



UNIVERSITÀ DI PISA

---

---

SCUOLA DI INGEGNERIA  
Dipartimento Ingegneria dell'Informazione  
Corso LARGE-SCALE AND MULTI-STRUCTURED DATABASES  
MSc in ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING

**REPORT TASK 1**

**Designing and Implementing a simple JAVA  
application connecting to  
a relational DB using JPA  
and  
Feasibility Study on the use of  
a Key-Value Data Storage**

PRESENTATO DA:

CARUSO Alberto  
TUMMINELLI Gianluca  
Di NOIA Antonio  
CALABRESE Pietro

Anno Accademico  
2019/2020

---

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Analisi del sistema</b>	<b>2</b>
2.1	Identificazione degli attori . . . . .	2
2.2	Casi d'uso . . . . .	2
2.3	Diagramma Classi . . . . .	3
<b>3</b>	<b>Implementazione</b>	<b>4</b>
3.1	Database . . . . .	6
3.2	Hibernate e JPA . . . . .	7
3.2.1	JPA Entities . . . . .	7
3.2.1.1	EventoDb . . . . .	7
3.2.1.2	OrganizzatoreDb . . . . .	10
3.2.1.3	PartecipanteDb . . . . .	13
3.3	Operazioni CRUD . . . . .	16
3.3.1	Create . . . . .	16
3.3.2	Read . . . . .	17
3.3.3	Update . . . . .	17
3.3.4	Delete . . . . .	18
<b>4</b>	<b>Studio di Fattibilità Database Key-Value</b>	<b>19</b>
4.1	Introduzione . . . . .	19
4.1.1	LevelDB . . . . .	20
4.2	Scenario 1 : MIGRAZIONE COMPLETA . . . . .	21
4.2.1	Modellazione dei buckets . . . . .	21
4.2.2	Modellazione delle chiavi . . . . .	22
4.2.3	Considerazioni sulle caratteristiche del sistema . . . . .	22
4.2.4	Conclusioni . . . . .	23
4.3	Scenario 2: APPROCCIO IBRIDO . . . . .	23
4.3.1	Modellazione dei database . . . . .	24
4.3.2	Considerazioni sulle caratteristiche del sistema . . . . .	26
4.3.3	Conclusioni . . . . .	28
4.3.4	Implementazione Database Key-Value . . . . .	28
4.3.4.1	Operazione Lettura da LevelDB . . . . .	28
<b>5</b>	<b>Manuale d'uso</b>	<b>30</b>
5.1	Login . . . . .	30
5.2	Registrazione nuovo utente . . . . .	31
5.3	Funzioni Organizzatore . . . . .	32

---

5.3.1	Creazione evento . . . . .	32
5.3.2	Visualizza eventi creati . . . . .	33
5.4	Funzioni Partecipante . . . . .	34
5.4.1	Pagina Utente . . . . .	34
5.4.2	Ricerca eventi . . . . .	35
5.4.3	Visualizza eventi . . . . .	36



---

## 1 Introduzione

La pratica della vita quotidiana è intrecciata con il digitale, in particolare i media mobili, e questo si estende anche alla ricerca di eventi vicini alla proprie passioni. Siti e app per la ricerca di questo tipo di contenuti, i servizi che supportano la ricerca di eventi sono sempre più numerosi sviluppati per dispositivi mobili, ma in realtà sono supportati da altri social network non sviluppati con questa finalità ma adattati allo scopo. Il boom dei social network negli ultimi anni ha alimentato l'hype nel condividere le proprie esperienze, ma allo stesso tempo ha creato un gap importante nella reale aggregazione al fine di cerare contenuti utili alla condivisione. Attualmente, si può venire a conoscenza di un evento attraverso le funzioni messe a disposizione dai social network, ad esempio, Facebook, ma la conoscenza di questa informazione non implica che si possa partecipare all'evento prenotando il proprio posto o azioni simili, inoltre c'è da considerare i dati generati anche da un punto di vista analitico ed economico, infatti essi ci possono dare una stima molto approssimativa dell'evento e delle persone che ipoteticamente possano prendere parte a questo, in quanto molto spesso il cliccare su "partecipa a evento" risulta essere un comportamento modaiolo e non effettivamente legato ai propri interessi, infine resta da considerare che alcuni social network molto attivi negli anni passati, risultano essere fuori dai trend del momento, a questo dunque, si associa un possibile scarsa visibilità del nostro contenuto. Lo scopo principale di questo lavoro è sviluppare una web app mobile oriented, che possa rendere la gestione di un evento molto più customer oriented, con questo si includono anche tutti i servizi legati alla partecipazione, ad esempio riservare il posto per l'evento o il pagamento del costo del biglietto. A questo si aggiungono tutte le funzionalità riservate all'organizzatore per monitorare l'andamento di un particolare evento, o in generale tutti gli eventi da esso promossi, per successivamente prendere decisioni strategiche sul proprio business e sulle campagne di marketing da attuare. Resta comunque fermo che la pubblicizzazione degli eventi non può prescindere dai canali di diffusione più usati al momento.

---

## 2 Analisi del sistema

In questa sezione si analizza in modo approfondito le scelte tecniche, i requisiti, le funzionalità e gli attori del progetto.

### 2.1 Identificazione degli attori

In questa sezione, ci soffermiamo nel definire gli attori che sono stati identificati, e che possono interagire con il sistema. Questi due attori sono: **Organizzatore** e **Partecipante**, ognuno di questi ha ruoli e operazioni differenti performabili sugli eventi.

**Organizzatore** – gestore di un’attività o un suo delegato – che al fine di promuovere la propria attività crea eventi. Le principali funzioni associate ad *Organizzatore* sono: visualizzare la lista degli eventi da esso creati, aggiungere o rimuovere un evento, modificare alcuni parametri legati ad un evento, visualizzare la lista dei partecipanti ad un evento. I privilegi legati a questo tipo di utente non possono essere in alcun modo inclusivi dei privilegi dell’utente *Partecipante*.

**Partecipante**, utente interessato a visionare gli eventi disponibili e nel caso a parteciparvi. Le principali funzioni associate a *Partecipante* sono: visualizzare la lista degli eventi disponibili, raffinare la ricerca attraverso dei filtri, inscriversi ad un evento, visualizzare la lista degli eventi a cui è iscritto, cancellare la partecipazione ad un evento. I privilegi legati a questo tipo di utente non possono essere in alcun modo inclusivi dei privilegi dell’utente *Organizzatore*.

### 2.2 Casi d’uso

Inizialmente ci siamo concentrati sul individuare i principali requisisti che il sistema deve soddisfare. Lo scopo principale del sistema è mostrare e prenotare degli eventi generati dagli organizzatori.

Dunque le principali azioni che il sistema eseguirà sono:

- ricerca di eventi, pubblicati dagli organizzatori, da parte dei partecipanti
- iscrizione ad un evento per un partecipante
- creazione di un evento da parte di un organizzatore

- gestione evento da parte dell'organizzatore.

Oltre a questo, per entrambi gli utenti si definiscono le operazioni basilari quali registrazione al servizio, login e logout. Inoltre si definisce il vincolo che email in atto di registrazione può essere usata per un unico account.

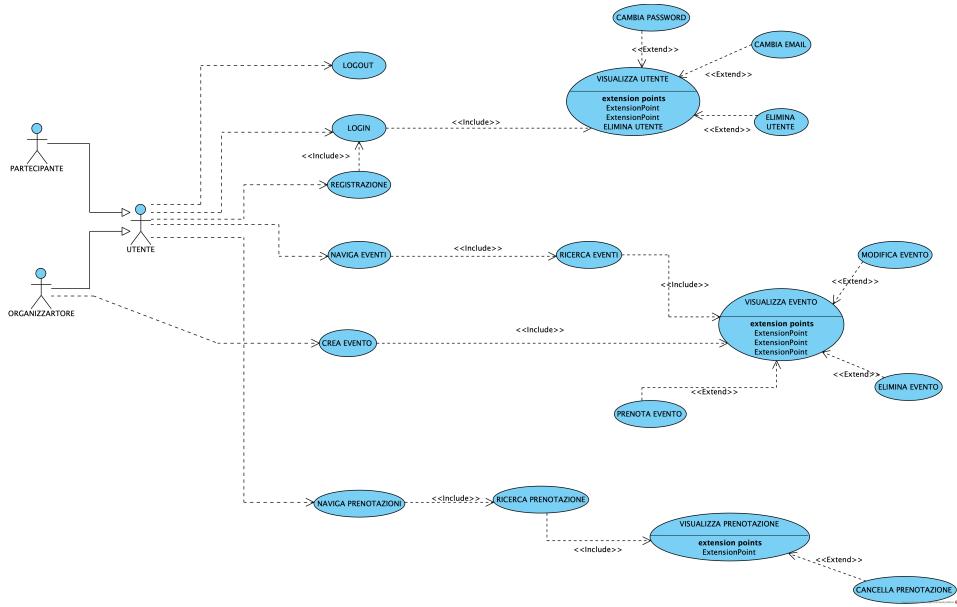


Figura 1: Use Cases Diagram

## 2.3 Diagramma Classi

Nella Figura 2 viene mostrato il diagramma delle classi ottenuto dopo l'analisi dei requisiti e la definizione degli attori. Per favorire la leggibilità, i dettagli e le procedure di ogni singola classe sono state omesse. Infatti, l'obiettivo principale è descrivere, attraverso l'uso delle associazioni, le relazioni che intercorrono tra le classi. Per fare questo è stato adottato il formalismo definito dal UML 2.

