

Sistemi informativi su Web

Introduzione a Spring Boot (parte 1)

aa 2024-2025



Paolo Merialdo
Università degli Studi Roma Tre

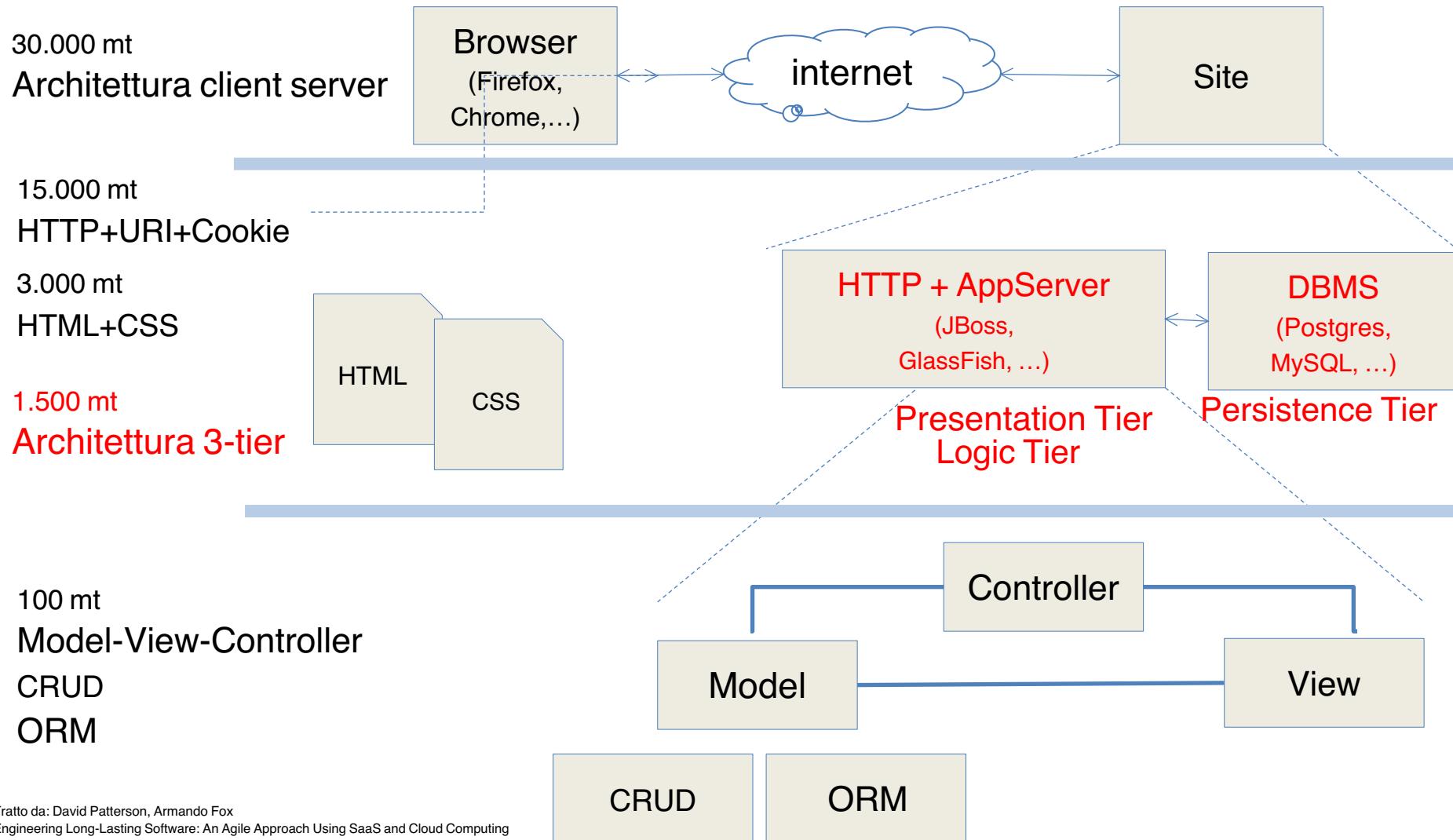


This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

Introduzione

- In questa primissima parte del corso, usiamo un framework, Spring Boot, per costruire rapidamente (e un po' magicamente) un sistema informativo su Web, molto semplice ma completo
- Rispetto a quello che vedremo in seguito, facciamo molte semplificazioni, che verranno evidenziate a fine lezione
- Oltre a dare una gratificazione immediata allo studente, questa esperienza permette di esemplificare il pattern MVC, e dimostra le potenzialità di un framework come Spring Boot

Anatomia di un sistema informativo su Web (nostra architettura)



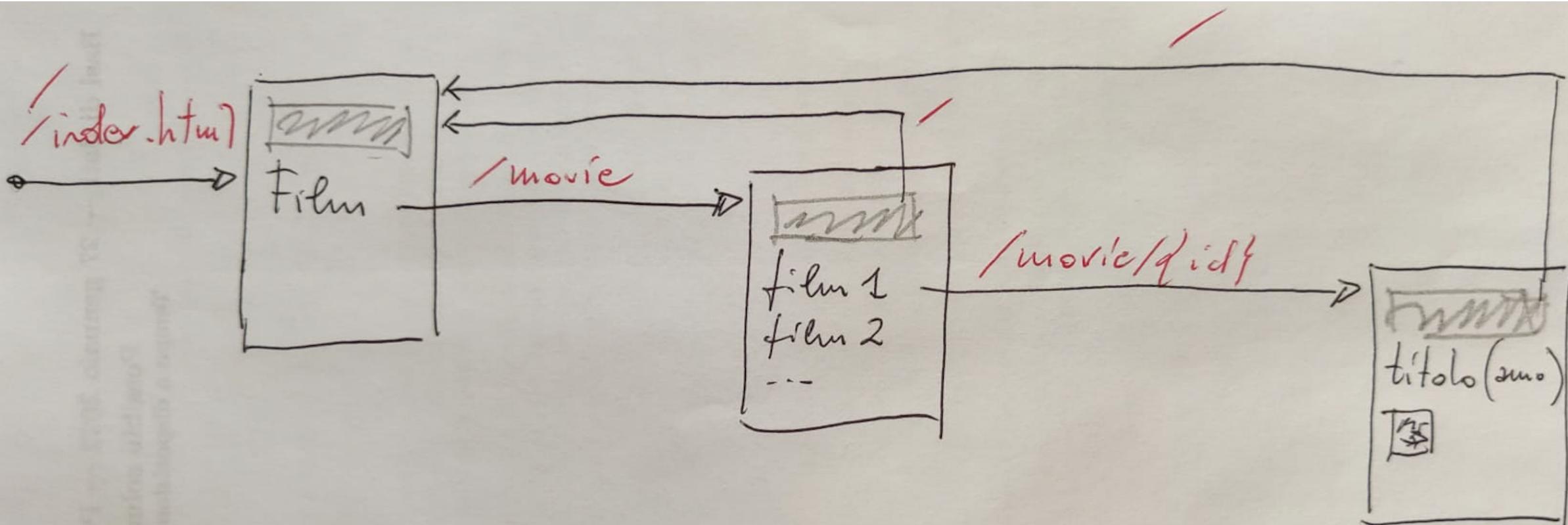
Web app di riferimento

- Diamo un primo sguardo all'applicazione
<http://sinai.inf.uniroma3.it:8080/> (attiva solo durante la lezione)
- Proviamo a guardarla e a modellarla da due punti di vista
 - utente: ricostruiamo l'interazione e mettiamo in evidenza le richieste che mandiamo al server quando interagiamo con il sistema attraverso il client (il browser)
 - progettista: ricostruiamo le entità e le operazioni del sistema sul lato server, le pagine e i fogli di stile lato client

Diamo un'occhiata all'applicazione

- L'homepage corrisponde alla richiesta index.html
- Nell'home page abbiamo un collegamento ipertestuale che ci offre la possibilità di inviare al server la richiesta **/movie**, alla quale il server risponde con una pagina che mostra l'elenco di tutti i film presenti nel sistema
- Il titolo di ciascun film dell'elenco è un collegamento ipertestuale, che ci offre la possibilità di inviare al server la richiesta **/movie/*id*** (dove *id* è un intero), alla quale il server risponde con una pagina che mostra i dettagli del film

Ricostruzione alla lavagna



Ricostruzione dal punto di vista del progettista

- entità
- operazioni del sistema sul lato server
- pagine HTML
- fogli di stile CSS

