

Progetto: Gestione degli Eventi

1. Descrizione del Progetto:

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

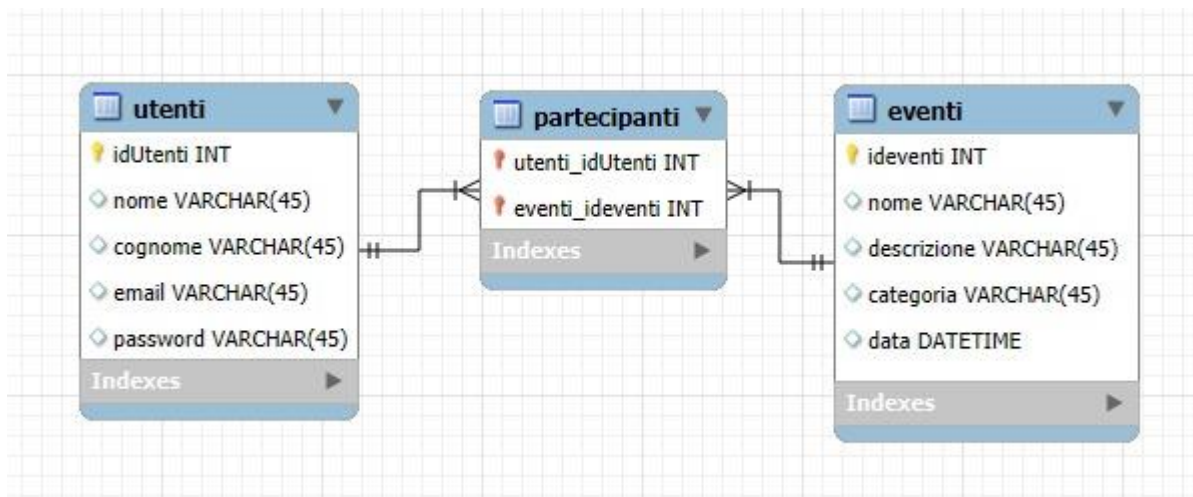
Gli utenti possono anche registrarsi per partecipare agli eventi.

Il progetto è sviluppato usando **Spring Boot**, con **Spring MVC** per la gestione delle richieste HTTP, **Spring Data JPA** per l'accesso ai dati del database relazionale nell'applicazione Java, **Thymeleaf** per la parte di visualizzazione (Front-End), 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;
- **Front-End (Thymeleaf):** fornisce una semplice interfaccia utente per visualizzare gli eventi, i dettagli e per registrarsi o loggarsi;
- **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-multi** (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

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"/>	

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 > gestioneEventi > model > J Utente.java > Utente > cognome

```
1 package com.example.gestioneEventi.gestioneEventi.model;
2
3 import java.util.List;
4
5 import jakarta.persistence.Column;
6 import jakarta.persistence.Entity;
7 import jakarta.persistence.GeneratedValue;
8 import jakarta.persistence.GenerationType;
9 import jakarta.persistence.Id;
10 import jakarta.persistence.ManyToMany;
11 import jakarta.persistence.Table;
12
13 @Entity
14 @Table(name = "utenti")
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     @ManyToMany(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     public String getEmail() {
66         return email;
67     }
68
69     public void setEmail(String email) {
70         this.email = email;
71     }
72
73     public String getPassword() {
74         return password;
75     }
76
77     public void setPassword(String password) {
78         this.password = password;
79     }
80
81     @Override
82     public String toString() {
83         return "Utente [id=" + id + ", nome=" + nome + ", cognome=" + cognome + ", email=" + email + ", password="
84             + password + "]\n";
85     }
86
87 }

```

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

```

43     public Evento(Long id, String nome, String descrizione, String categoria, LocalDate data) {
44         this.id = id;
45         this.nome = nome;
46         this.descrizione = descrizione;
47         this.categoria = categoria;
48         this.data = data;
49     }
50
51     public Long getId() {
52         return id;
53     }
54
55     public String getNome() {
56         return nome;
57     }
58
59     public void setNome(String nome) {
60         this.nome = nome;
61     }
62
63     public String getDescrizione() {
64         return descrizione;
65     }
66
67     public void setDescrizione(String descrizione) {
68         this.descrizione = descrizione;
69     }
70
71     public String getCategoria() {
72         return categoria;
73     }
74
75     public void setCategoria(String categoria) {
76         this.categoria = categoria;
77     }
78
79     public LocalDate getData() {
80         return data;
81     }
82
83     public void setData(LocalDate data) {
84         this.data = data;
85     }
86

```

```

87     @Override
88     public String toString() {
89         return "Evento [id=" + id + ", nome=" + nome + ", descrizione=" + descrizione + ", categoria=" + categoria
90             + ", data=" + data + "]";
91     }
92
93
94

```

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 **@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 > UtenteRepository
1  package com.example.gestioneEventi.repositories;
2
3  import java.util.List;
4  import java.util.Optional;
5
6  import org.springframework.data.jpa.repository.JpaRepository;
7  import org.springframework.data.jpa.repository.Query;
8  import org.springframework.data.repository.query.Param;
9  import org.springframework.stereotype.Repository;
10
11 import com.example.gestioneEventi.model.Utente;
12
13 @Repository
14 public interface UtenteRepository extends JpaRepository<Utente, Long> {
15
16     @Query(nativeQuery = true, value = "SELECT * FROM Utente U WHERE U.email = :email AND U.password = :password")
17     public Optional<Utente> findByEmailPassword(@Param("email") String email, @Param("password") String password);
18
19     @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")
20     public List<Utente> findAllByEvento(@Param("id_evento") Long id);
21
22 }
23
```

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:

- **findByEmailPassword(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 > J 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 > J EventoService.java > ...
1  package com.example.gestioneEventi.services;
2
3  import java.util.List;
4
5  import com.example.gestioneEventi.model.Evento;
6
7  public interface EventoService {
8
9      public List<Evento> recuperaTutti();
10
11     public Evento recuperaUno(long id);
12
13     public boolean salva(Evento evento);
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 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.

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.Utente;
10 import com.example.gestioneEventi.repositories.UtenteRepository;
11
12 @Service
13 public class UtenteServiceImpl implements UtenteService {
14
15     @Autowired
16     private UtenteRepository utenteRepo;
17
18     @Override
19     public Utente recuperaUno(long id) {
20
21         Optional<Utente> u = utenteRepo.findById(id);
22         return u.isEmpty() ? null : u.get();
23     }
24
25     public Utente recuperaByEmailAndPassword(String email, String password) {
26
27         Optional<Utente> u = utenteRepo.findByEmailAndPassword(email, password);
28
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     @Override
51     public List<Utente> recuperaByEvento(Long id) {
52
53         return utenteRepo.findAllByEvento(id);
54     }
55
56 }
57

```

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.