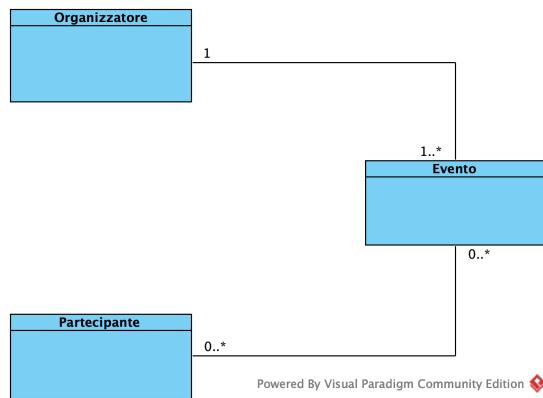


Figura 2: Class Diagram

Abbiamo tre classi principali: Organizzatore, Partecipante, Evento. Ogni evento è legato ad un solo organizzatore con una relazione Uno-a-Molti. Mentre esiste una relazione Molti-a-Molti tra le entità Partecipante ed Evento per mantenere la lista dei partecipanti ad ogni evento.

### 3 Implementazione

In questa sezione, si descrive, in modo più dettagliato, le scelte implementative adottata per lo sviluppo del sistema. Attualmente si tratta di una java app, basata su architettura Client-Server, il prototipo del sistema è stato sviluppato attenendosi al architettura multi-tier, questo per tenere separati il più possibile il lato Client da quello Server.

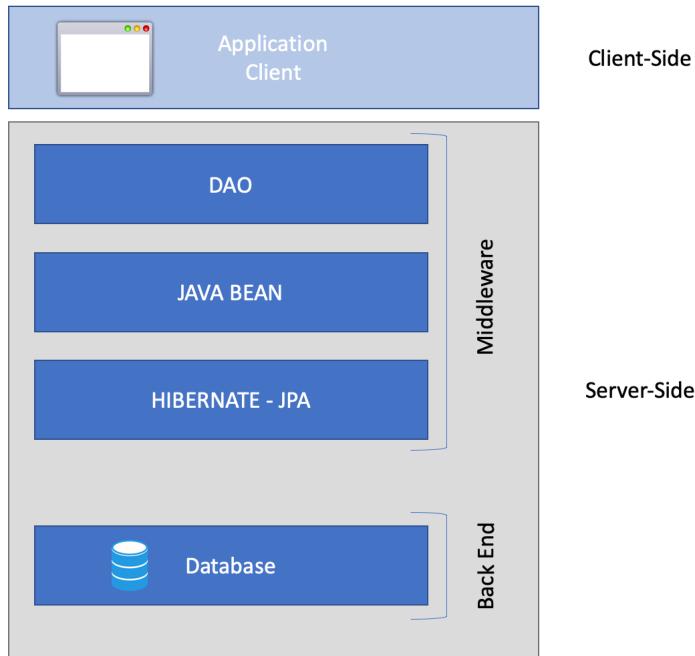


Figura 3: Multi-Tier Architecture Design

Come mostrato nella Figura 3, il lato Client è gestito con java application che si interfaccia con il lato server attraverso l'uso del framework Hibenate e le relative API JPA. Per lo sviluppo della parte di Back End si è scelto come DBMS MySQL, vista la buona affidabilità e la quasi piena implementazione dello Standard SQL. Ciò non toglie che per sviluppi futuri, si possa passare a POSTgre SQL, in quanto, come spiegato successivamente, sono previsti, per future estensioni del sistema, attributi che si riferiscono allo Standard Extended-SQL, è che POSTgre implementa e gestisce al meglio, nell'ambito dei DBMS open-source. Per assicurare la consistenza dei dati all'interno del DB, garantendo allo stesso tempo la concorrenza, il sistema è stato realizzato adottando il pattern architetturale MVC. Questo tipo di approccio risulta essere il metodo di controllo della concorrenza comunemente utilizzato dai sistemi di gestione per garantire accesso simultaneo al database e nei linguaggi di programmazione per implementare la memoria transazionale, senza questo ci potremo trovare di fronte a casi di inconsistenza dei dati all'interno del database.

---

### 3.1 Database

Nel definire le entità usate per descrivere i requisiti definiti dal sistema, si è partito dall’analisi dei requisisti e dalle considerazioni fatte sugli attori e sul diagramma delle classi, come linee guida principali. In particolare come evidenziato dalla figura 4, che descrive lo schema E-R, è presente l’entità Utente che è una generalizzazione delle due entità che descrivono gli utenti che sono stati previsti come utilizzatori del software. Poi è presente un’altra entità che descrive l’oggetto alla base di questo sistema, Evento. Infine vi sono due relazioni che servono per definire la principali operazioni all’interno del sistema.

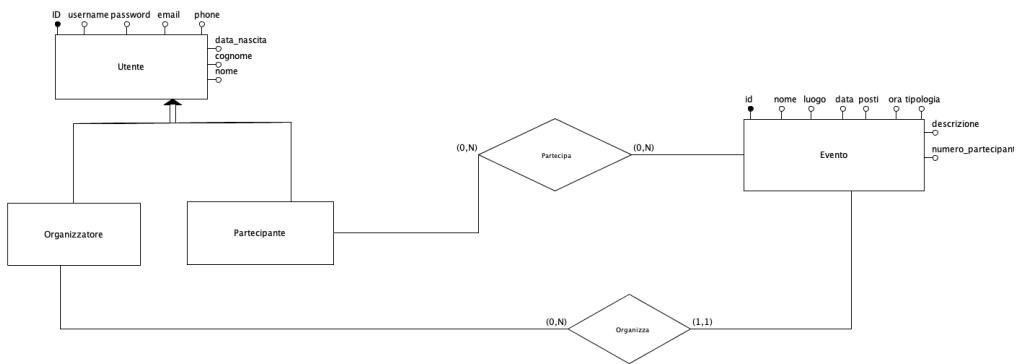


Figura 4: E-R Diagram

In fase di ristrutturazione, si è deciso per garantire maggiore velocità nell’esecuzione e per una migliore gestione dei dati di non aggregare le due entità figlie all’entità padre. Le tabelle modellate dopo la ristrutturazione risultano essere in Terza Forma Normale.

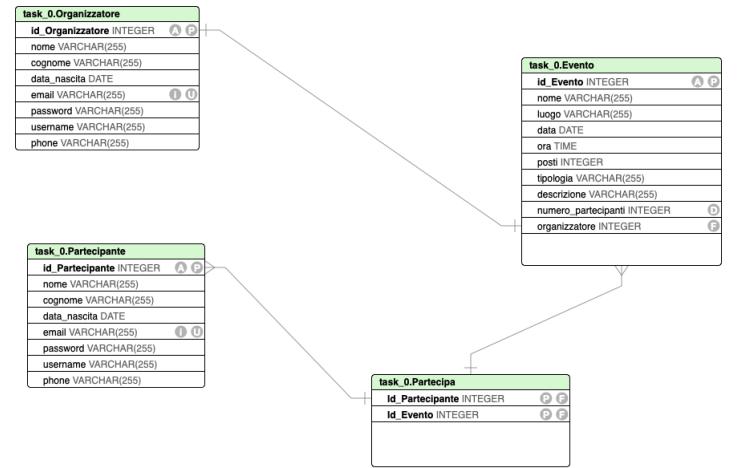


Figura 5: Relational Model

## 3.2 Hibernate e JPA

Hibernate è un popolare ORM (Object Relational Mapping) framework che permette di semplificare le operazioni dei programmatore che usano il paradigma della programmazione ad oggetti, implementando le Java Persistence Api che sviluppano un nuovo modo di fare le query orientate agli oggetti java. JPA sfrutta l'utilizzo di annotazioni nelle classi java per mappare le tabelle del database relazionale in oggetti.