Costruiamo l'applicazione con il framework Spring Boot

Le pagine con sfondo azzurro indicano passaggi
tecnici da fare al computer

Requisiti

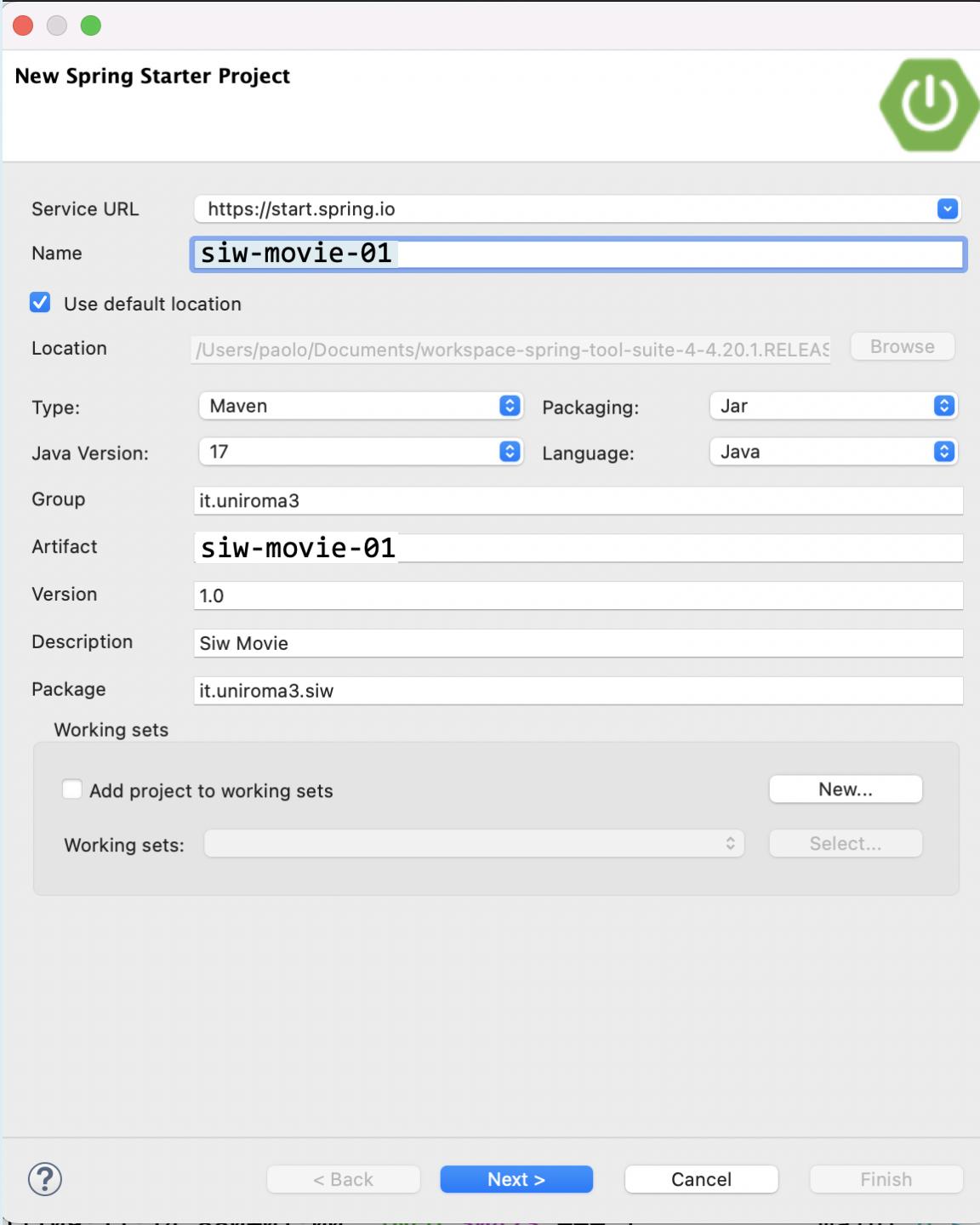
- Java 17 o superiore
- Postgres
- PgAdmin
- IDE: Spring boot Tool Suite
(o su Visual Studio Code, o IntelliJ: come preferite)

Step 1: creiamo il progetto

- Un progetto Web prevede l'uso di molte librerie
- Spring Boot mette a disposizione strumenti per creare un progetto specificando le librerie dalle quali dipende
 - online <https://start.spring.io/>
 - nell'IDE Spring Tool Suite
(File->New->Spring Starter Project)

Step 1: creiamo il progetto

- La creazione del progetto prevede inoltre di specificare:
 - Il sistema di gestione del progetto
Usiamo: Maven
 - Il tipo di packaging: va bene jar
 - La versione di Java: 17
 - da terminale, verificare la versione di Java che avete installato con i comandi:
`javac -version`
`java -version`
 - Group Id: identificatore univoco del progetto. Deve seguire le regole del nome di un package Java: nome del dominio seguito dal nome del progetto.
Usiamo: **it.uniroma3.siw**
 - Artifact Id: nome del jar che verrà prodotto. Lowercase, semplice. Rispetto a quanto fatto fino ad ora, è l'equivalente del nome del progetto Eclipse.
Usiamo: **siw-movie-01**
 - version: numero di versione.
Usiamo il default



Il tool di creazione del progetto nell'ide Spring Tool Suite

Nel passaggio successivo sceglieremo le librerie necessarie al nostro progetto

New Spring Starter Project Dependencies

Spring Boot Version: 3.2.3

Frequently Used:

PostgreSQL Driver Spring Data JPA Spring Web
 Thymeleaf

Available:

Type to search dependencies

- ▶ AI
- ▶ Developer Tools
- ▶ Google Cloud
- ▶ I/O
- ▶ Messaging
- ▶ Microsoft Azure
- ▶ NoSQL
- ▶ Observability
- ▶ Ops
- ▶ SQL
- ▶ Security
- ▶ Spring Cloud
- ▶ Spring Cloud Circuit Breaker
- ▶ Spring Cloud Config
- ▶ Spring Cloud Discovery
- ▶ Spring Cloud Messaging
- ▶ Spring Cloud Routing

Selected:

- X Spring Data JPA
- X PostgreSQL Driver
- X Thymeleaf
- X Spring Web

Make Default Clear Selection

< Back Next > Cancel Finish



Ci servono le librerie:

- Spring Web
- PostgresSQL Driver
- Thymeleaf
- Spring Data JPA

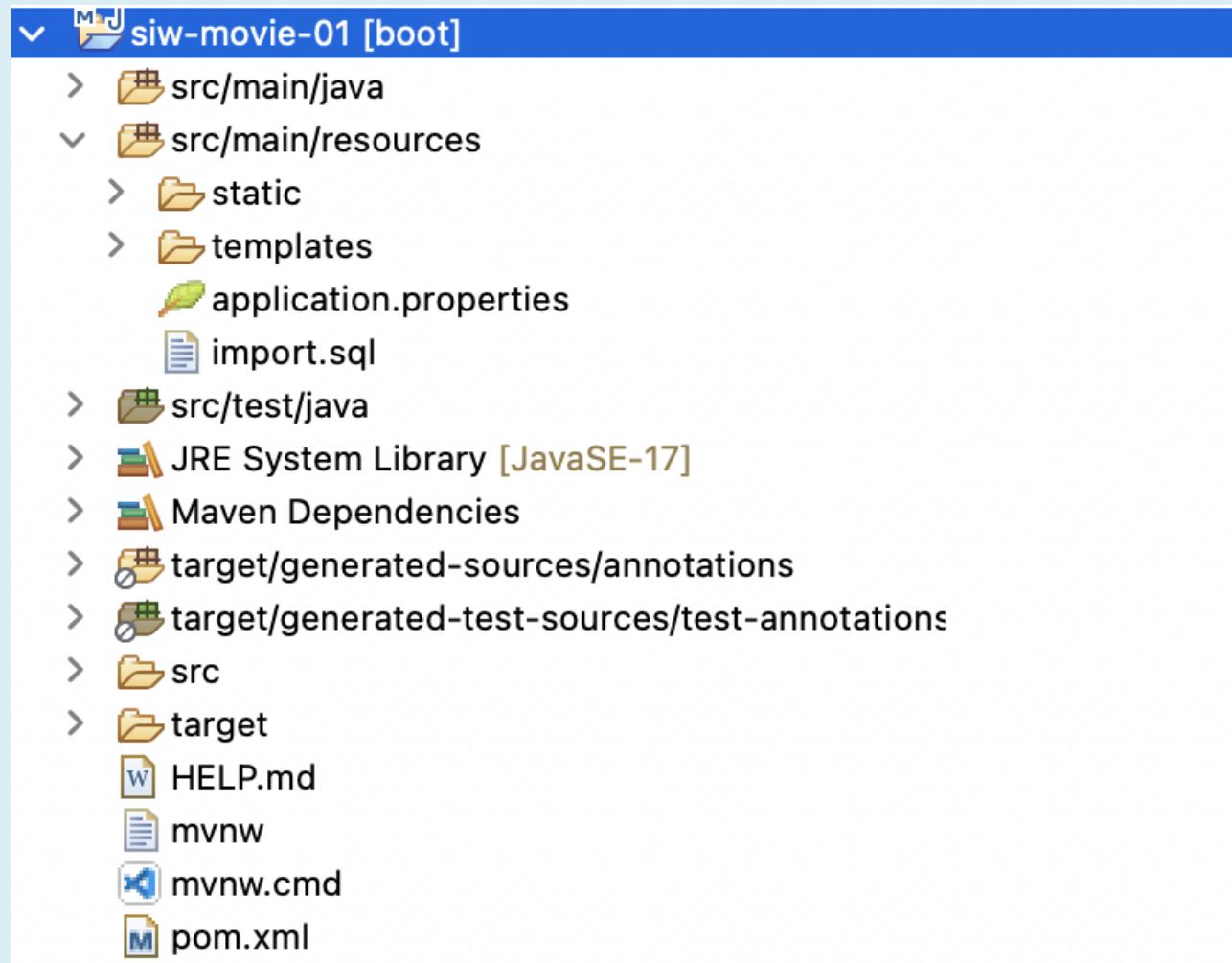
Per chi preferisce la versione online

The screenshot shows the Spring Initializr web interface for generating a Spring Boot project. The configuration includes:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 3.0.3 (selected)
- Project Metadata:**
 - Group: it.uniroma3.siw
 - Artifact: siw-movie-01
 - Name: siw-movie-01
 - Description: Gestione film
 - Package name: it.uniroma3.siw. siw-movie-01
 - Packaging: Jar (selected)
- Dependencies:**
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Spring Data JPA** (SQL): Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
 - PostgreSQL Driver** (SQL): A JDBC and R2DBC driver that allows Java programs to connect to a PostgreSQL database using standard, database independent Java code.
 - Thymeleaf** (TEMPLATE ENGINES): A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.
- Java:** 8 (selected)
- Buttons at the bottom:** GENERATE (⌘ + ↩), EXPLORE (CTRL + SPACE), SHARE...

