

Progetto: Gestione degli Eventi

1. Descrizione del Progetto:

L'applicazione "Gestione degli Eventi" permette agli utenti di creare, visualizzare, modificare e cancellare eventi.

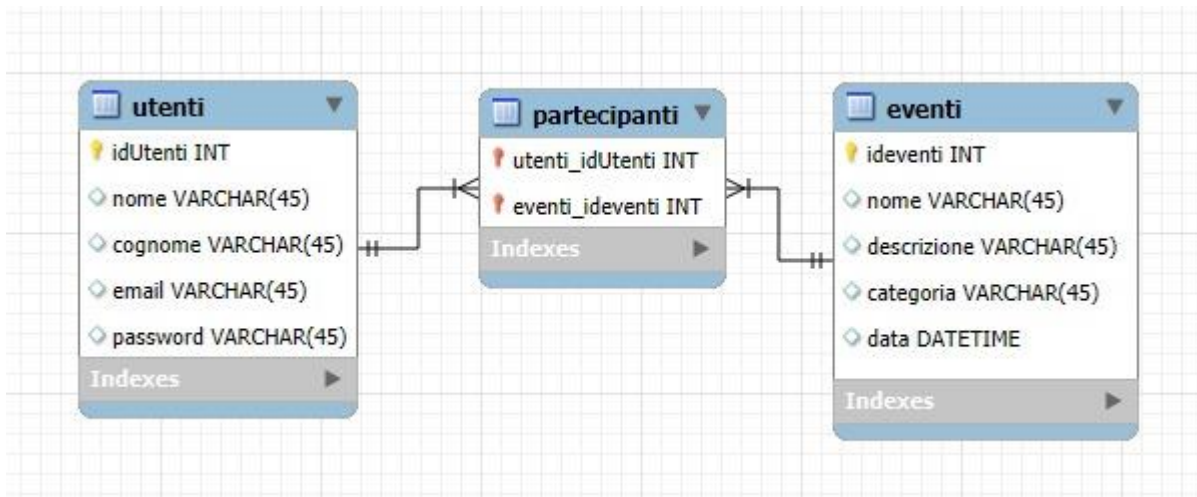
Gli utenti possono anche registrarsi per partecipare agli eventi.

Il progetto è sviluppato usando **Spring Boot**, con **Spring MVC** e **Postman** per la gestione delle richieste HTTP, **Spring Data JPA** per l'accesso ai dati del database relazionale nell'applicazione Java, e **MySQL** come database per memorizzare i dati relativi agli eventi e agli utenti.

2. Architettura del Sistema:

- **Back-End (Spring Boot):** gestisce la logica di business e l'interazione con il database. Include i servizi per la creazione, modifica, eliminazione degli eventi e per la gestione degli utenti;
- **Database (MySQL):** memorizza i dati degli eventi, degli utenti e le informazioni sulla registrazione agli eventi.

3. Progettazione del Database:



Realizzazione della **modellazione concettuale** (entità-relazione) per ottenere una visione semplificata della realtà d'interesse.

In questo caso, abbiamo ipotizzato due entità, ovvero **eventi** e **utenti**, con vari attributi, che tra loro hanno una relazione **multi-a-molti** (N:M), con un'associazione che abbiamo deciso di rappresentare con una terza entità chiamata **partecipanti**, che contiene le chiavi esterne di entrambe le entità.









Table Name:

Charset/Collation:

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
 id_utenti	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
 nome	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 cognome	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 email	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 password	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL

Realizzazione della **modellazione fisica** della tabella **utenti**, all'interno di un database chiamato **gestione_eventi**, che memorizza informazioni sugli utenti con i seguenti campi:

- **id_utenti**: chiave primaria dell'utente, con i vincoli **NOT NULL** per indicare che il valore della colonna non potrà mai essere mancante, e **AUTO_INCREMENT** per generare

automaticamente il valore dell'identificatore univoco durante il popolamento del database;

- **nome:** nome dell'utente;
- **cognome:** cognome dell'utente;
- **email:** indirizzo email dell'utente con un vincolo **UNIQUE** per indicare che il valore della colonna non potrà mai essere duplicato;
- **password:** password per l'accesso dell'utente.









Table Name:

Charset/Collation:

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
 id_evento	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
 nome	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 categoria	VARCHAR(45)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 descrizione	VARCHAR(255)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
 data	DATE	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	NULL
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Realizzazione della **modellazione fisica** della tabella **eventi**, all'interno di un database chiamato **gestione_eventi**, che memorizza informazioni sugli eventi, con i seguenti campi:

- **id_evento:** chiave primaria dell'evento, con i vincoli **NOT NULL** per indicare che i valori della colonna non potranno mai essere mancanti, e **AUTO_INCREMENT** per generare automaticamente i valori dell'identificatore univoco durante il popolamento del database;
- **nome:** nominativo dell'evento;
- **categoria:** tipologia di evento (es: Musica, Tecnologia, Sport, etc...);
- **descrizione:** una breve descrizione dell'evento;
- **data:** la data dell'evento.






Table Name:

Charset/Collation:

Comments:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
 utenti_idUtenti	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text"/>
 eventi_ideventi	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text"/>
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="text"/>

Realizzazione della **modellazione fisica** della tabella **partecipanti**, all'interno di un database chiamato **gestione_eventi**, che gestisce la relazione tra utenti ed eventi, con i seguenti campi:

- **utenti_idUtenti**: identificatore univoco dell'utente che partecipa (chiave esterna);
- **eventi_ideventi**: identificatore univoco dell'evento a cui partecipa l'utente (chiave esterna).

