.3.2. Linguaggio di interrogazione e query principali

Redis utilizza un linguaggio di interrogazione basato su comandi chiamato Redis CLI Commands, che permette di interagire direttamente con il database. A differenza dei database SQL, Redis non ha un linguaggio di interrogazione complesso e non permette di realizzare query articolate. La sintassi specifica con la quale comunicare con il server REDIS può cambiare in base alle librerie messe a disposizione dal linguaggio di programmazione utilizzato per il backend.

Nel caso specifico dell’applicativo oggetto di questo lavoro, Spring Boot offre il supporto nativo tramite Spring Data Redis che fornisce un'API ad alto livello per interagire con Redis. Il RedisTemplate è una classe fornita da Spring Data Redis che funge da interfaccia principale per interagire con Redis e permette di inviare comandi al server Redis senza dover gestire direttamente la connessione o scrivere il protocollo Redis. RedisTemplate è progettato per interagire con Redis utilizzando metodi specifici per ciascuna struttura dati. Preliminarmente, la classe RedisTemplate deve essere opportunatamente configurata ed istanziata.

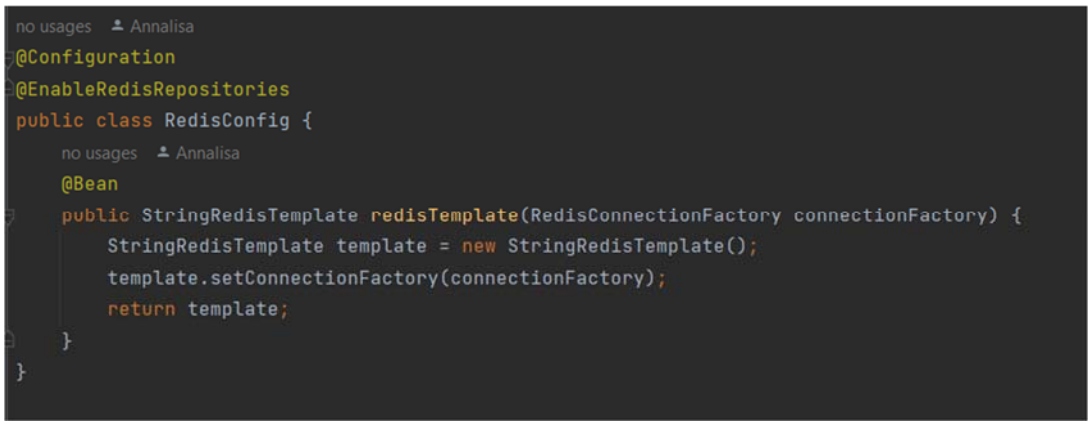

Figura 5: REDIS Template - Configurazione

La gestione del carrello nel suo ciclo di vita e nelle varie interazioni con l’utente ha richiesto le azioni primitive REDIS di GET, SET e DELETE.

A titolo di esempio si riporta il sorgente si riporta la GET del carrello utente.

GET http://localhost:8080/api/ecommerce/v1/cart/diana.goku1981@gmail.com

```java
@GetMapping("/cart/{mail}")
public ResponseEntity<ShoppingCart> getCartByMail(@NotNull @PathVariable String mail) throws JsonProcessingException {
    checkUser(mail);

    ShoppingCart c = cartService.getCart(mail);
    if(c == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    else {
        return new ResponseEntity<>(c, HttpStatus.OK);

    }
}
```

Figura 6:Chiamata REST - GET carrello utente

In questa GET viene passato come parametro la mail dell'utente che è la chiave della coppia chiave-valore nel data store REDIS creato e sulla quale si basa il recupero del valore associato. La funzione "cartService.getCart(mail)" realizza l'interrogazione vera e propria al server REDIS utilizzando il Redis Template.

```java
@Autowired
private StringRedisTemplate redisTemplate;

1 usage  Annalisa
public ShoppingCart getCart(String email) throws JsonProcessingException {
    String json = redisTemplate.opsForValue().get(ShoppingCart.REDIS_KEY_PREFIX+email);
    log.info("JSON LETTO DA REDIS: "+json);
    if(json == null) {
        return null;
    }
    else {
        ObjectMapper mapper = new ObjectMapper();
        ShoppingCart c = mapper.readValue(json, ShoppingCart.class);
        log.info("CARRELLO: "+c);
        return c;

    }
}
```

