
SPRING BOOT
MANUAL
SPRING BOOT

MANUALE SPRING BOOT

---

1. Introduzione

1.1 **Cosa troverai in questo manuale**
- Obiettivi e finalità del testo
- Panoramica dell'applicazione demo (derivata dal progetto Spring MVC)

1.2 **Perché passare a Spring Boot**
- Vantaggi e differenze rispetto a Spring MVC tradizionale
- Come il "bootstrapping" semplifica la configurazione

1.3 **Requisiti e pubblico di riferimento**
- Conoscenze pregresse (Java, Maven/Gradle, Spring di base)
- Strumenti necessari per seguire passo dopo passo

---

2. Installazione degli Strumenti e Configurazione dell'Ambiente

2.1 **Installazione di Java**
- Versione consigliata (JDK 11 o superiore)
- Installazione su Windows, macOS, Linux

2.2 **Installazione di Maven o Gradle**
- Differenze fra i due build tool
- Configurazione nelle variabili d'ambiente
- Installazione su Windows, macOS, Linux

2.3 **IDE e Editor di Testo**

- Spring Tool Suite (STS), IntelliJ IDEA, Eclipse
- Configurazione e plugin consigliati

2.4 **MySQL e strumenti correlati**

- Installazione di MySQL su Windows, macOS, Linux
- Strumenti di gestione (MySQL Workbench, Adminer, ecc.)
- Configurazione iniziale di database e utenti

2.5 **Verifica dell'ambiente**

- Test rapidi per assicurarsi che tutto sia installato correttamente

---

3. Panoramica di Spring Boot

3.1 **Cos'è e come funziona Spring Boot**

- Principi fondamentali (Convention over Configuration, Starter POM, autoconfigurazione)

3.2 **Le dipendenze "Starter" di Spring Boot**

- A cosa servono e come scegliere quelle necessarie

3.3 **Struttura di un progetto Spring Boot**

- Il "main class" e l'annotazione `@SpringBootApplication`
- File di configurazione (`application.properties` o `application.yml`)

3.4 **Differenze di concetto rispetto a Spring MVC classico**

- Cosa cambia nella gestione dei servlet, `web.xml` e file di configurazione XML

---

4. Creazione del Progetto Spring Boot

4.1 **Creazione tramite Spring Initializr**

- Utilizzo dell'interfaccia web e/o dell'IDE (STS/IntelliJ)
- Scelta delle dipendenze base

4.2 **Creazione manuale di un progetto**

- Configurazione del `pom.xml` (Maven) o `build.gradle` (Gradle)
- Aggiunta degli Starter di base (Web, JPA, DevTools, ecc.)

4.3 **Struttura delle directory**

- Pacchetti principali (`controller`, `service`, `repository`, `model`, ...)
- Organizzazione consigliata delle classi

---

5. Porting dell'Applicazione da Spring MVC a Spring Boot

5.1 **Analisi del progetto esistente**

- Revisione della struttura MVC classica
- Identificazione dei file di configurazione XML o Java-based

5.2 **Migrazione delle dipendenze**

- Come mappare le dipendenze di Spring MVC nelle dipendenze Starter di Spring Boot
- Rimozione dei file di configurazione non più necessari

5.3 **Aggiornamento del database MySQL**

- Verifica dello schema (diagramma allegato)
- Migrazione delle Entity (Hibernate/JPA) nel nuovo progetto Spring Boot

5.4 **Riconfigurazione di security (se presente)**

- Da `spring-security.xml` a configurazione Java con Spring Boot
- Integrazione con i "Security Starter"

5.5 **Gestione di template e static resources**

- Come spostare JSP/Thymeleaf (o altre view) in Spring Boot
- Configurazione del path per risorse statiche

5.6 **Verifica finale**

- Test end-to-end: controlli di rotta (controller mapping), viste e database

---

6. Configurazione di Spring Boot

6.1 `application.properties` e `application.yml`

- Differenze e best practice nell'organizzazione dei parametri
- Configurazioni di base: porta, context path, connessione al database

6.2 **Configurazioni di Profilo**

- `spring.profiles.active` e gestione di ambienti (dev, test, production)
- File separati per ogni profilo e come utilizzarli

6.3 **Logging in Spring Boot**

- Configurazione con Logback/Log4j2
- Livelli di log e log personalizzati

---

7. Accesso al Database e Persistenza Dati

7.1 **Panoramica di JPA/Hibernate in Spring Boot**

- Autoconfigurazione JPA e definizione delle Entity
- L'uso di `spring-boot-starter-data-jpa`

7.2 **Mappatura delle tabelle e validazione delle Entity**

- Revisione del diagramma MySQL allegato (Courses, Lessons, ecc.)
- Creazione delle classi model e delle relazioni (OneToMany, ManyToMany, ecc.)

7.3 **Repository Layer**

- `CrudRepository`, `JpaRepository`, query methods e query personalizzate
- Esempi di implementazione con i moduli `register`, `roles`, `subscription`

7.4 **Migrazione e Tooling Database**

- Flyway o Liquibase per la migrazione dello schema (opzionale)
- Test e popolamento dati iniziale

---

8. Strato Web: Controller, Service e View

8.1 **Creazione dei Controller**

- Annotazioni tipiche (`@RestController` vs `@Controller`, `@RequestMapping`, ecc.)
- Gestione dei parametri e dei path

8.2 **Service Layer**

- Gestione della logica di business
- Pattern consigliati di separazione dei layer

8.3 **Gestione delle View**

- Integrazione con Thymeleaf/JSP/altro (se necessario)
- Configurazione e path di default

8.4 **Validazione e gestione degli errori**

- Bean Validation (`@Valid`, `@NotNull`, ecc.)
- Gestione delle eccezioni con `@ControllerAdvice`

---

9. Sicurezza e Gestione Utenti

9.1 **Spring Security con Spring Boot**

- Configurazione rapida con `spring-boot-starter-security`
- Creazione di un Security Configuration File (Java Config)

9.2 **Integrazione con Database**

- Autenticazione e autorizzazione basate su tabelle `register`, `roles`, `register_roles`
- Persistenza delle credenziali, password encoding

9.3 **Gestione delle sessioni e Remember-Me**

- Sessioni in ambiente cluster/distribuito
- Opzionale: integrazione con token JWT (se previsto)

9.4 **Protezione degli endpoint**

- Configurazione di `HttpSecurity`, filtri, e regole di accesso
- Esempi di ruoli e permessi (admin, user)

---

10. Testing dell'Applicazione

10.1 **Testing Unitario e d'Integrazione**

- JUnit e Spring Boot Test
- MockMVC per testare i controller

10.2 **Test del Database**

- @DataJpaTest e test con database in-memory (H2)
- Esecuzione di query di integrazione

10.3 **Best Practice per i test in Spring Boot**

- Strutturazione dei package di test
- Test containers e test su environment Docker (opzionale)

---

11. Deployment e Distribuzione

11.1 **Creazione del file Jar o War**

- Differenze e vantaggi tra i due formati
- Configurazioni di packaging in Maven/Gradle

11.2 **Esecuzione stand-alone su diversi sistemi**

- Lancio su Windows, macOS, Linux con `java -jar`
- Configurazione di variabili d'ambiente e parametri di avvio

11.3 **Deployment su server esterno (Tomcat, Jetty, ecc.)**

- Configurazione e adeguamento (se War)
- Esempio di deployment su Tomcat remoto

11.4 **Docker e Containerizzazione**

- Creazione di una Dockerfile base
- Docker Compose per integrare DB e applicazione

---

12. Considerazioni e Approfondimenti Finali

12.1 **Manutenzione e aggiornamento di Spring Boot**

- Gestione delle versioni e del supporto a lungo termine
- Consigli su come rimanere aggiornati

12.2 **Monitoraggio e Logging avanzato**

- Actuator di Spring Boot
- Metriche e health checks

12.3 **Evoluzioni possibili dell'applicazione**

- Microservizi e Cloud (Breve cenno a Spring Cloud)
- Scalabilità e architettura distribuita

12.4 **Appendici**

- Troubleshooting comuni
- Risorse utili (documentazione ufficiale, community, plugin)

Introduzione manuale Spring Boot

Ragionato sull'introduzione del manuale Spring Boot per 4 secondi

1. Introduzione

L’obiettivo di questa sezione è fornire una panoramica chiara e completa di ciò che il lettore troverà all’interno del manuale dedicato a **Spring Boot**, spiegando sia le motivazioni per cui questo framework rappresenta un significativo passo avanti rispetto a Spring MVC tradizionale, sia i requisiti di base necessari per seguirne i contenuti.

1.1 Cosa troverai in questo manuale

Obiettivi e finalità del testo

In questo manuale, l’obiettivo principale è guidarti attraverso la realizzazione di un’applicazione web basata su **Spring Boot**, avvalendoci di un esempio pratico e completo. Il testo è concepito per essere un **riferimento passo-passo**: partendo dalle nozioni elementari del framework, si arriverà gradualmente a integrare funzionalità più complesse, consentendo di acquisire una visione d’insieme di Spring Boot e delle sue potenzialità. Nello specifico, le finalità del manuale includono:

1. **Fornire solide basi concettuali**: Comprendere i principi su cui si fonda Spring Boot, il suo modello di configurazione e le caratteristiche che lo contraddistinguono.
2. **Guidare l’implementazione tecnica**: Dimostrare come avviare un progetto, configurarlo, gestire le dipendenze e distribuire l’applicazione in modo sicuro ed efficiente.
3. **Offrire best practice**: Mettere in evidenza gli aspetti architetturali e di design (pattern) consigliati, consentendo di sviluppare progetti robusti e manutenibili.
4. **Stimolare l’approfondimento**: Fornire spunti e suggerimenti utili per estendere l’applicazione con funzionalità avanzate (sicurezza, integrazione con servizi esterni, monitoraggio e altro ancora).

Panoramica dell’applicazione demo (derivata dal progetto Spring MVC)

Il manuale include un’applicazione demo che prende le mosse da un classico progetto **Spring MVC**. Questa base di partenza permette di confrontare i paradigmi, la configurazione e il

deployment di un progetto nato in ambiente Spring tradizionale con quanto offre Spring Boot. In particolare, l'applicazione demo illustrerà:

- **Struttura dei package e dei moduli**: Come organizzare al meglio le classi Controller, le entità di dominio, i servizi e i repository.
- **Gestione delle dipendenze con Maven o Gradle**: Come integrare i vari **starter** Spring Boot e utilizzare la convenzione "opinionated" tipica del framework.
- **Funzionalità di base**: Creazione di controller REST, accesso a un database relazionale, gestione delle viste (se prevista un'interfaccia HTML) e sicurezza di base (Spring Security).
- **Esempi di configurazione**: Mostrare la differenza tra i file XML o le classi di configurazione Java di Spring MVC e le semplici annotazioni/parametri di configurazione di Spring Boot.

1.2 Perché passare a Spring Boot

Vantaggi e differenze rispetto a Spring MVC tradizionale

Spring Boot nasce come evoluzione naturale di Spring MVC, con l'intento di **ridurre al minimo la configurazione necessaria** e semplificare radicalmente la fase di avvio di un progetto Spring. Alcuni dei principali vantaggi sono:

1. **Configurazione automatica (Auto-Configuration)**: Evita la necessità di gestire manualmente file XML o classi Java per abilitare funzionalità comuni, come la persistenza dei dati o la sicurezza.
2. **Starter POM**: I cosiddetti "starter" (per esempio `spring-boot-starter-web`, `spring-boot-starter-data-jpa`, ecc.) includono un set già pronto di dipendenze per le funzionalità più diffuse, riducendo drasticamente il tempo speso nella ricerca e gestione delle versioni.
3. **Embedded Server**: Spring Boot integra server come Tomcat o Jetty "out of the box". Non è più obbligatorio caricare l'applicazione su un container esterno: è possibile avviarla con un semplice comando (`java -jar`) o tramite un IDE.
4. **Opinioni definite (Opinionated)**: Pur offrendo libertà di scelta, Spring Boot fornisce impostazioni di default e convenzioni che consentono di sviluppare più rapidamente senza rinunciare a flessibilità e possibilità di personalizzazione.

5. **Strumenti di monitoring e auditing**: Con facilità è possibile integrare librerie per il tracciamento delle prestazioni, la raccolta di metriche e il logging strutturato, grazie alle dipendenze pronte all'uso.

Nel passaggio da Spring MVC a Spring Boot, si notano dunque differenze significative in termini di **tempo di configurazione**, **avvio del progetto** e **rapidità** con cui si può ottenere un prodotto funzionale.

Come il "bootstrapping" semplifica la configurazione

Il concetto chiave è il **"bootstrapping"**: il framework rileva automaticamente librerie e annotazioni nel classpath e procede a configurarle con valori di default ragionevoli. Di conseguenza:

- **Meno file di configurazione**: Si riduce l'uso di XML e si centralizzano le impostazioni nel file `application.properties` (o `application.yml`).
- **Dipendenze coerenti**: Grazie agli "starter", le versioni delle librerie sono allineate e compatibili, evitando conflitti di dipendenze.
- **Orientamento "convention over configuration"**: Creazione e avvio di un progetto semplici, sfruttando le impostazioni predefinite di Spring Boot, modificabili all'occorrenza.

Questo approccio semplifica notevolmente la fase iniziale di qualsiasi sviluppo, diminuendo i tempi di setup e lasciando più spazio all'**implementazione della logica di business**.

