**GUIDA PROGETTI SPRING SU VSCODE  
Author: Romina Trazzi**

## Setup dell’IDE

**⚙️ Accesso rapido ai comandi (Command Palette)**

* Premi **Ctrl + Shift + P** per aprire la **Command Palette**.
* Digita e seleziona **Spring Initializr: Generate a Maven Project[[1]](#footnote-1)**
* Segui i passaggi guidati:
  + Scegli il linguaggio (**Java**).
  + Seleziona la versione di **Spring Boot**.
  + Immagine che contiene testo, schermata, Carattere

    Il contenuto generato dall'IA potrebbe non essere corretto.Aggiungi le **dipendenze** (es. Spring Web, Spring Data JPA, ThymeLeaf).
  + Assegna un nome al progetto e seleziona il percorso di salvataggio.
  + Impostare il file **src** → **main** → **resources** → application.properties
* **Ctrl + Shitf + P** per aggiungere Lombok (add startes).  
  Controllare poi il file pom.xml perché ci sia la dipendenza.

**File application.properties e .env**

***application.properties***

🔍 Contiene le **configurazioni dell’applicazione** Spring Boot:

* La porta su cui si avvia Tomcat (server.port=8080)
* L’indirizzzo del database
* Configurazioni personalizzate

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Dopo il nome dell’applicazione, troviamo il driver jdbc che ci connette al database all’URI indicato da Aiven usando un modo sicuro.  
  
Sempre per sicurezza rendiamo Username e Password criptate (variabili d’ambiente) e non le condividiamo su Github, tenendole in chiaro solo in un file chiamato .env da inserire nell’elenco di gitignore (cioè i file che vengono ignorati da Git e non pushati).

***.env***  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Qui ci sono i parametri della connessione al DB in chiaro (infatti non vengono pushati perché .env è nella lista di gitignore).  
  
Questi dati si trovano inizialmente nella configurazione di Aiven.  
  
**Nota:** nello scaffolding .env si trova allo stesso livello della cartella src.

***application.properties (continua da sopra)***Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

🔁 **spring.jpa.hibernate.ddl-auto=update**

**Significato:** dice a Hibernate cosa fare con lo schema del database (cioè tabelle, colonne, chiavi...) all'avvio dell'applicazione.

Immagine che contiene testo, schermata, Carattere, software

Il contenuto generato dall'IA potrebbe non essere corretto.  
  
**✅ Consiglio per ambienti di sviluppo:** usare update.

**⚠ Attenzione:** in produzione usare none o validate per non rischiare di perdere dati.

**🪵 spring.jpa.show-sql=true**

**Significato:** stampa tutte le query SQL generate da Hibernate nella console.

**📌 Utile per il debug:**

* Fa vedere cosa viene eseguito davvero sul database.
* Aiuta a ottimizzare o correggere le query.

**🧾 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect**

**Significato:** Hibernate può lavorare con tanti tipi di database (MySQL, H2, Oracle...).

Questa riga dice a Hibernate:

"Sto usando PostgreSQL, quindi usa la sintassi e ottimizzazioni SQL specifiche di quel database."

**✅** Senza questa riga, Hibernate proverebbe a indovinare il dialect — ma potresti avere comportamenti strani o inefficienti.

Immagine che contiene testo, schermata, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

**🖥️ Terminale / Shell** *(da testare)*

* Se esegui l’app da terminale:   
    
  **./mvnw spring-boot:run**
* Vedrai nel terminale i log dell’avvio Spring + query Hibernate.
* **🧩 Cosa mostra show-sql=true?**
* Solo le **query SQL**
* Senza parametri **bindati** (es. ?, non i veri valori)
* Solo le query invocate **automaticamente da Hibernate** (es. SELECT, INSERT, UPDATE, DDL)
* **🔎 Vuoi vedere anche i parametri reali?**
* Per visualizzare anche i **valori reali** passati nelle query (bind parameters), aggiungi questo nel tuo application.properties:   
    