### 3.2.1 JPA Entities

#### 3.2.1.1 EventoDb

```

1 package com.mycompany.hibernate;
2
3 import java.util.Date;
4 import java.util.HashSet;
5 import java.util.Set;
6 import javax.persistence.*;
7 import org.hibernate.annotations.Fetch;
8 import org.hibernate.annotations.FetchMode;
9
10 @Entity(name="EventoDb")
11 @Table(name="evento")
12 public class EventoDb {
13     @Column(name="id_Evento")

```

---

```
14 @Id
15 @GeneratedValue(strategy=GenerationType.IDENTITY)
16
17 private long id;
18 private String nome;
19 private String luogo;
20 @Temporal(javax.persistence.TemporalType.DATE)
21 @Column(name = "data")
22 private Date data;
23 private String ora;
24 private int posti;
25 private String tipologia;
26 private String descrizione;
27 private int numero_partecipanti;
28 @ManyToOne(fetch = FetchType.EAGER)
29 @JoinColumn(name = "id_Organizzatore")
30 private OrganizzatoreDb organizzatore;
31 @ManyToMany(cascade =
32     {CascadeType.PERSIST,CascadeType.MERGE},
33     fetch = FetchType.LAZY)
34 @JoinTable(
35     name="partecipa",
36     joinColumns=@JoinColumn(name="id_Evento"),
37     inverseJoinColumns=@JoinColumn(name="id_Partecipante"))
38
39
40 @Fetch(value = FetchMode.SUBSELECT)
41 private Set<PartecipanteDb> partecipazioni=new HashSet<>();
42
43 //Costruttori della classe
44 public EventoDb(){
45     //costruttore vuoto
46 }
47
48 public EventoDb(long id, String nome, String luogo, Date
49     data, String ora, int posti, String tipologia, String
50     descrizione, int numero_partecipanti, OrganizzatoreDb
      organizzatore, Set<PartecipanteDb> partecipazioni) {
51
52     this.id = id;
```

---

```
51     this.nome = nome;
52     this.luogo = luogo;
53     this.data = data;
54     this.ora = ora;
55     this.posti = posti;
56     this.tipologia = tipologia;
57     this.descrizione = descrizione;
58     this.numero_partecipanti = numero_partecipanti;
59     this.organizzatore = organizzatore;
60     this.partecipazioni = partecipazioni;
61 }
62 //funzioni utili
63 public void addPartecipante(PartecipanteDb p){
64     partecipazioni.add(p);
65     p.getBook().add(this);
66 }
67 public void removePartecipante(PartecipanteDb p){
68     partecipazioni.remove(p);
69     p.getBook().remove(this);
70 }
71 public void removeOrganizzatore(){
72     organizzatore.removeEvento(this);
73     organizzatore=null;
74 }
75 //hash and equals
76 @Override
77 public int hashCode() {
78     int hash = 3;
79     hash = 53 * hash + (int) (this.id ^ (this.id >>> 32));
80     return hash;
81 }
82
83
84 @Override
85 public boolean equals(Object obj) {
86     if (this == obj) {
87         return true;
88     }
89     if (obj == null) {
90         return false;
91     }
92 }
```

---

```

92     if (getClass() != obj.getClass()) {
93         return false;
94     }
95     final EventoDb other = (EventoDb) obj;
96     if (this.id != other.id) {
97         return false;
98     }
99     return true;
100 }
101
102 //Get and Setter
103
104 }
```

L'annotazione @GeneratedValue specifica come deve essere generato il valore della chiave primaria, nel caso dell'entità è stato usato il tipo IDENTITY. che specifica che il valore deve essere auto-incrementato e che deve essere unico nella tabella.

L'annotazione @Temporal viene usata per risolvere uno dei maggiori problemi di conversione della data e dell'ora dall'oggetto java al tipo compatibile sul database e per il recupero dell'informazione nell'applicazione. Nel caso specifico mantiene la data dell'evento.

Con @OneToMany descrive la relazione che esiste tra l'oggetto evento e l'oggetto organizzatore, aggiungendo organizzatore come Foreign-Key in evento. Mentre @ManyToMany collega i partecipanti ai vari eventi.

### 3.2.1.2 OrganizzatoreDb

```

1 package com.mycompany.hibernate;
2
3 import java.util.Date;
4 import java.util.HashSet;
5 import java.util.Set;
6 import javax.persistence.*;
7 import org.hibernate.annotations.Fetch;
8 import org.hibernate.annotations.FetchMode;
9
10 @Entity(name="OrganizzatoreDb")
11 @Table(name="organizzatore")
```

---

```
12 public class OrganizzatoreDb {
13     @Column(name="id_Organizzatore", updatable = false, nullable =
14             false)
15     @Id
16     @GeneratedValue(strategy=GenerationType.IDENTITY)
17     private long id;
18     private String nome;
19     private String cognome;
20     @Temporal(javax.persistence.TemporalType.DATE)
21     @Column(name = "data_nascita")
22     private Date data_nascita;
23     @Column(name="email", unique=true)
24     private String email;
25     private String password;
26     private String username;
27     private String phone;
28     @OneToMany(
29         mappedBy = "organizzatore",
30         cascade = CascadeType.ALL,
31         orphanRemoval = true,
32         fetch = FetchType.EAGER
33     )
34     @Fetch(value = FetchMode.SUBSELECT)
35     private Set<EventoDb> eventiCreati=new HashSet<EventoDb>();
36 
37     //funzioni utili per la classe
38     public void addEvento(EventoDb ev){
39         eventiCreati.add(ev);
40         ev.setOrganizzatore(this);
41     }
42     public void removeEvento(EventoDb ev){
43         eventiCreati.remove(ev);
44         ev.setOrganizzatore(null);
45     }
46     //costruttori della classe
47     public OrganizzatoreDb(){
48         //costruttore vuoto
49     }
50     public OrganizzatoreDb(long id, String nome, String cognome,
51             Date data_nascita, String email, String password, String
52             username, String phone) {
```

---

```
50     this.id = id;
51     this.nome = nome;
52     this.cognome = cognome;
53     this.data_nascita = data_nascita;
54     this.email = email;
55     this.password = password;
56     this.username = username;
57     this.phone = phone;
58 }
59
60 //hash and equals
61 @Override
62 public int hashCode() {
63     int hash = 5;
64     hash = 71 * hash + (int) (this.id ^ (this.id >>> 32));
65     return hash;
66 }
67
68 @Override
69 public boolean equals(Object obj) {
70     if (this == obj) {
71         return true;
72     }
73     if (obj == null) {
74         return false;
75     }
76     if (getClass() != obj.getClass()) {
77         return false;
78     }
79     final OrganizzatoreDb other = (OrganizzatoreDb) obj;
80     if (this.id != other.id) {
81         return false;
82     }
83     return true;
84 }
85
86 //Get and Setter
87
88 }
```

L'annotazione @GeneratedValue specifica come deve essere generato il valore

---

della chiave primaria, nel caso dell'entità è stato usato il tipo IDENTITY, che specifica che il valore deve essere auto-incrementato e che deve essere unico nella tabella.

L'annotazione @Temporal viene usata per risolvere uno dei maggiori problemi di conversione della data e dell'ora dall'oggetto java al tipo compatibile sul database e per il recupero dell'informazione nell'applicazione. Nel caso specifico mantiene la data di nascita.

Con @OneToOne descrive la relazione che esiste tra l'oggetto evento e l'oggetto organizzatore, aggiungendo organizzatore come Foreign-Key in evento. Mentre @ManyToMany collega i partecipanti ai vari eventi.

### 3.2.1.3 PartecipanteDb

```
1 package com.mycompany.hibernate;
2 import java.util.*;
3 import javax.persistence.*;
4 import org.hibernate.annotations.Fetch;
5 import org.hibernate.annotations.FetchMode;
6
7 @Entity(name="PartecipanteDb")
8 @Table(name="partecipante")
9 public class PartecipanteDb {
10     @Column(name="id_Partecipante")
11     @Id
12     @GeneratedValue(strategy=GenerationType.IDENTITY)
13     private long id;
14     private String nome;
15
16
17     private String cognome;
18     @Temporal(javax.persistence.TemporalType.DATE)
19     @Column(name = "data_nascita")
20     private Date data_nascita;
21     @Column(name="email",unique=true)
22     private String email;
23     private String password;
24     private String username;
25     private String phone;
26
```

---

```
27
28     @ManyToMany(
29         mappedBy="partecipazioni",
30         fetch = FetchType.LAZY)
31     @Fetch(value = FetchMode.SUBSELECT)
32     private Set<EventoDb> book=new HashSet<EventoDb>();
33
34     //funzioni utili
35
36     public void addBook(EventoDb ev){
37         book.add(ev);
38         ev.getPartecipazioni().add(this);
39     }
40
41     public void removeBook(EventoDb ev){
42         book.remove(ev);
43         ev.getPartecipazioni().remove(this);
44     }
45     //costruttori della classe
46     public PartecipanteDb(){
47         //costruttore vuoto
48     }
49
50     public PartecipanteDb(long id, String nome, String cognome,
51             Date data_nascita, String email, String password, String
52             username, String phone, Set<EventoDb> book) {
53         this.id = id;
54         this.nome = nome;
55         this.cognome = cognome;
56         this.data_nascita = data_nascita;
57         this.email = email;
58         this.password = password;
59         this.username = username;
60         this.phone = phone;
61         this.book = book;
62     }
63
64     //hash and equals
65     @Override
66     public int hashCode() {
```

---

```

66     int hash = 3;
67     hash = 53 * hash + (int) (this.id ^ (this.id >> 32));
68     return hash;
69 }
70
71 @Override
72 public boolean equals(Object obj) {
73     if (this == obj) {
74         return true;
75     }
76     if (obj == null) {
77         return false;
78     }
79     if (getClass() != obj.getClass()) {
80         return false;
81     }
82     final PartecipanteDb other = (PartecipanteDb) obj;
83     if (this.id != other.id) {
84         return false;
85     }
86     return true;
87 }
88
89 //Get and Setter
90
91 }
```

L'annotazione @GeneratedValue specifica come deve essere generato il valore della chiave primaria, nel caso dell'entità è stato usato il tipo IDENTITY, che specifica che il valore deve essere auto-incrementato e che deve essere unico nella tabella.

