Spring Boot

Passaggi Preliminari

1. **Creazione del progetto:**
    - Premi `F1`.
    - Seleziona **Initializer Maven Project**.
    - Configura il progetto con i seguenti parametri:
        - **Versione**: `3.4.1`
        - **Linguaggio**: `Java`
        - **Group**: `edu.unict.wsos`
        - **Artifact**: `esame`
        - **Tipo**: `jar`
        - **Java Version**: `17`
    - Aggiungi le seguenti dipendenze:
        - **Spring Web**
        - **Thymeleaf**
        - **MySQL Driver**
        - **Spring Data JPA**
2. **Salvataggio:**
    - Salva e apri la cartella del progetto nella tua home.

Configurazione del Progetto

1. **Compilazione:**
    - Esegui `maven package` nella cartella creata.
    - Java: Clean Workspace Cache.
2. **Configurazione delle proprietà di applicazione:**
    - Apri il file `src/main/resources/application.properties` e aggiungi la seguente configurazione:

```
spring.datasource.url=jdbc:mysql://localhost:3306/exam
spring.datasource.username=user
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
```

Struttura del Progetto

All'interno di `src/main/java/edu/unict/dmi/wsos/esami`, crea le seguenti cartelle:

- **Model** (step 1)
- **Data** (step 2)
- **Controller** (step 3)

Ordine di implementazione:

1. **Model**
   Crea una classe Java che rappresenta la tabella nel database.
   La classe deve includere:

   - `@Entity`
   - I campi del database
   - L'ID deve avere `@Id` e `@GeneratedValue(strategy = GenerationType.IDENTITY)`
   - Crea getter e setter
   - Costruttori con tutti i parametri e costruttore vuoto tramite l'azione "Azione origine di Visual Studio Code"

   Esempio:

```
package edu.unict.dmi.wsos.esami.Model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Esami {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    long id;
    String nome;
    String professore;
    int cfu;
    public Esami() {
    }
    public Esami(long id, String nome, String professore, int cfu) {
        this.id = id;
        this.nome = nome;
        this.professore = professore;
        this.cfu = cfu;
    }

     // Getter e Setter
}
```

2. **Repository**
   Crea un'interfaccia repository che estende `JpaRepository<NomeModel, Long>`.

   Esempio:

```
package edu.unict.dmi.wsos.esami.Data;

import org.springframework.data.jpa.repository.JpaRepository;
import edu.unict.dmi.wsos.esami.Model.Esami;

public interface EsamiRepository extends JpaRepository<Esami, Long> {

}
```

3. **Controller**
   Crea una classe `Controller` con l'annotazione `@Controller`, un riferimento **private final** alla repository e un costruttore che lo inizializzi.

   Esempio:

```
package edu.unict.dmi.wsos.esami.Controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import edu.unict.dmi.wsos.esami.Data.EsamiRepository;
import edu.unict.dmi.wsos.esami.Model.Esami;

@Controller
public class EsamiController {

    private final EsamiRepository repo;

    public EsamiController(EsamiRepository repo) {
        this.repo = repo;
    }

}
```

4. **Thymeleaf: Creazione dei template**

   All'interno della cartella `/src/main/resources/templates`, crea un file `index.html` tramite il comando `html:5` e modifica:

```
<html lang="en">
```

   in:

```
<html lang="it" xmlns:th="https://www.thymeleaf.org">
```

Sviluppo delle funzionalità CRUD

Read

Modifica il file `index.html` creando una tabella che contenga tutti i campi del model.
La riga relativa ai dati deve avere l'attributo `th:each="esame : ${esami}"`.

> **Nota bene**: Il campo tra parentesi graffe deve essere uguale a quello che scriviamo nel controller nel metodo `addAttribute()`. Ogni dato della tabella sarà dato dalla notazione `esame.attributo`, facendo attenzione ad usare la stessa notazione usata nel model.