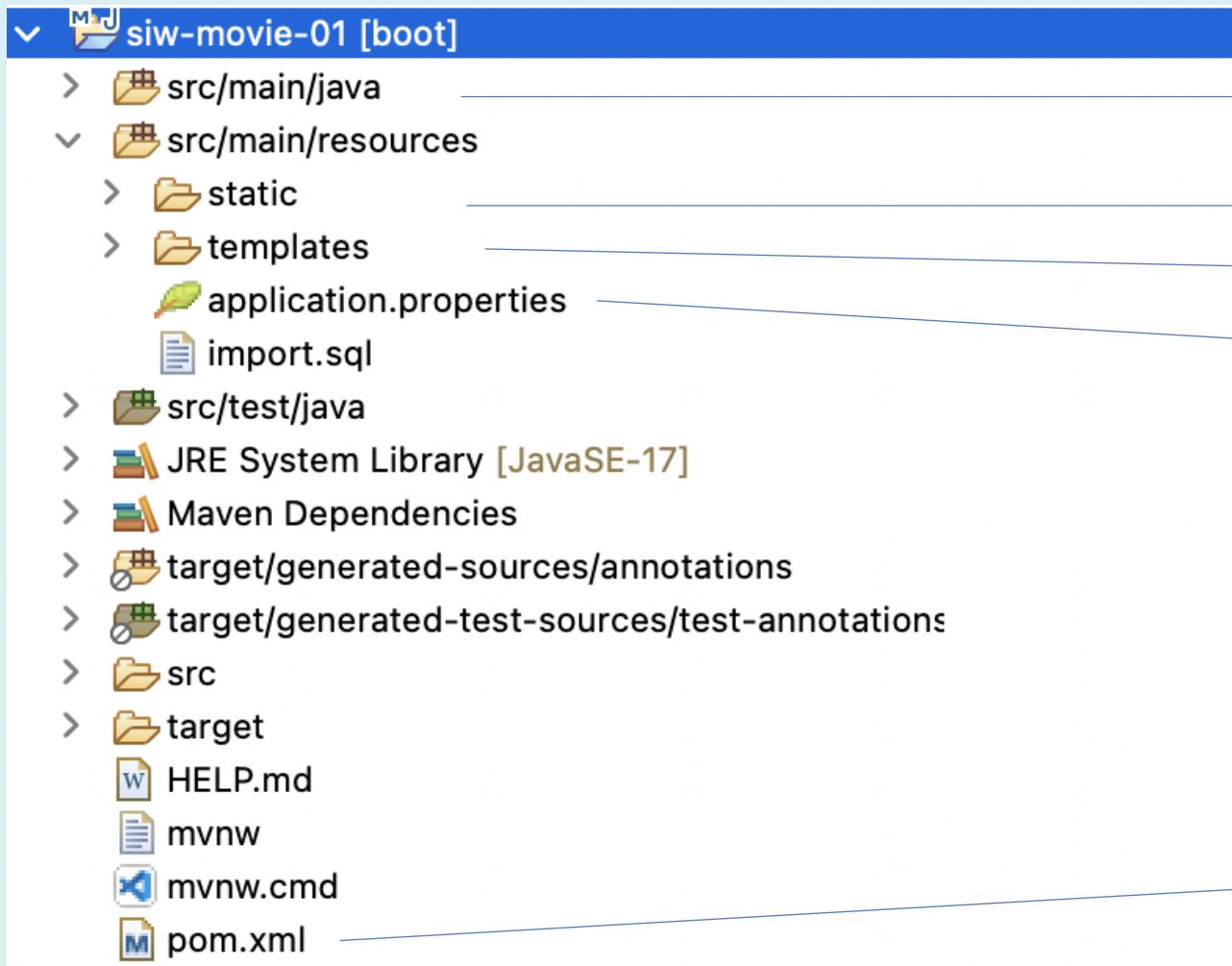
Bisogna scaricare lo zip che genera e importarlo nell'ide

Struttura progetto Spring Boot



Struttura progetto Spring Boot

(con la creazione dei package e di altri artefatti utili al progetto)



- Qui mettiamo le nostre classi
 - File statici (immagini, css)
 - Viste (template HTML per pagine dinamiche)
 - File di configurazione (JPA, Logging, etc.)
- File per la gestione progetto

Step 2: Configurazione del progetto

- Il nostro progetto avrà diversi parametri di configurazione (porta, dettagli per la connessione al DBMS, etc.)
- I parametri di configurazione del progetto si impostano nel file application.properties
- Ogni riga riporta l'impostazione di un parametro per mezzo di una coppia nome=valore

Nostra configurazione

```
#=====
# = Datasource
#=====
spring.jpa.database=POSTGRESQL
spring.jpa.show-sql=true
logging.level.org.hibernate.SQL=debug
logging.level.org.hibernate.type.descriptor.sql=trace
spring.jpa.hibernate.ddl-auto=create
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/movie
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
#=====
# = Web server
#=====
server.port=8080
#=====
# = Thymeleaf configurations
#=====
spring.thymeleaf.check-template-location=true
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.servlet.content-type=text/html
spring.thymeleaf.cache=false
spring.thymeleaf.enabled=true
spring.thymeleaf.encoding=UTF-8
```

una riga che comincia con il simbolo # è un commento

Configurazione

- Alcuni parametri li capiremo in seguito
- Osserviamone alcuni rilevanti

```
#=====
# = Datasource
#=====
spring.datasource.driver-class-name=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost/movie
spring.datasource.username=postgres
spring.datasource.password=postgres
#=====
# = Web server
#=====
server.port=8080
```

Step 3: creazione del database

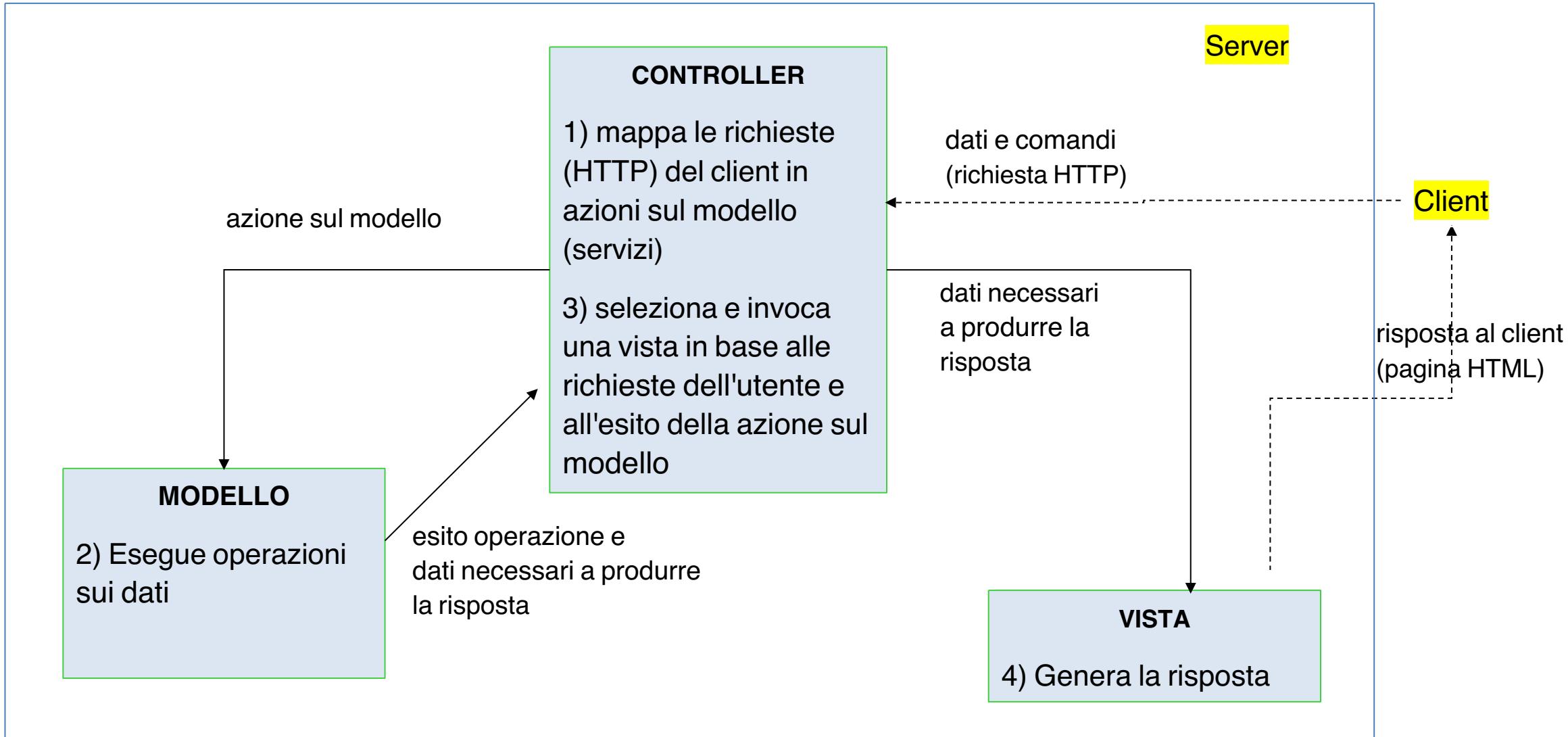
- Creiamo il database della applicazione
- Lo chiamiamo movie (vedi file di configurazione)
- Se usiamo postgres, possiamo creare il database con PgAdmin