1.3 Requisiti e pubblico di riferimento

Conoscenze pregresse (Java, Maven/Gradle, Spring di base)

Per trarre il massimo da questo manuale, è consigliabile che il lettore disponga di alcune conoscenze preliminari, tra cui:

- **Java**: Conoscenza di base delle caratteristiche del linguaggio (classi, interfacce, collezioni, eccezioni, ecc.).
- **Strumenti di build (Maven o Gradle)**: Familiarità con la struttura di un `pom.xml` o di un file `build.gradle` e con i concetti di dipendenze e repository.

- **Concetti di Spring di base**: Come funzionano le annotazioni tipiche (`@Controller`, `@Service`, `@Repository`, `@Configuration`), l'inversion of control (IoC) e l'injection delle dipendenze (DI).

Anche se non strettamente necessario, avere un'idea delle **architetture web** (RESTful, server-side rendering, microservizi) può facilitare la comprensione delle varie soluzioni illustrate.

Strumenti necessari per seguire passo dopo passo

Per la parte pratica, è utile disporre di:

1. **Ambiente Java**: Un JDK aggiornato (almeno la versione 8, anche se è consigliato utilizzare versioni più recenti come Java 17).
2. **IDE**: Un ambiente di sviluppo come IntelliJ IDEA, Eclipse o Visual Studio Code (con estensioni adeguate).
3. **Sistema di controllo di versione** (opzionale ma raccomandato): Git, per gestire e versionare il codice durante le sperimentazioni.
4. **Docker** (opzionale): Se si desidera testare l'applicazione in container o integrare servizi esterni con la massima facilità.
5. **Database locale** (es. MySQL, PostgreSQL, H2 embedded): Per gli esempi sulla persistenza dei dati.

Con questi prerequisiti e strumenti, sarai in grado di **seguire tutti gli esempi** e replicare o estendere l'applicazione demo proposta nel corso del manuale.

Conclusioni della sezione

L'introduzione fornisce i fondamenti per comprendere le motivazioni che hanno spinto la comunità Java verso un approccio più automatizzato e produttivo come quello di **Spring Boot**. Nei prossimi capitoli, si entrerà nel dettaglio dell'installazione degli strumenti e della configurazione iniziale, per poi passare alla realizzazione vera e propria di un'applicazione web completa, coprendo i concetti di sicurezza, persistenza, test e deployment. Il percorso proposto mira a trasformare le conoscenze teoriche in competenze pratiche, sfruttando al massimo l'ambiente offerto dal framework Spring Boot.

Installazione Spring Boot

---

2. Installazione degli Strumenti e Configurazione dell'Ambiente

2.1 Installazione di Java

1. **Versione consigliata (JDK 11 o superiore)**
   - Per utilizzare Spring Boot, è fortemente consigliato installare almeno la versione 11 di Java (JDK 11).
   - Spring Boot è generalmente compatibile anche con versioni successive (es. Java 17). Se si preferisce una Long-Term Support (LTS), Java 11 e Java 17 sono entrambe buone scelte.
2. **Installazione su Windows, macOS, Linux**
   Di seguito, uno schema generale per l'installazione su ciascun sistema operativo.
   - **Windows**
     1. Scaricare l'installer dal sito ufficiale di Oracle o da un distributore alternativo (Eclipse Temurin, Amazon Corretto, ecc.).
     2. Eseguire il file `.exe` e seguire le istruzioni del wizard.
     3. Al termine, verificare l'installazione aprendo il Prompt dei comandi ed eseguendo:

        ```bash
        java -version
        ```

        Dovreste vedere un output che conferma la versione di Java installata.
   - **macOS**
     1. Scaricare il `.dmg` o il pacchetto `.pkg` (ad esempio da Adoptium (Temurin)).

2. Aprire il pacchetto e seguire le istruzioni di installazione.
3. Per verificare, aprire il Terminale ed eseguire:

```bash
java -version
```

- **Linux**
  - Su distribuzioni basate su Debian/Ubuntu:

```bash
sudo apt-get update
sudo apt-get install openjdk-11-jdk
```

  - Su distribuzioni basate su Red Hat/Fedora/CentOS:

```bash
sudo dnf install java-11-openjdk
```

  - Verificare con:

```bash
java -version
```

In tutti i casi, assicurarsi che la variabile di ambiente `JAVA_HOME` (o `PATH`) sia correttamente configurata.

2.2 Installazione di Maven o Gradle

1. **Differenze fra i due build tool**
   - **Maven**: utilizza un file di configurazione XML (`pom.xml`), è basato su un modello di progetto standard e fornisce una struttura molto diffusa nella comunità Java.
   - **Gradle**: utilizza un file di configurazione testuale (`build.gradle` o `build.gradle.kts` in Kotlin DSL), è più flessibile e spesso più veloce.

- Entrambi gestiscono le dipendenze, i cicli di build, test e packaging. La scelta dipende spesso da preferenze personali o da standard di progetto.

2. **Configurazione nelle variabili d'ambiente**
   Sia per Maven sia per Gradle, dopo l'installazione è necessario assicurarsi che la cartella bin (contenente gli eseguibili `mvn` o `gradle`) sia inclusa nella variabile `PATH`.
   - Esempio di configurazione su Windows:
     1. Aprire *Pannello di Controllo* → *Sistema* → *Impostazioni di sistema avanzate* → *Variabili d'ambiente*.
     2. Trovare la variabile `PATH`, fare clic su "Modifica" e aggiungere il percorso, ad esempio `C:\Program Files\Apache\maven\bin`.
   - Su macOS e Linux, di solito si modifica il file `~/.bashrc`, `~/.zshrc` o `~/.profile` aggiungendo una riga come:

     ```bash
     export PATH=$PATH:/opt/maven/bin
     ```

3. **Installazione su Windows, macOS, Linux**
   - **Maven**
     1. Scaricare l'archivio `.zip` o `.tar.gz` dal sito ufficiale di Maven.
     2. Scompattare l'archivio in una cartella (ad es. `C:\apache-maven-x.x.x` o `/opt/maven`).
     3. Configurare la variabile d'ambiente `M2_HOME` (opzionale ma consigliata) e aggiungere `M2_HOME/bin` a `PATH`.
     4. Verificare con:

        ```bash
        mvn -v
        ```

   - **Gradle**
     1. Scaricare l'archivio `.zip` dal sito ufficiale di Gradle.
     2. Decomprimere nella cartella di preferenza (ad es. `C:\gradle` o `/opt/gradle`).
     3. Configurare la variabile d'ambiente `GRADLE_HOME` (opzionale) e aggiungere `GRADLE_HOME/bin` a `PATH`.

4. Verificare con:

```bash
gradle -v
```

2.3 IDE e Editor di Testo

1. **Spring Tool Suite (STS), IntelliJ IDEA, Eclipse**
   - **Spring Tool Suite (STS)**: un IDE basato su Eclipse con integrazioni specifiche per Spring. Ottimo per progetti Spring Boot grazie a wizard e template dedicati.
   - **IntelliJ IDEA**: molto popolare, offre una versione Community gratuita (ottima per Java e Spring Boot), e una versione Ultimate a pagamento con funzionalità aggiuntive.
   - **Eclipse**: IDE storico per Java, personalizzabile con molti plugin. STS si basa proprio su Eclipse, ma Eclipse "classico" può essere arricchito con il "Spring Tools" plugin.
2. **Configurazione e plugin consigliati**
   - **Spring Tools**: fornisce funzionalità dedicate (creazione veloce di progetti Spring Boot, assistenza nei file di configurazione, suggerimenti, integrazione con Spring Initializr). Disponibile per Eclipse/STS e IntelliJ.
   - **Lombok**: se il progetto utilizza Lombok, è necessario installare il plugin (o configurare l'agente Lombok) affinché l'IDE riconosca correttamente le annotation (`@Getter`, `@Setter`, ecc.).
   - **Editor YAML/Properties**: i progetti Spring Boot utilizzano spesso file `application.properties` o `application.yml`; disporre di evidenziazione della sintassi e auto-completamento è molto utile.
   - **Git Integration**: per il versionamento del codice e l'integrazione continua.

2.4 MySQL e strumenti correlati

1. **Installazione di MySQL su Windows, macOS, Linux**

- **Windows**
  1. Scaricare il MySQL Installer (Community Edition) dal sito ufficiale MySQL.
  2. Seguire il wizard per installare server, client e (opzionalmente) MySQL Workbench.
  3. Impostare username e password (di default `root`), e configurare la porta (di default 3306).
- **macOS**
  1. Scaricare il `.dmg` dal sito MySQL.
  2. Installare seguendo le istruzioni, impostando la password di root.
  3. Dopo l'installazione, è possibile avviare/fermare MySQL dal pannello Preferenze di Sistema o usando i comandi da terminale.
- **Linux**
  - Su Ubuntu/Debian:

    ```bash
    sudo apt-get update
    sudo apt-get install mysql-server
    ```

  - Su Fedora/Red Hat/CentOS:

    ```bash
    sudo dnf install mysql-server
    ```

  - Avviare il servizio:

    ```bash
    sudo service mysql start
    ```

  - Configurare la password di root e le impostazioni iniziali tramite lo script:

    ```bash
    sudo mysql_secure_installation
    ```

2. **Strumenti di gestione (MySQL Workbench, Adminer, ecc.)**

- **MySQL Workbench**: strumento ufficiale di gestione e progettazione database per MySQL. Permette di creare tabelle, eseguire query, fare modeling ER, ecc.
- **Adminer**: un'applicazione PHP leggera da installare sul server per gestire MySQL attraverso un browser. Alternativa semplice a phpMyAdmin.
- **Altri tool**: DBeaver, HeidiSQL, phpMyAdmin, etc.

3. **Configurazione iniziale di database e utenti**
   - Collegarsi al server MySQL (es. `mysql -u root -p`).
   - Creare un database (es. `springboot_db`):

     ```sql
     CREATE DATABASE springboot_db;
     ```

   - Creare un utente dedicato (es. `springuser`) e assegnare privilegi:

     ```sql
     CREATE USER 'springuser'@'%' IDENTIFIED BY 'springpass';
     GRANT ALL PRIVILEGES ON springboot_db.* TO 'springuser'@'%';
     FLUSH PRIVILEGES;
     ```

   - Verificare il corretto accesso al database con l'utente appena creato.

2.5 Verifica dell'ambiente

1. **Test rapidi per assicurarsi che tutto sia installato correttamente**
   - **Java**:

     ```bash
     java -version
     ```

     Deve mostrare la versione corretta (>= 11).
   - **Build Tool** (Maven o Gradle):

     ```bash
     ```

```
mvn -v
gradle -v
```

Devono mostrare la versione e l'installazione corretta.

- **IDE**:
  - Aprire l'IDE scelto (STS, IntelliJ, Eclipse).
  - Creare un nuovo progetto Spring Boot tramite wizard o Spring Initializr.
  - Verificare che non ci siano errori di configurazione.
- **MySQL**:
  - Avviare il server MySQL e connettersi con l'utente creato:

    bash
    ```
    mysql -u springuser -p
    ```
  - Eseguire qualche query di prova per testare la connessione e il database `springboot_db`.
- **Progetto Spring Boot di prova**:
  - Creare un progetto Spring Boot minimal (anche con Spring Initializr).
  - Configurare `application.properties` (o `application.yml`) per connettersi a MySQL:

    properties
    ```
    spring.datasource.url=jdbc:mysql://localhost:3306/springboot_db
    spring.datasource.username=springuser
    spring.datasource.password=springpass
    ```
  - Avviare l'applicazione e verificare che si avvii correttamente, senza errori di connessione.

Se tutti questi step si concludono positivamente, l'ambiente di sviluppo per Spring Boot è correttamente configurato.

Conclusione

Con questa procedura, si dispone di tutto il necessario per iniziare a sviluppare progetti Spring Boot: Java (almeno JDK 11), un build tool (Maven o Gradle), un IDE adeguato (STS, IntelliJ o Eclipse con plugin), e un database MySQL pronto per essere utilizzato dall'applicazione.

Spring Boot Panoramica

---

3. Panoramica di Spring Boot

Spring Boot nasce come estensione dell'ecosistema Spring per semplificare la configurazione e l'avvio di applicazioni Java. L'obiettivo principale è ridurre al minimo la complessità per lo sviluppatore, sfruttando un insieme di convenzioni che permettono di avere un progetto Spring funzionante con pochissime configurazioni iniziali.

3.1. Cos'è e come funziona Spring Boot

Principi fondamentali

1. **Convention Over Configuration**
   Spring Boot adotta il principio di "convenzione piuttosto che configurazione". Significa che, di default, mette a disposizione un insieme di configurazioni già pronte, basate su convenzioni comuni e "scelte sensate" (sensible defaults). Ad esempio, se si utilizza un database relazionale, Spring Boot può fornire automaticamente un DataSource, una gestione delle transazioni, e addirittura un database embedded per i test se non viene rilevata la presenza di un database esterno.
2. **Starter POM**
   Gli "Starter POM" sono dei pacchetti speciali pensati per ridurre la complessità delle dipendenze Maven (o Gradle, se si utilizza Gradle). Invece di dover specificare manualmente decine di librerie (Spring Core, Spring Web, Jackson, Validation, ecc.), si possono importare direttamente uno o più "starter" che raccolgono già le dipendenze tipiche di un certo ambito.
   - Ad esempio, `spring-boot-starter-web` include tutto il necessario per sviluppare un'applicazione web in Spring (Tomcat embedded, Spring MVC, Jackson per JSON, ecc.).
   - `spring-boot-starter-data-jpa` include le librerie di JPA e Hibernate per l'accesso ai dati su database relazionali, integrandosi con Spring Data.