  **logging.level.org.hibernate.SQL=DEBUG  
  logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE**

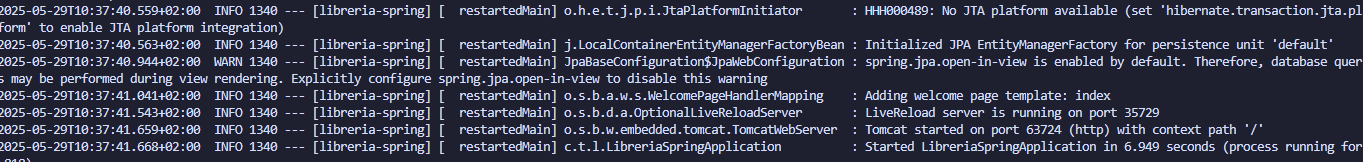
Immagine che contiene schermata, testo, software, Software multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

**Importante!**Di solito Tomcat si avvia a porta 8080, ma se non vi funziona usate la porta 0 (in questo modo Tomcat ogni volta che lo usate sceglierà lui la prima porta libera).   
Voi dovrete controllare che sulla barra degli indirizzi vi compaia il numero di porta giusto quando aprite il progetto.

* + Aprire la cartella diretta del progetto dall’Esagono in app (altrimenti non va)
  + Immagine che contiene testo, schermata, Carattere

    Il contenuto generato dall'IA potrebbe non essere corretto.Guardare a che porta si apre tomcat (finestra terminal di Vscode)

Qui si è aperto a porta **63724**

* + Schiacciate l’esagomo e poi il mappamondo in aprire per aprire il browser dentro VsCode

Immagine che contiene testo, schermata, software, multimediale

Il contenuto generato dall'IA potrebbe non essere corretto.

## Spring Spring è un framework di Java che funziona con le annotazioni (simbolo @ che separa la logica del Model dal Controller e dalla View e serve per velocizzare le connessioni con i database e il recupero dei dati. Al suo interno ha altri framework e librerie di terze parti come Hibernate e Thymeleaf e persino il server Tomcat. Scaffolding di un progetto Spring (= albero delle cartelle)

Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**Dove va la roba?**

* **src 🡪 main 🡪 resources 🡪 static**  
  Cartella css (a sua volta divisa con cartella mediaquery, partial ed altre eventuali)  
  Cartella js  
  Cartella img  
  Sono le risorse “statiche” del sito.
* **src 🡪 main 🡪 resources 🡪 template**Le viste (view) che sono pagine HTML, che grazie a ThymeLeaf diventano dinamiche  
  La pagina index.html che rappresenta la homepage del sito
* **src 🡪 main** 🡪 **java** 🡪 **com 🡪 nome\_dominio 🡪 nome\_progetto**

Contiene tutto il codice sorgente Java che viene compilato dalla JVM (file .java)

👉 Questo segue la **convenzione di naming dei package** in Java:

* com è il **dominio di primo livello**, che inizia con "com" per "company"
* trazzi è il **nome del dominio dell’azienda o dello sviluppatore**.
* nome\_progetto identifica il progetto specifico.

Cartella Controller (se non ci sono presentation e integration)  
Cartella Entities (ex Model)  
Cartella Repo  
Cartella Services  
Cartella Integration (per output file json)  
Cartella Presentation (per output file html e pattern mvc)

* **readup 🡪 .env**

È il file che contiene in chiaro username e password personali per connettersi al database. Va inserito nell’elenco di gitignore.   
Nello scaffolding è “fratello” di src (attenzione a non inserirlo dentro).

* **target**  
    
  Cartella “specchio” di src, dove va a finire il codice compilato dalla JVM (file .class), non va toccata.
* **file .gitignore**  
  Il file .gitignore è un file di testo speciale utilizzato in Git.  
    
