Java SQL

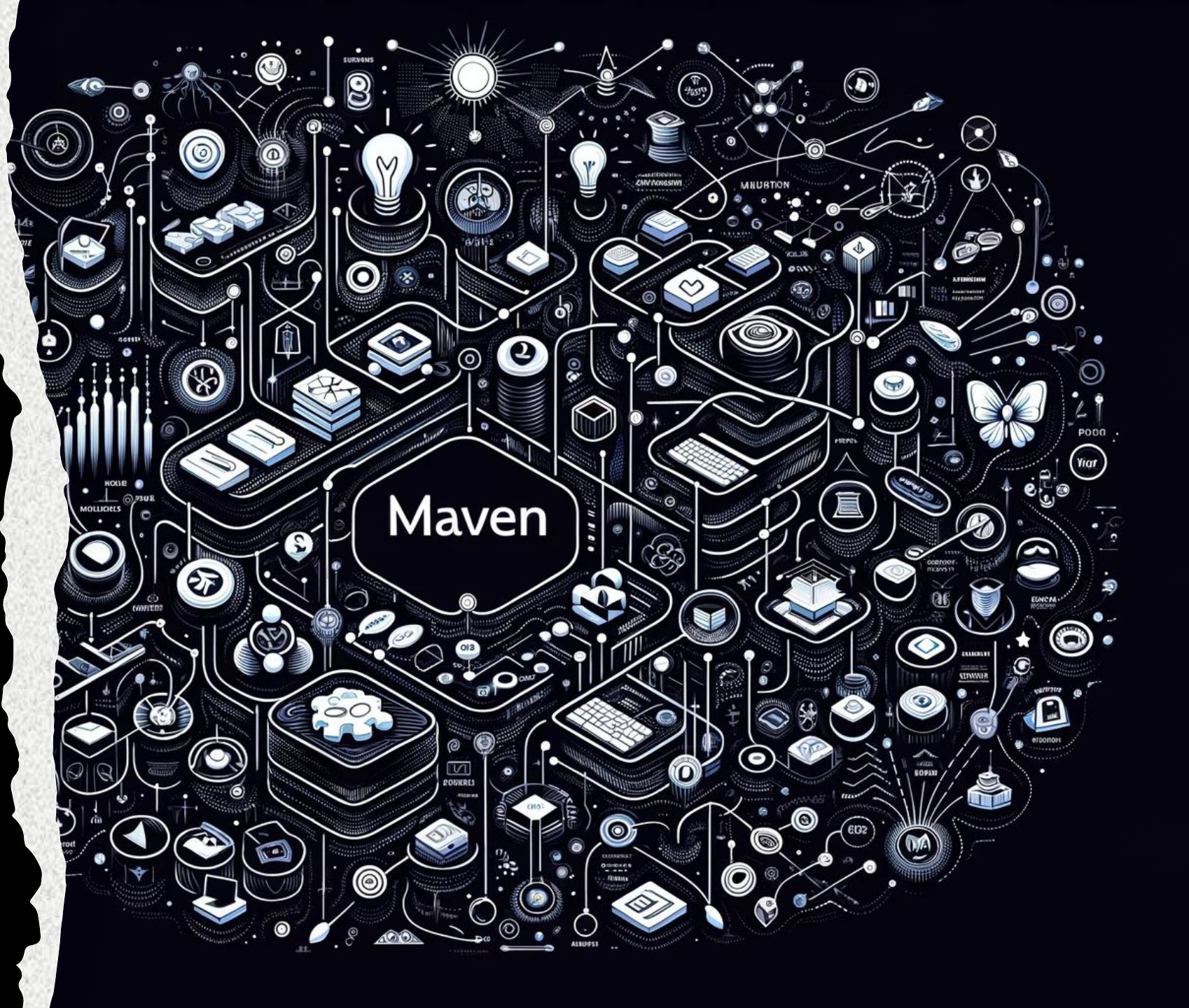
Quarta parte: **Spring Boot**

Giorno 23: 09.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Maven



Introduzione a Maven

Cos'è Maven?