3. **Autoconfigurazione**
   L'autoconfigurazione (o Auto-Configuration) è forse la funzionalità più caratteristica di Spring Boot. Se le dipendenze giuste sono incluse nel progetto, Spring Boot proverà a configurarle automaticamente, facendo "scansioni" dei componenti nel classpath e attivando i relativi Bean se soddisfatte certe condizioni.
   - Se nel classpath c'è un driver JDBC e si rileva un RDBMS, Spring Boot crea automaticamente i Bean di configurazione per connettersi al database.
   - Se nel classpath c'è Spring MVC (`spring-web`), verranno create le strutture di base per gestire le richieste HTTP, come un server integrato (Tomcat, Jetty o Undertow).

L'idea è che, in un progetto standard, molte configurazioni di base siano già preimpostate. Laddove servano personalizzazioni, è sempre possibile "sovrascrivere" le impostazioni di default e aggiungerne di proprie.

3.2. Le dipendenze "Starter" di Spring Boot

Le dipendenze Starter (i cosiddetti "starter modules") semplificano notevolmente la configurazione Maven o Gradle. Ciò evita di dover cercare e aggiungere manualmente ogni singola libreria.

A cosa servono

- **Aggregare dipendenze correlate**: se devi creare un'applicazione web con funzionalità REST e manipolazione JSON, ti basta aggiungere `spring-boot-starter-web`. Questo pacchetto include:
  - Spring MVC
  - Un container embedded come Tomcat
  - Jackson per la serializzazione/deserializzazione JSON
  - Altre dipendenze minori per logging, validazione, ecc.
- **Garantire compatibilità delle versioni**: gli starter POM sono curati da Spring. Ciò assicura che tutte le librerie al loro interno siano compatibili tra loro e allineate alla versione di Spring Boot che utilizzi.

Come scegliere quelle necessarie

- **Valutare il tipo di applicazione**: se stai realizzando un'applicazione Web, quasi certamente userai `spring-boot-starter-web`. Se lavori con database relazionali e vuoi utilizzare Spring Data JPA, sceglierai `spring-boot-starter-data-jpa`.
- **Consultare la documentazione ufficiale**: la documentazione di Spring Boot fornisce un elenco completo degli starter disponibili (`spring-boot-starter-test`, `spring-boot-starter-security`, `spring-boot-starter-actuator`, ecc.).
- **Modularità e dimensione del progetto**: aggiungi solo gli starter di cui hai realmente bisogno, evitando di appesantire il progetto con dipendenze inutili.

3.3. Struttura di un progetto Spring Boot

Un progetto Spring Boot segue una struttura semplice e generalmente coerente:

1. **La "main class" e l'annotazione** `@SpringBootApplication`
   - La classe principale, spesso posizionata nel package root del progetto, contiene il metodo `main(...)` da cui parte l'esecuzione.
   - L'annotazione `@SpringBootApplication` racchiude tre annotazioni chiave di Spring:
     - `@Configuration` (indica che la classe contiene configurazioni di Spring),
     - `@EnableAutoConfiguration` (abilita il meccanismo di autoconfigurazione),
     - `@ComponentScan` (attiva la scansione automatica dei componenti all'interno del package in cui si trova la classe stessa e dei suoi sottopackage).
   - Scrivendo ad esempio:

     ```java
     @SpringBootApplication
     public class MyApplication {
         public static void main(String[] args) {
             SpringApplication.run(MyApplication.class, args);
         }
     }
     ```

     Vengono avviati tutti i processi interni di Spring Boot (autoconfigurazione, avvio del server web integrato se presente, ecc.).
2. **File di configurazione (`application.properties` o `application.yml`)**

- **application.properties**: file testuale con proprietà di configurazione nella forma “chiave=valore”.
  Esempio:

  ```properties
  server.port=8081
  spring.datasource.url=jdbc:mysql://localhost:3306/mydb
  spring.datasource.username=myuser
  spring.datasource.password=mypassword
  ```

- **application.yml**: file YAML più flessibile e leggibile per molte persone.
  Esempio:

  ```yaml
  server:
    port: 8081

  spring:
    datasource:
      url: jdbc:mysql://localhost:3306/mydb
      username: myuser
      password: mypassword
  ```

- Questi file vivono tipicamente nella cartella `src/main/resources` e consentono di personalizzare l’autoconfigurazione di Spring Boot.
- È possibile definire più file di configurazione (ad es. `application-dev.properties`, `application-prod.yml`) e attivarli in base al profilo attivo (`spring.profiles.active=dev`), in modo da gestire ambienti di sviluppo, test e produzione.

3. **Organizzazione delle classi e dei package**
   - **Controller**: solitamente in un package come `com.example.demo.controller`, con classi annotate con `@RestController` o `@Controller`.
   - **Service**: le classi di servizio risiedono in un package dedicato (`com.example.demo.service`) e contengono la logica di business (annotate con `@Service`).

- **Repository**: se si utilizza Spring Data JPA o simili, i repository si mettono in `com.example.demo.repository` e sono interfacce annotate con `@Repository` (o implicitamente riconosciute quando estendono `JpaRepository`).
- **Model/Entity**: classi che rappresentano il dominio dell'applicazione, spesso in `com.example.demo.model` o `com.example.demo.entity`.

3.4. Differenze di concetto rispetto a Spring MVC classico

Nel modello "classico" di Spring MVC (prima di Spring Boot), lo sviluppatore doveva configurare manualmente diversi file e componenti. Eccone alcuni aspetti chiave e come cambiano in Spring Boot:

1. **Gestione dei Servlet e del file `web.xml`**
   - In un progetto Spring tradizionale, era spesso necessario definire i servlet (come il `DispatcherServlet`) all'interno di un file `web.xml`.
   - Con Spring Boot, di default, non c'è bisogno del `web.xml`. Il container embedded è configurato automaticamente, e il `DispatcherServlet` è creato da Spring Boot se trova le dipendenze di Spring MVC.
   - Se si vogliono configurazioni particolari del servlet, si può intervenire tramite classi Java-based (ad esempio estendendo `WebMvcConfigurer` o creando Bean specifici). Tuttavia, per la maggior parte dei casi standard, non serve intervenire manualmente.
2. **File di configurazione XML**
   - In Spring MVC classico, era comune utilizzare file XML (come `applicationContext.xml` o `dispatcher-servlet.xml`) per definire Bean, mappature, component-scan, ecc.
   - Spring Boot predilige **configurazioni Java-based** e l'autoconfigurazione, quindi il grosso del lavoro di setup è fatto in modo automatico o tramite annotazioni nelle classi. Anche se è ancora possibile integrare file XML, non è più la via raccomandata.
   - La maggior parte delle configurazioni che prima erano XML ora si trovano in `application.properties/.yml` o in classi @Configuration.
3. **Profilazione e ambienti**

- In progetti Spring classici, la configurazione per diversi ambienti veniva gestita spesso con più file XML o meccanismi personalizzati.
- Con Spring Boot, il concetto di **profili** (`spring.profiles.active` , `spring.profiles.include` , ecc.) è nativo e integrato. Puoi quindi avere `application-dev.yml` , `application-prod.yml` , ecc. e far sì che Spring Boot carichi automaticamente le proprietà in base al profilo attivo.

4. **Avvio e gestione del server**
   - Con Spring MVC classico, si aveva solitamente un file WAR distribuito su Tomcat/Jetty/GlassFish esterno.
   - Con Spring Boot, l'applicazione è di default un eseguibile "jar" (o "war" se necessario) dotato di un server embedded. Questo semplifica molto il deployment: basta lanciare `java -jar <nome_applicazione>.jar` e l'applicazione parte su un Tomcat integrato.
   - È comunque possibile configurare e utilizzare un server esterno se lo si desidera, ma non è più obbligatorio.

Conclusioni

I punti 3.1, 3.2, 3.3 e 3.4 forniscono uno sguardo d'insieme a ciò che rende Spring Boot diverso (e spesso preferibile) rispetto a un progetto Spring classico:

- **3.1**: Visione generale di Spring Boot e i suoi principi chiave (Convention Over Configuration, Starter POM, Autoconfigurazione).
- **3.2**: L'importanza degli Starter nel semplificare la gestione delle dipendenze.
- **3.3**: Come organizzare un progetto Spring Boot, dal main con `@SpringBootApplication` ai file di configurazione `application.properties/.yml` .
- **3.4**: Le differenze sostanziali rispetto al "vecchio" Spring MVC, in termini di assenza del `web.xml` , di minima necessità di file XML, e di server integrato per l'avvio dell'applicazione.

Questa impostazione rende Spring Boot uno strumento molto potente per avviare rapidamente progetti Spring, riducendo sensibilmente il tempo richiesto per la

configurazione e permettendo di concentrarsi principalmente sulla logica di business e sulle funzionalità dell'applicazione.

Creazione progetto Spring Boot

4. Creazione del Progetto Spring Boot
Questo punto del manuale illustra i vari modi per iniziare un progetto Spring Boot, concentrandosi sia sulla **creazione automatica** tramite l'**Initializr** (o gli IDE specializzati come STS/IntelliJ) sia sulla **creazione manuale** (configurando personalmente i file `pom.xml` o `build.gradle`). Infine, descrive la **struttura consigliata** delle directory e dei pacchetti di un progetto Spring Boot.

4.1 Creazione tramite Spring Initializr

1. **Introduzione a Spring Initializr**
   - Spring Initializr è un servizio (accessibile da start.spring.io) che permette di generare rapidamente l'impalcatura di un progetto Spring Boot.
   - È possibile utilizzarlo sia tramite interfaccia web, sia direttamente all'interno di alcuni IDE (Spring Tool Suite o IntelliJ IDEA) grazie a plugin integrati.
2. **Utilizzo dell'interfaccia web e/o dell'IDE (STS/IntelliJ)**
   - **Interfaccia web**:
     1. Si accede a start.spring.io.
     2. Si specificano i metadati principali (Group, Artifact, Nome del progetto, ecc.).
     3. Si seleziona il tipo di build system (Maven o Gradle).
     4. Si imposta la versione di Java e la versione di Spring Boot desiderata.
     5. Si aggiungono le dipendenze base (ad esempio: **Spring Web**, **Spring Data JPA**, **Spring DevTools**, **Lombok**, ecc.).
     6. Si genera il progetto e lo si scarica in un archivio ZIP, che successivamente potrà essere importato nell'IDE.
   - **Utilizzo dell'IDE (es. STS/IntelliJ)**:
     1. Dal menù "New → Spring Starter Project" (o "New Project" con template Spring Boot), si inseriscono le stesse informazioni di base (Group, Artifact, ecc.).

2. Si selezionano le dipendenze desiderate su un'interfaccia grafica analoga a quella disponibile sul sito.
3. L'IDE creerà automaticamente la struttura del progetto e i file necessari (`pom.xml` o `build.gradle` configurati con le dipendenze richieste).

3. **Scelta delle dipendenze base**
   - In genere, nelle prime fasi di un progetto Spring Boot, le dipendenze più comuni da includere sono:
     - **Spring Web**: per creare applicazioni web (REST o MVC).
     - **Spring Data JPA**: per l'accesso a database relazionali tramite JPA/Hibernate.
     - **Spring DevTools**: per agevolare lo sviluppo, grazie al live reload e altre funzionalità utili.
     - **Lombok** (opzionale): per ridurre il boilerplate code (getter/setter, log, ecc.).

4.2 Creazione manuale di un progetto

1. **Configurazione del `pom.xml` (Maven) o `build.gradle` (Gradle)**
   - Se non si utilizza lo Spring Initializr, è possibile creare un progetto da zero:
     1. Creare manualmente la struttura di cartelle (ad es. `src/main/java`, `src/main/resources`, ecc.).
     2. Aggiungere un file di build: per Maven sarà `pom.xml`, per Gradle sarà `build.gradle`.
     3. Inserire al loro interno le dipendenze necessarie, come `spring-boot-starter-web`, `spring-boot-starter-data-jpa`, `spring-boot-devtools`, ecc., specificando la versione di Spring Boot desiderata.
2. **Aggiunta degli Starter di base**
   - Gli "Starter" di Spring Boot (come `spring-boot-starter-web`, `spring-boot-starter-test`, ecc.) racchiudono in un unico artefatto un insieme di librerie comuni per sviluppare funzionalità specifiche.
   - Per esempio, se desideriamo un'applicazione web REST, basterà aggiungere la dipendenza:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

(in Maven) o, in Gradle:

```groovy
implementation 'org.springframework.boot:spring-boot-starter-web'
```

- Con questa semplice dipendenza, otteniamo già Tomcat (il server integrato) e tutto il necessario per scrivere controller REST.

3. **Configurazione di base**
   - Nel caso di un progetto Maven, all'interno del file `pom.xml` devono inoltre essere presenti le sezioni relative al **plugin Spring Boot** (che si occupa di gestire l'esecuzione con il classico comando `mvn spring-boot:run`).
   - Lo stesso vale per Gradle, dove è necessario applicare il plugin di Spring Boot e definire la relativa versione.