L'annotazione @Temporal viene usata per risolvere uno dei maggiori problemi di conversione della data e dell'ora dall'oggetto java al tipo compatibile sul database e per il recupero dell'informazione nell'applicazione. Nel caso specifico mantiene la data di nascita.

@ManyToMany collega i partecipanti ai vari eventi.

---

### 3.3 Operazioni CRUD

#### 3.3.1 Create

Per memorizzare un oggetto creato sul database è necessario utilizzare l'oggetto Entity Manager per iniziare una transazione, rendere effettivo il caricamento ed infine chiudere la connessione. Nell'esempio seguente il codice usato per l'inserimento di un nuovo partecipante o di un nuovo organizzatore nel DB.

```
1 public static int registrazione(OrganizzatoreDb
2   ↳ organizzatore,PartecipanteDb partecipante){
3
4
5   if(((organizzatore==null)&&(partecipante==null))
6     ||((organizzatore!=null)&&(partecipante!=null)))
7
8   {
9     System.err.println("parametri non validi riprova");
10    return 0;
11
12  }
13  try{
14    creaConnessione();
15    entityManager.getTransaction().begin();
16    if(organizzatore!=null)
17      entityManager.persist(organizzatore);
18    else
19      entityManager.persist(partecipante);
20
21    entityManager.getTransaction().commit();
22  }
23  catch(PersistenceException pe)
24  {
25    pe.printStackTrace();
26    errore=0;
27  }
28  catch(Exception e)
29  {
30    e.printStackTrace();
```

---

```

31     System.out.println("Errore durante la transizione
32         ↳ riprova!");
33     errore=0;
34 }
35 finally
36 {
37     chiudiConnessione();
38 }
39 return errore;
}

```

### 3.3.2 Read

Eseguiamo un interrogazione al DB tramite la chiamata del metodo getResultList() dell'Entity Manager che restituisce una lista oggetti. Nell'esempio seguente il metodo restituisce tutti gli eventi ancora disponibili e al quale il partecipante non si è ancora iscritto.

```

1 public static ArrayList<Evento> ricercaEventi(OrganizzatoreDb
2     ↳ organizzatore){
3
4     ArrayList<Evento> ev=new ArrayList<>();
5
6     for (EventoDb evento : organizzatore.getEventiCreati()) {
7         ev.add( new Evento((int)evento.getId(), evento.getNome(),
8             ↳ evento.getLuogo(), evento.getData(),
9             evento.getOra(), evento.getPosti(),
10            ↳ evento.getTipologia(), evento.getDescrizione(),
11            int)evento.getOrganizzatore().getId(),
12            ↳ evento.getNumero_partecipanti()));
13    }
14
15    return ev;
16
17
18 }

```

### 3.3.3 Update

Nel seguente esempio verranno modificati gli attributi di un partecipante, tramite l'apertura della transazione, il merge dell'oggetto partecipanteDb della chiusura della transazione.

---

```
1 public static int modificaDati(PartecipanteDb partecipante) {
2
3     int errore=2;
4
5     try{
6         entityManager = factory.createEntityManager();
7         entityManager.getTransaction().begin();
8         entityManager.merge(partecipante);
9         entityManager.getTransaction().commit();
10        errore=1;
11
12    } catch(Exception ex){
13        ex.printStackTrace();
14        System.out.println("A problem occured in insert
15                           events!");
16        errore=2;
17
18    }
19    entityManager.close();
20
21
22    return errore;
23
24 }
```

### 3.3.4 Delete

Cancellazione dell'evento sfruttando il metodo remove dell'oggetto Entity Manager.

```
1 public static int eliminaEvento(long id) {
2
3     int errore = 1;
4     try{
5
6         entityManager = factory.createEntityManager();
7         entityManager.getTransaction().begin();
8         EventoDb ev=entityManager.find(EventoDb.class, id);
9         for(Iterator<PartecipanteDb>
10            it=ev.getPartecipazioni().iterator();it.hasNext();){
11
12             PartecipanteDb p=it.next();
13             if(p.getId()==id)
14                 it.remove();
15
16         }
17         entityManager.getTransaction().commit();
18         errore=0;
19
20     }
21
22 }
```

---

```

10         PartecipanteDb p=it.next();
11         it.remove();
12         ev.removePartecipante(p);
13     }
14     ev.removeOrganizzatore();
15     entityManager.remove(ev);
16     entityManager.getTransaction().commit();
17     System.out.println("EVENTO Cancellato");
18
19 } catch(Exception ex){
20     ex.printStackTrace();
21     entityManager.getTransaction().rollback();
22     System.out.println("A problem occured in delete an
23     ↳ event!");
24     errore = 0;
25 }
26 finally{
27     entityManager.close();
28 }
29
30
31 return errore;
32
33 }
```

## 4 Studio di Fattibilità Database Key-Value

### 4.1 Introduzione

Lo sviluppo dei database NoSQL è fortemente connesso con il fenomeno dei Big DATA. Questo a sua volta collegato con le sempre più importanti richieste di spazio per la memorizzazione e la gestione di un enorme quantità di dati, molto complessi dinamici e provenienti da fonti differenti. Il termine NoSQL non si riferisce a una specifica soluzione, ma a soluzioni basate su differenti finalità e modelli hardware. Le caratteristiche principali di questi sistemi sono garantire bassa latenza nell'accesso a grandi dataset e la pressocché perfetta disponibilità del sistema anche in ambienti di utilizzo inaffidabili. I database NoSQL sono classificabili in quattro categorie:

- Key-Value

- 
- Column-Family
  - Document
  - Graph

I database Key-Value sono la forma più semplice di database NoSQL, sono modellati su due componenti: *Chiave* e *Valore*. Le chiavi sono identificatori associati a dei valori, queste devono essere uniche all'interno di un namespace, anche detto bucket. I valori, sono memorizzati insieme alle chiavi, e possono essere di tipi diversi, stringhe, interi o oggetti più complessi, come le immagini. La completa mancanza di forte tipizzazione, da una parte, garantisce una grande flessibilità durante la memorizzazione dei dati, ma dall'altra impone agli sviluppatori del software di effettuare un controllo sui tipi. Le caratteristiche principali dei database Key-Value sono: semplicità, velocità e scalabilità. Gestendo un tipo di strutture molto semplice, fa sì che possa tenere questi dati in memoria, e questo garantisce che le operazioni siano più veloci. Quindi fare un tuning delle prestazioni per questo tipo database, significa, gestire le strutture in modo che la loro dimensione possa stare all'interno della memoria, questo può essere fatto, ad esempio, tramite la compressione. Come è facile capire le operazioni sui valori sono basate sulle chiavi, infatti si possono recuperare, memorizzare e cancellare i valori, ma sempre attraverso la chiave. Questa risulta essere una limitazione, poiché non è possibile usare dei costrutti tipici dei database SQL, ad esempio non è possibile cercare delle occorrenze con un determinato valore, come nel caso di SQL con la clausola WHERE. Questo però, può essere comunque, implementato all'interno dell'applicazione.

#### 4.1.1 LevelDB

Per questo progetto si è scelta come tecnologia implementativa LevelDB. Si tratta di un database Key-Value open-source progettato da Google. In LevelDB i dati sono ordinati per chiave. Supporta molto bene le scritture batched e la compressione. Poiché si tratta di un NoSQL database, non ha modello relazionale per i dati e non supporta le query SQL. Inoltre, non supporta l'uso di indici. Le performance di LevelDB sono di gran lunga migliori, se si compara con altri DBMS, come in Figura 6, in termini di operazioni di scrittura e lettura di dati in ordine sequenziale. Si nota un calo delle prestazioni con valori molto grandi in scrittura. Di fatto le sue potenzialità risiedono principalmente nelle operazioni di lettura. Giusto per

---

citare un esempio d’uso, Google Chorme lo utilizza come database di backend per l’IndexDB.

#### A. Sequential Reads



Figura 6: Reading Performance Comparison

## 4.2 Scenario 1 : MIGRAZIONE COMPLETA

In questa prima proposta, si vuole modellare il sistema implementandolo completamente su un database NoSQL di tipo Key-Value. Questo per avere tutti i vantaggi in termini di fruibilità e velocità di recupero dei dati offerti da questo tipo di database ed avere altresì una semplicità d’uso, cioè implementare tutto su un unico database. Considerati i requisiti esistenti, sopra descritti, si è scelto di affrontare il problema attraverso la modellazione con design pattern “*Emulating Tables*”, in cui si ripropone la struttura del database relazione definendo le chiavi e buckets in modo da essere consistenti con il problema.

