

Programmazione avanzata

Lezione 9

Creazione di un servizio web per l'invio dei dati memorizzati in un database in formato JSON alle applicazioni

JPI e Hibernate per il recupero dati

Realizzazione di un servizio web usando Spring

Spring può essere utilizzato anche per la creazione di un servizio web il quale ad esempio può essere programmato per fornire i dati alle applicazioni leggendoli da un database¹.

Il primo passaggio per creare un servizio web è di creare lo scheletro dell'applicazione tramite Spring Boot:

The screenshot shows the Spring Initializr web application generator interface. It is divided into several sections:

- Project:** Includes radio buttons for **Maven Project** (selected) and **Gradle Project**.
- Language:** Includes radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
- Spring Boot:** Includes radio buttons for versions: 3.0.0 (SNAPSHOT), 3.0.0 (M5), 2.7.5 (SNAPSHOT), **2.7.4** (selected), 2.6.13 (SNAPSHOT), and 2.6.12.
- Project Metadata:** Includes input fields for Group (it.unipi), Artifact (ServerApp), Name (ServerApp), Description (Prima app esempio server), and Package name (it.unipi.ServerApp). It also has a **Packaging** section with **Jar** (selected) and **War**, and a **Java** version section with 19, 17, **11** (selected), and 8.
- Dependencies:** Includes a button **ADD DEPENDENCIES... CTRL + B** and a list of selected dependencies: **Spring Web** (WEB), **MySQL Driver** (SQL), and **Spring Data JPA** (SQL). Each dependency has a brief description.

A differenza dell'applicazione web creata in precedenza, questo servizio non esporrà un'interfaccia web ma fornirà solo dati. Per questo le dipendenze del progetto sono le seguenti:

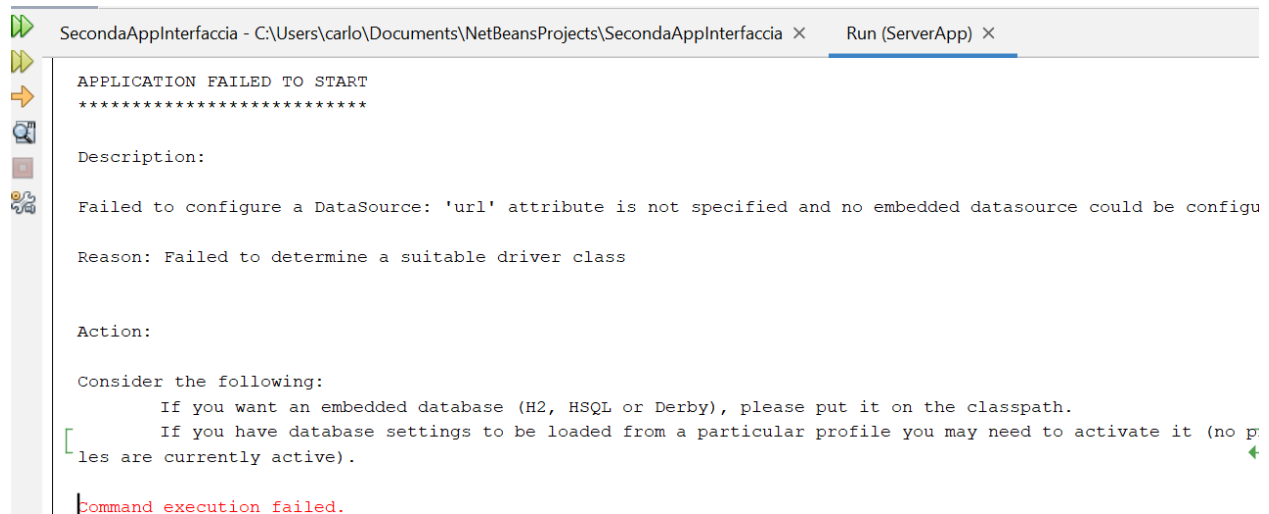
- Spring Web, come scheletro di un servizio web generico