Inoltre, la combinazione di questi due campi, è la chiave primaria della tabella.

4. Progettazione Spring:

Dopo aver generato il progetto di base con le dovute dipendenze con **Spring Initializr** e aver compilato il file di configurazione **application.properties** con le dovute componenti, come le informazioni per la connessione al database e la porta in ascolto al server, sono pronto a modellare il mio progetto con le specifiche richieste:

Utente.java:

```
src > main > java > com > example > gestioneEventi > model > J Utente.java > {} com.example.gestioneEventi.model
1 package com.example.gestioneEventi.model;
2
3 import java.util.List;
4
5 import com.fasterxml.jackson.annotation.JsonIgnore;
6
7 import jakarta.persistence.Column;
8 import jakarta.persistence.Entity;
9 import jakarta.persistence.GeneratedValue;
10 import jakarta.persistence.GenerationType;
11 import jakarta.persistence.Id;
12 import jakarta.persistence.ManyToOne;
13
14 @Entity
15 public class Utente {
16
17     @Id
18     @GeneratedValue(strategy = GenerationType.IDENTITY)
19     private Long id;
20
21     @Column(name = "nome")
22     private String nome;
23     @Column(name = "cognome")
24     private String cognome;
25     @Column(name = "email")
26     private String email;
27     @Column(name = "password")
28     private String password;
29
30     @ManyToOne(mappedBy = "utenti")
31     private List<Evento> eventi;
32
33     public Utente() {
34
35     }
36
37     public Utente(Long id, String nome, String cognome, String email, String password) {
38         this.id = id;
39         this.nome = nome;
40         this.cognome = cognome;
41         this.email = email;
42         this.password = password;
43     }
44
45     public Long getId() {
46         return id;
47     }
48 }
```

```

49     public String getNome() {
50         return nome;
51     }
52
53     public void setNome(String nome) {
54         this.nome = nome;
55     }
56
57     public String getCognome() {
58         return cognome;
59     }
60
61     public void setCognome(String cognome) {
62         this.cognome = cognome;
63     }
64
65     @JsonIgnore
66     public List<Evento> getEventi() {
67         return eventi;
68     }
69
70     public String getEmail() {
71         return email;
72     }
73
74     public void setEmail(String email) {
75         this.email = email;
76     }
77
78     public String getPassword() {
79         return password;
80     }
81
82     public void setPassword(String password) {
83         this.password = password;
84     }
85
86     @Override
87     public String toString() {
88         return "Utente [id=" + id + ", nome=" + nome + ", cognome=" + cognome + ", email=" + email + ", password="
89             + password + "]\n";
90     }
91
92 }
93

```

La seguente classe Java, creato all'interno di una cartella chiamata **models**, rappresenta l'entità utente utilizzando **JPA** (Java Persistence API) per la gestione dei dati in un database.

Infatti, ritroviamo delle annotazioni che vanno ad indicare la classe come un'entità JPA, cioè una rappresentazione di una tabella nel database.

Una tra queste è la **@GeneratedValue(strategy = GenerationType.IDENTITY)** associata alla variabile id, che va ad indicare che quest'ultima viene generata automaticamente dal database.

Ovviamente, essendo la relazione tra la tabella evento e utente una molti-a-molti, utilizzeremo un'annotazione **@ManyToMany(mappedBy = "utenti")**, dove per l'appunto un utente può partecipare a più eventi e un evento può avere più utenti.

Evento.java:

src > main > java > com > example > gestioneEventi > model > J Evento.java > Evento > Evento(Long, String, String, String, LocalDate)

```
1 package com.example.gestioneEventi.model;
2
3 import java.time.LocalDate;
4 import java.util.List;
5
6 import com.fasterxml.jackson.annotation.JsonFormat;
7 import com.fasterxml.jackson.annotation.JsonIgnore;
8
9 import jakarta.persistence.Column;
10 import jakarta.persistence.Entity;
11 import jakarta.persistence.GeneratedValue;
12 import jakarta.persistence.GenerationType;
13 import jakarta.persistence.Id;
14 import jakarta.persistence.JoinColumn;
15 import jakarta.persistence.JoinTable;
16 import jakarta.persistence.ManyToMany;
17
18 @Entity
19 public class Evento {
20
21     @Id
22     @GeneratedValue(strategy = GenerationType.IDENTITY)
23     private Long id;
24
25     @Column(name = "nome")
26     private String nome;
27     @Column(name = "descrizione")
28     private String descrizione;
29     @Column(name = "categoria")
30     private String categoria;
31
32     @Column(name = "data")
33     @JsonFormat(pattern = "dd-MM-yyyy")
34     private LocalDate data;
35
36     @ManyToMany
37     @JoinTable(name = "partecipanti", joinColumns =
38     @JoinColumn(name = "id_evento", referencedColumnName = "id"), inverseJoinColumns =
39     @JoinColumn(name = "id_utente", referencedColumnName = "id", nullable = true))
40     private List<Utente> utenti;
41
42     public Evento() {
43
44     }
```

```

45
46 public Evento(Long id, String nome, String descrizione, String categoria, LocalDate data) {
47     this.id = id;
48     this.nome = nome;
49     this.descrizione = descrizione;
50     this.categoria = categoria;
51     this.data = data;
52 }
53
54 public Long getId() {
55     return id;
56 }
57
58 public void setId(Long id) {
59     this.id = id;
60 }
61
62 public String getNome() {
63     return nome;
64 }
65
66 public void setNome(String nome) {
67     this.nome = nome;
68 }
69
70 public String getDescrizione() {
71     return descrizione;
72 }
73
74 public void setDescrizione(String descrizione) {
75     this.descrizione = descrizione;
76 }
77
78 public String getCategoria() {
79     return categoria;
80 }
81
82 public void setCategoria(String categoria) {
83     this.categoria = categoria;
84 }
85
86 public LocalDate getData() {
87     return data;
88 }
89
90 public void setData(LocalDate data) {
91     this.data = data;
92 }
93
94 @Override
95 public String toString() {
96     return "Evento [id=" + id + ", nome=" + nome + ", descrizione=" + descrizione + ", categoria=" + categoria
97         + ", data=" + data + "];"
98 }
99
100 @JsonIgnore
101 public List<Utente> getUtenti() {
102     return utenti;
103 }
104
105 }
106