  ✅ **Scopo:** dire a Git quali file o cartelle non devono essere tracciati o inclusi nel repository (in pratica è una sorta di magazzino)  
    
  💡 In pratica, serve per evitare di caricare file inutili o sensibili, come:
  + File temporanei generati dall’ambiente di sviluppo.
  + Configurazioni locali (es. .idea, \*.iml).
  + File compilati o di output (\*.class, target/).
  + Credenziali e configurazioni private (.env, config/)

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**📚Annotazioni**

* **JPA (Java Persistence API)**: è uno **standard Java** per salvare oggetti Java in un database relazionale.
* **Hibernate**: è una delle implementazioni più usate di JPA. Fa da ponte tra Java e il database.

Questi strumenti permettono di usare **le annotazioni @** invece di scrivere direttamente SQL per leggere/salvare dati.

Le **annotazioni** sono funzioni che servono a **definire comportamenti specifici** per classi, metodi e variabili, semplificando la configurazione e separando le responsabilità nell’MVC.  
Si scrivono come **@Annotazione**.  
  
Permettono di **gestire Bean**, mappare richieste HTTP, e collegare Spring a **Hibernate/JPA** per fare l’ORM, ovvero per “riflettere” quello che facciamo nell’app all’interno del database.  
  
**Bean =** Oggetto gestito da Spring (esempio: servizi, controller, repository).

**✅ Annotazioni principali di Spring**

| **Annotazione** | **📌 Significato e Utilizzo** |
| --- | --- |
| @Component | Indica che la classe è un **Bean gestito da Spring**. |
| @Service | Specializzazione di @Component, per classi di **logica di business**. |
| @Repository | Specializzazione di @Component, per **data access** (DAO). |
| @Controller | Definisce una classe come **controller MVC** per gestire richieste web. |
| @RestController | Unisce @Controller e @ResponseBody, per API REST. |
| @Autowired | Inietta automaticamente i **bean** nelle classi. |
| @RequestMapping | Mappa URL e metodi HTTP a funzioni specifiche. |
| @GetMapping, @PostMapping | Shortcut di @RequestMapping per HTTP GET e POST. |
| @Value | Inietta valori da configurazioni (es. application.properties). |

**Azioni principali di Spring  
  
Iniezione delle dipendenze (Dependency Injection)**

Spring può **iniettare automaticamente** le dipendenze nelle classi grazie a:

* @Autowired (inietta Bean).
* @Value("${chiave}") (inietta valori da configurazioni).

**Mappatura delle richieste (Routing)**

Spring permette di mappare **URL** a metodi:

* @RequestMapping("/path"): generico.
* @GetMapping("/path"): specifico per GET.
* @PostMapping("/path"): specifico per POST.

**Annotazioni per la persistenza dei dati**

Quando Spring gestisce dati su database, si usano annotazioni di **JPA** (Hibernate):

* @Entity: definisce una classe come tabella.
* @Id: definisce la chiave primaria.
* @GeneratedValue: genera automaticamente il valore ID.
* @Column: personalizza le colonne.

**START**

**LIVELLI DI SPRING  
  
**

Dopo aver creato i quattro \ cinque package (in VS Code cartelle) all’interno di:  
  
**src 🡪 main** 🡪 **java** 🡪 **com 🡪 nome\_dominio 🡪 nome\_progetto**

**CONTROLLER**

Si inizia dal controller

Il Controller reagisce ad una richiesta da parte del client quando l’utente interagisce con la nostra pagina (es. click su un bottone che fa accadere qualcosa, invia un form o apre un link).  
Secondo il pattern MVC, il Controller si occupa di far comunicare la View (quindi la pagina HTML se siamo su un sito web) con il Model (cioè i dati estratti dal database e fatti diventare una classe Java).

📄 Un Controller è una **classe Java** annotata con **@Controller** o **@RestController**, a seconda se deve restituire una vista HTML (quindi una pagina web), oppure un file in formato JSON.

Fa una cosa molto importante:

* 🔗 Si occupa del **routing** (instradamento): decide quale cosa restituire all’utente, in base alla **sitemap** progettata dagli sviluppatori.

Per fare il routing ha bisogno di creare delle **route (rotte)**, cioè degli indirizzi delle pagine web \ risorse del sito (potrebbe essere anche un file .pdf ad esempio).