Figura 7: Esempio di utilizzo della classe Redis Template

Essendo un database chiave-valore, REDIS utilizza le chiavi per accedere ai valori memorizzati. Il metodo opsForValue() è specifico per interagire con le chiavi di tipo String. La GET restituisce il Json associato alla chiave data in input e in questo caso è proprio il contenuto del carrello associato all'utente passato come parametro. Il valore attribuito alla variabile statica *REDIS_KEY_PREFIX="cart:"* della classe ShoppingCart concatenato con la mail dell'utente passata in input riscostruisce il path corretto per la chiave salvata in REDIS.

Infine, la classe ObjectMapper converte il Json in un oggetto java in modo da poter manipolare agevolmente i dati a backend.

3.3.3. Architettura e scalabilità

Si procede, ora, con un focus su le possibili architetture supportate da REDIS.

REDIS supporta architetture dati distribuite. Un REDIS cluster può essere configurato con nodi di replica e con nodi di sharding rispondendo all'esigenza di fault tollerance e di scalabilità. La figura seguente illustra l'architettura di un REDIS Cluster con shard e repliche.

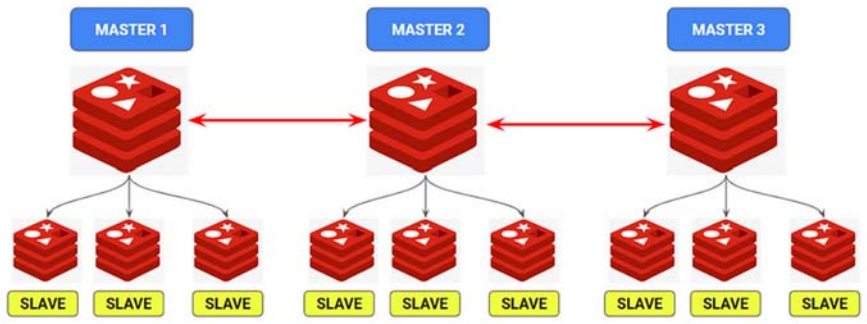

Figura 8: Architettura REDIS- Redis Cluster

Ci sono più nodi master (in questo caso 3) e ciascun master può avere una o più replica. I dati sono suddivisi tra i nodi master ottenendo scalabilità nella memorizzazione dei dati e nelle scritture, dato che ciascun nodo Master è l'unico responsabile per l'elaborazione delle richieste di scritture. Se un nodo master va in fault, una delle sue repliche viene promossa a master e questo aumenta la disponibilità del database anche in caso di guasti ai nodi.

Un’altra possibile configurazione è composta solamente da repliche.

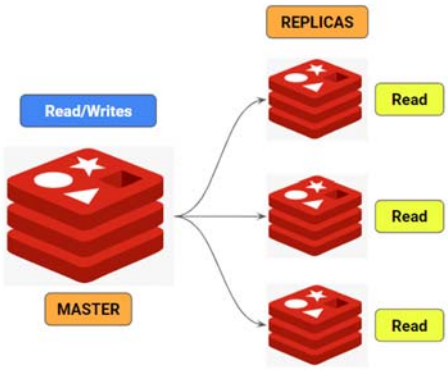

Figura 9: Architettura REDIS- Redis Repliche

In questa configurazione, il Master rimane l’unico point of failure per le scritture, mentre le letture possono essere gestite indifferentemente dalle varie repliche. È una configurazione particolarmente utile in scenari dove l'affidabilità e la continuità del servizio sono cruciali ma non si ha una forte necessità di scalabilità orizzontale.