¹ <https://spring.io/guides/gs/accessing-data-mysql/>

- MySQL Driver per l'interazione con il database
- Spring Data JPA per interagire con il database attraverso la framework JPA

L'interazione con il database può avvenire attraverso delle query SQL, come abbiamo visto finora, oppure utilizzando delle astrazioni di più alto livello che semplificano l'interazione con il database. Queste funzioni di accesso al database di alto livello sono fornite attraverso il meccanismo JPA, Java Persistence API, una serie di funzioni che servono a modellare le interazioni al database attraverso la serializzazione di oggetti, senza ricorrere a query SQL.

Una volta importata l'applicazione e dopo averla importata in NetBeans si può procedere a lanciare l'applicazione scheletro la quale darà il seguente errore:



```

SecondaAppInterfaccia - C:\Users\carlo\Documents\NetBeansProjects\SecondaAppInterfaccia × Run (ServerApp) ×
APPLICATION FAILED TO START
*****

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured

Reason: Failed to determine a suitable driver class

Action:

Consider the following:
    If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
    If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

Command execution failed.
  
```

Questo errore è dovuto al fatto che l'applicazione ha bisogno di avere le credenziali di accesso al database che vanno specificate creando il seguente file:

src/main/resources/application.properties

```

1. spring.jpa.hibernate.ddl-auto=update
2. spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/drivers
3. spring.datasource.username=root
4. spring.datasource.password=root
5. spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
6. #spring.jpa.show-sql: true
  
```

Il file contiene la stringa di connessione al database MySQL e le credenziali di accesso. A questo punto lo scheletro dell'applicazione è pronto.

Esercizio

Creare lo scheletro del servizio web

Interazione con il database con JPA

Come abbiamo già visto uno dei primi passi è quello di creare una classe che rappresenta il dato che vogliamo memorizzare.

Supponiamo nel nostro caso di voler creare un servizio alternativo che fornisce dati sui piloti di F1 leggendo i dati dal database che abbiamo creato in precedenza. Il primo passaggio è quello di creare un JavaBean che rappresenta i dati nel database:

```
1. @Entity
2. @Table(name="drivers", schema="drivers")
3. public class Driver {
4.     @Id
5.     @GeneratedValue(strategy=GenerationType.AUTO)
6.     private Integer id;
7.
8.     @Column(name="name")
9.     private String name;
10.    @Column(name="lastname")
11.    private String lastName;
12.    @Column(name="nationality")
13.    private String nationality;
14.    @Column(name="birthdate") // La maiuscola genera l'aggiunta di un _
15.    private Date birthDate;
16.
17.    public Integer getId() {
18.        return id;
19.    }
20.
21.    public void setId(Integer id) {
22.        this.id = id;
23.    }
24.
25.    public String getName() {
26.        return name;
27.    }
28.
29.    public void setName(String Name) {
30.        this.name = Name;
31.    }
32.
33.    public String getLastName() {
34.        return lastName;
35.    }
36.
37.    public void setLastName(String LastName) {
38.        this.lastName = LastName;
39.    }
40.
41.    public String getNationality() {
42.        return nationality;
43.    }
44.
45.    public void setNationality(String Nationality) {
46.        this.nationality = Nationality;
47.    }
48.
49.    public Date getBirthDate() {
50.        return birthDate;
51.    }
52.
53.    public void setBirthDate(Date BirthDate) {
54.        this.birthDate = BirthDate;
55.    }
```

```

56.
57.     public Driver(Integer id, String Name, String LastName, String Nationality, Date
      BirthDate) {
58.         this.id = id;
59.         this.name = Name;
60.         this.lastName = LastName;
61.         this.nationality = Nationality;
62.         this.birthDate = BirthDate;
63.     }
64.
65.     public Driver() {
66.     }
67.
68. }
69.