Esempio:

```
<!DOCTYPE html>
<html lang="it" xmlns="https://www.thymeleaf.org">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Esami</title>
</head>

<body>
    <h1>
        <center>Esami in Spring Boot</center>
    </h1>

    <table border="1px">
        <tr>
            <th>Nome</th>
            <th>Professore</th>
            <th>CFU</th>
            <th>Modifica</th>
            <th>Elimina</th>
        </tr>
```

```
        <tr th:each="esame : ${esami}">
            <td th:text="${esame.nome}"></td>
            <td th:text="${esame.professore}"></td>
            <td th:text="${esame.cfu}"></td>
            <td>
                <form action="/update" method="post">
                    <input type="hidden" name="id" th:value="${esame.id}">
                    <input type="submit" value="Modifica">
                </form>
            </td>
            <td>
                <form action="/delete" method="post">
                    <input type="hidden" name="id" th:value="${esame.id}">
                    <input type="submit" value="Elimina">
                </form>
            </td>
        </tr>
    </table>
</body>

</html>
```

Modifica il `controller` aggiungendo un metodo `@GetMapping` che prende come parametro il `Model`.
Verrà richiamato `model.addAttribute()` con i seguenti parametri:

- **Nome della variabile** che usiamo nel template Thymeleaf nella notazione `${}` presente nel `th:each`.
- `repo.findAll()` che permette di selezionare tutti i dati presenti nel database MySQL.

Esempio:

```
@GetMapping("/")
public String getEsami(Model model) {
    model.addAttribute("esami", repo.findAll());
    return "index";
}
```

Create

Modifica il file `index.html` creando un form con metodo `POST` e action `/create`.
Il form avrà tanti campi quanti sono i dati da inserire nella tabella del database.

> **Nota bene**: L'attributo `name` di ogni tag `input` deve corrispondere esattamente con il nome delle variabili utilizzate nel model.

Esempio:

```
<h3>Inserisci un nuovo esame</h3>

<form action="/create" method="post">
    <span>Inserisci nome esame: </span>
    <input type="text" name="nome"><br>
    <span>Inserisci nome professore: </span>
    <input type="text" name="professore"><br>
    <span>Inserisci CFU: </span>
    <input type="number" name="cfu"><br>
    <input type="submit" value="Invia">
</form>
```

Aggiungi al `controller` un metodo `@PostMapping` che prende come parametro la classe creata allo step 1.
Verrà richiamato `repo.save()` per salvare l'oggetto nel database.
Successivamente, reindirizziamo l'utente alla home.

Esempio:

```
@PostMapping("/create")
public String creaEsame(Esami esame) {
    repo.save(esame);
    return "redirect:/";
}
```

Delete

Aggiungi al `controller` un metodo `@PostMapping` che prende come parametro l'id dell'oggetto da eliminare (passato come `hidden` nel form della tabella).
Verrà richiamato `repo.deleteById()` per eliminare l'oggetto dal database.
Successivamente, reindirizziamo l'utente alla home.

Esempio:

```
@PostMapping("/delete")
public String deleteEsame(Long id) {
    repo.deleteById(id);
    return "redirect:/";
}
```

Update

Crea un file `modifica.html` nella stessa directory di `index.html`. Questo file conterrà un form simile a quello utilizzato per l'invio, ma con alcune accortezze per precompilare i dati da modificare.

- Il form avrà `method=post`, `action=/create` e `th:object=${esame}`.
- Ogni campo avrà un attributo `th:value={esame.parametro}`, dove `esame` è il nome della variabile passata nel controller.

Esempio di form:

```
<body>
    <h3>Modifica esame</h3>

    <form action="/create" method="post" th:object="${esame}">
        <input type="hidden" name="id" th:value="${esame.id}">
        <span>Modifica nome esame: </span>
        <input type="text" name="nome" th:value="${esame.nome}"><br>
        <span>Modifica nome professore: </span>
        <input type="text" name="professore" th:value="${esame.professore}"><br>
        <span>Modifica CFU: </span>
        <input type="number" name="cfu" th:value="${esame.cfu}"><br>
        <input type="submit" value="Modifica">
    </form>
</body>

</html>
```

Modifica il `controller` aggiungendo un metodo `@PostMapping` che prende come parametri:

- Il **Model**
- L'**ID** dell'oggetto da modificare (passato come `hidden` nel form della tabella di `index.html`).
  Il compito del metodo sarà:

Richiamare `model.addAttribute()` passando la variabile per Thymeleaf (assicurati che il nome sia coerente tra controller e template) ed utilizzando `repo.getReferenceById()` per ottenere i dati dal database e precompilare il form.

Esempio:

```
@PostMapping("/update")
public String updateEsame(Model model, Long id) {
    model.addAttribute("esame", repo.getReferenceById(id));
    return "modifica";
}
```

Filtri

Creiamo un filtro che, tramite i model:

- **Abbigliamento**
- **Brand**

Permetta di visualizzare tutti i capi di abbigliamento di un determinato brand.