4.3 Struttura delle directory

1. **Pacchetti principali**
   Nella maggior parte dei progetti Spring Boot, la struttura consigliata (sotto la cartella principale `src/main/java/…`) prevede almeno i seguenti pacchetti:
   - **controller**: contiene le classi che espongono le API REST o gestiscono le richieste MVC. Di solito si trovano annotati con `@RestController` o `@Controller`.
   - **service**: contiene la logica di business dell'applicazione, spesso classi annotati con `@Service`. Qui risiede la "parte centrale" dell'elaborazione.
   - **repository**: contiene le interfacce (o le classi) di accesso ai dati, tipicamente annotate con `@Repository` o che estendono le interfacce di `JpaRepository` (se si utilizza Spring Data JPA).

- **model** (o **entity**, a seconda delle convenzioni): contiene le classi che rappresentano i dati dell'applicazione o le entità di database se si utilizza JPA (annotate con `@Entity`).

2. **Organizzazione consigliata delle classi**
   - **Controller**:
     - Se l'applicazione prevede diversi "moduli" funzionali, è buona pratica organizzare i controller secondo la logica dei domini.
     - Le classi dovrebbero essere pulite e focalizzate sulla gestione della richiesta e sulla delega verso il livello di servizio.
   - **Service**:
     - Le classi di servizio dovrebbero contenere la logica applicativa principale.
     - In un progetto strutturato, il service è l'unico punto di accesso alla logica di dominio e "nasconde" i dettagli di implementazione (accesso al database, ecc.) al controller.
   - **Repository**:
     - Di norma, si scrivono interfacce estendendo `CrudRepository`, `JpaRepository`, o altre interfacce di Spring Data.
     - Le query personalizzate vengono definite o con i metodi a convenzione di Spring Data (findBy...), o con annotazioni come `@Query`.
   - **Model/Entity**:
     - Le classi rappresentano gli oggetti principali con cui l'applicazione lavora.
     - Nel caso delle entità JPA, presentano annotazioni come `@Entity`, `@Table` e campi annotati con `@Id`, `@Column`, ecc.
     - In caso di semplici DTO/POJO, possono stare nello stesso package o in un package separato dedicato alle classi di trasporto dati (ad es. `dto`).

In sintesi, questa **organizzazione a livelli** (controller, service, repository, model) è fortemente incoraggiata da Spring perché aiuta a separare le responsabilità, a rendere il codice più modulare e a facilitare la manutenzione e i test.

Conclusione

Seguendo l'ordine del punto 4 del manuale:

- Con **Spring Initializr** (4.1) si creano in modo rapido progetti già configurati.
- In alternativa, con una **creazione manuale** (4.2) si ha più controllo sulle dipendenze e sulla struttura iniziale, ma occorre definire tutto a mano (file di build, directory, ecc.).
- Infine, la **struttura delle directory** (4.3) con pacchetti `controller`, `service`, `repository`, `model` è la base di una buona architettura per applicazioni Spring Boot.

Porting a Spring App

Di seguito troverai:

1. **Una spiegazione dettagliata del punto 5 del menù** (Porting dell'Applicazione da Spring MVC a Spring Boot) e delle sue sottosezioni (5.1 – 5.6).
2. **La definizione delle entità JPA** corrispondenti alle tabelle riportate nel diagramma del database, con le relative relazioni.
3. **Un esempio di configurazione Java** (classe di configurazione) tipica in un progetto Spring Boot.
4. **L'infrastruttura completa (Model, Repository, Service, Controller) d'esempio** riferita alla tabella `Courses`.

---

1. Panoramica del Punto 5: Porting dell'Applicazione da Spring MVC a Spring Boot

5.1 Analisi del progetto esistente

- **Revisione della struttura MVC classica**: in un progetto Spring MVC "tradizionale" spesso troviamo file `web.xml`, configurazioni XML dedicate (ad esempio `dispatcher-servlet.xml`, `spring-security.xml`, ecc.) e un'architettura di cartelle più manuale.
- **Identificazione dei file di configurazione** (XML o Java-based): prima di migrare a Spring Boot, occorre mappare tutti i file di configurazione esistenti (ad esempio i bean definiti in XML, la configurazione di sicurezza, le impostazioni dei datasource, i View Resolver, ecc.).

5.2 Migrazione delle dipendenze

- **Uso degli Starter di Spring Boot**: al posto delle dipendenze singole di Spring MVC (spring-webmvc, spring-jdbc, ecc.), con Spring Boot si ricorre normalmente agli *starter* (`spring-boot-starter-web`, `spring-boot-starter-data-jpa`, `spring-boot-starter-security`, ecc.).
- **Rimozione dei file di configurazione non più necessari**: molte configurazioni, come il `web.xml`, possono essere eliminate perché Spring Boot fornisce un auto-configuratore

interno.

5.3 Aggiornamento del database MySQL

- **Verifica dello schema**: si controlla che lo schema (tabelle, campi, relazioni) sia coerente con le Entity che verranno mappate con JPA/Hibernate.
- **Migrazione delle Entity**: spostamento (o creazione ex-novo) delle classi Entity in base alle tabelle esistenti. Con Spring Boot, si configura la connessione in `application.properties` o `application.yml`, evitando configurazioni XML aggiuntive.

5.4 Riconfigurazione di Spring Security (se presente)

- **Dalla configurazione XML a quella Java-based**: tipicamente si sostituisce il file `spring-security.xml` con una classe annotata con `@EnableWebSecurity` e i relativi metodi di override per definire le regole di accesso, form login, ecc.
- **Integrazione con "Security Starter"**: grazie a `spring-boot-starter-security`, non serve più un file XML dedicato, ma solo la classe di configurazione che estende `WebSecurityConfigurerAdapter` o, nelle versioni recenti, utilizza le nuove API di Spring Security (SecurityFilterChain).

5.5 Gestione di template e risorse statiche

- **Spostare JSP/Thymeleaf (o altre view) in Spring Boot**: in un progetto Maven/Gradle standard, i template si posizionano sotto `src/main/resources/templates` (per Thymeleaf) o `src/main/webapp/WEB-INF/` (per JSP, anche se con Boot è più comune utilizzare Thymleaf o un altro motore).
- **Configurazione del path per risorse statiche**: static resources (CSS, JS, immagini) di default si trovano in `src/main/resources/static`. La risoluzione delle view e delle risorse statiche è in gran parte gestita automaticamente da Spring Boot.

5.6 Verifica finale

- **Test end-to-end**: una volta "portata" l'applicazione su Spring Boot, è fondamentale eseguire test di rotta (controller mapping), test sulle viste e sugli accessi al database, nonché test di sicurezza per verificare che i vecchi controlli e la nuova configurazione coincidano con i requisiti iniziali.

---

2. Definizione delle Entità (JPA/Hibernate) e Relazioni

Di seguito una possibile mappatura delle tabelle riportate nel diagramma in entità Java. Ogni classe è annotata con `@Entity` e riflette la corrispondente tabella del database MySQL. Le relazioni vengono dichiarate con le annotazioni `@OneToMany`, `@ManyToOne`, `@OneToOne`, `@ManyToMany`, a seconda dei casi. Le classi seguenti usano nomi di package fittizi, che potrai cambiare in base al tuo progetto.

2.1 Entity `Register` (utente)

java

```java
package com.example.mycourse.model;

import jakarta.persistence.*;
import java.util.Set;

@Entity
@Table(name = "register")
public class Register {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "username", length = 100, nullable = false)
    private String username;

    @Column(name = "fullname", length = 255)
    private String fullname;

    @Column(name = "password", length = 100, nullable = false)
    private String password;

    @Column(name = "email", length = 100)
    private String email;

    @Column(name = "enabled")
    private boolean enabled;

    // Relazione con i ruoli (molti-a-molti)
    @ManyToMany
```

```
    @JoinTable(
       name = "register_roles",
       joinColumns = @JoinColumn(name = "user_id"),
       inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private Set<Roles> roles;

    // Relazione con Subscription (uno-a-molti)
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private Set<Subscription> subscriptions;

    // Relazione con Courses (uno-a-molti) -> chi crea i corsi
    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private Set<Courses> courses;

    // Getter e Setter ...
    // Costruttori ...
}
```

2.2 Entity `Roles`

java

```java
package com.example.mycourse.model;

import jakarta.persistence.*;
import java.util.Set;

@Entity
@Table(name = "roles")
public class Roles {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "name", length = 50, nullable = false)
    private String name;

    @ManyToMany(mappedBy = "roles")
    private Set<Register> registers;
```

```
    // Getter e Setter ...
    // Costruttori ...
}
```

> Si noti che la tabella di join `register_roles` è esplicitamente mappata con l'annotazione `@JoinTable`. Non è necessario creare un'entità a parte per `register_roles` se vogliamo utilizzare la relazione @ManyToMany in modo diretto.

2.3 Entity `Courses`

java

```java
package com.example.mycourse.model;

import jakarta.persistence.*;
import java.math.BigDecimal;
import java.util.Set;

@Entity
@Table(name = "courses")
public class Courses {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "Title", length = 450)
    private String title;

    @Column(name = "Description", columnDefinition = "LONGTEXT")
    private String description;

    @Column(name = "ImagePath", length = 450)
    private String imagePath;

    @Column(name = "Author", length = 450)
    private String author;

    @Column(name = "Email", length = 450)
    private String email;

    @Column(name = "Rating", precision = 18, scale = 2)
```

```
    private BigDecimal rating;

    @Column(name = "FullPrice_Amount", precision = 18, scale = 1)
    private BigDecimal fullPriceAmount;

    @Column(name = "FullPrice_Currency", length = 45)
    private String fullPriceCurrency;

    @Column(name = "CurrentPrice_Amount", precision = 18, scale = 2)
    private BigDecimal currentPriceAmount;

    @Column(name = "CurrentPrice_Currency", length = 45)
    private String currentPriceCurrency;

    @Column(name = "RowVersion", length = 45)
    private String rowVersion;

    @Column(name = "Status", length = 45)
    private String status;

    // Relazione con Register (molti-a-uno) -> idUser
    @ManyToOne
    @JoinColumn(name = "idUser")
    private Register user;

    // Relazione con Lessons (uno-a-molti)
    @OneToMany(mappedBy = "course", cascade = CascadeType.ALL)
    private Set<Lessons> lessons;

    // Relazione con Subscription (uno-a-molti)
    @OneToMany(mappedBy = "course", cascade = CascadeType.ALL)
    private Set<Subscription> subscriptions;

    // Getter e Setter ...
    // Costruttori ...
}
```

2.4 Entity `Lessons`

java

```java
package com.example.mycourse.model;

import jakarta.persistence.*;
import java.math.BigDecimal;

@Entity
@Table(name = "lessons")
public class Lessons {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "Title", length = 4500)
    private String title;

    @Column(name = "Description", length = 10000)
    private String description;

    @Column(name = "Duration", length = 100)
    private String duration;

    @Column(name = "Order", length = 100)
    private String order;

    @Column(name = "RowVersion", length = 450)
    private String rowVersion;

    @Column(name = "orderLesson", length = 100)
    private String orderLesson;

    // Relazione con Courses (molti-a-uno)
    @ManyToOne
    @JoinColumn(name = "CourseId")
    private Courses course;

    // Getter e Setter ...
    // Costruttori ...
}
```

2.5 Entity `Subscription`

java

```java
package com.example.mycourse.model;

import jakarta.persistence.*;
import java.math.BigDecimal;

@Entity
@Table(name = "subscription")
public class Subscription {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @Column(name = "UserId")
    private Integer userIdFK;
    // A volte si preferisce mappare direttamente l'oggetto user, ma qui seguiamo la colonna se serve.

    @Column(name = "CourseId")
    private Integer courseIdFK;

    @Column(name = "PaymentDate", length = 45)
    private String paymentDate;

    @Column(name = "PaidAmount", precision = 18, scale = 2)
    private BigDecimal paidAmount;

    @Column(name = "PaidCurrency", length = 45)
    private String paidCurrency;

    @Column(name = "PaymentType", length = 450)
    private String paymentType;

    @Column(name = "TransactionId", length = 450)
    private String transactionId;

    @Column(name = "Vote")
    private Integer vote;

    // Relazione con Register (molti-a-uno)
```

```
    @ManyToOne
    @JoinColumn(name = "UserId", insertable = false, updatable = false)
    private Register user;

    // Relazione con Courses (molti-a-uno)
    @ManyToOne
    @JoinColumn(name = "CourseId", insertable = false, updatable = false)
    private Courses course;

    // Getter e Setter ...
    // Costruttori ...
}
```

> Nel caso di `Subscription`, se preferisci, puoi eliminare le colonne `userIdFK` e `courseIdFK` e usare direttamente i campi `user` e `course` con la colonna join, come fatto nelle altre entità.