```

La seguente classe Java, creato all'interno di un cartella chiamata **model**, rappresenta l'entità evento utilizzando **JPA** (Java Persistence API) per la gestione dei dati in un database.

Infatti, ritroviamo delle annotazioni che vanno ad indicare la classe come un'entità JPA, cioè una rappresentazione di una tabella nel database.

Una tra queste è la **@GeneratedValue(strategy = GenerationType.IDENTITY)** associata alla variabile `id`, che va ad indicare che quest'ultima viene generata automaticamente dal database.

Ovviamente, essendo la relazione tra la tabella evento e utente una molti-a-molti, utilizzeremo un'annotazione **@ManyToMany**, dove per l'appunto un utente può partecipare a più eventi e un evento può avere più utenti.

Con la **@JsonFormat** impostiamo il formato della data di tipo `LocalDate`, che sarà utilizzata quando l'oggetto `Evento` sarà serializzato in JSON con il pattern **dd-MM-yyyy** che specifica che la data deve essere rappresentata nel formato giorno-mese-anno.

Con la **@JoinTable** ci riferiamo alla tabella partecipanti, ovvero la join, che contiene entrambe le chiavi primarie di entrambe le entità correlate.

Con **@JoinColumn** ci riferiamo alla colonna che rappresenta la chiave esterna della tabella di join riferita all'entità principale (in questo caso la classe **Evento**), mentre con **inverseJoinColumn** ci riferiamo alla colonna che rappresenta la chiave esterna dell'entità inversa (in questo la classe **Utente**) nella tabella partecipanti.

UtenteRepository.java:

```
src > main > java > com > example > gestioneEventi > repositories > UtenteRepository.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.repositories
1 package com.example.gestioneEventi.repositories;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.JpaRepository;
6 import org.springframework.data.jpa.repository.Query;
7 import org.springframework.data.repository.query.Param;
8 import org.springframework.stereotype.Repository;
9
10 import com.example.gestioneEventi.model.Utente;
11
12 @Repository
13 public interface UtenteRepository extends JpaRepository<Utente, Long> {
14
15     @Query(nativeQuery = true, value = "SELECT U.nome, U.cognome FROM Utenti u JOIN Partecipanti p ON u.id_utente = p.id_utente WHERE P.id_evento = :id_evento")
16     public List<Utente> findAllByEvento(@Param("id_evento") Long id);
17
18 }
19
```

La seguente interfaccia Java, creata all'interno di un cartella chiamata **repositories**, estende lo standard `JpaRepository` della tecnologia Spring Data JPA, per ereditare tutte le operazioni di base della **CRUD** (Create, Read, Update, Delete) per la gestione dell'entità utente, fornendo metodi per interrogare il database e ottenere informazioni sugli utenti:

- **findByEmailAndPassword(String email, String password):**

Questo metodo consente di recuperare un utente in base alla sua email e password, restituendo un oggetto di tipo **Optional<Utente>**, per la ricerca di un utente che corrisponderà ai parametri della ricerca (email e password), altrimenti mi restituirà un **Optional vuoto**, cioè senza un valore dentro;

- **findAllByEvento(Long id_evento):**

Questo metodo consente di ottenere tutti gli utenti che partecipano a un determinato evento in base all'id dell'evento, restituendo una lista di oggetti **List<Utente>**, contenente gli utenti che partecipano all'evento identificato dal parametro id_evento.

L'annotazione **@Repository** indica che questa interfaccia è un componente di persistenza che si occupa della gestione degli utenti nell'applicazione.

L'annotazione **@Query** è utilizzata per eseguire query personalizzate, in questo caso, nello specifico, scritte in **SQL nativo**, e quindi senza l'uso di astrazioni di Java o altri linguaggi di alto livello.

Ovviamente, utilizziamo l'annotazione **@Param** è utilizzata per associare i parametri nominati Java (**:email** e **:password**) ai segnaposto della query SQL.

EventoRepository.java:

```

src > main > java > com > example > gestioneEventi > repositories > J EventoRepository.java > ...
1  package com.example.gestioneEventi.repositories;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4  import org.springframework.stereotype.Repository;
5
6  import com.example.gestioneEventi.model.Evento;
7  import java.util.List;
8  import java.time.LocalDate;
9
10 @Repository
11 public interface EventoRepository extends JpaRepository<Evento, Long> {
12
13     public List<Evento> findByData(LocalDate data);
14
15     public List<Evento> findByCategoria(String categoria);
16
17 }
18

```

La seguente interfaccia Java, creata all'interno di una cartella chiamata **repositories**, estende lo standard `JpaRepository` della tecnologia Spring Data JPA, per ereditare tutte le operazioni di base della **CRUD** (Create, Read, Update, Delete) per la gestione dell'entità evento, fornendo metodi per interrogare il database e ottenere informazioni sugli eventi:

- **findByData(LocalDate data):**

Questo metodo consente di recuperare una lista di eventi che si svolgono in una determinata data, restituendo una lista di oggetti **List<Evento>** che corrispondono alla data specificata;

- **findByCategoria(String categoria):**

Questo metodo consente di recuperare una lista di eventi appartenenti a una determinata categoria, restituendo una lista di oggetti **List<Evento>** che appartengono alla categoria specificata.