### 4.2.1 Modellazione dei buckets

Come già discusso nella parte introduttiva, le chiavi all’interno del namespace devono essere uniche; nel nostro caso vista la presenza delle entità *Partecipante* e *Organizzatore*, le quali contengono gli stessi attributi e usano una chiave di tipo meaningless, si è scelto di modellare ogni tabella presente nel database con un bucket differente.



Figura 7: Buckets model

---

#### 4.2.2 Modellazione delle chiavi

Seguendo il pattern “Emulating Tables”, la definizione delle chiavi risulta essere molto standard:

*Entity\_Name+ : +Entity\_ID+ : +Entity\_Attribute = Value;*

In questo modo si riesce a replicare in modo molto fedele la struttura del database relazionale, ma senza avere nessun vantaggio dall’uso del database Key-Value, anzi si complica inutilmente la programmazione della logica del sistema, in quanto la consistenza delle foreign key deve essere garantita a questo livello, così come la ricerca tramite filtri che non può essere gestita direttamente dal database manager system. Sicuramente vista la lunghezza delle chiavi è preferibile utilizzare una funzione Hash, al fine di rendere l’accesso molto più performante.

#### 4.2.3 Considerazioni sulle caratteristiche del sistema

Come ci suggerisce il teorema CAP, in un sistema distribuito non possono essere garantite contemporaneamente tutte e tre le seguenti proprietà: Consistency, Availability e Partition Tolerance. Nel caso particolare di LevelDB, possono essere garantite solo Availability e Partition Tolerance. Analizzando i requisisti del sistema sicuramente questo tipo di caratteristiche, può essere utile, si veda la Tabella Evento è tollerabile un parziale inconsistenza dei dati, ma invece la consistenza risulta essere estremamente necessaria nelle tabelle che mantengono i dati degli utenti.

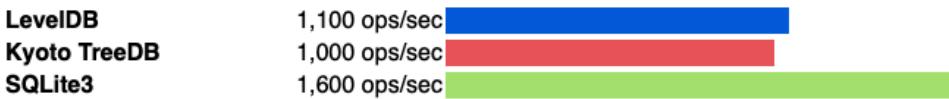
Altri elementi che mancano a un sistema così realizzato sono l’indicizzazione e la possibilità di fare join. La mancanza di indici potrebbe essere soppiata dalla creazione di chiavi che contengano al loro interno elementi degli attributi che rendono più facile il recupero dei dati, ma questo per prima cosa aumenta notevolmente la dimensione dei dati da memorizzare in quanto si avrebbero diverse copie dello stesso dato ma con chiavi diverse, inoltre favoriscono ancora di più una possibile inconsistenza dei dati, in quanto alcune manipolazioni dei dati non potrebbero essere fatte su tutte le varie copie.

Per quanto riguarda la possibilità di fare Join, anche questa potrebbe essere aggirata con un recupero completo dei dati dal database e poi uniti e manipolati lato client, ma questo metodo presenta alcune problematiche in termini di uso del dispositivo da parte dell’utente. Bisogna pensare a un uso principalmente su dispositivi mobili, che pure avendo prestazioni molto

---

elevate in termini di velocità di elaborazione, a volte i modelli più economici presentano limitazioni della dimensione della memoria (RAM) e dello spazio di archiviazione, quindi bisogna considerare che non si può lavorare in uso esclusivo delle risorse del dispositivo, che inficerebbero le performance generali dell'apparecchio con un conseguente abbandono dell'applicazione da parte dell'utente.

#### **Sequential Writes**



#### **Random Writes**

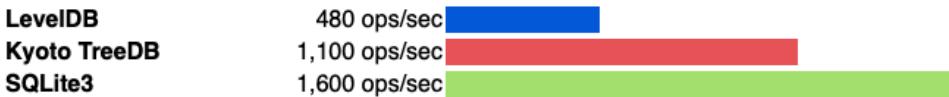


Figura 8: Write Performance Comparison

#### **4.2.4 Conclusioni**

Come già ampiamente discusso nelle sezioni precedenti, lo scenario descritto, migrazione totale verso un database Key-Value, non può essere implementato, in quanto si perderebbe la consistenza di alcune parti di dati, si perderebbe la strutturazione dei dati come in forma relazionale, in ultimo si noterebbe un calo importante delle prestazioni in scrittura, cosa da evidenziare in modo particolare in quanto, nel sistema si prevedono molte scritture specialmente da parte dei Partecipanti che si inscrivono a un evento o ne cancellano l'iscrizione, introducendo così una latenza dei tempi di attesa molto grande. Sicuramente in un'altra forma un database NoSQL, potrebbe essere un approccio interessante per questo tipo di sistemi.

### **4.3 Scenario 2: APPROCCIO IBRIDO**

In questo secondo tipo di implementazione viene proposto un approccio di tipo ibrido. Come molte volte descritto in letteratura e anche realizzato in diverse implementazioni di applicazioni reali, per citare un esempio il social network Facebook, usa un database di tipo relazionale per mantenere i dati degli utenti, mentre il resto dei contenuti immagini post ecc., sono memorizzati in un database NoSQL, nel caso specifico un Column-Family. Prima di

---

sviluppare l'idea che sta dietro questo tipo di implementazione, risulta necessario soffermarci sui dei requisiti non funzionali ma altrettanto importanti per il sistema, in particolare un analisi accurata di cosa ogni utente chiede al servizio e in quale dimensione economica si pone per la crescita del servizio stesso. L'utente Organizzatore, vede il servizio come un luogo dove possa essere pubblicizzato il suo evento al fine di avere torna conto economico, quindi esso valuta in modo relativo la velocità con cui il database memorizza l'informazione, l'importante è che questa sia corretta e fruibile. D'altra parte l'utente Partecipante, valuta in modo particolare anche la velocità con cui riesce a recuperare i dati relativi agli eventi, maggiori sono la velocità e la facilità dell'utilizzo dell'applicazione, maggiore sarà anche l'afflusso di utenti che sceglieranno il servizio, ciò garantirà sia un ritorno economico per gli sviluppatori del servizio, sia per gli organizzatori in quanto vedranno un afflusso di clienti ai loro eventi. Fatta questa premessa, l'idea è quella di lasciare come database principale quello relazionale e aggiungere un database Key-Value in qui sarà contenuta una copia di un sotto-insieme di tuple della tabella Evento, come se questa fosse una cache per il database relazionale, così da favorire il recupero dei dati degli eventi a bassa latenza da parte dei Partecipanti.

#### 4.3.1 Modellazione dei database

Alla base del sistema resta come database quello descritto in Figura 5, come già implementato nella versione esistente, sui vengo effettuate tutte le operazioni di scrittura da parte degli utenti e quelle di lettura, inoltre si occupa di gestire l'attributo Posti\_Disponibili in Evento, per impedire eventuali problemi di over-booking.

Al suo fianco, viene memorizzata la tabella Evento in un database Key-Value, questa è accessibile in sola lettura dagli utenti Partecipanti, che una volta fatto il recupero dei dati, gestiscono in locale, attraverso delle funzioni proprie dell'applicazione, tutte le ricerche basate su i filtri imposti dall'utente. Alla parte server dell'applicazione è anche deputato il compito di tenere aggiornato il database Key-Value con gli Eventi con posti ancora disponibili e con data ancora valida. Così da ridurre in modo drastico le operazioni di scrittura sul LevelDB.

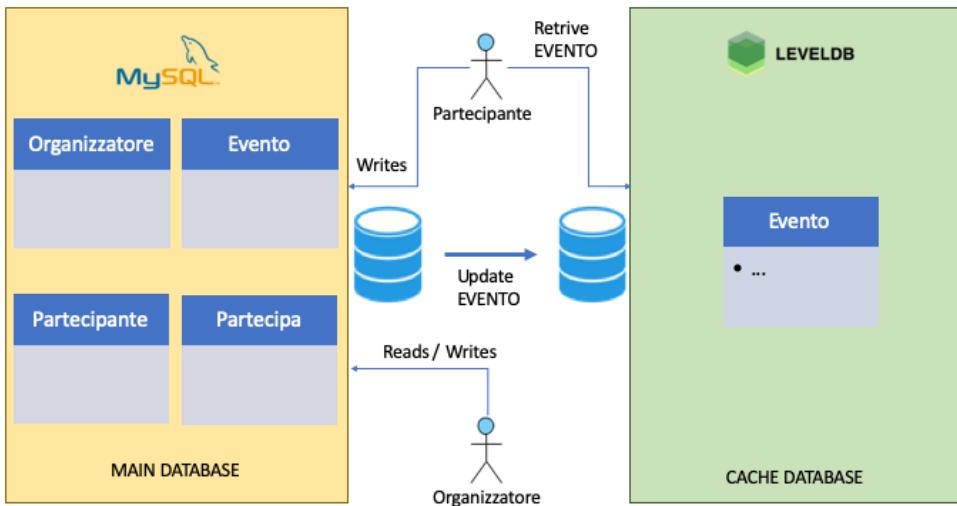


Figura 9: Databases Integration

Si era pensato di portare in NoSQL anche la tabella Partecipa, in cui è tenuta traccia di tutti i partecipanti ad un evento. Questa ipotesi però è stata scartata, poiché se si tiene traccia delle sole foreign-key risulterebbe necessario un ulteriore accesso al RDB; o altrimenti complicare in modo sostanziale aggiungendo ulteriori bucket e dati all'interno del database Key-Value, perdendo in consistenza dei dati visto le molteplici copie memorizzate in diverse posizioni.