---

3. Esempio di Configurazione Java (Spring Boot)

Di seguito un esempio minimale di classe di configurazione per JPA in Spring Boot. In genere, è sufficiente inserire le proprietà nel file `application.properties` e lasciare che lo **starter** di Spring Boot configuri automaticamente JPA/Hibernate.

application.properties (esempio)

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/mycourse_db
spring.datasource.username=root
spring.datasource.password=secret
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Classe di configurazione opzionale (se hai bisogno di personalizzazioni particolari):

java

```
package com.example.mycourse.config;

import org.springframework.context.annotation.Configuration;
```

```java
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@Configuration
@EnableJpaRepositories(basePackages = "com.example.mycourse.repository")
public class JpaConfig {
    // Se hai bean specifici da dichiarare, li aggiungerai qui.
}
```

Con Spring Boot, spesso basta annotare la classe principale (`@SpringBootApplication`) ed eventualmente usare `@EnableJpaRepositories`. Tutti i componenti (Repository, Service, ecc.) verranno rilevati automaticamente se si trovano nello stesso package o in package figli.

---

4. Infrastruttura Completa per la Tabella `Courses`

Qui di seguito un esempio di come strutturare Model, Repository, Service e Controller per la gestione delle entità `Courses`. (Le entità l'abbiamo già vista sopra come "model" stesso.)

4.1 Entity (Model)

L'**Entity** l'abbiamo già definita in precedenza con la classe `Courses`. La riporto in forma semplificata:

java

```java
@Entity
@Table(name = "courses")
public class Courses {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String title;
    private String description;
    private String imagePath;
    // ... e tutti gli altri campi

    @ManyToOne
    @JoinColumn(name = "idUser")
```

```
    private Register user;

    // Getter e Setter ...
}
```

4.2 Repository

Un repository per `Courses` che estende `JpaRepository` di Spring Data JPA. In questo modo, si ottengono già metodi CRUD di base.

java

```java
package com.example.mycourse.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import com.example.mycourse.model.Courses;

@Repository
public interface CoursesRepository extends JpaRepository<Courses, Integer> {
    // Metodi custom, se necessari
    // es: List<Courses> findByTitleContaining(String title);
}
```

4.3 Service

È consigliato creare un layer di servizio che incapsuli la logica di business e sfrutti il repository per l'accesso ai dati.

Interfaccia Service:

java

```java
package com.example.mycourse.service;

import com.example.mycourse.model.Courses;
import java.util.List;

public interface CoursesService {
    List<Courses> findAll();
    Courses findById(Integer id);
    Courses save(Courses course);
```

```
    void deleteById(Integer id);
}
```

Implementazione del Service:

java

```java
package com.example.mycourse.service.impl;

import com.example.mycourse.model.Courses;
import com.example.mycourse.repository.CoursesRepository;
import com.example.mycourse.service.CoursesService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;

@Service
public class CoursesServiceImpl implements CoursesService {

    private final CoursesRepository coursesRepository;

    @Autowired
    public CoursesServiceImpl(CoursesRepository coursesRepository) {
        this.coursesRepository = coursesRepository;
    }

    @Override
    public List<Courses> findAll() {
        return coursesRepository.findAll();
    }

    @Override
    public Courses findById(Integer id) {
        Optional<Courses> optional = coursesRepository.findById(id);
        return optional.orElse(null);
    }

    @Override
    public Courses save(Courses course) {
        return coursesRepository.save(course);
    }
```

```
    @Override
    public void deleteById(Integer id) {
        coursesRepository.deleteById(id);
    }
}
```

4.4 Controller

Il **Controller** gestisce le richieste HTTP (ad esempio REST) relative ai corsi. Utilizza il layer Service per compiere le operazioni di CRUD.

java

```java
package com.example.mycourse.controller;

import com.example.mycourse.model.Courses;
import com.example.mycourse.service.CoursesService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/courses")
public class CoursesController {

    private final CoursesService coursesService;

    @Autowired
    public CoursesController(CoursesService coursesService) {
        this.coursesService = coursesService;
    }

    // GET all
    @GetMapping
    public List<Courses> getAllCourses() {
        return coursesService.findAll();
    }

    // GET by ID
```

```
    @GetMapping("/{id}")
    public ResponseEntity<Courses> getCourseById(@PathVariable Integer id) {
        Courses course = coursesService.findById(id);
        if (course == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(course);
    }

    // CREATE
    @PostMapping
    public ResponseEntity<Courses> createCourse(@RequestBody Courses course) {
        Courses saved = coursesService.save(course);
        return ResponseEntity.ok(saved);
    }

    // UPDATE
    @PutMapping("/{id}")
    public ResponseEntity<Courses> updateCourse(@PathVariable Integer id,
@RequestBody Courses updatedCourse) {
        Courses existing = coursesService.findById(id);
        if (existing == null) {
            return ResponseEntity.notFound().build();
        }
        // Aggiorna i campi che vuoi
        existing.setTitle(updatedCourse.getTitle());
        existing.setDescription(updatedCourse.getDescription());
        // ... e così via per gli altri campi ...

        Courses saved = coursesService.save(existing);
        return ResponseEntity.ok(saved);
    }

    // DELETE
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteCourse(@PathVariable Integer id) {
        Courses course = coursesService.findById(id);
        if (course == null) {
            return ResponseEntity.notFound().build();
        }
        coursesService.deleteById(id);
        return ResponseEntity.noContent().build();
```

```
    }
}
```

> In un'applicazione MVC classica (non REST) potresti invece usare `@Controller` e restituire "view name" come `return "coursesList"`, integrando i template Thymeleaf/JSP, e utilizzare i modelli (`Model` o `ModelAndView`). L'approccio qui sopra è tipicamente RESTful.

Conclusioni

1. **Punto 5 del menù** riassume i passaggi chiave per migrare un'applicazione da Spring MVC tradizionale a Spring Boot: analisi della struttura, migrazione di dipendenze e configurazioni, aggiornamento del database, riconfigurazione di Spring Security, passaggio a template e static resources gestiti da Boot, e infine la verifica finale end-to-end.
2. **Le entità** illustrate (Courses, Lessons, Register, Roles, Subscription, ecc.) dimostrano come impostare correttamente le relazioni in JPA/Hibernate, sostituendo file XML con annotazioni.
3. **L'esempio di configurazione** mostra come la maggior parte delle impostazioni di Spring Boot avvenga tramite `application.properties` (o `.yml`), con la possibilità di aggiungere classi `@Configuration` personalizzate.
4. **L'infrastruttura Courses** (Model, Repository, Service, Controller) rappresenta la struttura tipica a strati in Spring Boot:
    - **Model**: Entità JPA che mappa la tabella `courses` .
    - **Repository**: Interfacce che estendono `JpaRepository` , fornendo CRUD e query personalizzate.
    - **Service**: Logica di business, incapsula le operazioni del repository.
    - **Controller**: Espone endpoint REST o gestisce il flusso MVC, collegandosi al servizio.

Utilizzando questi esempi come guida, puoi replicare lo stesso approccio per le altre tabelle (`Lessons` , `Subscription` , `Register` , `Roles`) e completare così l'applicazione MyCourse in ambiente Spring Boot.

Configurazione Spring Boot

Ecco un approfondimento sul punto **6. Configurazione di Spring Boot**, suddiviso nelle tre sezioni che hai indicato:

6.1 *application.properties* e *application.yml*

Differenze e best practice nell'organizzazione dei parametri

In un progetto Spring Boot è possibile utilizzare due formati principali per la configurazione:

1. **application.properties**
2. **application.yml**

La scelta fra `.properties` e `.yml` spesso è una questione di preferenza, ma ci sono alcune differenze:

- **application.properties** è un file di testo con una struttura semplice a coppie *chiave=valore*. Esempio:

  properties

  ```
  server.port=8080
  spring.datasource.url=jdbc:mysql://localhost:3306/db_example
  spring.datasource.username=root
  spring.datasource.password=root
  ```

- **application.yml** usa la notazione YAML (un formato gerarchico), che risulta spesso più leggibile quando si hanno molte proprietà correlate fra loro. Esempio:

  yaml

  ```yaml
  server:
    port: 8080

  spring:
    datasource:
      url: jdbc:mysql://localhost:3306/db_example
  ```

```
    username: root
    password: root
```

Best Practice:

- Organizzare le configurazioni in modo logico. Per esempio, raggruppare le impostazioni relative al database sotto la sezione `spring.datasource`.
- Evitare di lasciare credenziali sensibili in chiaro all'interno dei file di configurazione (ad esempio password), preferendo meccanismi di *external configuration* o *config server* (Spring Cloud Config).
- Utilizzare i *profiling concept* (sezione 6.2) per separare configurazioni differenti (ambiente di sviluppo, test, produzione).

Configurazioni di base: porta, context path, connessione al database

- **Porta**: Spring Boot per default gira sulla porta 8080. Per modificarla, si può impostare la proprietà `server.port`. Ad esempio:

  properties
  ```
  server.port=9090
  ```

  Oppure in YAML:

  yaml
  ```
  server:
    port: 9090
  ```

- **Context path**: Il *context path* è il prefisso che verrà aggiunto a tutte le URL della tua applicazione. Per impostarlo, puoi utilizzare:

  properties
  ```
  server.servlet.context-path=/mio-servizio
  ```

  In YAML:

  yaml
  ```
  server:
    servlet:
  ```

```
    context-path: /mio-servizio
```

- **Connessione al database**: Per configurare un database, tipicamente si impostano queste proprietà:

  properties

  ```
  spring.datasource.url=jdbc:mysql://localhost:3306/db_example
  spring.datasource.username=nome_utente
  spring.datasource.password=la_password
  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
  ```

  Oppure in YAML (come mostrato sopra).

In ambienti reali, è spesso preferibile leggere url, utente e password da variabili d'ambiente per non esporre dati sensibili nel codice sorgente.

6.2 Configurazioni di Profilo

`spring.profiles.active` e gestione di ambienti (*dev*, *test*, *production*)

Spring Boot rende molto semplice la creazione e l'uso di *profili* (o *profiles*) per gestire diverse configurazioni in base all'ambiente.

- `spring.profiles.active` definisce qual è il profilo (o i profili) correntemente attivo.
- Nei file di configurazione, puoi specificare dei *profili* aggiungendo il suffisso `-<profilo>` al nome del file. Ad esempio:
  - `application-dev.properties` o `application-dev.yml`
  - `application-test.properties`
  - `application-prod.properties`

Poi, all'interno di `application.properties` o `application.yml` *principale*, puoi indicare:

properties

```
spring.profiles.active=dev
```

oppure utilizzare variabili d'ambiente o parametri da linea di comando (`--spring.profiles.active=prod`) per attivare il profilo desiderato.

Esempio di file separati:

- `application-dev.yml`

```yaml
server:
  port: 8081
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/db_dev
    username: dev_user
    password: dev_pwd
```

- `application-prod.yml`

```yaml
server:
  port: 8080
spring:
  datasource:
    url: jdbc:mysql://prod-db-server:3306/db_prod
    username: prod_user
    password: secret_pwd
```

Spring Boot, una volta attivato il profilo tramite `spring.profiles.active=prod`, caricherà in automatico anche il file `application-prod.yml` (o `.properties`).

File separati per ogni profilo e come utilizzarli

Suddividere le configurazioni per singolo profilo permette di:

- Mantenere più pulita la configurazione.
- Ridurre errori di "confusione" fra ambienti diversi.
- Eseguire in modo semplice *deployment* in ambienti diversi (basta cambiare una sola variabile d'ambiente per selezionare la configurazione corretta).

È possibile che alcuni parametri siano comuni a tutti i profili (ad esempio un messaggio generico). Questi possono restare nel file `application.properties` (o `.yml`), mentre quelli

specifici di un profilo finiscono nei rispettivi file. In caso di conflitto, ha la precedenza il file del profilo attivo.

6.3 Logging in Spring Boot

Configurazione con Logback/Log4j2

Di default, Spring Boot utilizza **Logback** come implementazione di logging. Tuttavia, è possibile configurare **Log4j2** come alternativa.

- Per **Logback**, basta inserire un file di configurazione `logback-spring.xml` nella directory `src/main/resources`. Esempio di un file minimal:

  ```xml
  <configuration>
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
      <encoder>
        <pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level [%thread] %logger{36} -
  %msg%n</pattern>
      </encoder>
    </appender>

    <root level="INFO">
      <appender-ref ref="CONSOLE"/>
    </root>
  </configuration>
  ```

  Imposta il livello di log e il formato di output.

- Per **Log4j2**, puoi inserire un file `log4j2-spring.xml` e includere le configurazioni equivalenti per gli appender, i logger personalizzati, ecc.

Spring Boot riconosce automaticamente i file di configurazione se sono nominati in modo corretto (ad esempio `logback-spring.xml`, `log4j2-spring.xml`).

Livelli di log e log personalizzati

I livelli di log più comuni sono: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`. In ordine crescente di gravità.

- Puoi configurare i livelli a livello di `root` (globale) o di singolo *package* o classe. Ad esempio in `application.properties`:

  properties

  ```
  logging.level.root=INFO
  logging.level.org.springframework.web=DEBUG
  logging.level.com.mio.progetto=TRACE
  ```

  Questo permette di stampare più (o meno) log a seconda di ciò che ti interessa monitorare.