📚 Collega le **rotte** alle azioni (metodi) presenti nel service usando annotazioni come:

* **@GetMapping** → per richieste di visualizzazione (GET).
* **@PostMapping** → per invio di dati (POST).

Le route sono delle **funzioni** annotate con **@GetMapping** (se vogliamo inviare la richiesta di vedere una specifica pagina web), oppure **@PostMapping** (se vogliamo inviare i dati di un form per salvarli su un database)

Se la risorsa restituità è una pagina HTML, la stringa di ciascuna funzione “get” del Controller DEVE ASSOLUTAMENTE essere uguale al nome del file .html della vista presente nella cartella template, altrimenti non vedrà la pagina (in questo esempio devono esistere nella cartella template vista\_libri.html e index.html per la home.   
In caso del post nell’esempio si utilizza la dicitura redirect:/nome\_route per far restare l’utente sulla stessa pagina\vista)

Il controller fa anche un’altra cosa molto importante:

* **Deve essere connesso al service tramite l’annotazione @Autowired per poter recuperare i dati del Model e passarli alla View.**

Immagine che contiene testo, schermata, schermo, software

Il contenuto generato dall'IA potrebbe non essere corretto.

**Qui abbiamo 3 rotte:** la home, libri (get) e libri (post).

**Abbiamo la “chiamata” al service**, che ci dà accesso alle funzioni getLibri e addLibro (specificate nella classe service. Vedi dopo).  
  
**Abbiamo accesso ad un’istanza del Model** (che possiamo chiamare m, model o simili).  
Model è **un oggetto** fornito da Spring che serve per **trasportare dati dal Controller alla View** (cioè alla pagina HTML che vedrà l’utente).  
Questo oggetto come tutti gli oggetti ha dei suoi metodi e delle sue proprietà.

Uno dei suoi metodi è **addAttribute**, che funziona in questo modo:  
  
Immagine che contiene testo, Carattere, bianco, Elementi grafici

Il contenuto generato dall'IA potrebbe non essere corretto.

* "nome" = è il nome con cui il dato sarà disponibile nella pagina HTML.
* valore = è il dato vero e proprio (es: un numero, una stringa, una lista).

Internamente, funziona come una Map<String, Object>, dove puoi associare un nome (la chiave) e un valore (oggetto o dato).

In questo modo la View (es: vista\_libri.html) potrà usare il nome "libri" per **stampare i dati nella pagina html**.  
Questo processo si chiama data binding, perché lega alla variabile espressa da “nome” uno specifico “valore”.   
Chi si occupa di sostituire il nome con il value corrispondente è un template engine come Thymeleaf.  
  
Esempio di vista\_libri.html:

Immagine che contiene testo, Carattere, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene software, Software multimediale, Software per la grafica, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

Questo è invece un Controller REST (pacchetto integration), formato da una interfaccia:  
  
Immagine che contiene testo, Carattere, schermata, Elementi grafici

Il contenuto generato dall'IA potrebbe non essere corretto.  
  
E da una classe ObjREST:  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.L’annotazione PostMapping ha associata l’associazione RequestBody a cui passiamo una risorsa (un oggetto, o un file).

**SERVICE**

📄 **Service** è una classe Java che contiene **le funzioni** per lavorare con i dati.  
È marcato con l’annotazione **@Service** per farlo riconoscere a Spring come componente.  
Viene iniettato nel Controller tramite l’annotazione **@Autowired**.  
  
Per poter lavorare con i dati, ha bisogno di prenderli dal DAO, il quale a sua volta viene iniettato con l’annotazione **@Autowired**.  
Questo dà inoltre accesso alle funzioni predefinite del Dao in Springboot.