Anche in questo scenario per modellare le chiavi per i vari valori si usa il design pattern “Emulating Table” quindi risulterà così composta:

*Entity\_Name : +Entity\_Citta : +Entity\_ID+ : +Entity\_Attribute = Value;*

Un esempio del caso specifico risulta:

```

Evento : Pisa : 1 : nome = Value;
Evento : Pisa : 1 : data = Value;
Evento : Pisa : 1 : ora = Value;
Evento : Pisa : 1 : tipologia = Value;

```

La gestione di questo database è del tutto trasparente al livello middletier dell'applicazione, infatti l'aggiornamento dei dati viene fatto dal server o nei momenti di minor utilizzo dell'applicazione, come ad esempio gli orari in cui

---

sono in svolgimento la maggior parte degli eventi, il che presuppone che gli utenti che stanno utilizzando il sistema sia molto ridotto. Può essere prevista una procedura di aggiornamento, anche quando, si verifica un'eccezione, ad esempio, un evento non ha più posti disponibili ma viene visualizzato nella lista dei prenotabili, all'atto della procedura della prenotazione, questa non va a buon fine e lancia a sua volta una procedura di aggiornamento per quell'evento.

Al fine di rendere ancora più veloce l'accesso ai dati è stata aggiunta un metodo di indicizzazione secondaria inserendo nella chiave la città in cui si svolge l'evento così da avere dei cicli di lettura molto più veloci

#### 4.3.2 Considerazioni sulle caratteristiche del sistema

Un implementazione di questo tipo, cerca di sfruttare tutte le caratteristiche peculiari di ogni tecnologia utilizzata. Facendo riferimento al teorema CAP, si prende il vantaggio della consistenza garantita dai RDB da un lato, dall'altro tutte le altre caratteristiche di availability e partition tolerance dei database Key-Value, oltre a tutte le proprietà di scalabilità ad esso connesse.

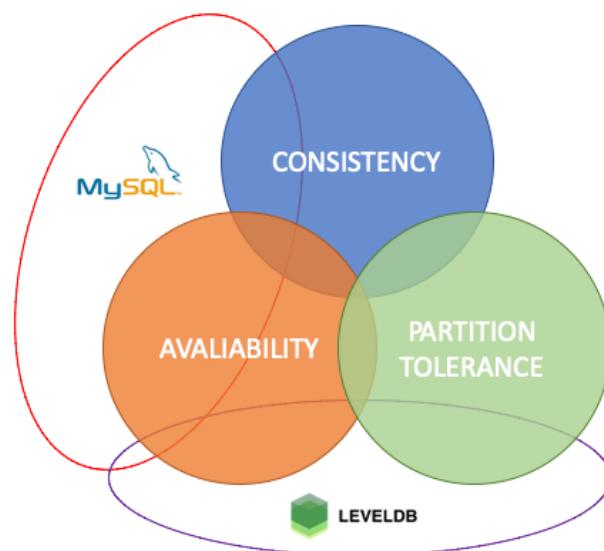


Figura 10: CAP Property in System

Dalle descrizioni fatte nelle sezioni precedenti, si nota una discrepanza nella gestione della consistenza. Infatti siamo di fronte a due copie dello stesso oggetto che dovrebbero essere costantemente consistenti entrambe. Quello che

---

realmente deve essere consistente è il database relazionale che accoglie tutte le scritture, per quanto riguarda il database Key-Value, siamo in un caso di eventualy consistency, infatti la cosa importante è che l'utente Partecipante possa visualizzare la lista di tutti gli eventi, poi se qualcuno di questi non sia più prenotabile non è un problema poiché sarà la logica dietro l'applicazione a gestire questo caso e non far procedere con l'operazione. Per citare un esempio di questo tipo, si pensi a un contenuto postato come storia su Instagram, non risulta necessario che a tutti i follower dell'utente il contenuto sia fruibile allo stesso momento, anche se qualcuno di questi abbia il contenuto nel suo feed con del ritardo l'importante è che questo sia presente e visualizzabile.

Dai test eseguiti per valutare la velocità di riposta dei due database in funzione del numero di record da leggere, si può notare come LevelDB mantiene delle prestazioni abbastanza livellate su tutto lo spettro, mentre MySQL ha un'incremento eccessivo al crescere dei record, ciò considerando un numero di richieste concorrenti abbastanza elevato produrrebbe un notevole rallentamento nelle prestazioni.

I test sono stati effettuati su una macchina con INTEL coreI7-8850U, RAM 8 Gb, SSD 128 Gb. Inoltre non sono stati valutati eventuali colli di bottiglia introdotti dalla rete.

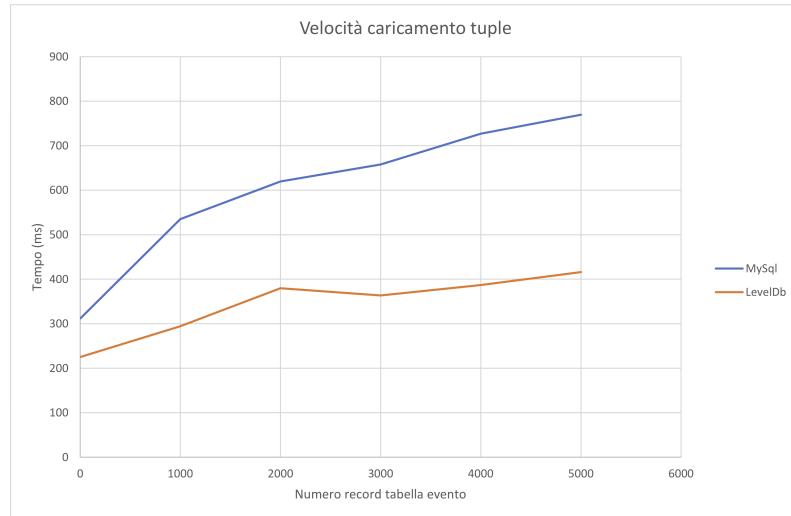


Figura 11: Statistiche Lettura

---

### 4.3.3 Conclusioni

Da tutto ciò detto precedentemente, dall’analisi dei requisisti, delle prestazioni, un’implementazione con l’uso dei database misto o ibrido, risulta essere quella che più si adatta allo sviluppo del nostro sistema. Questo garantisce, velocità d’uso ed elevate prestazioni agli utenti Partecipanti che valutano principalmente questo aspetto prima ancora di valutare i contenuti presentati dall’applicazione; consistenza e possibilità di avere una strutturazione dei dati per successive analisi, per gli utenti Organizzatori, i quali potranno programmare eventuali future campagne di marketing per incrementare i loro profitti. Dal punto di vista dello sviluppo del software, non si hanno eccessive complicazioni, in quanto, il core resta quello già attualmente in uso, a questo vengo affiancate delle semplici funzioni per il recupero dei dati e la gestione del filtering dei dati. In termini economici l’investimento che il gestore del sistema deve fare risulta essere non troppo oneroso, data la scalabilità orizzontale di questo tipo di sistemi e la richiesta di hardware anche non troppo potenti.

### 4.3.4 Implementazione Database Key-Value

Si precisa che, vista la natura didattica dell’elaborato, non è stato creato un server ad-hoc per LevelDB, ma è stato gestito in locale attraverso funzioni java.