Puoi anche creare dei logger personalizzati nel codice, ad esempio usando la notazione standard di Spring Boot e Slf4j:

java

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
public class MyController {
    private static final Logger logger =
LoggerFactory.getLogger(MyController.class);

    @GetMapping("/test")
    public String test() {
        logger.info("Esempio di log INFO");
        logger.debug("Esempio di log DEBUG");
        return "ok";
    }
}
```

Conclusione

La configurazione in Spring Boot (e la gestione dei profili) rappresentano uno strumento estremamente potente per adattare l'applicazione a contesti diversi, senza dover cambiare il codice. Combinando correttamente i file `.properties` / `.yml`, i *profili* e un sistema di logging

ben organizzato, sarai in grado di gestire in modo efficace la tua applicazione in qualunque ambiente (sviluppo, test o produzione).

LA PERSISTENZA

7.1 Panoramica di JPA/Hibernate in Spring Boot

- **Autoconfigurazione JPA:** Spring Boot semplifica la configurazione di JPA (Java Persistence API) e Hibernate. Basta aggiungere la dipendenza `spring-boot-starter-data-jpa` nel `pom.xml` (o nel `build.gradle`) perché Spring Boot si occupi di gran parte della configurazione.
- **Definizione delle Entity**: in questa fase si creano le classi che mappano le tabelle del database (le cosiddette "entità"). Attraverso annotazioni come `@Entity`, `@Table`, `@Id`, etc., si definisce come i campi Java corrispondano alle colonne del DB.

7.2 Mappatura delle tabelle e validazione delle Entity

- **Revisione del diagramma MySQL allegato (Courses, Lessons, ecc.)**: si parte dal modello concettuale/relazionale del database (lo schema con tabelle come `Courses`, `Lessons`, ecc.). Lo si analizza per capire le relazioni tra le varie tabelle (uno-a-molti, molti-a-molti, ecc.).
- **Creazione delle classi model e delle relazioni (OneToMany, ManyToMany, ecc.)**: tramite annotazioni JPA/Hibernate (`@OneToMany`, `@ManyToOne`, `@ManyToMany`, `@OneToOne`), si definiscono i legami tra le entità. Questo permette di gestire automaticamente i join e le chiavi esterne senza dover scrivere manualmente query SQL complesse.
- **Validazione delle Entity**: usando le annotazioni di Bean Validation (ad esempio `@NotNull`, `@Size`, `@Email`, ecc.), è possibile abilitare il controllo sui dati in

entrata/uscita. In combinazione con Spring, si può configurare facilmente la validazione per assicurare che le entità rispettino i requisiti di business e di consistenza.

7.3 Repository Layer

- **CrudRepository, JpaRepository**: Spring Data JPA fornisce interfacce pronte all'uso (`CrudRepository`, `JpaRepository`) che semplificano l'implementazione delle operazioni di base (CRUD). Creando un'interfaccia che estende queste repository, si ottengono già i metodi standard (ad esempio `save`, `findById`, `delete`, ecc.).
- **Query methods e query personalizzate**: oltre ai metodi di default, è possibile definire metodi che seguono la convenzione di naming di Spring Data (tipo `findByEmail`, `findByLastNameAndFirstName`, ecc.), i quali generano automaticamente le query necessarie. Se serve più flessibilità, si possono scrivere query personalizzate con l'annotazione `@Query` (JPQL o SQL nativo).
- **Esempi di implementazione con i moduli register, roles, subscription**: vengono mostrati esempi pratici su come creare repository specifiche per gestire moduli quali l'area di registrazione utenti, la gestione dei ruoli, la sottoscrizione ad un servizio, ecc.

7.4 Migrazione e Tooling Database

- **Flyway o Liquibase per la migrazione dello schema (opzionale)**: questi strumenti gestiscono la versione del database nel tempo. Permettono di definire script di migrazione (SQL o in forma Java) per aggiornare in modo incrementale lo schema man mano che il progetto evolve. Spring Boot supporta nativamente sia Flyway sia Liquibase.
- **Test e popolamento dati iniziale**: spesso si include una fase di "seed" dei dati per popolare il DB con valori di test o con dati di base. Flyway e Liquibase offrono la possibilità di eseguire script di inserimento, mentre Spring Boot permette anche di usare file `data.sql` o meccanismi come `data-loader` per facilitare la configurazione iniziale.

In sintesi, il punto 7 descrive l’intero flusso di lavoro legato alla persistenza e all’accesso ai dati in una tipica applicazione Spring Boot: si parte dalla configurazione automatica, si definiscono le entità e le loro relazioni, si implementa il layer di repository per le operazioni CRUD e infine si aggiungono le procedure di migrazione, test e popolamento del database. Questo costituisce la base di un’architettura ben organizzata, modulare e manutenibile per la gestione dei dati.

STRATO WEB

8.1 Creazione dei Controller

1. **Annotazioni tipiche:**
   - **@Controller** vs **@RestController**
     - `@Controller` si usa in genere con applicazioni che restituiscono viste (ad esempio pagine HTML, JSP, Thymeleaf) o che utilizzano i `ModelAndView`.
     - `@RestController` è un'estensione di `@Controller` che combina anche `@ResponseBody`, ideale quando vuoi produrre output JSON o XML direttamente, come in applicazioni REST.
   - **@RequestMapping** (e derivati come `@GetMapping`, `@PostMapping`, ecc.)
     - `@RequestMapping` definisce l'endpoint (URL) a cui il metodo risponde e il tipo di richiesta (GET, POST, PUT, DELETE, ecc.).
     - I più recenti `@GetMapping`, `@PostMapping`, ecc. sono scorciatoie specifiche per ogni metodo HTTP.
2. **Gestione dei parametri e dei path**
   - **Path Variable**: con `@PathVariable` puoi catturare parti del path URL (es. `/utente/{id}`) e passarle come parametri al metodo.
   - **Request Param**: con `@RequestParam` gestisci parametri query (es. `?nome=Mario`).
   - **Request Body**: con `@RequestBody` ricevi dati (spesso in JSON) nel body della richiesta, tipico nelle API REST.

In sostanza, la responsabilità dei controller è rispondere alle chiamate HTTP, delegare al livello di business (service) la logica e restituire una risposta formattata (sia sotto forma di vista che di JSON/XML).

8.2 Service Layer

- **Gestione della logica di business**
  Il Service Layer contiene tutta la logica "centralizzata" dell'applicazione, cioè le operazioni che di solito vanno oltre la semplice gestione dei dati. Qui si decide:
  - Come elaborare le informazioni provenienti dal controller.
  - Come interagire con il data layer (repository, DAO) per leggere/scrivere i dati.
  - Come gestire eventuali transazioni (se necessario).
- **Pattern consigliati di separazione dei layer**
  In un'architettura Spring tipica ci sono almeno tre layer ben definiti:
  1. **Controller** (o Web layer): gestisce le richieste HTTP, le validazioni preliminari e la formattazione della risposta.
  2. **Service** (business layer): contiene la logica di business.
  3. **Repository** (data access layer): interagisce con il database (con JPA, JDBC, ecc.).

  Questo approccio favorisce la **manutenibilità**, la **testabilità** (puoi fare unit test su service e repository in modo indipendente) e la **separazione dei compiti**.

8.3 Gestione delle View

- **Integrazione con Thymeleaf/JSP/altro**
  A seconda delle esigenze, se non stiamo producendo soltanto servizi REST, possiamo usare un engine di template per generare pagine HTML (ad es. **Thymeleaf**).
  - Con Thymeleaf, tipicamente i template si trovano in `src/main/resources/templates`.
  - Un `@Controller` "classico" può restituire stringhe che rappresentano il nome del template Thymeleaf, completando il modello (`Model`) con dati dinamici.
- **Configurazione e path di default**
  - In Spring Boot, di default, Thymeleaf cerca i file HTML in `classpath:/templates/`.

- Per JSP, spesso si configura `prefix` e `suffix` nel `application.properties` o `application.yml` (ad es. `spring.mvc.view.prefix=/WEB-INF/views/` e `spring.mvc.view.suffix=.jsp`).

In pratica, il controller passa il `Model` con i dati alla vista, che li visualizza in HTML, JSP, ecc.

8.4 Validazione e gestione degli errori

1. **Bean Validation** (con `@Valid`, `@NotNull`, ecc.)
   - Spring integra la **Java Bean Validation** (Hibernate Validator è l'implementazione più comune).
   - Inserendo annotazioni come `@NotNull`, `@Size`, `@Email`, ecc. sulle classi di modello o DTO, puoi far sì che Spring verifichi automaticamente tali vincoli.
   - Nel controller, usando `@Valid` prima del parametro (es. `public String salva(@Valid @ModelAttribute UserForm form, ... )`), viene eseguita la validazione. In caso di errori, vengono passati a un `BindingResult` o generata un'eccezione.
2. **Gestione delle eccezioni con `@ControllerAdvice`**
   - `@ControllerAdvice` permette di definire a livello globale la gestione di determinati errori o eccezioni che possono verificarsi in un qualsiasi controller.
   - Con `@ExceptionHandler` all'interno di una classe annotata con `@ControllerAdvice`, puoi gestire eccezioni specifiche (ad es. `NullPointerException`, `DataAccessException`, ecc.) e restituire la risposta HTTP adeguata (un codice di errore, un messaggio JSON, oppure una pagina di errore dedicata).

Grazie a questo meccanismo, l'applicazione risulta coerente nella gestione degli errori, evitando di dover replicare la stessa logica in ogni singolo controller.

In sintesi

- **Controller**: si occupa di “parlare” con l’esterno (HTTP). Contiene metodi mappati alle URL, gestisce parametri e path, valida le richieste, e richiama il **Service**.
- **Service**: racchiude la logica di business e fa da intermediario con il **Repository** (o DAO).
- **View**: se l’applicazione non è solo REST ma ha pagine HTML, si usa un engine come Thymeleaf/JSP; la parte “visuale” è nel livello delle viste.
- **Validazione & Eccezioni**: annotate i vostri DTO/Model per la validazione automatica e sfruttate `@ControllerAdvice` per gestire in modo centralizzato gli errori.

Questo è il cuore del “Punto 8” – l’architettura tipica di un’applicazione web in Spring.

Sicurezza in Spring Boot

9.1 Spring Security con Spring Boot

Dipendenze di base

Per abilitare Spring Security in un progetto Spring Boot, in genere è sufficiente aggiungere la dipendenza:

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
  <version>${spring-boot.version}</version>
</dependency>
```

Questa dipendenza:

- Fornisce le classi e le configurazioni di base per proteggere l'applicazione.
- Aggiunge di default un meccanismo di login (form di autenticazione) e un utente predefinito (in memoria) se non si specifica nulla.

Creazione di una configurazione di sicurezza personalizzata

Con Spring Boot 3 (e Spring Security 6), l'approccio consigliato prevede la definizione di un **SecurityFilterChain**. Un esempio minimale:

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
```

```
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
        // Configurazione base
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll()    // Endpoint pubblici
                .anyRequest().authenticated()                 // Tutto il resto
richiede autenticazione
            )
            .formLogin(Customizer.withDefaults()) // Form di login standard
            .httpBasic(Customizer.withDefaults()); // Abilita Basic Auth (se
desiderato)

        return http.build();
    }
}
```

Punti fondamentali:

1. **authorizeHttpRequests(...)**: definisce le regole di accesso alle risorse.
2. **formLogin(...)**: abilita la form di login generata da Spring Security.
3. **httpBasic(...)**: abilita l'autenticazione Basic (utile soprattutto per API o test).

È possibile arricchire questa configurazione aggiungendo filtri personalizzati, configurando logout, CSRF, cors, ecc.

9.2 Integrazione con Database

Struttura del database

Per gestire utenti e ruoli dal database, si creano tipicamente due (o più) tabelle, ad esempio:

- **users** (o *register*): contiene i dati degli utenti (username, password, ecc.).
- **roles**: definisce i ruoli disponibili (es. ROLE_ADMIN, ROLE_USER).
- **users_roles** (o *register_roles*): tabella di relazione N:N tra utenti e ruoli.

Un esempio di schema (MySQL) potrebbe essere:

sql

```sql
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL UNIQUE,
    password VARCHAR(100) NOT NULL,
    enabled BOOLEAN NOT NULL DEFAULT TRUE
);

CREATE TABLE roles (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE users_roles (
    user_id BIGINT NOT NULL,
    role_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(id),
    FOREIGN KEY (role_id) REFERENCES roles(id),
    PRIMARY KEY (user_id, role_id)
);
```

(I nomi delle tabelle possono ovviamente variare: `register`, `roles`, `register_roles` come specificato.)

Persistenza delle credenziali e password encoding

1. **Password Encoding**: è importante **non** memorizzare mai la password in chiaro. Spring Security fornisce encoder come il `BCryptPasswordEncoder`.
2. **Creazione di un UserDetailsService personalizzato**:
   - È una classe che implementa l'interfaccia `UserDetailsService`.
   - Recupera l'utente dal database (ad es. tramite repository JPA) e restituisce un `UserDetails` con username, password e ruoli.

Esempio semplificato di `UserDetailsService`:

java

```java
@Service
public class CustomUserDetailsService implements UserDetailsService {
```

```
    @Autowired
    private UserRepository userRepository; // Interfaccia JPA

    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
        UserEntity userEntity = userRepository.findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException("Utente non trovato"));

        // Conversione in UserDetails
        return new org.springframework.security.core.userdetails.User(
                userEntity.getUsername(),
                userEntity.getPassword(),
                // Conversione dei ruoli in GrantedAuthority
                userEntity.getRoles().stream()
                    .map(role -> new SimpleGrantedAuthority(role.getName()))
                    .collect(Collectors.toList())
        );
    }
}
```

Dopodiché, nella `SecurityConfig`:

java

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public UserDetailsService userDetailsService() {
        return new CustomUserDetailsService();
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        // BCryptEncoder o un altro
        return new BCryptPasswordEncoder();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
```

```
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
            .formLogin(Customizer.withDefaults())
            .userDetailsService(userDetailsService()); // Indichiamo il nostro
userDetailsService

        return http.build();
    }
}
```

9.3 Gestione delle sessioni e Remember-Me

Sessioni in ambiente distribuito/cluster

Se l'applicazione gira in cluster (ad esempio su più istanze dietro un load balancer), bisogna gestire **sessioni condivise** (es. tramite Redis, database, o sessioni stateless con token).