Spring Boot: la classe Application

- Un progetto Spring Boot ha una classe **Application**, che rappresenta l'intera applicazione
- Questa classe ha un metodo main
 - lancia l'**application server** e carica tutti i componenti
- **Quindi non abbiamo bisogno di un application server su cui installare l'applicazione: è già incluso nell'applicazione**
- Quando eseguiamo l'applicazione, viene lanciato l'**application server** con la nostra app già installata
 - L'**application server** è in ascolto sulla porta impostata nel file di configurazione (8080 se manteniamo quanto riportato nella slide precedente)

Step 4: Scriviamo il codice

Model-View-Controller



Organizzazione package

- Creiamo i package
 - it.uniroma3.siw.model
 - it.uniroma3.siw.controller
 - it.uniroma3.siw.repository
 - it.uniroma3.siw.service

Il modello

- La nostra applicazione prevede un'entità (il film)
- Per ogni film sono di interesse:
 - il titolo
 - l'anno di uscita
 - l'url di un'immagine relativa al film
 - inoltre, ad ogni film associamo come identificatore un intero Long
- Scriviamo il codice di una semplice classe, **Movie**, che modella questa entità

La classe Movie

```
public class Movie {  
  
    private Long id;  
    private String title;  
    private Integer year;  
    private String urlImage;  
  
    // seguono (devono essere scritti) metodi setter e getter  
    // seguono (devono essere scritti) metodi equals e hashCode:  
    //      due oggetti Movie sono uguali se hanno  
    //      stesso titolo e stesso anno  
  
}
```

La classe Movie

- Gli oggetti della classe Movie devono essere persistenti
 - Dovremmo creare una tabella che possa memorizzare gli oggetti
 - Gestire la creazione degli id

La classe Movie

Sfruttiamo le potenzialità dell'ORM JPA:

- Annotiamo la classe con
`@Entity`
per comunicare al framework che va predisposta una tabella per memorizzare gli oggetti della classe Movie
- Annnotiamo l'identificatore con
`@Id`
`@GeneratedValue(strategy = GenerationType.AUTO)`
per comunicare al framework che la variabile che segue corrisponde ad una chiave primaria il cui valore deve essere creato automaticamente dal DBMS ogni volta che viene inserita una nuova ennupla Movie

`@Entity`, `@Id`, `@GeneratedValue` sono nel package `jakarta.persistence`

L'entità Movie

```
@Entity
public class Movie {

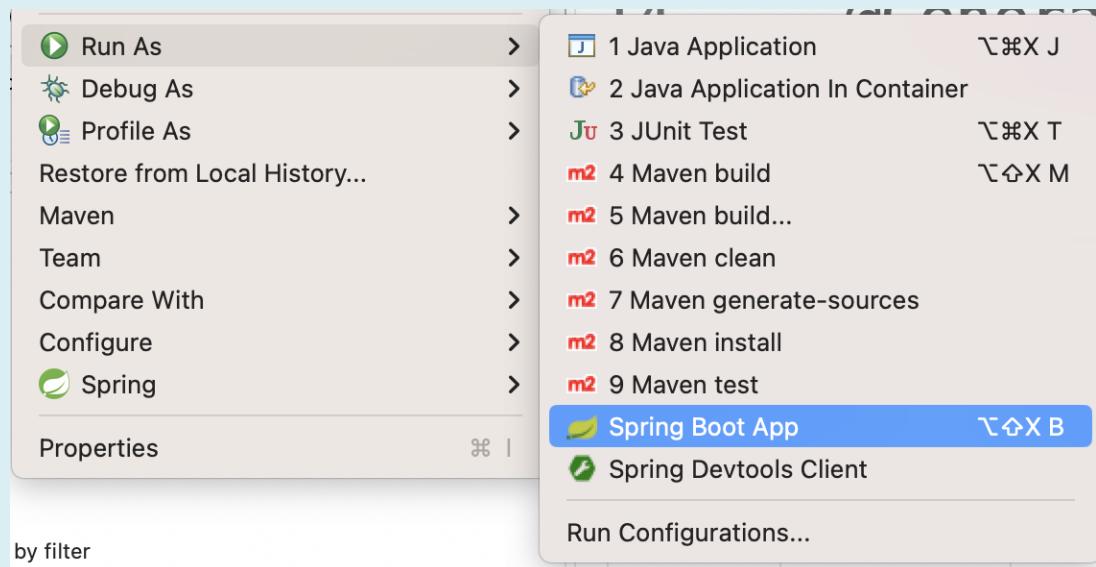
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String title;
    private Integer year;
    private String urlImage;

    // seguono metodi setter e getter
    // seguono metodi equals e hashCode:
    //     due oggetti Movie sono uguali se hanno
    //     stesso titolo e stesso anno

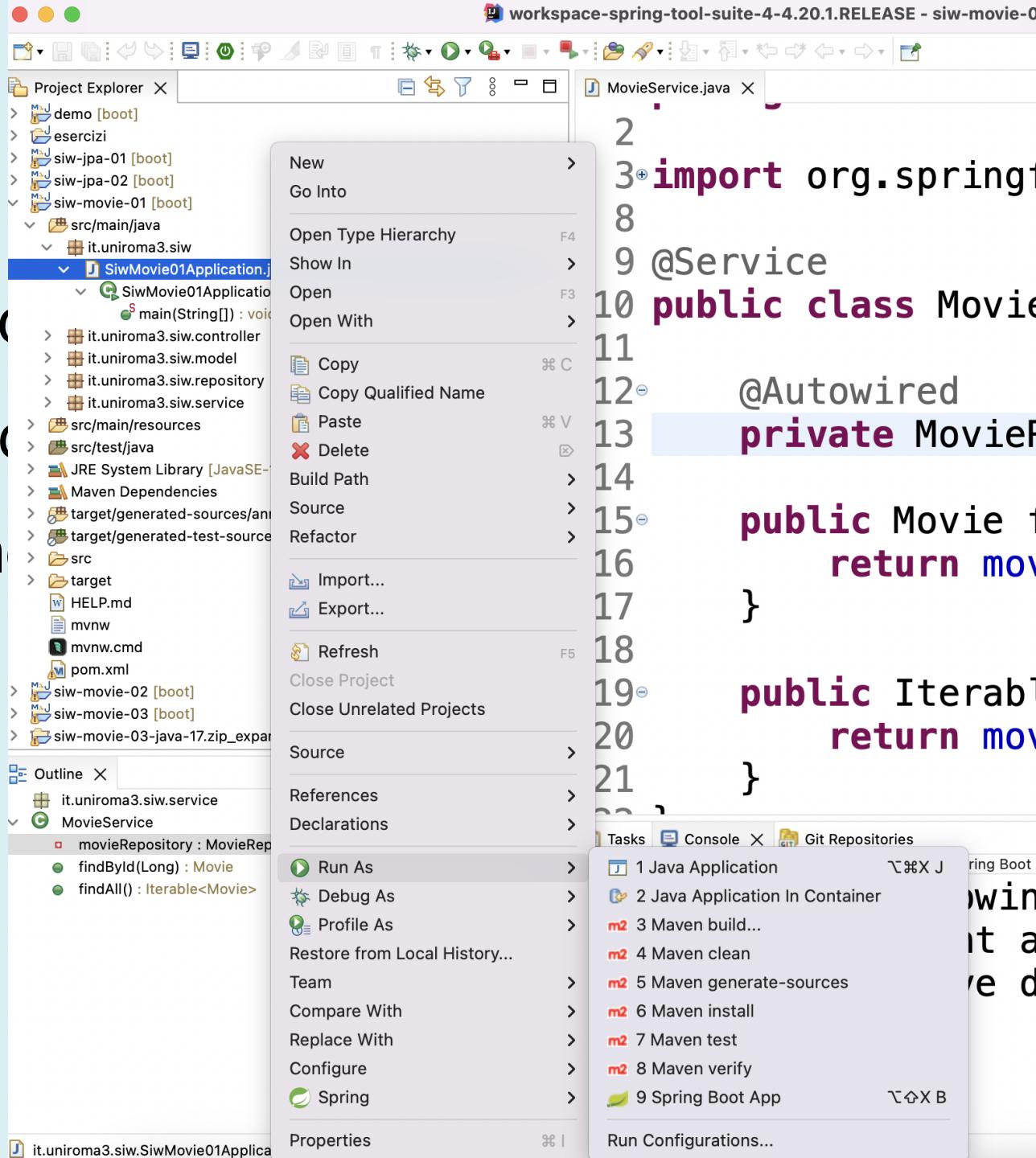
}
```

Lanciamo l'applicazione

- Cosa succede dopo aver introdotto queste annotazioni?
- Proviamo ad eseguire il codice
- Lanciamo l'applicazione come Spring Boot App



- Cosa succede?
- Proviamo a...
- Lanciamo!