#### 4.3.4.1 Operazione Lettura da LevelDB

```
1 public static ArrayList<Evento> RicercaEventi(String
2   ↵ citta,PartecipanteDb partecipante,boolean TestDAccesso) {
3   ArrayList<Evento> ev = new ArrayList<>();
4   try {
5     if(!TestDAccesso)
6       populaLevelDb.join();
7   } catch (InterruptedException ex) {
8     java.util.logging.Logger
9       getLogger(RicercaEventi.class.getName()).log(Level.SEVERE,
10      ↵ null, ex);
11   }
12   Options options = new Options();
13   options.createIfMissing(true);
14   try {
15     levelDBStore = factory.open(new File("eventi"), options);
```

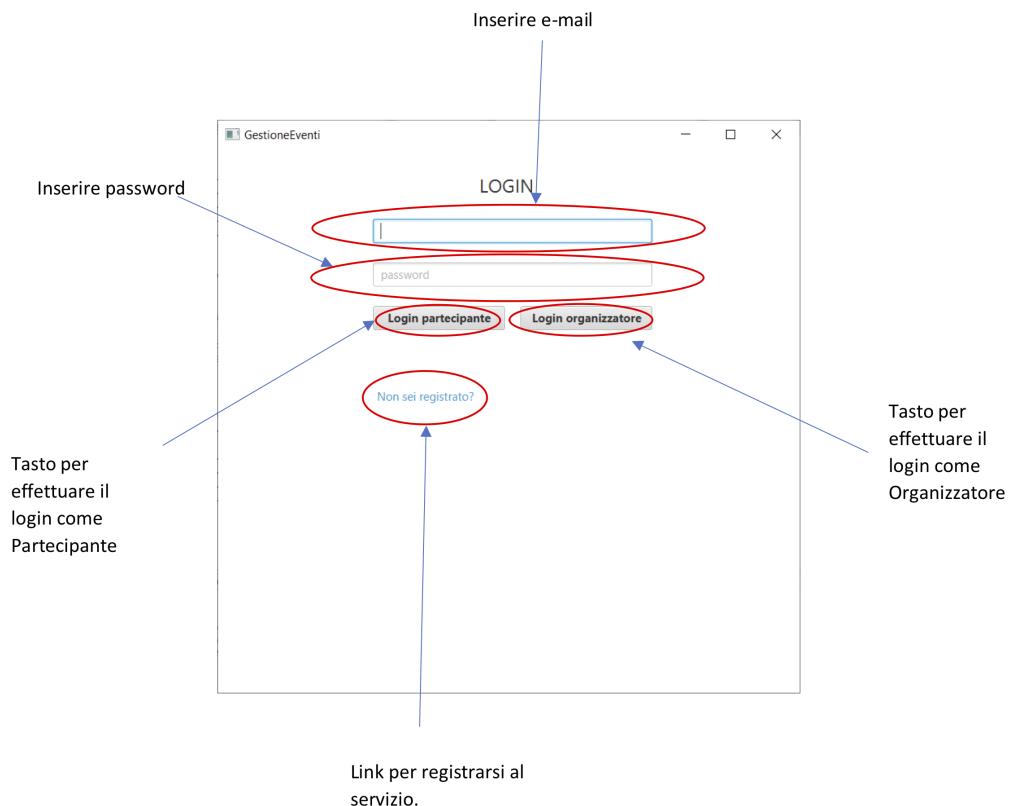
---

```
15 } catch (IOException ex) {
16     java.util.logging.Logger.getLogger(RicercaEventi.class.getName()).
17         log(Level.SEVERE, null, ex);
18 }
19 DBIterator iterator = levelDBStore.iterator();
20 setNullAllEventAttributes();
21 String argSeek;
22 if(citta.equals("")){
23     argSeek = "Evento:";
24 } else{
25     argSeek = "Evento:"+citta;
26 }
27 iterator.seek(bytes(argSeek));
28 try{
29     while(iterator.hasNext()){
30         String key = asString(iterator.peekNext().getKey());
31         String value = asString(iterator.peekNext().getValue());
32         String[] dividiKey=key.split(":");
33         Integer id=Integer.parseInt(dividiKey[2]);
34         luogo = dividiKey[1];
35         if(!citta.equals("")&&!citta.equals(luogo)){
36             break;
37         }
38
39         SetAnEventAttributeFromKeyValue(dividiKey,value);
40         if(id !=null && data!=null && nome!=null && luogo!=null
41             && ora!=null
42             && posti!=null && tipologia!=null && descrizione!=null
43             && idOrganizzatore!=null && numeroPartecipanti!=null){
44             addEventAtList(id,partecipante,ev);
45         }
46         iterator.next();
47     }
48 }finally {
49     try {
50         iterator.close();
51         levelDBStore.close();
52     } catch (IOException ex) {
53         java.util.logging.Logger.getLogger(RicercaEventi.
```

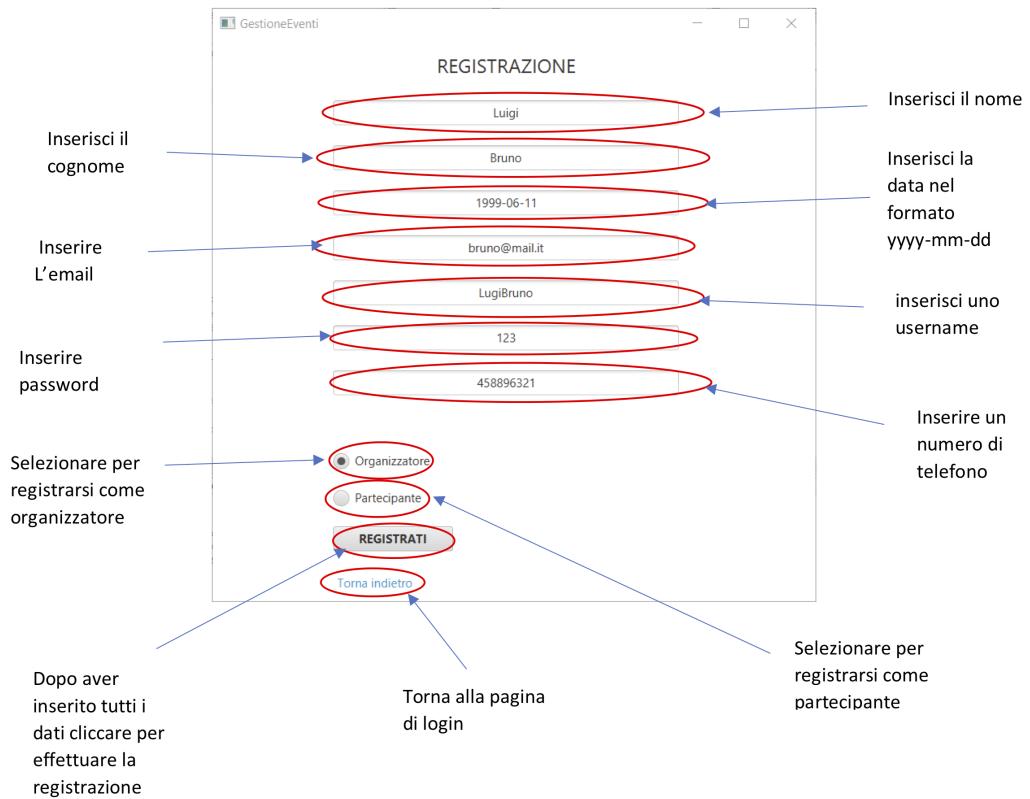
---

## 5 Manuale d'uso

### 5.1 Login



## 5.2 Registrazione nuovo utente



## 5.3 Funzioni Organizzatore

### 5.3.1 Creazione evento

GestioneEventi

### CREAZIONE EVENTO

ESCI

Inserire il nome dell'evento  
Inserire l'orario d'inizio evento  
Indicare il tipo d'evento  
Dare una breve descrizione dell'evento.

Festa in maschera  
Lecce  
20:30  
770  
Festa  
2019-12-30

Descrizione  
Evento in maschera per tutte le età

Cliccare per effettuare il log-out.