Maven è uno strumento di project management e comprensione software che segue l'approccio di configurazione basato su **POM (Project Object Model)**, utilizzando un file XML (**pom.xml**) per descrivere il progetto software, le sue dipendenze da altri moduli e componenti esterni, e l'ordine di build.

Questo strumento automatizza il processo di compilazione, test, e packaging, promuovendo la **convenzione su configurazione**.

Comprendere Maven è fondamentale per semplificare e standardizzare la costruzione di progetti software.

Perché Usare Maven?

Automazione e Standardizzazione

Maven porta numerosi vantaggi come :

- l'automazione della compilazione
- la gestione semplificata delle dipendenze
- la standardizzazione dei progetti
- la promozione delle migliori pratiche e standard attraverso l'automazione della build
- la gestione delle dipendenze e la standardizzazione.

Maven migliora l'efficienza dello sviluppo e facilita la collaborazione tra team.

Convenzione su Configurazione

Semplificare la Configurazione

Maven segue il principio della convenzione su configurazione, predefinendo una struttura di progetto standard e regole di naming per ridurre al minimo la configurazione richiesta.

Questo include cartelle come ***src/main/java*** per il codice e ***src/test/java*** per i test, promuovendo le best practice e accelerando l'avvio di nuovi progetti.

Adottare la convenzione su configurazione accelera l'avvio di progetti e promuove best practice.

Lifecycle di Build di Maven

Fasi Principali

Maven organizza il processo di build in tre lifecycle principali: **clean**, **default** e **site**.

Ogni lifecycle comprende diverse fasi, come:

- ***clean***: la rimozione dei file generati da build precedenti
- ***default***: la gestione della compilazione, il test e il packaging del progetto
- ***site***:la generazione di documentazione per il progetto

```
<dependency>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-clean-plugin</artifactId>
  <version>3.1.0</version>
</dependency>
```

Conoscere i lifecycle di Maven aiuta a personalizzare e ottimizzare il processo di build.

Gestione delle Dipendenze

Il Cuore di Maven

Una delle caratteristiche più potenti di Maven è la sua capacità di gestire automaticamente le dipendenze del progetto, specificate nel file **pom.xml**.

Maven si occupa di scaricarle e aggiornarle quando necessario dal repository centrale, facilitando così la gestione delle versioni e la condivisione di artefatti.

```
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.9.8</version>
</dependency>
```

La gestione delle dipendenze automatizzata riduce significativamente il rischio di conflitti e semplifica l'aggiornamento delle librerie.

Comandi Maven Fondamentali

Operazioni di Base

I comandi base di Maven consentono di gestire facilmente tutte le fasi di sviluppo del progetto. Tra questi:
mvn clean: per pulire il progetto
mvn compile: per compilarlo
mvn test: per eseguire i test
mvn package: per pacchettizzare il progetto in un formato distribuibile
mvn install: per installare il pacchetto nel repository locale

```
mvn clean install
```

Conoscere i comandi base di Maven consente di gestire facilmente le fasi di sviluppo del progetto.

Configurazione di Maven in Eclipse

Installazione e Setup

Per utilizzare Maven in Eclipse, assicurati che il plugin ***M2Eclipse*** sia installato.

Questo plugin integra pienamente Maven in **Eclipse**, fornendo funzionalità come:

- l'importazione di progetti Maven
- la gestione delle dipendenze
- l'esecuzione di goals Maven direttamente dall'IDE.

Il plugin M2Eclipse facilita l'uso di Maven in Eclipse, integrando le funzionalità di Maven direttamente nell'ambiente di sviluppo.

Importare un Progetto Maven in Eclipse

Passi da Seguire

Per importare* un progetto Maven esistente in Eclipse:

1. vai su ***File > Import... > Maven > Existing Maven Projects***
2. naviga alla directory del Progetto
3. completa l'importazione.