```

La struttura della classe è molto simile a quella già creata per le lezioni precedenti la principale differenza in questo caso è la presenza di una serie di attributi associati con la classe stessa e i dati al suo interno. Questi attributi sono definiti direttamente dal framework JPA e servono per legare la classe ai dati del database:

- `@Entity` – serve a definire una classe che viene gestita tramite il framework JPA
- `@Table(name="nome_database", schema="nome_tabella")` – serve a legare l'entità al database e allo schema preciso che rappresenta
- `@Column(name="nome_campo_database")` – serve a legare un membro della classe ad un campo del database
- `@Id` – serve ad identificare il campo della classe che è legato ad un campo del database che è chiave
- `@GeneratedValue(strategy=GenerationType.AUTO)` – serve a specificare che il campo è generato dal database al momento dell'inserimento

Da notare inoltre che insieme ai metodi Getter e Setter (righe dalla 21 alla 55) e al costruttore con tutti gli argomenti (righe 57 – 63), abbiamo anche un costruttore di default vuoto, necessario in questo caso (righe 65-66).

Una volta definita la classe legata ad una tabella specifica abbiamo bisogno di creare una classe di tipo Repository che serve a definire la politica attraverso la quale verranno gestiti i dati nel database. Il recupero dei dati dal database e' ampiamente automatizzato, alcune funzionalità possono essere implementate automaticamente dalla libreria a partire dal nome del metodo.

Per fare questo si deve creare una nuova interfaccia che estende la classe `CrudRepository` che fornisce le funzioni di base per l'estrazione/memorizzazione dei dati dal/al database.

```

1. public interface DriverRepository extends CrudRepository<Driver, Integer> {
2. }

```

Il prossimo passo è quello di creare un Controllore per gestire le richieste in arrivo, creando una nuova classe:

```

1. @Controller
2. @RequestMapping(path="/drivers")
3. public class MainController {
4.     @Autowired
5.     private DriverRepository driverRepository;
6.
7.     @GetMapping(path="/all")
8.     public @ResponseBody Iterable<Driver> getAllUsers() {
9.         return driverRepository.findAll();
10.    }
11.
12. }
13.

```

La classe, che avrà l'attributo `@Controller` del framework Spring inizialmente avrà un solo metodo legato all'URL "drivers/all" che serve per recuperare l'elenco dei piloti in formato JSON, il linguaggio di default per la serializzazione delle strutture dati in Spring.

Questo metodo, la funzione "getAllUsers" nel nostro caso (riga 8), è contraddistinto dall'attributo `@GetMapping(path="/all")`. La funzione avrà come ritorno una lista iterabile di elementi Driver i quali verranno serializzati in formato JSON direttamente dal framework, per effetto dell'attributo `@ResponseBody`. L'interno della funzione è semplicemente una chiamata alla funzione "findAll" esposta dalla classe DriverRepository. Il risultato dell'implementazione può essere recuperato attraverso il browser:



```

[{"id":1,"name":"Carlo","lastName":"Vallati","nationality":"Italian","birthDate":"1983-02-19T08:00:00.000+00:00"},
{"id":2,"name":"Charles","lastName":"Leclerc","nationality":"Monaco","birthDate":"1992-02-03T08:00:00.000+00:00"}]

```

Nel caso in cui si voglia scegliere XML come linguaggio di codifica dobbiamo aggiungere tra le dipendenze la seguente libreria che viene utilizzata da Spring per la gestione dell'encoding/decoding²:

```

1. <dependency>
2.     <groupId>com.fasterxml.jackson.dataformat</groupId>
3.     <artifactId>jackson-dataformat-xml</artifactId>
4. </dependency>

```

E poi modificare gli attributi della funzione come segue:

```

1. @GetMapping(path="/all", produces=MediaType.APPLICATION_XML_VALUE)

```

² Spring di default supporta JSON come linguaggio di encoding. La libreria Jackson in pratica converte internamente JSON in XML

```

▼ <ArrayList>
  ▼ <item>
    <id>1</id>
    <name>Carlo</name>
    <lastName>Vallati</lastName>
    <nationality>Italian</nationality>
    <birthDate>1983-02-19T08:00:00.000+00:00</birthDate>
  </item>
  ▼ <item>
    <id>2</id>
    <name>Charles</name>
    <lastName>Leclerc</lastName>
    <nationality>Monaco</nationality>
    <birthDate>1992-02-03T08:00:00.000+00:00</birthDate>
  </item>

```

Esercizio

Completare il servizio web con l'implementazione del metodo per il recupero dell'elenco dei piloti.