In questo elaborato, è stato utilizzato un unico server REDIS. La scelta è stata dettata dalle dimensioni e dalle caratteristiche dell’applicazione e soprattutto dai limiti imposti dall’hardware disponibile. Un applicativo di e-commerce di prodotti per l’equitazione può essere considerato un’applicazione “di nicchia” di prodotti non critici con un numero di utenti limitato ai soli appassionati. Sulla base delle conoscenze del settore, non si prevedono grosse esigenze di scalabilità orizzontale. Prestazioni più critiche, come già detto, possono essere quelle più direttamente connesse con la user experience come l’alta disponibilità e la bassa latenza. Con lo schema di dati attuale è possibile, in ogni caso, creare repliche e sfruttare i vantaggi che queste offrono.

Pur utilizzando un unico server, in realtà, REDIS permette comunque di creare più nodi su di esso sia di replica che di shard attraverso la definizione di un REDIS Cluster virtuale.

Tuttavia, se la creazione di più nodi in un unico server può essere una strategia ammissibile in una fase di sviluppo, è sconsigliabile applicarla in produzione poiché l’utilizzo di un unico server non permette di sfruttare a pieno i vantaggi di un REDIS Cluster. Eseguire più nodi Redis su un singolo server ha limiti significativi, specialmente per quanto riguarda CPU, memoria, disco e affidabilità. L’unico server diventerebbe, inoltre, un singolo punto di errore (single point of failure) il cui fallimento porterebbe tutti i nodi Redis offline. La configurazione single server con più nodi, in pratica, annulla di fatto tutti i vantaggi garantiti da una “vera” architettura distribuita.

4. Architettura dell'applicazione

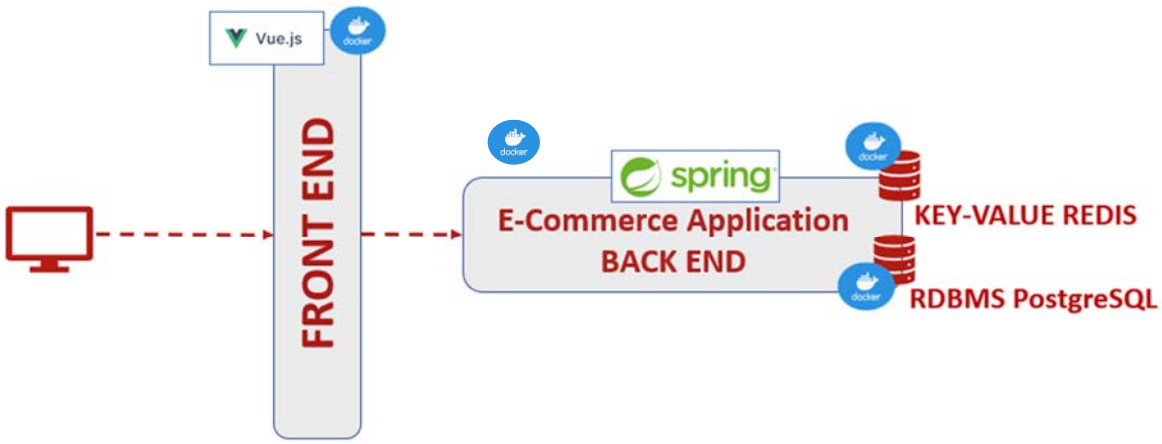

Figura 10:Application Architecture- Tecnologie utilizzate

L'applicazione è basata su Spring Boot (Java) con il quale è stato realizzato il backend. Spring Boot è ben integrato sia con PostGreSQL che con Redis. In entrambi i casi, infatti, i moduli di Spring come "Spring Data JPA" per PostgreSQL e "Spring Data Redis", se aggiunti correttamente al file delle dipendenze, semplificano l'integrazione e l'interazione con i database. Simile a Spring Data JPA, Spring Data Redis permette di usare repository per interagire con Redis, permettendo operazioni CRUD (Create, Read, Update, Delete) e offrendo un'interfaccia di alto livello per lavorare con Redis senza utilizzare i comandi nativi.