Immagine che contiene testo, schermata, software, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

È possibile avere anche un “doppio” service composto da:  
- Una interfaccia ObjService   
- Una classe ObjServiceImpl

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

**REPOS (DAO)**

📄 Il **DAO** è una interfaccia che estende una interfaccia speciale chiamata JPA Repository che semplifica l’accesso ai database e le operazioni CRUD (Create Read Update Delete) sui dati dell’entità specificata (es. Libro), senza dover scrivere codice SQL.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Questi metodi vengono poi usati nel service:  
  
Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

Se vogliamo inserire altri metodi più customizzati, che dovranno per forza essere implementati, li possiamo definire qui in modo generico, senza specificare il corpo della funzione.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**ENTITIES**L’entity è una classe Java che corrisponde ad una tabella del database.  
L’annotazione **@Entity** indica che questa classe rappresenta una **tabella** nel database (grazie a JPA).  
**@Entity**

* **Definizione:** L’annotazione @Entity (javax.persistence.Entity) dichiara che la classe è una *managed entity* di JPA e corrisponde a una tabella del database.
* **Comportamento JPA:** Traccia i cambiamenti riguardanti la tabella che dall’applicazione si riflettono sul database.

**@Table (opzionale)**

* Immagine che contiene testo, schermata, Carattere, numero

  Il contenuto generato dall'IA potrebbe non essere corretto.**Definizione:** L’annotazione @Table (javax.persistence.Table) specifica i dettagli della tabella che viene mappata dall’Entity, quando il nome della classe (di default) non coincide con quello desiderato.  
  Se non dichiarata @Table è implicita e il nome tabella è infatti preso dalla classe.
* **Comportamento JPA:** Specifica il nome della tabella se diverso dal nome della classe.

**@Id**

* **Definizione:** L’annotazione @ID (javax.persistence.Id) Identifica il campo come chiave primaria della tabella del database.  
  Senza @Id, JPA non sa come distinguere le righe e genera un errore di mapping.
* **Requisito:** Ogni Entity deve avere esattamente un campo annotato con @Id.

**@GeneratedValue(strategy = GenerationType.XXX)**

* **Definizione:** L’annotazione @GeneratedValue (javax.persistence.GeneratedValue) configura la strategia di generazione automatica dell’ID (es. auto-increment)
* **Strategie disponibili:**
  + **AUTO (default):** JPA sceglie la migliore per il database.   
    Va bene quando siamo sul branch di sviluppo (dev), ma in produzione è meglio esplicitare cosa vogliamo usare.
  + **IDENTITY:** usa auto‐increment nativo (MySQL, PostgreSQL…), va bene per piccole applicazioni.
  + **SEQUENCE:** usa una sequenza database (Oracle, PostgreSQL…).   
    Si definisce anche @SequenceGenerator per personalizzare nome e increment. Viene usato in applicazioni enterprise.
  + **TABLE:** usa una tabella per generare ID (meno performante). Viene usato con quei database che non supportano Sequence.

**@Column (opzionale)**

* **Definizione:** L’annotazione @Column (javax.persistence.Column) mappa il campo come una colonna e consente di personalizzarla.

Attributi principali:

* **name** – nome della colonna
* **length** – lunghezza massima (per tipi String)
* **nullable** – se ammette valori null
* **unique** – vincolo di unicità
* **Comportamento JPA:**  
  Se omessa, JPA usa il nome del campo come nome di colonna e valori di default (nullable=true, length standard).

**Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.Esempio di configurazione enterprise base:**

**Esempio di configurazione enterprise avanzata:**   
  
Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

**@Override dei metodi ereditati dalla classe Object**

Equals e HashCode sono due metodi a disposizione di default per ogni oggetto di ciascuna classe (come ToString) perché vengono ereditati dalla superclasse Object e quindi possono essere sovrascritti.  
Definire equals() e hashCode() nella classe **Libro** (o **Utente**, o **Sfida**) permette di:

* Confrontare due oggetti.
* Lavorare correttamente con **Set**, **Map** e collezioni basate su hash.
* Assicurare operazioni di inserimento, ricerca e rimozione **veloci** e **coerenti**.
* Mantenere il **persistence context** di JPA pulito, senza duplicati in memori.