L'annotazione **@Repository** indica che questa interfaccia è un componente di persistenza che si occupa della gestione degli utenti nell'applicazione.

UtenteService.java:

```
src > main > java > com > example > gestioneEventi > services > UtenteService.java > {} com.example.gestioneEventi.services
1 package com.example.gestioneEventi.services;
2
3 import java.util.List;
4
5 import com.example.gestioneEventi.model.Utente;
6
7 public interface UtenteService {
8
9     public Utente recuperaUno(long id);
10
11     public List<Utente> recuperaByEvento(Long id);
12
13     public Boolean salva(Utente utente);
14
15     public void elimina(Long id);
16
17 }
18
```

La seguente interfaccia Java, creata all'interno di una cartella chiamata **services**, definisce i metodi per la gestione degli utenti:

- **recuperaUno(long id):**
Questo metodo consente di recuperare un singolo utente identificato dal suo id, restituendo l'oggetto Utente corrispondente;
- **recuperaByEvento(Long id):**
Questo metodo consente di recuperare una lista di utenti associati a un evento specifico, identificato dal suo id, restituendo una lista di oggetti Utente;
- **salva(Utente utente):**
Questo metodo consente di salvare un nuovo utente o aggiornare uno esistente nel sistema, restituendo un valore booleano che indica se l'operazione di salvataggio è riuscita (vero se l'operazione è riuscita, falso in caso contrario);
- **elimina(Long id):**
Questo metodo consente di eliminare l'utente identificato dal suo id, e non restituisce alcun valore.

EventoService.java:

```

src > main > java > com > example > gestioneEventi > services > EventoService.java > EventoService
1  package com.example.gestioneEventi.services;
2
3  import java.time.LocalDate;
4  import java.util.List;
5
6  import com.example.gestioneEventi.model.Evento;
7
8  public interface EventoService {
9
10     public List<Evento> recuperaTutti();
11
12     public Evento recuperaUno(long id);
13
14     public boolean salva(Evento evento);
15
16     public void elimina(Long id);
17
18     public List<Evento> recuperaEventiByCategoria(String categoria);
19
20     public List<Evento> recuperaEventiByData(LocalDate data);
21
22 }
23

```

La seguente interfaccia Java, creata all'interno di una cartella chiamata **services**, definisce i metodi per la gestione degli eventi:

- **recuperaTutti():**
Questo metodo consente di restituire una lista di tutti eventi registrati nel sistema;
- **recuperaUno(long id):**
Questo metodo consente di recuperare un singolo evento identificato dal suo id, restituendo l'oggetto Evento corrispondente;
- **salva(Evento evento):**
Questo metodo consente di salvare un nuovo evento o aggiornare uno esistente nel sistema, restituendo un valore booleano che indica se l'operazione di salvataggio è riuscita (vero se l'operazione è riuscita, falso in caso contrario);
- **elimina(Long id):**
Questo metodo consente di eliminare l'evento identificato dal suo id, e non restituisce alcun valore;

- **recuperaEventiByCategoria(String categoria):**

Questo metodo consente di restituire una lista di eventi filtrati per categoria;

- **recuperaEventiByData(LocalDate data):**

Questo metodo consente di restituire una lista di eventi che si svolgono in una specifica data.

PartecipantiService.java:

```
src > main > java > com > example > gestioneEventi > services > J PartecipantiService.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.services
1  package com.example.gestioneEventi.services;
2
3  import java.util.List;
4  import java.util.Optional;
5
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.stereotype.Service;
8
9  import com.example.gestioneEventi.model.Evento;
10 import com.example.gestioneEventi.model.Utente;
11 import com.example.gestioneEventi.repositories.EventoRepository;
12 import com.example.gestioneEventi.repositories.UtenteRepository;
13
14 @Service
15 public class PartecipantiService {
16
17     @Autowired
18     private EventoRepository eventoRepo;
19
20     @Autowired
21     private UtenteRepository utenteRepo;
22
23     public boolean registrazioneUtente(Long idUtente, Long idEvento) {
24
25         Optional<Utente> utenteDB = utenteRepo.findById(idUtente);
26         Optional<Evento> eventoDB = eventoRepo.findById(idEvento);
27
28         Utente utente = utenteDB.get();
29         Evento evento = eventoDB.get();
30
31         if (utente != null && evento != null) {
32             utente.getEventi().add(evento);
33             evento.getUtenti().add(utente);
34
35             utenteRepo.save(utente);
36             eventoRepo.save(evento);
37
38             return true;
39         }
40
41         return false;
42     }
43 }
44
```

```

45     public List<Utente> getUtentyByEvento(Long idEvento) {
46
47         Optional<Evento> eventoDB = eventoRepo.findById(idEvento);
48         Evento evento = eventoDB.get();
49
50         if (evento == null)
51             return null;
52
53         return evento.getUtenti();
54     }
55
56     public List<Evento> getEventiByUtente(Long idUtente) {
57
58         Optional<Utente> utenteDB = utenteRepo.findById(idUtente);
59         Utente utente = utenteDB.get();
60
61         if (utente == null)
62             return null;
63
64         return utente.getEventi();
65     }
66
67 }
68

```

La seguente classe Java, creata all'interno di una cartella chiamata **services**, definisce i metodi per la gestione delle operazioni relative alla registrazione di utenti a eventi e alla gestione delle relazioni tra gli utenti e gli eventi:

- **registrazioneUtente(Long idUtente, Long idEvento):**

Questo metodo gestisce la registrazione di un utente a un evento, ricevendo gli id di un utente e di un evento come parametri.