2
3+ **import org.springframework.
8
9 @Service
10 public class MovieService {
11
12 @Autowired
13 private MovieRepository movieRe
14
15 public Movie findById(Long id) {
16 return movieRe
17 }
18
19 public Iterable<Movie> findAll()
20 }
21 }**

public Iterable<Movie> findAll()

L'entità Movie

- Apparentemente non succede nulla
- Ma diamo un'occhiata ai messaggi della console e al nostro database siwmovie

L'entità Movie

- Apparentemente non succede nulla
- Ma diamo un'occhiata ai messaggi della console e al nostro database movie

```
2023-03-05 18:06:33.570 DEBUG 35885 --- [           main] org.hibernate.SQL
Hibernate: drop table if exists movie cascade
2023-03-05 18:06:33.575 DEBUG 35885 --- [           main] org.hibernate.SQL
Hibernate: drop sequence if exists hibernate_sequence
2023-03-05 18:06:33.578 DEBUG 35885 --- [           main] org.hibernate.SQL
Hibernate: create sequence hibernate_sequence start 1 increment 1
2023-03-05 18:06:33.580 DEBUG 35885 --- [           main] org.hibernate.SQL
Hibernate: create table movie (id int8 not null, title varchar(255), url_image var
2023-03-05 18:06:33.584 INFO 35885 --- [           main] o.h.t.schema.internal.Sc
2023-03-05 18:06:33.586 INFO 35885 --- [           main] i.LocalContainerEntityManager
```

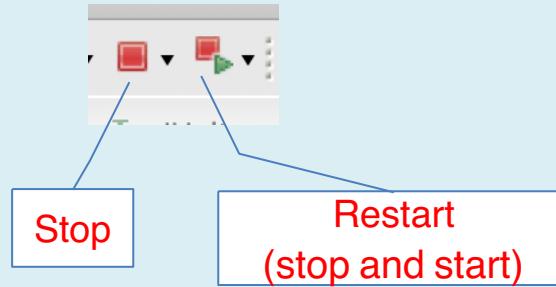
- E' stata creata la tabella movie ed una sequenza hibernate_sequence

L'applicazione Spring Boot

- Quando lanciamo un'applicazione Spring Boot
 1. Il framework cerca tutte le classi `@Entity` e per ciascuna di esse crea un'opportuna tabella nel database
 2. Viene avviato l'application server, che mette in ascolto l'applicazione sulla porta specificata nel file di configurazione
- Proviamo ad accedere attraverso il browser:
<http://localhost:8080>
 - Il server risponde! Per ora con un messaggio di errore (404 page not found): non trova la pagina richiesta, ma d'altronde non abbiamo ancora predisposto nessuna pagina

Nota pratica

- Se usiamo STS, possiamo lanciare (e fermare) l'applicazione direttamente dall'IDE
 - ovviamente serve solo in fase di sviluppo
 - in produzione, viene semplicemente eseguito il jar



Gestione della persistenza

- Per ora abbiamo definito una classe per la quale il framework ha predisposto nel database una tabella per memorizzare in maniera persistente gli oggetti
- Come gestiamo però le operazioni tipiche (CRUD) della persistenza?
- Ci affidiamo al pattern Repository

Il pattern Repository

- Il pattern Repository viene usato negli approcci Domain Driven Design (DDD)*
- I repository sono oggetti che encapsulano la logica necessaria per l'accesso al database
- Favoriscono la manutenibilità del codice disaccoppiando la tecnologia utilizzata per accedere al database dal modello di dominio

* Vedi corso di APS

Il pattern Repository

- Martin Fowler descrive il pattern repository così:
 - *A repository performs the tasks of an **intermediary between the domain model layers and data mapping**, acting in a similar way to a set of domain objects in memory*
 - *Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer*
 - *Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping*

<https://martinfowler.com/eaaCatalog/repository.html>

Il pattern Repository

- Possiamo pensare un repository come ad una collezione persistente
 - Il programmatore/utilizzatore usa i repository senza che gli sia richiesto di conoscere i dettagli di come internamente viene gestita la persistenza
- Idealmente definiamo una classe Repository per ogni entità del modello di dominio
- La classe Repository dovrà offrire metodi per le operazioni CRUD (salva, recupera, modifica, cancella) sugli oggetti dalla collezione, più altre funzioni di utilità (ad es., conteggio, verifica di esistenza)

Il pattern Repository

- Da un punto di vista implementativo, dovremmo quindi definire una classe che offre metodi per le operazioni CRUD

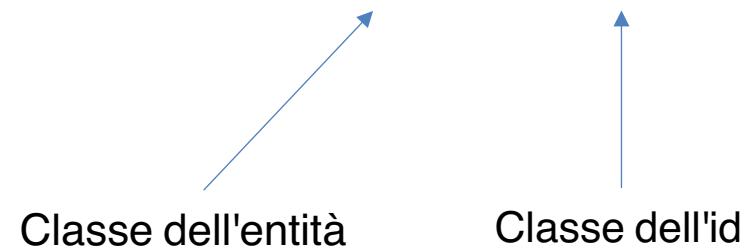
Repository con Spring Boot

- Il framework Spring Boot offre un sistema di librerie per la definizione dei Repository
- Per scrivere una classe Repository è sufficiente definire una Java interface che estende
`org.springframework.data.repository.CrudRepository`
- Automaticamente avremo a disposizione i metodi crud per la nostra classe. Vedi:

<https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html>

CrudRepository: Esempio

```
package it.uniroma3.siw.repository;  
  
import org.springframework.data.repository.CrudRepository;  
import it.uniroma3.siw.model.Movie;  
  
public interface MovieRepository extends CrudRepository<Movie, Long> {  
}
```



I metodi di CrudRepository

long	count() Returns the number of entities available.
void	delete(T entity) Deletes a given entity.
void	deleteAll() Deletes all entities managed by the repository.
void	deleteAll(Iterable<? extends T> entities) Deletes the given entities.
void	deleteAllById(Iterable<? extends ID> ids) Deletes all instances of the type T with the given IDs.
void	deleteById(ID id) Deletes the entity with the given id.
boolean	existsById(ID id) Returns whether an entity with the given id exists.
Iterable<T>	findAll() Returns all instances of the type.
Iterable<T>	findAllById(Iterable<ID> ids) Returns all instances of the type T with the given IDs.
Optional<T>	findById(ID id) Retrieves an entity by its id.
<code><S extends T> S</code>	save(S entity) Saves a given entity.
<code><S extends T></code>	saveAll(Iterable<S> entities) Saves all given entities.
Iterable<S>	

Il modello

- Le operazioni che il sistema offre per manipolare le classi del modello sono offerte da una classe apposita
- In questa classe abbiamo un metodo per ogni operazione
- Questa organizzazione del codice risponde al pattern *Façade*
 - L'idea è quella di mostrare una facciata, attraverso la quale si può accedere ai servizi offerti da un sistema

Il modello

- Mettiamo in una classe del package service i metodi alle operazioni offerte dal sistema
- Diverse soluzioni:
 - una classe per ogni entità
 - un'unica classe con le tutte le operazioni offerte dal sistema
 - una classe per ruolo degli utenti (ad es. professori, studenti, amministrazione)
- Per semplicità, seguiamo la prima soluzione

Il modello

- Le nostre applicazioni per ora sono molto semplici: i servizi offerti coincidono con operazioni CRUD
- Di conseguenza l'implementazione dei metodi delle classi service è quasi sempre una banale invocazione di un metodo del repository

La classe MovieService

```
@Service
public class MovieService {

    @Autowired
    private MovieRepository movieRepository;

    public Movie getMovieById(Long id) {
        return movieRepository.findById(id).get();
    }

    public Iterable<Movie> getAllMovies() {
        return movieRepository.findAll();
    }
}
```

@Autowired

- Spring utilizza il pattern *Dependency Injection* per iniettare automaticamente le variabili da cui una classe dipende
- Nel nostro caso, la classe **MovieService** dipende dalla classe (che implementa) **MovieRepository**
- Ci pensa Spring a creare un oggetto **MovieRepository** e ad assegnarlo alla variabile **movieRepository**

Riassumiamo

- Abbiamo definito il modello: l'entità movie
- Grazie all'ORM incluso nel framework, l'abbiamo associata ad una tabella per gestire la persistenza dei dati
- Abbiamo creato una classe Repository, che ci permette di astrarre l'accesso ai dati memorizzati nel database associati alle istanze delle entità
- Abbiamo definito una classe Service che espone le operazioni offerte dal sistema

Controller

- Come abbiamo visto nella panoramica del corso, le nostre applicazioni adottano il pattern MVC: la responsabilità per la gestione delle richieste viene assegnata ad una classe che nel pattern viene chiamata Controller
- Creiamo una classe controller per ogni entità (potremmo usare soluzioni diverse, ma questa è efficace e sufficientemente generica)
- Possiamo immaginare il Controller come una classe che contiene un metodo Java per ogni richiesta HTTP
 - richiesta HTTP = metodo (GET o POST) + URL