Eclipse riconoscerà automaticamente la struttura del progetto e le configurazioni dal file ***pom.xml***.

** Il Progetto viene importato ma rimane nel file system di origine*

Importare progetti Maven esistenti in Eclipse consente di sfruttare le potenti funzionalità di Maven direttamente all'interno dell'IDE.

Aggiornare un Progetto Maven in Eclipse

Sincronizzazione con il pom.xml

Dopo aver apportato modifiche al file ***pom.xml*** di un progetto Maven in Eclipse, è possibile che tu debba aggiornare il progetto per riflettere queste modifiche.

Clicca con il tasto destro sul progetto e seleziona:
Maven > Update Project...

Aggiornare il progetto Maven assicura che tutte le modifiche al pom.xml siano correttamente applicate e riconosciute da Eclipse.

Eseguire Goals Maven in Eclipse

Utilizzo della Vista Maven Projects

Per eseguire specifici goals Maven in Eclipse:

1. apri la vista **Maven Projects**
2. espandi il tuo progetto
3. clicca con il tasto destro su **Lifecycle** o su uno specifico goal
4. seleziona **Run As**.

Eseguire goals Maven direttamente in Eclipse permette di automatizzare compiti come la compilazione, i test e il packaging senza uscire dall'IDE.

Risoluzione Problemi di Dipendenze

Gestione delle Dipendenze in Eclipse

Se incontri problemi di dipendenze in un progetto Maven all'interno di Eclipse, utilizza la funzionalità **Maven > Update Project...** e assicurati di selezionare **Force Update of Snapshots/Releases**.
Questo costringerà Maven a ricontrollare le dipendenze e aggiornarle se necessario.

```
<dependency>
    <groupId>com.esempio</groupId>
    <artifactId>libreria-esempio</artifactId>
    <version>1.0</version>
</dependency>
```

Forzare l'aggiornamento delle dipendenze può risolvere problemi legati a versioni mancanti o obsolete nel repository Maven.

Configurazione Avanzata di Maven in Eclipse

Personalizzazione del pom.xml

Per configurazioni avanzate di Maven in Eclipse, come la definizione di profili specifici o la configurazione di plugin, modifica direttamente il file pom.xml del tuo progetto.
Eclipse rileverà automaticamente queste modifiche e applicherà le configurazioni necessarie.

```
<profiles>
    <profile>
        <!-- Dettagli del profilo -->
    </profile>
</profiles>
```

La personalizzazione diretta del pom.xml offre la massima flessibilità nella configurazione del progetto Maven all'interno di Eclipse.