Recupera l'utente e l'evento dai rispettivi repository, e se entrambi esistono, aggiunge l'evento alla lista degli eventi dell'utente e l'utente alla lista degli utenti dell'evento.

Se l'operazione è riuscita (utente ed evento esistono), restituisce true, altrimenti restituisce false;

- **getUtentiByEvento(Long idEvento):**

Questo metodo recupera tutti gli utenti registrati a un evento specifico, dato l'id dell'evento.

Se l'evento esiste, restituisce la lista degli utenti associati, altrimenti restituisce null;

- **getEventiByUtente(Long idUtente):**

Questo metodo recupera tutti gli eventi a cui un utente è registrato, dato l'id dell'utente.

Se l'utente esiste, restituisce la lista degli eventi associati, altrimenti restituisce null.

UtenteServiceImpl.java:

```
src > main > java > com > example > gestioneEventi > services > J UtenteServiceImpl.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.services
1  package com.example.gestioneEventi.services;
2
3  import java.util.List;
4  import java.util.Optional;
5
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.stereotype.Service;
8
9  import com.example.gestioneEventi.model.Evento;
10 import com.example.gestioneEventi.model.Utente;
11 import com.example.gestioneEventi.repositories.EventoRepository;
12 import com.example.gestioneEventi.repositories.UtenteRepository;
13
14 import jakarta.persistence.EntityNotFoundException;
15
16 @Service
17 public class UtenteServiceImpl implements UtenteService {
18
19     @Autowired
20     private UtenteRepository utenteRepo;
21
22     @Autowired
23     private EventoRepository eventoRepo;
24
25     @Override
26     public Utente recuperaUno(long id) {
27
28         Optional<Utente> u = utenteRepo.findById(id);
29         return u.isEmpty() ? null : u.get();
30     }
31
32     @Override
33     public Boolean salva(Utente utente) {
34         boolean esito = true;
35
36         try {
37             utenteRepo.save(utente);
38         } catch (Exception e) {
39             esito = false;
40         }
41
42         return esito;
43     }
44 }
```



```

45 // @Override
46 // public void elimina(Long id) {
47 //     utenteRepo.deleteById(id);
48 // }
49
50 public void elimina(Long userId) {
51     Utente utente = utenteRepo.findById(userId)
52         .orElseThrow(() -> new EntityNotFoundException(message:"Utente non trovato"));
53
54     for (Evento evento : utente.getEventi()) {
55         evento.getUtenti().remove(utente);
56         eventoRepo.save(evento);
57     }
58
59     // Elimina l'utente
60     utenteRepo.delete(utente);
61 }
62
63 @Override
64 public List<Utente> recuperaByEvento(Long id) {
65     return utenteRepo.findAllByEvento(id);
66 }
67
68 }
69
70
71
72

```

La seguente classe Java, creata all'interno di una cartella chiamata **services**, implementa l'interfaccia `UtenteService` e i suoi metodi per la gestione degli utenti.

L'annotazione **@Autowired** permette di "iniettare" automaticamente un'istanza di `UtenteRepository` chiamata **utenteRepo**, che è il componente responsabile della persistenza dei dati relativi agli utenti nel database.

Strumento potente di **Dependency Injection (DI)** in Spring, dato che semplifica la gestione delle dipendenze e rende il codice più pulito e manutenibile.

EventoServiceImpl.java:

```
src > main > java > com > example > gestioneEventi > services > J EventoServiceImpl.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.services
1  package com.example.gestioneEventi.services;
2
3  import java.time.LocalDate;
4  import java.util.List;
5  import java.util.Optional;
6
7  import org.springframework.beans.factory.annotation.Autowired;
8  import org.springframework.stereotype.Service;
9
10 import com.example.gestioneEventi.model.Evento;
11 import com.example.gestioneEventi.repositories.EventoRepository;
12
13 @Service
14 public class EventoServiceImpl implements EventoService {
15
16     @Autowired
17     private EventoRepository eventoRepo;
18
19     @Override
20     public List<Evento> recuperaTutti() {
21
22         return eventoRepo.findAll();
23     }
24
25     @Override
26     public Evento recuperaUno(long id) {
27
28         Optional<Evento> e = eventoRepo.findById(id);
29
30         return e.isEmpty() ? null : e.get();
31     }
32
33     @Override
34     public boolean salva(Evento evento) {
35
36         boolean esito = true;
37
38         try {
39             eventoRepo.save(evento);
40         } catch (Exception e) {
41             esito = false;
42         }
43
44         return esito;
45     }
46 }
```

```
47     @Override
48     public void elimina(Long id) {
49
50         eventoRepo.deleteById(id);
51     }
52
53     public List<Evento> recuperaEventiByCategoria(String categoria) {
54
55         return eventoRepo.findByCategoria(categoria);
56     }
57
58     public List<Evento> recuperaEventiByData(LocalDate data) {
59
60         return eventoRepo.findByData(data);
61     }
62
63 }
64
```

La seguente classe Java, creata all'interno di una cartella chiamata **services**, implementa l'interfaccia **EventoService** e i suoi metodi per la gestione degli eventi.