Il frontend dell'applicazione "Scuderia on-line" è una single page implementata con il framework Vue.js.

Il progetto è composto da un set di container Docker indipendenti così articolati:

- un container per il back end Spring boot;
- un container per i database in PostgreSQL;
- un container per il database REDIS;
- un container per il front end implementato utilizzando il framework Vue.js.

L'utilizzo di un container Docker per REDIS semplifica la creazione di più nodi replica. Nel file Docker Compose, ad ogni nodo replica viene assegnata una porta e un volume.

L'entry point per un utente è un sito web disponibile all'indirizzo: http://localhost:5173/

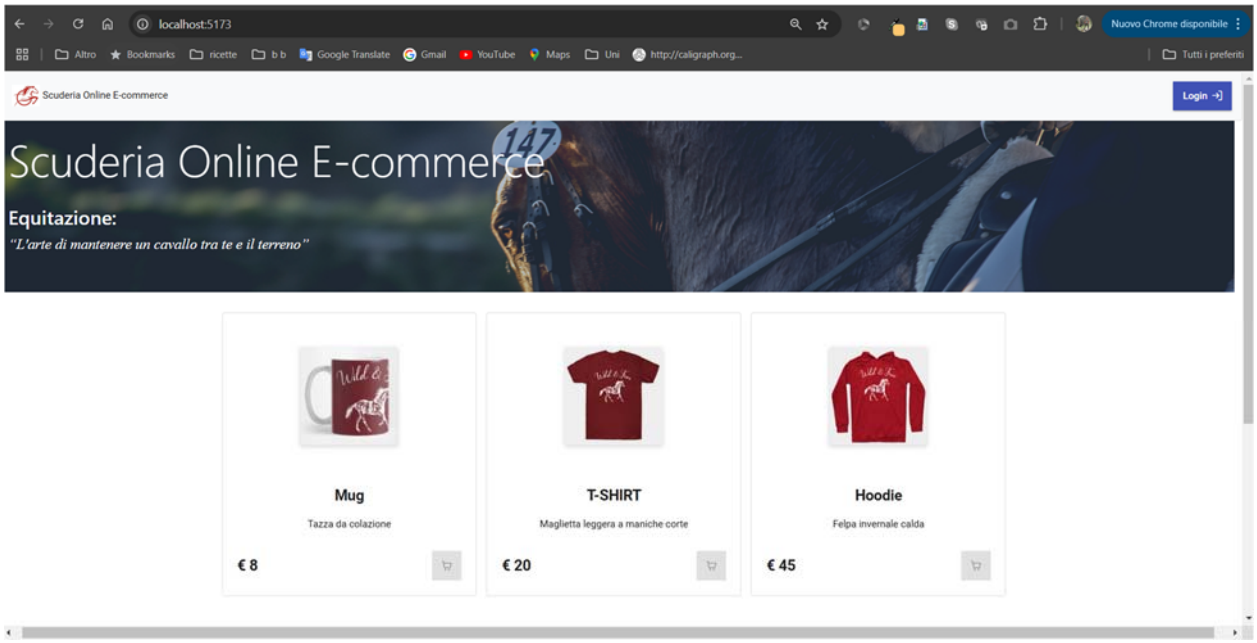

Figura 11:Scuderia Online E-commerce- Home page

Per eseguire questa applicazione è necessario installare due strumenti: Docker e Docker Compose. Esistono diversi file di installazioni di Docker a seconda del sistema operativo dell'host.

L'intera applicazione può essere eseguita con un singolo comando da terminale partendo dalla directory «maad-lab»:

```
$ docker-compose up -d
```

Allo stesso modo la sua esecuzione può essere fermata tramite il comando:

```
$ docker-compose down
```

Il codice sorgente è contenuto nel repository github al seguente link https://github.com/annagoku/maad-lab.