Implementazioni di altri metodi

L'implementazione di altri metodi può avvenire definendo altre funzioni nella classe controllore e nella classe Repository.

Aggiungiamo un nuovo metodo per il recupero dei dati del pilota che ha un determinato nome. Il primo passo è quello di aggiungere nell'interfaccia del Repository la dichiarazione di un nuovo metodo:

```

1. public interface DriverRepository extends CrudRepository<Driver, Integer> {
2.     Driver findByName(String n);
3. }

```

La nuova funzione "findByName" è un metodo auto-implementato dal sistema che va ad implementare una funzione di ricerca basata sul campo Nome.

Il prossimo passaggio è quello di implementare una nuova funzione nel controllore per esporre la ricerca per nome:

```

1. @PostMapping(path="/driver")
2. public @ResponseBody Driver getUserByName(@RequestParam String name) {
3.     return driverRepository.findByName(name);
4. }

```

La quale semplicemente invoca la funzione findByName su un parametro ricevuto, il nome del pilota. Il parametro viene automaticamente recuperato come parametro della richiesta utilizzando l'attributo @RequestParam. La risposta sarà un unico elemento Driver la che verrà convertito in JSON³.

Per testare questo metodo si può utilizzare il comando "curl", un comando testuale presente anche in sistemi operativi Windows, che può essere utilizzato per effettuare richieste http

³ <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation>

```
1. curl localhost:8080/drivers/driver -d name=Carlo
```

```
Command Prompt
Microsoft Windows [Version 10.0.22000.1098]
(c) Microsoft Corporation. All rights reserved.

C:\Users\carlo>curl localhost:8080/drivers/driver -d name=Carlo
{"id":1,"name":"Carlo","lastName":"Vallati","nationality":"Italian","birthDate":"1983-02-19T08:00:00.000+00:00"}
C:\Users\carlo>
```

Vediamo adesso come si possono realizzare dei metodi che aggiungono degli elementi nel database, ad esempio un nuovo metodo attraverso il quale si riesce a aggiungere i dati di nuovi piloti.

Il primo passo è quello di definire un nuovo metodo nel controllore:

```
1. @PostMapping(path="/add")
2. public @ResponseBody String addNewDriver (@RequestBody Driver d) {
3.
4.     driverRepository.save(d);
5.     return "Saved";
6. }
```

In questo caso definiamo un nuovo metodo “addNewDriver” che prende in ingresso una nuova istanza di Driver. L’istanza verrà creata dal framework Spring utilizzando l’input fornito dall’utente come payload della richiesta http de-serializzata in oggetto Driver.

L’aggiunta della nuova istanza al database può essere effettuata semplicemente invocando il metodo ‘save’ del Repository il quale andrà a salvare la nuova riga nel database.

Anche in questo caso il nuovo metodo può essere testato attraverso il comando curl:

```
1. curl localhost:8080/drivers/add -H "Content-Type: application/json" -X POST --data "{
  \"name\": \"Carlos\", \"lastName\": \"Saintz\", \"nationality\": \"Spagna\", \"birthDate\":
  \"1992-03-12\" }"
```

```
C:\Users\carlo>curl localhost:8080/drivers/add -H "Content-Type: application/json" -X POST --data "{ \"name\": \"Charles
\", \"lastName\": \"Leclerc\", \"nationality\": \"Monaco\", \"birthDate\": \"1992-02-12\" }"
Saved
C:\Users\carlo>
```

Esercizio

Aggiungere I nuovi metodi all’implementazione

Utilizzo di JPI per la semplificazione di applicazioni non Spring

La stessa procedura per l'interazione semplificata con il database può essere utilizzata anche da applicazioni che non usano il framework Spring, attraverso [hibernate](https://hibernate.org/)⁴, un'implementazione di JPI.