Vediamo le richieste del nostro sistema

Get `/movie/{id}`

Risponde con una pagina che mostra i dati del film con il codice specificato nell'ultima parte dell'URL

Get `/movie`

Risponde con una pagina che mostra la lista di tutti i film

Controller in Spring Boot

- Scriviamo un metodo per ogni richiesta HTTP
- Ogni metodo del controller:
 - gestisce i dati che arrivano con la richiesta
 - prepara e mette a disposizione della vista i dati necessari a produrre la risposta
 - ritorna una stringa, che corrisponde alla vista che deve essere invocata per la produzione della risposta

Vediamo i dati necessari per produrre le risposte

Get `/movie/{id}`

Risponde con una pagina che mostra **i dati del film** con il codice specificato nell'ultima parte dell'URL

Get `/movie`

Risponde con una pagina che mostra la **lista di tutti i film**

Controller in Spring Boot

- La classe controller è annotata `@Controller`
- Ogni metodo è annotato con `@PostMapping(URL)` o `@GetMapping(URL)` per associare il metodo alla richiesta.
 - L'URL è una stringa, eventualmente parametrica
- Per mettere a disposizione del componente che genera la risposta i dati usiamo il metodo `addAttribute(String, Object)` di un oggetto Model, che viene passato come parametro nel metodo
- Per invocare la vista che produrrà la risposta, il metodo ritorna una stringa con il nome della vista

MovieController.java

```
@Controller
public class MovieController {
    @Autowired MovieService movieService;

    @GetMapping("/movie/{id}")
    public String getMovie(@PathVariable("id") Long id, Model model) {
        model.addAttribute("movie", this.movieService.getMovieById(id));
        return "movie.html";
    }

    @GetMapping("/movie")
    public String showMovies(Model model) {
        model.addAttribute("movies", this.movieService.getAllMovies());
        return "movies.html";
    }
}
```

Controller: dettagli

```
@Controller  
public class MovieController {  
    @Autowired MovieService movieService;  
  
    @GetMapping("/movie/{id}")  
    public String getMovie(@PathVariable("id") Long id, Model model) {  
        model.addAttribute("movie", this.movieService.getMovieById(id));  
        return "movie.html";  
    }  
    ...  
}
```

Risponde a richiesta GET
con URL del tipo:
/movie/1231

NON FARE COPIA E INCOLLA DEL CODICE DALLE SLIDE:
POWERPOINT INTRODUCE CARATTERI BIANCHI CHE SPORCANO IL CODICE

Controller: dettagli

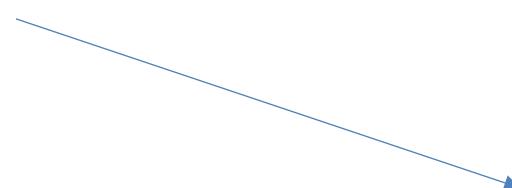
```
@Controller  
public class MovieController {  
    @Autowired MovieService movieService;  
  
    @GetMapping("/movie/{id}")  
    public String getMovie(@PathVariable("id") Long id, Model model) {  
        model.addAttribute("movie", this.movieService.getMovieById(id));  
        return "movie.html";  
    }  
    ...  
}
```

Il parametro id, che fa parte del path viene convertito in Long e passato come parametro al metodo

NON FARE COPIA E INCOLLA DEL CODICE DALLE SLIDE:
POWERPOINT INTRODUCE CARATTERI BIANCHI CHE SPORCANO IL CODICE

Controller: dettagli

```
@Controller  
public class MovieController {  
    @Autowired MovieService movieService;  
  
    @GetMapping("/movie/{id}")  
    public String getMovie(@PathVariable("id") Long id, Model model) {  
        model.addAttribute("movie", this.movieService.getMovieById(id));  
        return "movie.html";  
    }  
    ...  
}
```



La vista selezionata dal controller è
movie.html

Controller: dettagli

```
@Controller  
public class MovieController {  
    @Autowired MovieService movieService;  
  
    @GetMapping("/movie/{id}")  
    public String getMovie(@PathVariable("id") Long id, Model model) {  
        model.addAttribute("movie", this.movieService.getMovieById(id));  
        return "movie.html";  
    }  
    ...  
}
```

Model è un oggetto che permette di scambiare dati tra vista e controller



Chiediamo di mettere a disposizione della vista un oggetto (il secondo parametro di questo metodo), a cui la vista potrà far riferimento con il nome "movie"

@Autowired

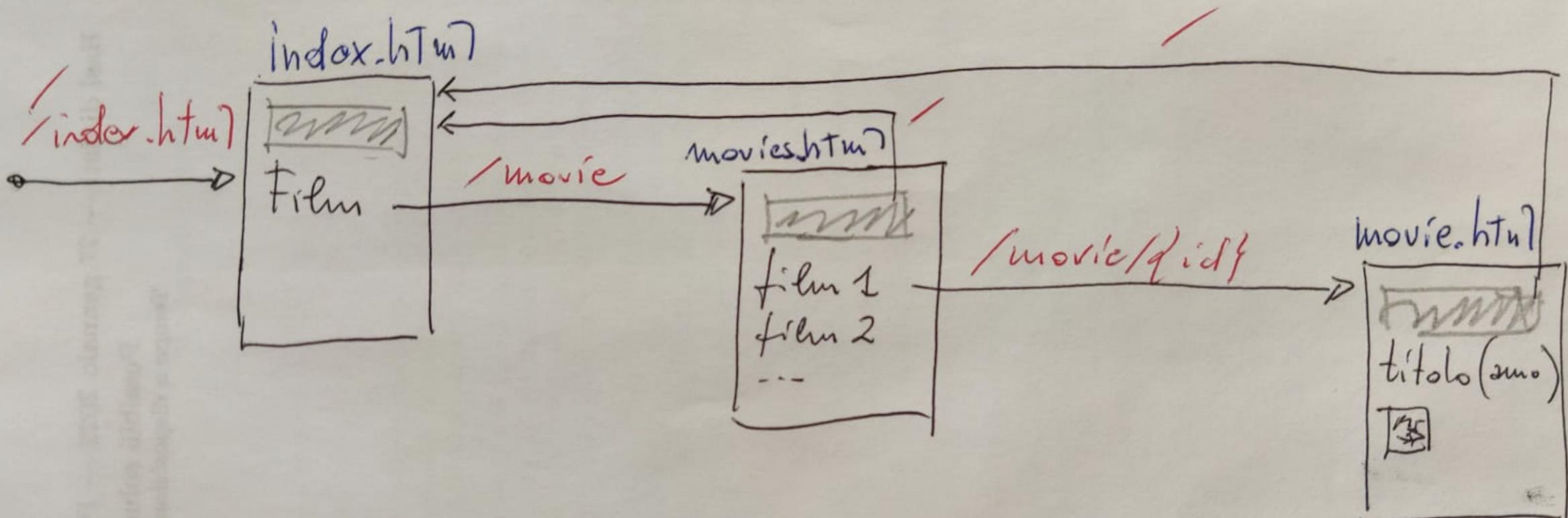
- Anche in questo caso sfruttiamo la *Dependency Injection* per iniettare automaticamente le variabili da cui una classe dipende
- Qui, la classe **MovieController** dipende dalla classe (che implementa) **MovieService**
- Ci pensa Spring a creare un oggetto **MovieService** e ad assegnarlo alla variabile **movieService**

Gestione delle Risposte

- La gestione delle risposte è affidata ai componenti della vista, che hanno la responsabilità di generare la risposta
- Nel nostro caso le risposte sono pagine HTML
- Per la produzione di pagine HTML usiamo un paradigma di programmazione che prevede di scrivere un template di codice HTML nel quale sono immerse espressioni ed istruzioni di controllo (condizioni e cicli) opportunamente codificate
 - Il server, quando genera la risposta, sostituisce l'espressione con il suo valore

Spring Boot: View

- Spring Boot supporta molte tecnologie di templating
- La tecnologia che usiamo noi è Thymeleaf



Thymeleaf

- Thymeleaf è una tecnologia di templating che introduce il paradigma "natural templates"
 - Espressioni e istruzioni di controllo sono espressi tramite attributi HTML (quindi trasparenti al browser)
 - È possibile mettere valori fintizi in corrispondenza delle variabili
 - In questo modo il template può essere manipolato in fase di sviluppo, visualizzandolo correttamente nel browser anche senza far girare l'applicazione

Thymeleaf: esempio

```
<!DOCTYPE html>
<html>
  <head>
    <title>Film</title>
    <link rel="stylesheet" href="/stile.css?version=2" />
  </head>

  <body>
    <div><a href="/"></a></div>
    <div th:if="${movie}">
      <h1 th:text="${movie.title}">Titolo film</h1>
      <div>Anno: <span th:text="${movie.year}">2000</span></div>
      movie.urlImage
    </div>
    <div th:unless="${movie}">Non esiste</div>
  </body>
</html>
```

movie.html

movie è il nome di un oggetto che è stato aggiunto nel Model dal Controller. Possiamo accedere ai campi dell'oggetto con la dot notation

Testo fittizio: ci permette di visualizzare la pagina HTML anche senza l'applicazione. Verrà sostituito con il valore dell'espressione assegnata a th:text

Thymeleaf: istruzioni di controllo

- Le istruzioni di controllo sono espresse tramite attributi nei tag
- Il tag in cui viene messo l'attributo corrispondente ad una istruzione corrisponde anche all'inizio di una istruzione blocco: il tag di chiusura corrisponde alla fine dell'istruzione di blocco