L'annotazione **@Autowired** permette di "iniettare" automaticamente un'istanza di **EventoRepository** chiamata **eventoRepo**, che è il componente responsabile della persistenza dei dati relativi agli eventi nel database.

Strumento potente di **Dependency Injection (DI)** in Spring, dato che semplifica la gestione delle dipendenze e rende il codice più pulito e manutenibile.

UtenteController.java:

```
src > main > java > com > example > gestioneEventi > controller > J UtenteController.java > Language Support for Java(TM) by Red Hat > UtenteController
1  package com.example.gestioneEventi.controller;
2
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.http.HttpStatus;
5  import org.springframework.http.ResponseEntity;
6  import org.springframework.web.bind.annotation.DeleteMapping;
7  import org.springframework.web.bind.annotation.GetMapping;
8  import org.springframework.web.bind.annotation.PathVariable;
9  import org.springframework.web.bind.annotation.RequestMapping;
10 import org.springframework.web.bind.annotation.RestController;
11
12 import com.example.gestioneEventi.model.Utente;
13 import com.example.gestioneEventi.services.UtenteService;
14
15 @RestController
16 @RequestMapping("/utenti")
17 public class UtenteController {
18
19     @Autowired
20     private UtenteService service;
21
22     // http://localhost:8080/gestione_eventi/utenti/1
23     @GetMapping("/{id}")
24     public ResponseEntity<Utente> getUser(@PathVariable Long id) {
25
26         if (service.recuperaUno(id) != null) {
27             return ResponseEntity.status(HttpStatus.OK).body(service.recuperaUno(id));
28         } else {
29             return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
30         }
31     }
32
33     // http://localhost:8080/gestione_eventi/utenti/1
34     @DeleteMapping("/{id}")
35     public ResponseEntity<Utente> elimina(@PathVariable Long id) {
36
37         Utente u = service.recuperaUno(id);
38
39         if (u != null) {
40             service.elimina(id);
41             return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
42         } else {
43             return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
44         }
45     }
46 }
47
```

La seguente classe Java, creata all'interno di una cartella chiamata **controller**, è per l'appunto un controller di tipo REST che gestisce le operazioni HTTP relative agli utenti.

Essa fornisce endpoint per recuperare, creare, aggiornare ed eliminare utenti, utilizzando il servizio **UtenteService** per la logica di business.

L'annotazione **@RestController** indica che classe è un controller REST e che i metodi restituiscono dati direttamente, non visualizzazioni.

Inoltre, può rispondere solo se gli viene assegnato un input completamente corretto, altrimenti lo ignora.

L'annotazione **@RequestMapping("/utenti")** definisce il prefisso degli endpoint, quindi tutte le richieste di questa classe che iniziano con /utenti.

Utilizziamo anche oggetti **ResponseEntity** per controllare i codici di stato HTTP e restituire i dati appropriati.

Funzionalità principali:

- **GET /utenti/{id}:**
Questo metodo gestisce le richieste GET per ottenere un utente specifico, dato un id tramite un variabile path.
Se l'utente esiste, restituisce un codice di stato 200 (OK) con i dettagli dell'utente, altrimenti restituisce un codice di stato 404 (NOT FOUND);
- **DELETE /utenti/{id}:**
Questo metodo gestisce le richieste DELETE per eliminare un utente specifico, dato un id tramite una variabile path.
Se l'utente esiste, viene eliminato e viene restituito un codice di stato HTTP 204 (NO CONTENT), che indica che l'eliminazione è avvenuta con successo, altrimenti viene restituito un codice di stato http 404 (NOT FOUND).

EventoController.java:

```
src > main > java > com > example > gestioneEventi > controller > J EventoController.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.controller

1  package com.example.gestioneEventi.controller;
2
3  import java.time.LocalDate;
4  import java.util.List;
5
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.http.HttpStatus;
8  import org.springframework.http.ResponseEntity;
9  import org.springframework.web.bind.annotation.DeleteMapping;
10 import org.springframework.web.bind.annotation.GetMapping;
11 import org.springframework.web.bind.annotation.PathVariable;
12 import org.springframework.web.bind.annotation.PostMapping;
13 import org.springframework.web.bind.annotation.PutMapping;
14 import org.springframework.web.bind.annotation.RequestBody;
15 import org.springframework.web.bind.annotation.RequestMapping;
16 import org.springframework.web.bind.annotation.RestController;
17
18 import com.example.gestioneEventi.model.Evento;
19 import com.example.gestioneEventi.services.EventoService;
20
21 import jakarta.validation.Valid;
22
23 @RestController
24 @RequestMapping("/eventi")
25 public class EventoController {
26
27     @Autowired
28     private EventoService eventoService;
29
30     // http://localhost:8080/gestione_eventi/eventi
31     @GetMapping
32     public List<Evento> getEventi() {
33
34         return eventoService.recuperaTutti();
35     }
36
37     // http://localhost:8080/gestione_eventi/eventi/1
38     @GetMapping("/{id}")
39     public ResponseEntity<Evento> getEventiById(@PathVariable Long id) {
40
41         Evento evento = eventoService.recuperaUno(id);
42
43         if (evento != null)
44             return ResponseEntity.ok(evento);
45         else
46             return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
47     }
48 }
```

```

49 // http://localhost:8080/gestione_eventi/eventi/categoria/sport
50 @GetMapping("/categoria/{categoria}")
51 public ResponseEntity<List<Evento>> getEventiByCategoria(@PathVariable String categoria) {
52
53     List<Evento> eventi = eventoService.recuperaEventiByCategoria(categoria);
54
55     if (eventi != null)
56         return ResponseEntity.status(HttpStatus.OK).body(eventi);
57     else
58         return ResponseEntity.status(HttpStatus.BAD_GATEWAY).build();
59 }
60
61 // http://localhost:8080/gestione_eventi/eventi/data/2024-03-10
62 @GetMapping("/data/{data}")
63 public ResponseEntity<List<Evento>> getEventiByData(@PathVariable LocalDate data) {
64
65     List<Evento> eventi = eventoService.recuperaEventiByData(data);
66
67     if (eventi != null)
68         return ResponseEntity.status(HttpStatus.OK).body(eventi);
69     else
70         return ResponseEntity.status(HttpStatus.BAD_GATEWAY).build();
71 }
72
73 // http://localhost:8080/gestione_eventi/eventi/crea
74 @PostMapping("/crea")
75 public ResponseEntity<Evento> creaEvento(@RequestBody @Valid Evento evento) {
76
77     if (eventoService.salva(evento))
78         return ResponseEntity.status(HttpStatus.CREATED).body(evento);
79     else
80         return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).build();
81 }
82