- **Sessioni sticky**: se il load balancer permette di "fissare" la sessione a un nodo (sticky sessions), la configurazione può rimanere standard.
- **Spring Session**: se si sceglie di mantenere lo stato della sessione in un archivio centralizzato (ad es. Redis), si può integrare la dipendenza `spring-session-data-redis`.

Esempio di configurazione minima in `application.properties`:

properties

```
spring.session.store-type=redis
```

Ciò permette di condividere le sessioni tra più nodi.

Remember-Me

Il "Remember-Me" consente all'utente di rimanere loggato oltre la singola sessione, tramite un **cookie persistente**.

Per abilitarlo:

java

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated()
        )
        .formLogin(Customizer.withDefaults())
        .rememberMe(rm -> rm
            .tokenValiditySeconds(14 * 24 * 60 * 60)  // 14 giorni, ad esempio
            .key("chiaveSegreta")                     // chiave segreta per generare
token
        );

    return http.build();
}
```

Solitamente si associa il “remember-me” a un token salvato in database o in un cookie crittato.

(Opzionale) Integrazione con token JWT

Un approccio più moderno (spesso usato in applicazioni REST stateless) è **non** memorizzare la sessione sul server, ma restituire un **JWT** (JSON Web Token) al client, che lo include nelle richieste (ad es. in un header Authorization).

- Serve una libreria per generare/validare i token JWT (es. `io.jsonwebtoken`).
- L’`HttpSecurity` andrà configurato per essere **stateless**, disabilitando la sessione:

  java

  ```
  http.sessionManagement(session ->
  session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
  ```

9.4 Protezione degli endpoint

Configurazione di HttpSecurity, filtri e regole di accesso

In Spring Security 6, la configurazione classica per proteggere gli endpoint passa attraverso `HttpSecurity`:

1. **Regole di autorizzazione** (chi può accedere a cosa):

```java
http.authorizeHttpRequests(auth -> auth
    .requestMatchers("/public/**").permitAll()
    .requestMatchers("/admin/**").hasRole("ADMIN")
    .requestMatchers("/user/**").hasRole("USER")
    .anyRequest().authenticated()
);
```

   - `permitAll()` permette l'accesso a risorse pubbliche senza login.
   - `hasRole("ADMIN")` richiede che l'utente abbia il ruolo `ROLE_ADMIN`.
   - `anyRequest().authenticated()` è la "chiusura" che protegge tutte le rotte non specificate.

2. **Filtri**:
   - Spring Security include filtri di default (es. per l'autenticazione formLogin, Basic Auth, gestione delle eccezioni).
   - È possibile aggiungere filtri custom implementando `OncePerRequestFilter` o altri e inserirli nella catena (`http.addFilterBefore(...)`).

Esempi di ruoli e permessi (admin, user)

- **admin**: tipicamente ha accesso completo a tutte le operazioni critiche (CRUD su risorse sensibili, gestione di altri utenti, ecc.).
- **user**: può accedere solo alle funzionalità di base (ad es. visualizzare e modificare i propri dati).

Nella pratica, i ruoli vengono salvati nel database (es. `roles`), l'utente li acquisisce attraverso la relazione `users_roles` e Spring Security li carica come `GrantedAuthority` durante l'autenticazione.

Conclusioni e best practice

1. **Sicurezza a più livelli**: oltre a Spring Security, conviene curare anche la sicurezza di rete, la protezione dagli attacchi CSRF (se l'applicazione usa form e sessione), la validazione degli input, ecc.
2. **Cifratura delle password**: sempre utilizzare un `PasswordEncoder` (ad es. `BCrypt`) per salvare le password in modo sicuro.
3. **Ruoli vs. permessi**: in progetti più complessi, si preferisce spesso una gestione a "permessi" (authority) anziché a singoli ruoli "macro" (admin/user). Questo permette maggiore granularità.
4. **Log e monitoraggio**: tenere traccia degli accessi, tentativi di login falliti, e altri eventi di sicurezza è fondamentale.
5. **Distribuzione/cluster**: se l'app cresce, pensare a soluzioni di session management centralizzate (Spring Session) o passare all'autenticazione stateless con token (JWT o altri) per scalare più facilmente.

Sviluppando così il "punto 9", si ottiene una panoramica sulle principali best practice e funzionalità offerte da **Spring Security**, fornendo sia uno scheletro di configurazione sia concetti chiave (user details, password encoding, sessioni distribuite, ruoli e permessi).

Testing Applicazione Spring Boot

---

10. Testing dell'Applicazione

10.1 Testing Unitario e d'Integrazione

10.1.1 JUnit e Spring Boot Test

- **JUnit 5** (Jupiter) è il framework di test più diffuso in ambiente Java. Si integra alla perfezione con Spring Boot grazie alla dipendenza `spring-boot-starter-test`, che include:
  - **JUnit 5** per i test unitari
  - **Spring Test & Spring Boot Test** per l'integrazione con il contesto di Spring
  - **Mockito** per il mocking
  - **AssertJ/Hamcrest** per le asserzioni fluide
- L'annotazione chiave per i test di integrazione in Spring Boot è `@SpringBootTest`, che:
  1. Avvia l'intero **ApplicationContext** di Spring.
  2. Consente di iniettare bean, repository e servizi reali all'interno del test.
  3. Può essere personalizzata (ad es. limitando la scansione dei componenti con `classes = ...` o disabilitando alcune feature).

java

```
@SpringBootTest
class MyIntegrationTest {

    @Autowired
```

```
    private MyService myService;

    @Test
    void testIntegrationScenario() {
        // Testa il comportamento dell'intera applicazione
        // ad esempio chiamando un metodo del servizio
        // e verificando la logica end-to-end
        String result = myService.businessLogic("input");
        assertEquals("expected", result);
    }
}
```

10.1.2 MockMVC per testare i Controller

- **MockMVC** permette di simulare richieste HTTP ai tuoi **Controller** senza avviare un server reale (come Tomcat o Jetty).
- Per abilitare MockMVC, puoi usare:
  - `@AutoConfigureMockMvc` in combinazione con `@SpringBootTest`
  - Oppure creare uno slice test per il Web layer con `@WebMvcTest(controllers = ...)`
- Esempio di test con **MockMVC**:

java

```
@SpringBootTest
@AutoConfigureMockMvc
class MyControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void testGetEndpoint() throws Exception {
        mockMvc.perform(get("/api/myresource"))
               .andExpect(status().isOk())
               .andExpect(content().string("Hello World"));
    }
}
```

- In questo modo verifichi che il mapping dei controller, la serializzazione/deserializzazione JSON e i flussi di richieste HTTP funzionino come previsto, senza dover alzare davvero un server esterno.

10.2 Test del Database

10.2.1 @DataJpaTest e test con database in-memory (H2)

- Quando occorre testare la **persistenza** (repository JPA, query personalizzate, mapping con le entità), puoi utilizzare l'annotazione `@DataJpaTest`.
- `@DataJpaTest`:
  1. Carica soltanto i componenti relativi a JPA nel contesto (EntityManager, Repository, ecc.).
  2. Utilizza di default un database in-memory **H2** (a meno che non ne configuri un altro).
  3. Effettua il **rollback** automatico delle transazioni dopo ciascun test, per garantire l'isolamento.

java

```java
@DataJpaTest
class MyRepositoryTest {

    @Autowired
    private MyRepository myRepository;

    @Test
    void testFindByName() {
        // Arrange
        MyEntity entity = new MyEntity();
        entity.setName("TestName");
        myRepository.save(entity);

        // Act
        Optional<MyEntity> found = myRepository.findByName("TestName");
```

```
        // Assert
        assertTrue(found.isPresent());
        assertEquals("TestName", found.get().getName());
    }
}
```

- Usare H2 rende il test molto veloce e isolato dal database reale di produzione. Puoi comunque specificare un diverso DB di test nelle **application.properties** di test, se vuoi.

10.2.2 Esecuzione di query di integrazione

- Se hai query SQL custom o procedure complesse, oltre ai test unitari, potresti voler fare **test di integrazione** più ampi, che:
  - Avviano l'intera applicazione (`@SpringBootTest`)
  - Utilizzano un database in-memory o esterno (anche tramite Docker / Testcontainers)
  - Verificano l'intero percorso: salvataggio, recupero, elaborazione dei dati a livello di repository/servizio/controllore
- In questi casi, puoi combinare `@SpringBootTest` e le funzionalità di JPA:

java

```java
@SpringBootTest
class MyFullIntegrationTest {

    @Autowired
    private MyService myService;

    @Autowired
    private MyRepository myRepository;

    @Test
    void testComplexDatabaseScenario() {
        // Esempio di test che salva un'entità, invoca il servizio
        // e infine verifica il risultato finale su DB
    }
}
```

10.3 Best Practice per i test in Spring Boot

10.3.1 Strutturazione dei package di test

- **Convenzione Maven/Gradle**: i test si trovano solitamente nella cartella `src/test/java`.
  - La struttura dei package dovrebbe rispecchiare quella dei sorgenti principali, es.:
    - `com.example.myapp.controller` → test in `com.example.myapp.controller` (all'interno di `src/test/java`)
    - `com.example.myapp.service` → test in `com.example.myapp.service`, e così via.
- Nominare le classi di test con suffisso `Test` (p.es. `UserControllerTest`).
- Se hai **molti test** e vuoi ordine, potresti creare sotto-pacchetti come `repository`, `integration`, `unit` ecc., a seconda della complessità del tuo progetto.

10.3.2 Test containers e test su environment Docker (opzionale)

- Se il tuo progetto necessita di test di integrazione più **realistici** (ad esempio su un DB reale come PostgreSQL, MySQL, MongoDB, ecc.), puoi usare **Testcontainers**.
  - **Testcontainers** utilizza container Docker "usa e getta" per avviare e testare servizi esterni (database, broker di messaggi, ecc.) durante il ciclo di test.
  - In questo modo, ogni test parte da un ambiente pulito e automatizzato, senza dover configurare manualmente un database locale.
- Esempio di configurazione **Testcontainers** con PostgreSQL:

java

```java
@SpringBootTest
@Testcontainers
class MyPostgresIntegrationTest {

    @Container
    static PostgreSQLContainer<?> postgres = new PostgreSQLContainer<>
("postgres:15.1")
        .withDatabaseName("testdb")
        .withUsername("testuser")
        .withPassword("testpwd");

    @DynamicPropertySource
    static void overrideProps(DynamicPropertyRegistry registry) {
```

```
        registry.add("spring.datasource.url", postgres::getJdbcUrl);
        registry.add("spring.datasource.username", postgres::getUsername);
        registry.add("spring.datasource.password", postgres::getPassword);
    }