1. **Aggiunta del metodo nella repository di Abbigliamento**

Nella repository **Abbigliamento**, aggiungiamo un metodo personalizzato chiamato `findByBrandId(Brand brand)`, che accetta come parametro un oggetto `Brand` per effettuare la ricerca.

```
public interface AbbigliamentoRepository extends JpaRepository<Abbigliamento,
Long> {
    List<Abbigliamento> findByBrandId(Brand brand);
}
```

2. **Implementazione del metodo nel controller**

Nel controller di **Abbigliamento**, aggiungiamo un metodo che utilizza il metodo appena definito nella repository. È importante cercare l'oggetto **Brand** tramite `getReferenceById()` prima di passarlo alla funzione.

```
    @PostMapping("/abbigliamento/findByBrand")
    public String findByBrand(Model model, @RequestParam Long brandId) {
        // Recupera l'oggetto Brand tramite il suo ID
        Brand brand = brandRepository.getReferenceById(brandId);

        // Aggiunge gli elementi filtrati e tutti i brand al model
        model.addAttribute("abbigliamenti",
abbigliamentoRepository.findByBrandId(brand));
        model.addAttribute("brands", brandRepository.findAll());

        // Restituisce la vista della lista
        return "abbigliamento/list";
    }
```

Gestione dell'eliminazione a cascata

Supponiamo di avere i seguenti modelli:

- **Studenti**
- **Esami**

Se tentassimo di eliminare un esame sostenuto da uno o più studenti, ciò causerebbe un errore dovuto alla violazione delle dipendenze. Per risolvere questo problema, possiamo utilizzare l'annotazione `@OneToMany()` nella classe `Esami` per gestire l'eliminazione a cascata e la rimozione degli orfani:

```
@OneToMany(mappedBy = "examId", cascade = CascadeType.REMOVE)
    private List<Student> students = new ArrayList<>();
```

In questo modo, l'eliminazione di un esame comporterà automaticamente l'eliminazione delle associazioni con gli studenti.

Somma dei valori

Il metodo `findAll()` restituisce una lista di oggetti del modello, consentendo di accedere facilmente ai metodi di ciascun oggetto. Di seguito, un esempio per calcolare la somma dei CFU degli esami utilizzando un ciclo `for` avanzato:

```
public String getExams(Model model) {
    // Recupera tutti gli esami dal repository
    List<Exam> allExams = repo.findAll();
    model.addAttribute("exams", allExams);

    // Calcola la somma dei CFU
    int totalCfu = 0;
    for (Exam exam : allExams) {
        totalCfu += exam.getCfu();
    }

    // Aggiungi la somma dei CFU come attributo del modello
    model.addAttribute("countCFU", totalCfu);

    return "exam/list";
}
```

In questo esempio, la somma dei CFU viene calcolata iterando sulla lista degli esami, e il risultato viene aggiunto al modello per essere mostrato nella vista.

Modifica dei CFU

Supponiamo di voler uniformare i CFU delle materie, portando a 6 CFU tutte quelle che attualmente ne hanno meno di 6. Per farlo, aggiungiamo un metodo alla repository per filtrare le materie in base al numero di CFU:

```
List<Exam> findByCfuLessThan(int cfu);
```

Successivamente, implementiamo nel controller un metodo che aggiorna i CFU delle materie filtrate, impostandoli a 6:

```
@PostMapping("/exam/fixCFU")
public String fixCFU(Model model) {
    List<Exam> esami = repo.findByCfuLessThan(6);

    for (Exam elem : esami) {
        elem.setCfu(6);
        repo.save(elem);
    }

    return "redirect:/exam";
}
```

In questo modo, tutte le materie con meno di 6 CFU saranno aggiornate automaticamente.

Colorazione dinamica delle righe di una tabella

Per modificare dinamicamente il colore delle righe di una tabella in base a una condizione, possiamo utilizzare Thymeleaf. Ad esempio, per evidenziare in rosso le righe relative a uno specifico esame (ad esempio, "WSOS"), possiamo scrivere:

```
<tr th:if="${student.examId.name == 'WSOS'}"
    style="background-color: red; color: white;"
    th:each="student : ${students}">
    ...
</tr>

<tr th:unless="${student.examId.name == 'WSOS'}"
    th:each="student : ${students}">
    ...
</tr>
```