```

```

83 // http://localhost:8080/gestione_eventi/eventi/8
84 @PutMapping("/{id}")
85 public ResponseEntity<Evento> modificaEvento(@PathVariable Long id, @RequestBody @Valid Evento evento) {
86
87     Evento eventoTrovato = eventoService.recuperaUno(id);
88
89     if (eventoTrovato != null) {
90
91         evento.setId(eventoTrovato.getId());
92         eventoService.salva(evento);
93         return ResponseEntity.status(HttpStatus.OK).body(evento);
94     } else
95         return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
96 }
97
98 // http://localhost:8080/gestione_eventi/eventi/8
99 @DeleteMapping("/{id}")
100 public ResponseEntity<Evento> elimina(@PathVariable Long id) {
101
102     Evento eventoTrovato = eventoService.recuperaUno(id);
103
104     if (eventoTrovato != null) {
105
106         eventoService.elimina(id);
107         return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
108     } else
109         return ResponseEntity.status(HttpStatus.NOT_FOUND).build();
110 }
111
112 }
113
114

```

La seguente classe Java, creata all'interno di una cartella chiamata **controller**, è per l'appunto un controller di tipo REST che gestisce le operazioni HTTP relative agli eventi.

Essa fornisce endpoint per recuperare, creare, aggiornare ed eliminare eventi, utilizzando il servizio **EventoService** per la logica di business.

L'annotazione **@RestController** indica che classe è un controller REST e che i metodi restituiscono dati direttamente, non visualizzazioni.

Inoltre, può rispondere solo se gli viene assegnato un input completamente corretto, altrimenti lo ignora.

L'annotazione **@RequestMapping("/eventi")** definisce il prefisso degli endpoint, quindi tutte le richieste di questa classe che iniziano con /eventi.

L'annotazione **@Valid** valida, per l'appunto, l'oggetto Evento nelle richieste di tipo POST e PUT.

Utilizziamo anche oggetti **ResponseEntity** per controllare i codici di stato HTTP e restituire i dati appropriati.

Funzionalità principali:

- **GET /eventi:**
Recupera tutti gli eventi tramite il metodo recuperaTutti() di EventoService e li restituisce come lista;
- **GET /eventi/{id}:**
Recupera un evento specifico per ID tramite il metodo recuperaUno().
Se l'evento è trovato, restituisce un codice di stato 200 (OK), altrimenti restituisce un codice di stato 404 (NOT FOUND);
- **GET /eventi/categoria/{categoria}:**
Recupera una lista di eventi filtrata per categoria utilizzando il metodo recuperaEventiByCategoria().
Restituisce una risposta con codice di stato 200 (OK) se la lista non è vuota, altrimenti restituisce un codice di stato 502 (BAD GATEWAY);
- **GET /eventi/data/{data}:**
Recupera eventi per una data specifica utilizzando recuperaEventiByData().
Restituisce una risposta con codice di stato 200 (OK) o 502 (BAD GATEWAY) in base alla disponibilità degli eventi;
- **POST /eventi/crea:**
Crea un nuovo evento.

Se l'evento viene salvato correttamente, restituisce un codice di stato 201 (CREATED) con i dettagli dell'evento, altrimenti in caso di errore, restituisce un codice di stato 500 (INTERNAL SERVER ERROR);

- **PUT /eventi/{id}:**

Modifica un evento esistente.

Se l'evento è trovato, aggiorna l'evento e restituisce un codice 200 (OK) con il nuovo evento, altrimenti restituisce un codice di stato 404 (NOT FOUND);

- **DELETE /eventi/{id}:**

Elimina un evento specifico.

Se l'evento viene trovato ed eliminato con successo, restituisce un codice di stato 204 (NO CONTENT), altrimenti se l'evento non esiste, restituisce un codice di stato 404 (NOT FOUND).