Inserire la località dell'evento

Selezionare il numero dei posti disponibili

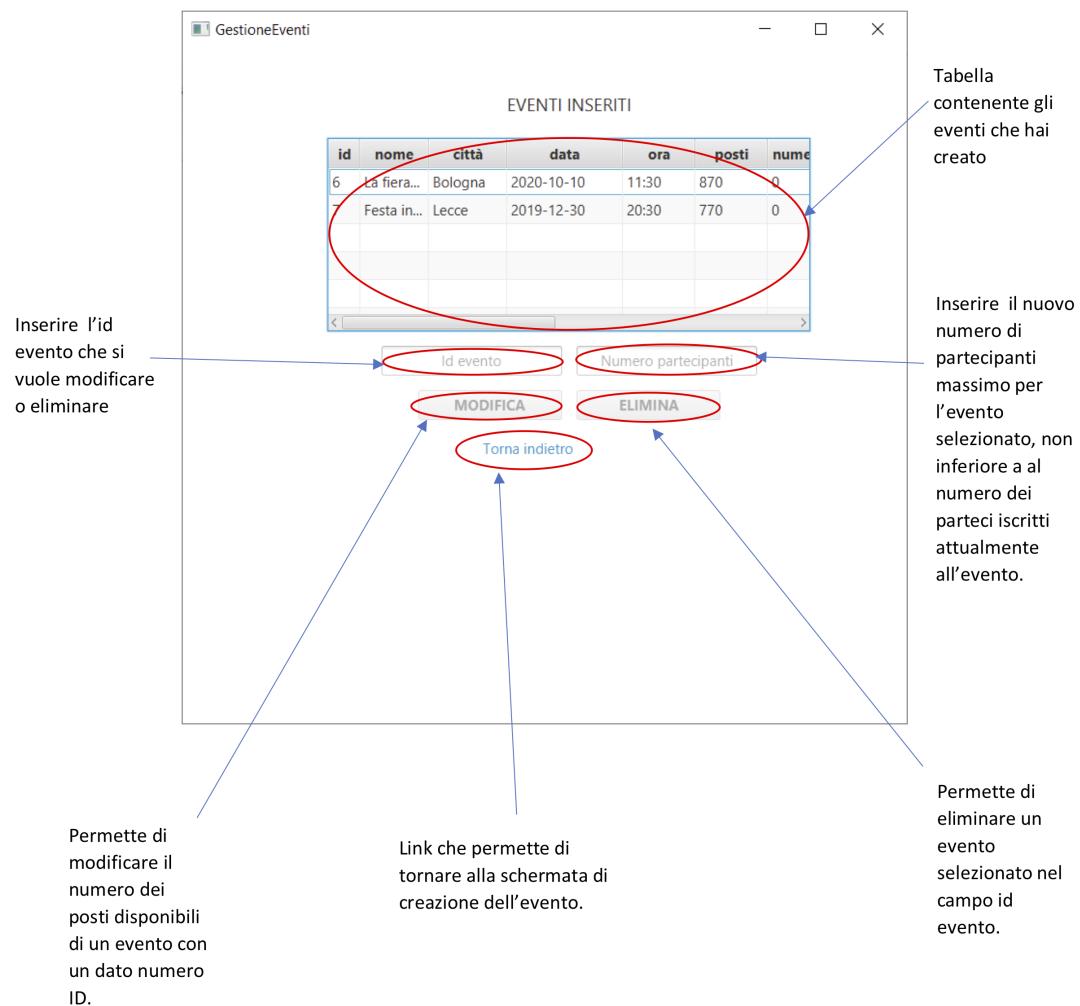
Indicare la data dell'evento

Una volta compilati tutti i campi premere il pulsante per creare un nuovo utente.

Premere il pulsante per visualizzare e gestire gli eventi che hai creato fino ad ora.

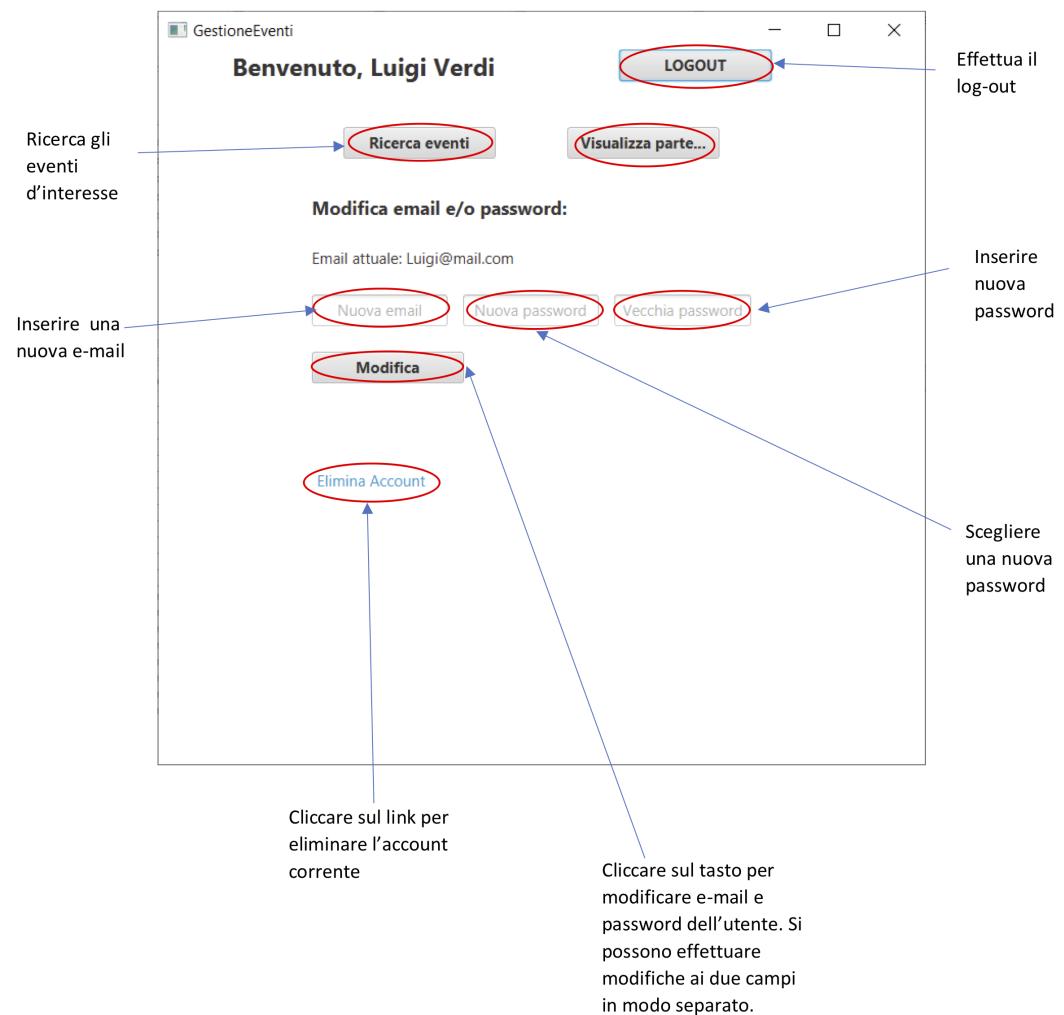
Crea Visualizza Eventi

### 5.3.2 Visualizza eventi creati

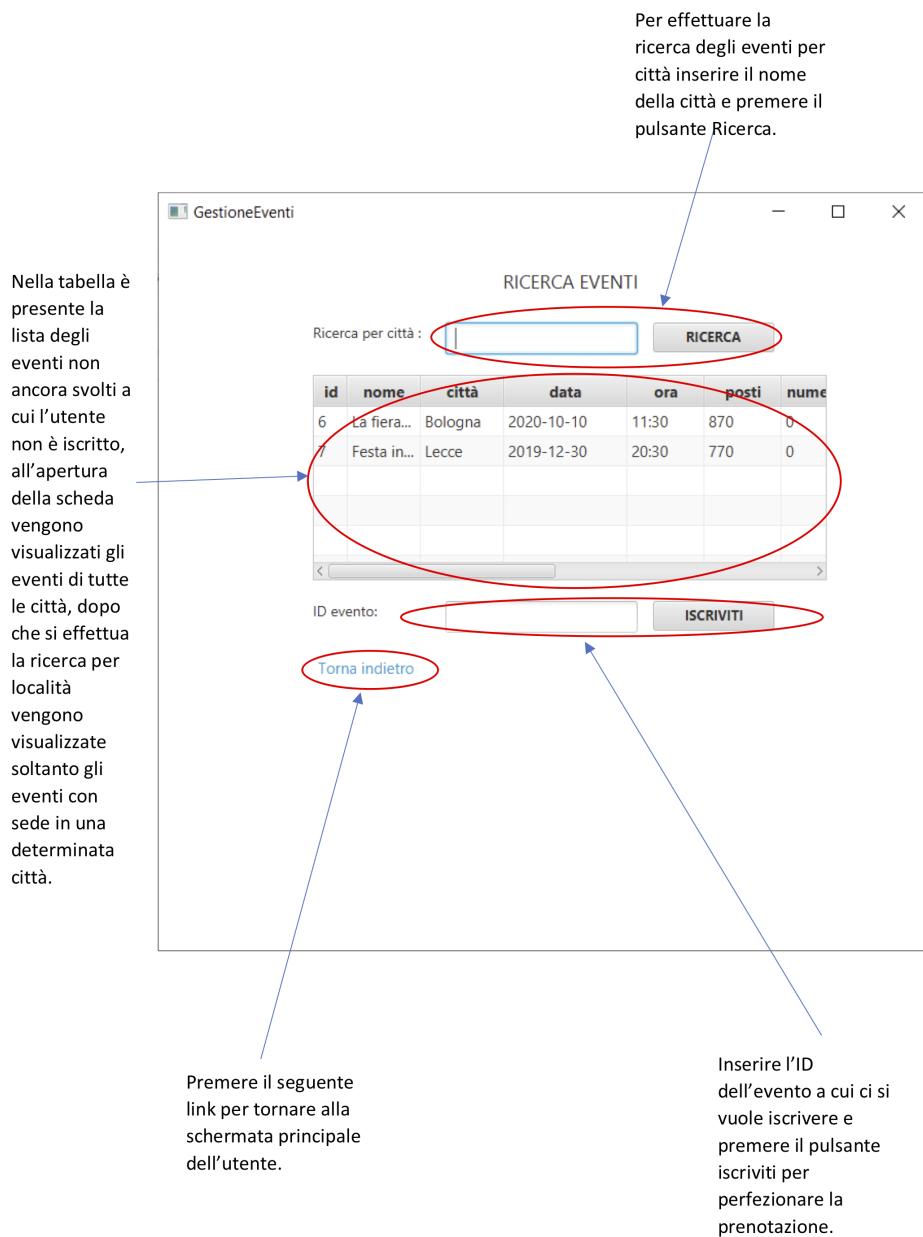


## 5.4 Funzioni Partecipante

### 5.4.1 Pagina Utente



### 5.4.2 Ricerca eventi



### 5.4.3 Visualizza eventi