Thymeleaf: esempio for-each

```
<!DOCTYPE html>
<html>
<head>
    <title>Film</title>
    <link rel="stylesheet" href="/stile.css?version=2" />
</head>

<body>
    <div><a href="/"></a></div>
    <h1>Film</h1>
    <div th:if="${movies.isEmpty()}">Non ci sono film nel sistema</div>
    <ul>
        <li th:each="movie : ${movies}">
            <a th:href="@{'/movie' + '/' + ${movie.id}}"
               th:text="${movie.title}">titolo</a>
        </li>
    </ul>
</body>
</html>
```

movies.html

NON FARE COPIA E INCOLLA DEL CODICE DALLE SLIDE:
POWERPOINT INTRODUCE CARATTERI BIANCHI CHE SPORCANO IL CODICE

```
<!DOCTYPE html>
<html>
  <head>
    <title>Film</title>
    <link rel="stylesheet" href="/stile.css?version=2" />
  </head>

  <body>
    <div><a href="/"></a></div>
    <div th:if="${movie}">
      <h1 th:text="${movie.title}">Titolo film</h1>
      <div>Anno: <span th:text="(' + ${movie.year} + ')'">2000</span></div>
      <span></span>
    </div>
    <div th:unless="${movie}">Non esiste</div>
  </body>
</html>
```

Thymeleaf: if-unless

movie.html

NON FARE COPIA E INCOLLA DEL CODICE DALLE SLIDE:
POWERPOINT INTRODUCE CARATTERI BIANCHI CHE SPORCANO IL CODICE

Thymeleaf: espressioni

`${...}`- *Variable expressions*: sono simili alle espressioni EL: contengono variabili che sono nel modello o nella sessione

`*{...}`- *Selection expressions*: fanno riferimento ai campi di una variabile definita in uno scope esterno

`#{...}`- *Message (i18n) expressions*: fanno riferimento a messaggi di testo scritti nei file di messaggi (predisposti all'internazionalizzazione)

`@{...}`- *Link (URL) expressions*: espressioni per costruire riferimenti ipertestuali

`~{...}`- *Fragment expressions*: espressioni per includere frammenti di template

Per dettagli: <https://www.thymeleaf.org/doc/articles/standarddialect5minutes.html>

Organizzazione dei file relativi alla Vista

- I file thymeleaf vanno nella directory **templates**
- I file CSS e le immagini vanno nella directory **static**

Dettagli sulla creazione del database

- Impostando il parametro di configurazione
spring.jpa.hibernate.ddl-auto=create
Stiamo specificando che il database deve essere creato ex-novo
(rimuovendo le tabelle, se esistono) ogni volta che avviamo
l'applicazione
- In fase di sviluppo puo' essere comodo, ma ovviamente quando
l'applicazione è in esercizio non vogliamo perdere i dati. In
esercizio, imposteremo così
spring.jpa.hibernate.ddl-auto=none

Dettagli sulla creazione del database

- In fase di sviluppo è comodo che ad ogni esecuzione le tabelle vengano cancellate e create da zero ad ogni esecuzione
- Ma può essere comodo anche inizializzare il database con alcuni valori
- Per fare questo ci basta predisporre in un file **import.sql** una sequenza di istruzioni SQL per popolare il database
 - Il file **import.sql** deve stare nella stessa directory del file di **application.properties**

Esempio file import.sql

```
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Full metal jacket',  
'https://pad.mymovies.it/filmclub/2006/04/020/locandina.jpg', 1987);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Non e" un paese per vecchi',  
'https://musicart.xboxlive.com/7/89d04600-0000-0000-0000-000000000002/504/image.jpg',2007);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'The founder',  
'https://pad.mymovies.it/filmclub/2016/03/235/locandina.jpg',2016);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Harry Potter e la pietra filosofale',  
'https://pad.mymovies.it/filmclub/2001/12/001/locandina.jpg',2001);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Il pianeta delle scimmie', 'https://media-  
assets.wired.it/photos/615daad62707bc568326abfa/master/w_1600,c_limit/war-for-the-planet-of-the-apes1.jpg',2001);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Lo chiamavano Jeeg Robot',  
'https://www.rai.it/dl/img/2016/02/23/1280x720_1456237082397_jeegrobot.jpg',2015);  
insert into movie (id, title, url_image, year) values(nextval('movie_seq'), 'Yesterday', 'https://citynews-today.stgy.ovh/~media/horizontal-  
mid/20648620006811/yesterday-film-poster-movie-universal-pictures-2.jpg',2019);
```

Nota: **movie_seq** è il nome della sequenza Postgres che genera gli identificativi incrementali dell'entità.
Verificare che il nome sia corretto nel proprio database

Introduzione a Spring Boot: sintesi

- Entità: classi i cui oggetti devono essere persistenti
- Service: classi i cui metodi rappresentano le operazioni offerte dal sistema
- Repository: interface che estende CrudRepository
 - Implementazione dei metodi CRUD
 - Facile estensione con semplice convenzione sui nomi dei metodi
- Controller: classe annotata @Controller
 - Facile associazione metodo-path
 - La stringa ritornata da ciascun metodo corrisponde alla view selezionata dal controller
 - Oggetto Model per scambio dati con la vista
- Vista: template HTML con espressioni
 - Le espressioni permettono di riportare valori degli oggetti passati dal controller e istruzioni condizionali e di ciclo
- @Autowired permette di iniettare dipendenze

Riferimenti

- <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- <https://spring.io/guides/gs/spring-boot/>
- <https://www.thymeleaf.org/>
- <https://www.baeldung.com/spring-boot>

Esercizio

- Creare un nuovo progetto my-siw-movie-01
- Modificare l'applicazione affinchè il nostro sistema gestisca anche informazioni su un'entità che modella gli artisti. A tal fine:
 - Oltre all'entità **Movie**, introdurre l'entità **Artist**
 - Per ogni artista sono di interesse:
 - il nome
 - Il cognome
 - La data di nascita (di tipo `java.time.LocalDate`)
 - inoltre, ad ogni artista associamo come identificatore un intero `Long`
 - Scrivere il codice di **Artist**, **ArtistService**, **ArtistController**, **ArtistRepository**
 - Creare i template thymeleaf opportuni per visualizzare l'elenco degli artisti e per il dettaglio di ciasun artista
 - Popolare opportunamente il database

Esercizio

- Perchè quando proviamo ad accedere a <http://localhost:8080> ci viene restituita una pagina che riporta l'errore 404?
 - Suggerimento 1: cosa è l'errore HTTP 404?
 - Suggerimento 2: c'è qualche metodo del controller che mappa la richiesta HTTP al path "/" ?
- Perchè se lanciamo l'applicazione senza prima stopparla la console ci dice che non può essere avviata perchè la porta 8080 è già occupata?
 - Come risolviamo il problema?
 - La prossima volta, cosa facciamo prima di lanciare l'applicazione?

Esercizio

- Aggiungere all'applicazione
 - un semplice foglio di stile (il nome deve coincidere con quello riportato nel template)
 - l'immagine che appare nell'intestazione della pagina (usare un'immagine qualunque, che potete generare con un LLM multimediale, purchè abbia il nome che coincide con quello indicato nei template)

Sistemi informativi su Web

Introduzione a Spring Boot (parte 2)