PartecipantiController.java:

```
src > main > java > com > example > gestioneEventi > controller > J PartecipantiController.java > Language Support for Java(TM) by Red Hat > {} com.example.gestioneEventi.controller
1  package com.example.gestioneEventi.controller;
2
3  import java.util.List;
4
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.http.HttpStatus;
7  import org.springframework.http.ResponseEntity;
8  import org.springframework.web.bind.annotation.GetMapping;
9  import org.springframework.web.bind.annotation.PathVariable;
10 import org.springframework.web.bind.annotation.PostMapping;
11 import org.springframework.web.bind.annotation.RequestMapping;
12 import org.springframework.web.bind.annotation.RestController;
13
14 import com.example.gestioneEventi.model.Evento;
15 import com.example.gestioneEventi.model.Utente;
16 import com.example.gestioneEventi.services.PartecipantiService;
17
18 @RestController
19 @RequestMapping("/registrazione")
20 public class PartecipantiController {
21
22     @Autowired
23     private PartecipantiService partecipantiService;
24
25     // http://localhost:8080/gestione_eventi/registrazione/2/3
26     @PostMapping("/{idEvento}/{idUtente}")
27     public ResponseEntity<String> registraUtente(@PathVariable Long idEvento, @PathVariable Long idUtente) {
28
29         if (partecipantiService.registrazioneUtente(idUtente, idEvento))
30             return ResponseEntity.status(HttpStatus.OK).body(body: "Ti sei registrato con successo all'evento.");
31         else
32             return ResponseEntity.status(HttpStatus.NOT_FOUND).body(body: "Registrazione fallita.");
33     }
34
35
36     // http://localhost:8080/gestione_eventi/registrazione/2/utenti
37     @GetMapping("/{idEvento}/utenti")
38     public ResponseEntity<List<Utente>> getUtentiRegistratiByEvento(@PathVariable Long idEvento) {
39
40         List<Utente> utenti = partecipantiService.getUtentiByEvento(idEvento);
41
42         if (utenti.isEmpty())
43             return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
44         else
45             return ResponseEntity.status(HttpStatus.OK).body(utenti);
46     }
47
48 }
```



```

49      // http://localhost:8080/gestione_eventi/registrazione/2/eventi
50      @GetMapping("/{idUtente}/eventi")
51      public ResponseEntity<List<Evento>> getEventiByUtente(@PathVariable Long idUtente) {
52
53          List<Evento> eventi = partecipantiService.getEventiByUtente(idUtente);
54
55          if (eventi.isEmpty())
56              return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
57          else
58              return ResponseEntity.status(HttpStatus.OK).body(eventi);
59
60      }
61
62  }
63

```

La seguente classe Java, creata all'interno di una cartella chiamata **controller**, è per l'appunto un controller di tipo REST che gestisce le operazioni HTTP relative alla registrazione degli utenti agli eventi.

Essa fornisce endpoint per recuperare gli utenti registrati a un evento e gli eventi a cui un utente è registrato, utilizzando il servizio **PartecipantiService** per la logica di business.

L'annotazione **@RestController** indica che classe è un controller REST e che i metodi restituiscono dati direttamente, non visualizzazioni.

Inoltre, può rispondere solo se gli viene assegnato un input completamente corretto, altrimenti lo ignora.

L'annotazione **@RequestMapping("/registrazione")** definisce il prefisso degli endpoint, quindi tutte le richieste di questa classe che iniziano con /registrazione.

Utilizziamo anche oggetti **ResponseEntity** per controllare i codici di stato HTTP e restituire i dati appropriati.

Funzionalità principali:

- **POST /registrazione/{idEvento}/{idUtente}:**

Questo metodo gestisce le richieste POST per registrare un utente a un determinato evento, ricevendo due parametri nel path, ovvero l'id dell'evento e dell'utente.

Il metodo delega la logica di registrazione al servizio **PartecipantiService**, e se la registrazione è avvenuta con successo, restituisce un codice di stato 200 (OK), altrimenti restituisce il codice di stato 404 (NOT FOUND);

- **GET /registrazione/{idEvento}/utenti:**

Questo metodo gestisce le richieste GET per ottenere tutti gli utenti registrati a un determinato evento, ricevendo l'id dell'evento come parametro nel path.

Se sono presenti utenti registrati all'evento, restituisce la lista degli utenti con il codice di stato 200 (OK), altrimenti restituisce il codice di stato 204 (NO CONTENT);

- **GET /registrazione/{idUtente}/eventi:**

Questo metodo gestisce le richieste GET per ottenere tutti gli eventi a cui un determinato utente è registrato, ricevendo l'id dell'utente come parametro nel path.

Se l'utente ha eventi registrati, restituisce la lista degli eventi con il codice di stato 200 (OK), altrimenti restituisce il codice di stato 204 (NO CONTENT).

5. Limiti di sistema:

Primo caso:

Se inserisci nel metodo di ricerca per data all'interno di un URL una data in un formato diverso da **YYYY-MM-DD**, il link non funzionerà correttamente.

In altre parole, se la data non è nel formato previsto (anno-mese-giorno), l'URL potrebbe restituire un errore o non mostrare i risultati desiderati.

Per esempio:

- Un formato come **DD/MM/YYYY** o **MM/DD/YYYY** potrebbe non essere riconosciuto dal sistema che sta elaborando la ricerca;
- Il sistema si aspetta che la data sia nel formato **YYYY-MM-DD** (Anno-Mese-Giorno), che è un formato che abbiamo rappresentato nel pattern del @JsonFormat nella classe Evento all'interno della cartella model.

Secondo caso:

Non è stata implementata alcuna forma di **autenticazione** o **autorizzazione** all'interno del progetto (esempio: login, ruoli, etc...).

- **Conseguenze:**

Gli endpoint sono aperti e vulnerabili agli attacchi con una mancanza di controllo sugli accessi;

- **Possibile miglioramento:**

Integrare Spring Security e definire ruoli (esempio: admin, utente, etc...).