Senza un’implementazione corretta di equals() e hashCode(), il comportamento delle classi può diventare imprevedibile in diversi contesti:

* **Collezioni** basate su hash (HashSet, HashMap, ecc.)
* **Duplicati “invisibili”:** due oggetti con stessi valori (titolo, autore, …) verranno considerati diversi perché ereditano l’equals() di Object, che confronta solo il riferimento in memoria.
* **contains() e remove()** falliscono: se cerchi o rimuovi un oggetto “identico” a uno già presente, non lo troverai perché il loro hashCode() e equals() non coincidono.

**Persistence Context di JPA/Hibernate**

**Duplicazione di entità in cache:** Hibernate usa equals() per capire se un’istanza è già gestita; senza di esso potresti avere più copie della stessa riga in memoria, con possibili incoerenze e update inattesi.

**Problemi in relazioni Set<…>:** se una @OneToMany Set<Libro> contiene entità senza equals()/hashCode(), Hibernate non riconosce duplicati e potrebbe generare record ripetuti nelle tabelle di join.

**Cache e chiavi di mapping**

Qualsiasi sistema di caching (Spring Cache, EHCache, …) che usa i tuoi oggetti come chiavi non funzionerà correttamente: le lookup in cache falliranno, vanificando i benefici prestazionali.

**Logica di business e confronti**

Ogni volta che scrivi if (libro1.equals(libro2)) otterrai sempre false a meno che non sia proprio lo stesso oggetto in memoria (stessa istanza), rendendo inutile ogni confronto “logico” basato sui campi.

**Video utile:** <https://www.youtube.com/watch?v=N2EszxG-1qM>

**Esempio semplice**

Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**(ecc. set e get per tutte le proprietà\colonne)  
  
(costruttore)**

**Esempio completo**Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, software

Il contenuto generato dall'IA potrebbe non essere corretto.

Immagine che contiene testo, schermata, Carattere

Il contenuto generato dall'IA potrebbe non essere corretto.Immagine che contiene testo, schermata, Carattere, linea

Il contenuto generato dall'IA potrebbe non essere corretto.

**LE VISTE HTML**

Nella cartella templates vanno le pagine HTML del sito.  
  
**INDEX.HTML**È la pagina di accesso al sito\web app e ha come route il semplice /.  
In questo esempio vengono aggiunti qui sia Thymeleaf, che il framework Pico.css, che un inizio di navbar.  
Thymeleaf è un template engine di Spring che ci permette di sbloccare l’attributo “th:value” che permette di potenziare e rendere dinamico l’HTML grazie al data binding.  
Immagine che contiene testo, schermata

Il contenuto generato dall'IA potrebbe non essere corretto.

**VISTA\_LIBRI.HTML**Immagine che contiene testo, schermata, design

Il contenuto generato dall'IA potrebbe non essere corretto.

**Per compilare le classi Java e creare un file Jar Eseguibile:**maven package (in console)  
  
Poi su Bash scrivere java -jar

**🧩 Passaggi tecnici (a prova di Shadi)**

1️. Il **client invia una richiesta** (es: GET /libri).  
  
2. Spring intercetta la richiesta e la **passa al controller**.  
  
3️. l **controller**:

* 🔗 **Richiama il Service** per ottenere i dati dal database.

Il Service a sua volta si collega al DAO (Repository) che interroga il database tramite ORM (es: JpaRepository).

* 📦 **Aggiunge i dati al Model** (oggetto Model di Spring) usando addAttribute("nome", valore).
* 📜 **Restituisce il nome della View** (esempio: "vista\_libri"), che corrisponde al nome del file HTML da mostrare.

4️. Spring cerca un file HTML in **src/main/resources/templates** chiamato vista\_libri.html.

* 📌 In questo file i dati passati nel Model sono disponibili come variabili (es: ${libri}).

5️. Il server Tomcat restituisce la **pagina HTML** al client, il browser la interpreta e l’utente la vede.

1. Maven è il gestore dei pacchetti di SpringBoot (come npm e composer) [↑](#footnote-ref-1)