Il primo passo è quello di aggiungere hibernate tra le dipendenze nel progetto Maven:

```
1.      <dependency>
2.          <groupId>org.hibernate.orm</groupId>
3.          <artifactId>hibernate-core</artifactId>
4.          <version>6.1.3.Final</version>
5.      </dependency>
```

I progetti che hanno il file "module-info.java" richiedono anche l'aggiunta delle seguenti righe:

```
1.      requires jakarta.persistence;
2.      requires org.hibernate.orm.core;
3.      requires java.naming;
```

Il secondo è quello di creare un file di configurazione di hibernate dal nome 'hibernate.cfg.xml' nella cartella 'src/main/resources':

```
1.  <?xml version = "1.0" encoding = "utf-8"?>
2.  <hibernate-configuration>
3.      <session-factory>
4.
5.          <!-- Set URL -->
6.          <property name =
7.              "hibernate.connection.url">jdbc:mysql://localhost:3306/drivers</property>
8.
9.          <!-- Set User Name -->
10.         <property name = "hibernate.connection.username">root</property>
11.
12.         <!-- Set Password -->
13.         <property name = "hibernate.connection.password">root</property>
14.
15.         <!-- Set Driver Name -->
16.         <property name =
17.             "hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
18.
19.         <property name = "hibernate.show_sql">true</property>
20.     </session-factory>
21. </hibernate-configuration>
```

Il file di configurazione in formato XML contiene tutti i parametri per la configurazione della libreria, tra cui:

- Percorso di connessione al database, con nome del database (riga 6)
- Username per accesso al database (riga 9)

⁴ <https://hibernate.org/>

- Password per accesso al database (riga 12)
- Driver di accesso (riga 15)
- Opzioni di log del codice SQL (riga 17)

Le operazioni di interazione con il database a questo punto possono avvenire utilizzando l'interfaccia fornita da hibernate che lavora con oggetti. Il primo passo e' quello di definire l'oggetto da memorizzare creando un JavaBeans che rappresenti il dato allo stesso modo in cui e' stato fatto per il server web.

Successivamente ogni volta che si vuole interagire con il database basta inizializzare hibernate con il seguente codice:

```
1. Configuration configuration = new Configuration();
2. configuration.configure("hibernate.cfg.xml");
3. configuration.addAnnotatedClass(Driver.class);
4.
5. SessionFactory sessionFactory = configuration.buildSessionFactory();
6. Session session = sessionFactory.openSession();
```

Il codice carica il file di configurazione (righe 1-2) e carica la classe che rappresenta il dato nel framework (riga 3). Successivamente per leggere/scrivere dal database si deve aprire una sessione attraverso le due istruzioni alle righe 5 e 6.

Per scrivere dei nuovi dati sul database le operazioni a questo punto sono molto semplici:

```
1. Driver nd = new Driver (...);
2. session.save(nd);
3. session.getTransaction().commit();
```

la scrittura avviene attraverso la funzione save che salva nel database una nuova riga in base ai dati contenuti in un'istanza dell'oggetto Driver.

```
1. Query query = session.createQuery("FROM drivers", Driver.class);
2. List<Driver> dr = query.getResultList();
```

La lettura invece avviene tramite la creazione di un oggetto Query. La funzione createQuery prende in ingresso la query descritta in un linguaggio pseudo-sql⁵, che nel caso dell'esempio recupera tutti i guidatori attualmente memorizzati.

⁵ https://www.tutorialspoint.com/hibernate/hibernate_query_language.htm