    @Test
    void testRepositoryWithRealPostgres() {
        // Qui puoi avviare l'applicazione, iniettare i repository e testare query
reali.
    }
}
```

- Con Testcontainers:
  1. Avvii il container Docker in modo trasparente ai test.
  2. Terminata l'esecuzione, il container viene chiuso, evitando di lasciare risorse in esecuzione.
  3. Assicuri che il tuo codice funzioni **esattamente** come farebbe in produzione su un database vero.

Riepilogo

1. **Test unitari**: concentrati su singole classi e metodi (servizi, metodi di utilità, ecc.) usando JUnit e Mockito (per simulare dipendenze esterne).
2. **Test di integrazione**: sfrutta `@SpringBootTest` per caricare l'intera applicazione o slice test (come `@WebMvcTest` per i controller) e validare la corretta interazione tra i componenti Spring.
3. **Test del database**:
   - Per testare esclusivamente la parte JPA/Repository, `@DataJpaTest` con H2 è la strada più semplice.
   - Se vuoi un test di integrazione "end-to-end" con un DB reale, puoi optare per Testcontainers (o un DB di test configurato ad hoc).
4. **Best practice**:

- Mantieni la stessa struttura dei package tra `src/main` e `src/test`.
- Dai nomi chiari e coerenti ai test (`MyServiceTest`, `UserRepositoryTest`, etc.).
- Se servono risorse esterne reali (database, broker) in modo automatizzato, usa Docker + Testcontainers.

Con questa suddivisione e spiegazione, dovresti avere una panoramica di come organizzare i test della tua applicazione Spring Boot: dai test più semplici (unitari) a quelli più complessi (integrazione con DB e container).

DEPLOYMENT E DISTRIBUZIONE

11.1 Creazione del file Jar o War

Differenze e vantaggi tra i due formati

1. **Jar (Java ARchive)**
    - Viene usato principalmente per applicazioni standalone, ovvero eseguibili con il comando `java -jar nomefile.jar`.
    - Contiene di solito il codice compilato (class), le librerie dipendenti e i file di configurazione necessari all'applicazione per girare autonomamente.
    - È pensato per applicazioni che non si appoggiano a un application server (come Tomcat, Jetty o altri). Se l'applicazione include un embedded server (ad esempio Spring Boot con Tomcat embedded), allora può essere lanciata come servizio web senza bisogno di installare un server esterno.
2. **War (Web ARchive)**
    - Viene usato per distribuire applicazioni web su un application server (Tomcat, Jetty, WildFly, GlassFish, ecc.).
    - È sostanzialmente uno zip che contiene, oltre alle classi e alle librerie, anche la struttura delle cartelle tipiche delle web-app Java (WEB-INF, file di configurazione come web.xml, ecc.).
    - Il vantaggio è che può essere caricato ("deployato") su qualsiasi server compatibile con lo standard Java EE (o Jakarta EE), rendendo il passaggio da un server a un altro relativamente semplice (portabilità).
    - Di contro, se si vuole avviare l'applicazione senza un server esterno, un war non è di per sé eseguibile direttamente con `java -jar` (a meno che non si usino framework

o plugin che includano un server embedded all'interno del War).

Configurazioni di packaging in Maven/Gradle

- **Maven**
  - Nel file `pom.xml` si può specificare, all'interno della sezione `<packaging>`, il tipo di archivio desiderato: `<packaging>jar</packaging>` o `<packaging>war</packaging>`.
  - Per esempio, in un progetto Spring Boot, spesso si crea un eseguibile jar con Tomcat embedded. Se invece si desidera un War da caricare su un Tomcat esterno, si cambia il packaging e si adattano le dipendenze.
  - Il comando tipico per generare gli artefatti è `mvn clean package` oppure `mvn clean install`.
- **Gradle**
  - Nel file `build.gradle` si indica il plugin corrispondente:

    ```groovy
    apply plugin: 'java'
    apply plugin: 'war' // se si vuole produrre un file .war
    ```

  - Oppure si usa l'applicazione come Jar (spesso con plugin `application` o `java`).
  - Come in Maven, basta poi lanciare `gradle build` (o il task specifico) per generare l'artefatto.

11.2 Esecuzione stand-alone su diversi sistemi

Lancio su Windows, macOS, Linux con `java -jar`

- Se abbiamo un file Jar **eseguibile** (cioè contiene un `Main-Class` nel Manifest o un bootstraper interno come in Spring Boot), possiamo lanciarlo con:

  ```bash
  java -jar nomefile.jar
  ```

- Funziona allo stesso modo su tutti i sistemi operativi dove sia installata la JVM (Windows, macOS, Linux).
- Per farlo funzionare, è spesso necessario che la versione di Java installata localmente sia compatibile con quella usata per compilare l'applicazione.

Configurazione di variabili d'ambiente e parametri di avvio

1. **Variabili d'ambiente**
   - Possono essere utilizzate per configurare le impostazioni di runtime, come ad esempio la porta dell'applicazione, le credenziali del database, percorsi di log, ecc.
   - In Windows si impostano dal pannello di controllo o da prompt, in Linux/macOS si possono impostare a riga di comando (ad esempio `export NOME_VARIABILE=valore`) prima di lanciare l'applicazione.
2. **Parametri di avvio**
   - Spesso si usa la sintassi `java -jar nomefile.jar --parametro=valore` (se l'applicazione prevede un parsing di questi parametri).
   - È utile per passare configurazioni senza dover ri-compilare il Jar o alterare l'ambiente.

11.3 Deployment su server esterno (Tomcat, Jetty, ecc.)

Configurazione e adeguamento (se War)

- Se vogliamo distribuire un War su Tomcat, Jetty o altri server, dobbiamo:
  1. Assicurarci che la struttura del War sia corretta (`WEB-INF`, `META-INF`, eventuale `web.xml` se non si usano approcci "configurazione by code").
  2. Controllare le dipendenze: eventuali librerie fornite di solito dal container (ad es. servlet API) non dovrebbero essere incluse in conflitto.
  3. Definire correttamente le configurazioni di contesto nel server (ad es. un file `context.xml` in Tomcat, se necessario).

Esempio di deployment su Tomcat remoto

- Tomcat consente il deployment di un War in diversi modi:

1. **Copia manuale** del file War nella cartella `webapps` di Tomcat (ad esempio via FTP, scp, o attraverso un filesystem condiviso). Al riavvio (o all'hot deploy) Tomcat riconosce il War e lo decomprime in una cartella con lo stesso nome.
2. **Manager web interface**: se attivata, si può caricare il War dall'interfaccia web di Tomcat.
3. **Script/CICD**: tramite Maven plugin o altri script di automazione (ad es. Jenkins, GitLab CI), si può configurare il deploy remoto:

   ```xml
   <plugin>
     <groupId>org.apache.tomcat.maven</groupId>
     <artifactId>tomcat7-maven-plugin</artifactId>
     <version>2.2</version>
     <configuration>
       <url>http://indirizzo-server:8080/manager/text</url>
       <server>TomcatServer</server>
       <path>/nomeApp</path>
     </configuration>
   </plugin>
   ```

   In questo modo si "spinge" il War su Tomcat remoto direttamente da Maven.

- Una volta distribuita l'applicazione, Tomcat la renderà accessibile all'indirizzo configurato, ad esempio `http://indirizzo-server:8080/nomeApp`.

11.4 Docker e Containerizzazione

Creazione di una Dockerfile base

- Se vogliamo containerizzare un'applicazione Java (Jar o War), tipicamente creiamo un file `Dockerfile` simile a questo:

  ```dockerfile
  # Usare un'immagine base con Java (ad esempio OpenJDK 17)
  FROM openjdk:17-jdk-alpine

  # Creiamo una directory per l'app nel container
  ```

```
WORKDIR /app

# Copiamo il Jar (o War) all'interno del container
COPY target/nomefile.jar /app/nomefile.jar

# Esporre la porta (se ad esempio l'app gira sulla 8080)
EXPOSE 8080

# Comando di avvio
CMD ["java", "-jar", "nomefile.jar"]
```

- Costruendo l'immagine con `docker build -t nome-immagine .`, si ottiene un'immagine Docker contenente l'applicazione pronta a essere eseguita.

Docker Compose per integrare DB e applicazione

- Spesso l'app ha bisogno di un database (MySQL, PostgreSQL, ecc.). Per avviare facilmente più container insieme, si usa **Docker Compose**.
- Esempio di `docker-compose.yml` che avvia un container con la nostra applicazione e uno con MySQL:

yaml

```yaml
version: '3.8'

services:
  db:
    image: mysql:8
    container_name: my_db
    environment:
      MYSQL_ROOT_PASSWORD: root
      MYSQL_DATABASE: myappdb
      MYSQL_USER: myuser
      MYSQL_PASSWORD: mypass
    ports:
      - "3306:3306"
    networks:
      - my_network

  app:
    image: nome-immagine:latest
    container_name: my_app
```

```
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:mysql://db:3306/myappdb
      - SPRING_DATASOURCE_USERNAME=myuser
      - SPRING_DATASOURCE_PASSWORD=mypass
    depends_on:
      - db
    networks:
      - my_network

networks:
  my_network:
    driver: bridge
```

- In questo modo, con un solo comando `docker-compose up -d`, si avvia sia il database che l'applicazione, e l'app "vede" il database tramite il DNS interno di Docker (`db`).

Conclusioni e Best Practice

- **Jar vs War**: Usa Jar con server embedded (Spring Boot) se vuoi un eseguibile portabile, semplice da lanciare. Usa War se devi integrare con un server esistente o in un contesto Java EE classico.
- **Gestione configurazioni**: Mantieni le configurazioni fuori dall'artefatto finale, preferibilmente con variabili d'ambiente o file di configurazione separati.
- **Deployment**:
  - Se l'applicazione è in Spring Boot, lanciare `java -jar` o containerizzare può essere la strada più rapida.
  - Se ci sono più applicazioni su uno stesso server, un War su Tomcat/Jetty può essere preferibile per una gestione centralizzata.
- **Containerizzazione**:
  - Docker semplifica la gestione delle dipendenze e l'isolamento.
  - Docker Compose integra facilmente più servizi (DB, code server, cache, ecc.).

- In produzione, potresti valutare orchestratori come Kubernetes per scalare e gestire i container in maniera più sofisticata.

In sintesi, il punto 11 copre l'intero flusso di come passare dal codice sorgente a un artefatto distribuibile (Jar o War), come eseguirlo o deployarlo, e infine come gestire tutto con Docker. Sperabilmente queste note ti aiutano a organizzare e approfondire ogni sotto-punto, passando da un semplice file Jar stand-alone a una soluzione containerizzata e pronta per ambienti di produzione.

Spring Boot Manutenzione e Aggiornamenti

12.1 Manutenzione e aggiornamento di Spring Boot

- **Gestione delle versioni e del supporto a lungo termine**
  Spring Boot mette a disposizione un modello di versioning che segue da vicino le versioni di Spring Framework. Esistono versioni di **supporto a lungo termine (LTS)** che ricevono aggiornamenti di sicurezza e bugfix per un periodo prolungato, oltre alle versioni "di punta" con nuove funzionalità ma con un ciclo di vita più breve. È importante scegliere la versione in base alle esigenze di stabilità o di adozione delle ultime novità.
- **Consigli su come rimanere aggiornati**
  1. **Monitorare il blog ufficiale** di Spring (spring.io/blog) per annunci di nuove release e note di rilascio.
  2. **Iscriversi alle newsletter** o ai feed RSS di Spring per ricevere aggiornamenti su eventi, webinar e annunci.
  3. **Partecipare alla community** (forum, Stack Overflow, GitHub) per segnalazioni di bug, soluzioni alternative e suggerimenti.
  4. **Aggiornare con regolarità** la versione di Spring Boot e dipendenze correlate, testando accuratamente eventuali modifiche in ambienti di staging.

12.2 Monitoraggio e Logging avanzato

- **Actuator di Spring Boot**
  Spring Boot Actuator fornisce endpoint utili per monitorare lo stato dell'applicazione:

`/actuator/health`, `/actuator/info`, `/actuator/metrics`, ecc. È uno strumento indispensabile per tenere sotto controllo le performance, il carico e la salute complessiva del sistema.

  - È possibile **personalizzare** quali endpoint vengono esposti e con quali permessi di accesso (ad esempio, abilitando la sicurezza su endpoint sensibili).

- **Metriche e health checks**
  In combinazione con Actuator, librerie come **Micrometer** consentono di raccogliere metriche dettagliate e inviarle a piattaforme di monitoring (Prometheus, Grafana, Datadog, ecc.). Implementare health checks personalizzati permette di rendere più granulare il monitoraggio, ad esempio per verificare la connettività a un database o un servizio esterno.

- **Logging centralizzato**
  In scenari più complessi, conviene affidarsi a soluzioni di **log centralizzati** (Elastic Stack, Splunk, Graylog) che semplificano la ricerca e l'analisi di grandi quantità di log provenienti da più istanze dell'applicazione.

12.3 Evoluzioni possibili dell'applicazione

- **Microservizi e Cloud (Breve cenno a Spring Cloud)**
  Un'evoluzione naturale per molte applicazioni Spring Boot è quella verso un'architettura a **microservizi**. In questo contesto, **Spring Cloud** fornisce una serie di moduli (Netflix OSS, Config Server, Eureka, Ribbon, Feign, ecc.) per affrontare sfide tipiche dell'architettura distribuita: service discovery, load balancing, configurazione centralizzata, circuit breaker, e così via.
  Se l'applicazione viene distribuita in ambiente Cloud (ad esempio su AWS, Azure o Google Cloud), Spring Cloud semplifica l'integrazione con i servizi nativi della piattaforma (messaggistica, storage, monitoring, ecc.).

- **Scalabilità e architettura distribuita**
  Quando il carico aumenta, uno dei vantaggi principali di Spring Boot è la possibilità di **scalare orizzontalmente** il numero di istanze (ad esempio, in un cluster Kubernetes). L'architettura a microservizi, affiancata da metodologie DevOps (CI/CD, containerizzazione, orchestrazione), permette di gestire e mantenere più facilmente progetti di grandi dimensioni.

12.4 Appendici

- **Troubleshooting comuni**
  - **Port binding**: errori nella configurazione della porta HTTP/HTTPS. Verificare le proprietà `server.port` e la presenza di conflitti con altri processi.
  - **Problemi di dipendenze**: quando i jar confliggono (ad esempio versioni diverse di librerie comuni). Utilizzare lo strumento `mvn dependency:tree` (o Gradle equivalent) per individuare le versioni importate.
  - **Timeout e prestazioni**: controllare le impostazioni di pool di connessione e i parametri di timeout (Tomcat, DataSource, ecc.).
  - **Errori di configurazione**: ricordarsi di definire correttamente i profili (`spring.profiles.active`) e i file di configurazione (`application.properties` o `application.yml`).
- **Risorse utili (documentazione ufficiale, community, plugin)**
  - Documentazione Ufficiale di Spring Boot
  - Spring Initializr per generare progetti di base configurati
  - Forum e Community Spring
  - **Plugin e IDE**: Eclipse, IntelliJ IDEA e Visual Studio Code offrono plugin dedicati per semplificare lo sviluppo e la configurazione di progetti Spring.

Link al codice su GitHub

- Esempio di applicazione Spring MVC “classica”: https://github.com/malbasini/Spring-MVC
- Esempio di applicazione Spring Boot: https://github.com/malbasini/spring-boot