aa 2024-2025



Paolo Merialdo
Università degli Studi Roma Tre

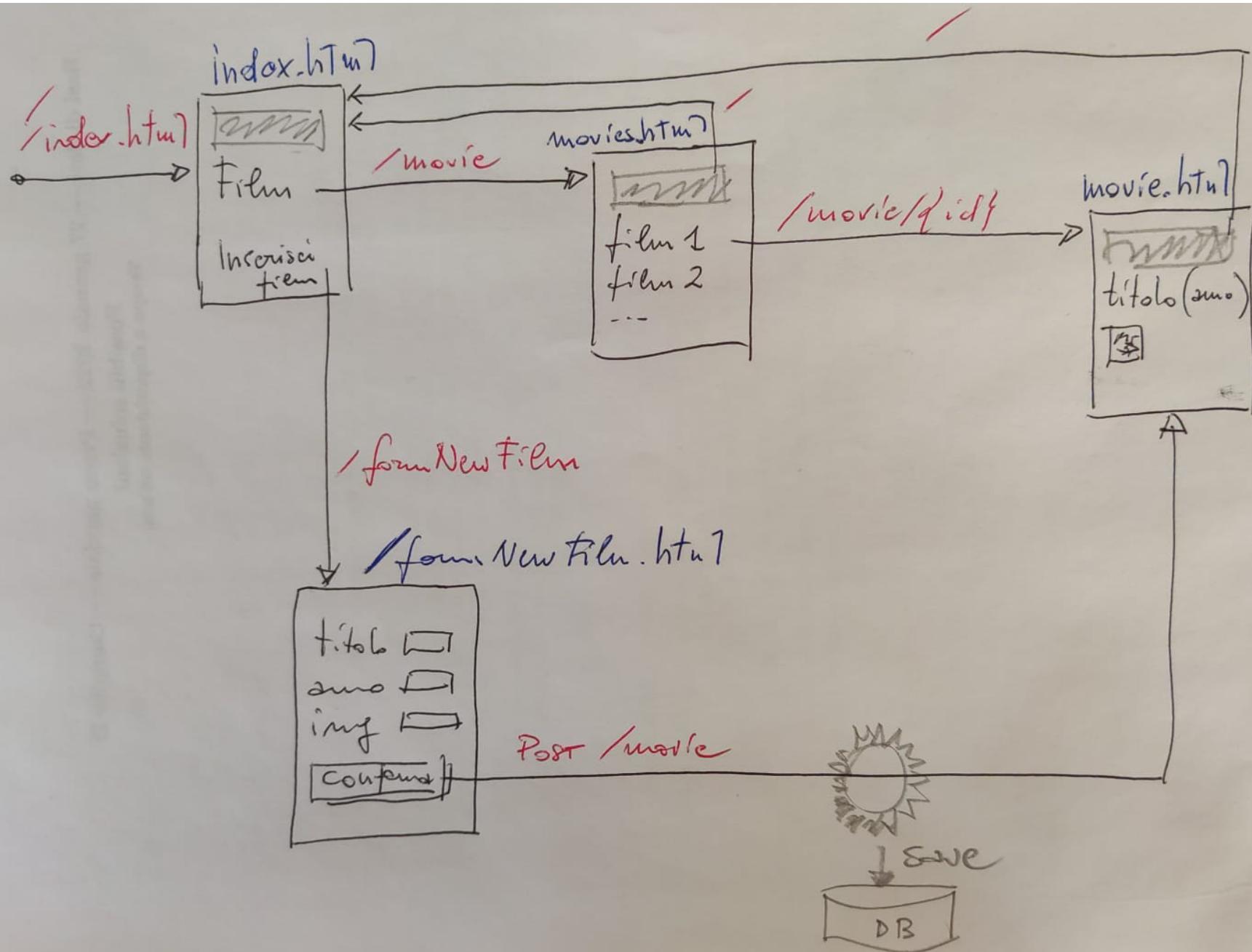


This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> for details

Introduzione a Spring boot

- Fino ad ora abbiamo visto solo operazioni di lettura dei dati
- In un sistema reale ci sono anche operazioni per modificare il database, in particolare per inserire, modificare, cancellare tuple
- Nel nostro esempio, vediamo come possiamo gestire l'inserimento di un nuovo film

- Vediamo dapprima l'applicazione funzionante (in aula)



Vediamo le richieste del nostro sistema

Get `/movie/{id}`

Risponde con una pagina che mostra i dati del film con il codice specificato nell'ultima parte dell'URL

Get `/movie`

Risponde con una pagina che mostra la lista di tutti i film

Get `/formNewMovie`

Risponde con una pagina che contiene la form per inserire i dati di un nuovo film

Post `/movie`

Gestisce i dati di un nuovo film raccolti dalla form: se il film non esiste, salva i dati nel db e risponde con una pagina che mostra i dati così come sono salvati nel db, se il film esiste già risponde con la pagina che mostra la form e con un messaggio di errore (il film esiste)

HTML: Form

- Per creare una form in HTML usiamo il tag `<form>`
- Ogni elemento `<form>` può contenere testo (eventualmente marcato) ed elementi per raccogliere dati (caselle di testo, menu a tendina, scelte opzionali, etc).
- Per ora, noi usiamo l'elemento più semplice: la casella di testo
`<input type="text"/>`

Che nel browser viene visualizzato così:



HTML: Form

- Una form HTML raccoglie i dati inseriti dall'utente e li manda (attraverso un messaggio HTTP GET o POST) al server, affinché siano processati da una risorsa
- Il metodo HTTP (GET o POST) e la risorsa che deve processare i dati sono espressi all'interno del tag `<form>` attraverso gli attributi `method` e `action`

```
<form action="path della risorsa" method="POST">
```

Thymeleaf: form

- Per la gestione delle form, Thymeleaf permette di associare alla form:
 - Il path specificato nel controller attraverso l'attributo th:action
 - <form th:action="@{/movie}" method="POST" th:object="\${movie}">
 - L'oggetto i cui campi vengono raccolti attraverso la form
 - <form th:action="@{/movie}" method="POST" th:object="\${movie}">
 - I campi dell'oggetto:
 - <input type="text" th:field="\${movie.title}" />

Thymeleaf: form

```
<!DOCTYPE html>
<html>
<head>
<title>New Movie</title>
<link rel="stylesheet" href="/stile.css?version=3" />
</head>
<body>
<div><a href="/"></a></div>
<h1>Aggiungi un film</h1>
<form th:action="@{/movie}" method="POST" th:object="${movie}">
<div>
    Titolo: <span><input required type="text" th:field="${movie.title}" /></span>
</div>
<div>
    Anno: <span><input required type="text" th:field="${movie.year}" /></span>
</div>
<div>
    URL immagine: <span><input type="url" th:field="${movie.urlImage}" /></span>
</div>
<div>
    <span><button type="submit">Conferma</button></span>
</div>
</form>
</body>
</html>
```

formNewMovie.html

NON FARE COPIA E INCOLLA DEL CODICE DALLE SLIDE:
POWERPOINT INTRODUCE CARATTERI BIANCHI CHE SPORCANO IL CODICE

MovieController.java

```
@Controller
public class MovieController {
    @Autowired MovieService movieService;

    @GetMapping("/formNewMovie")
    public String formNewMovie(Model model) {
        model.addAttribute("movie", new Movie());
        return "formNewMovie.html";
    }

    @PostMapping("/movie")
    public String newMovie(@ModelAttribute("movie") Movie movie, Model model) {
        this.movieService.save(movie);
        model.addAttribute("movie", movie);
        return "movie.html";
    }

// continua
```

```
@GetMapping("/movie/{id}")
public String getMovie(@PathVariable("id") Long id, Model model) {
    model.addAttribute("movie", this.movieService.getMovieById(id).get());
    return "movie.html";
}

@GetMapping("/movie")
public String showMovies(Model model) {
    model.addAttribute("movies", this.movieService.getAllMovies());
    return "movies.html";
}

}
```

Dettagli Thymeleaf vengono discussi a lezione

Un problema

- Dopo un inserimento, se l'utente ricarica la pagina (con i dettagli del film), viene mandata la stessa richiesta precedente (quindi viene inserito nuovamente lo stesso film: fare un prova con l'applicazione). Infatti l'ultima richiesta era la richiesta POST di inserimento dei dati
- Per evitare questo inconveniente, possiamo fare in modo che il Controller, anzichè inoltrare la richiesta alla vista, mandi un messaggio al client chiedendogli di fare la richiesta per la pagina con i dettagli del film appena inserito

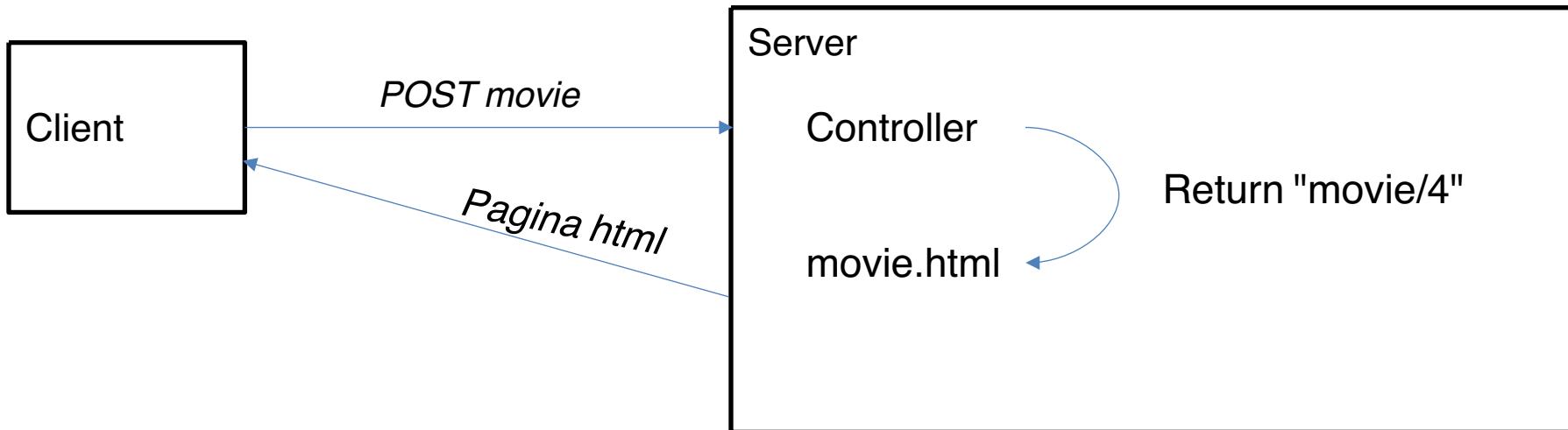
redirect

- Questa operazione, tecnicamente viene mettendo nella stringa ritornata dal controller il prefisso "redirect:"
- Esempio:

```
@PostMapping("/movie")
public String newMovie(@ModelAttribute("movie") Movie movie) {
    this.movieService.save(movie);
    return "redirect:/"+movie.getId();
}
```

redirect

- Se non mettiamo redirect:, il controller invoca direttamente (sul server) un metodo interno all'application server che corrisponde alla costruzione della risposta con il template passato come parametro
- Con redirect: il controller invia un messaggio HTTP 302 al client imponendogli di fare una richiesta HTTP GET ad una risposta specificata (nel nostro caso a movie/{id})



REDIRECT