Conclusioni

Il Valore Aggiunto di Maven

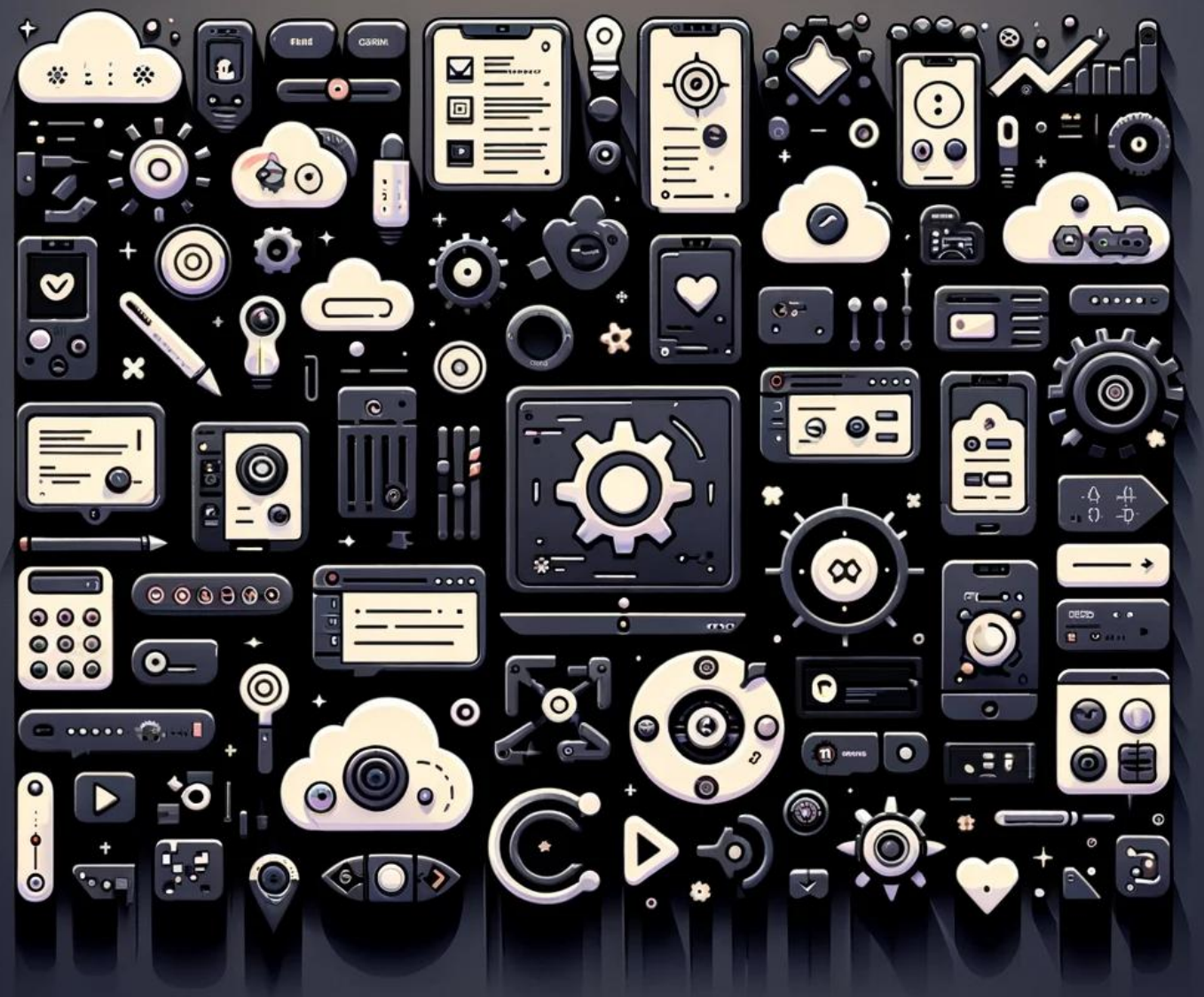
Maven non è solo uno strumento di automazione della build ma un vero e proprio **ecosistema** che supporta lo sviluppatore:

- gestione del ciclo di vita del software dalla gestione delle dipendenze
- standardizzazione dei progetti

facilitando così la collaborazione e l'adozione di best practice.

Maven rappresenta una soluzione completa per l'automazione dello sviluppo software, in modo da promuovere efficienza e qualità nel processo di costruzione del software.

Spring Boot



Introduzione a Spring Boot

Che cos'è Spring Boot?

Spring Boot **è un progetto all'interno dell'ecosistema Spring** che mira a **semplificare il processo di** configurazione e **pubblicazione** delle **applicazioni** basate su **Spring**.

Offre una configurazione **convention over configuration** per ridurre il numero di decisioni che uno sviluppatore deve prendere mentre mantiene la flessibilità.

Spring Boot facilita la creazione di applicazioni stand-alone che possono essere eseguite *out of the box* senza richiedere un server di applicazioni esterno.

Semplificazione dello sviluppo di applicazioni Spring

Caratteristiche principali di Spring Boot

Cosa offre Spring Boot?

Spring Boot offre numerose caratteristiche, tra cui:

- start-up rapido di progetti con ***Spring Initializr***;
- configurazione automatica basata sulle dipendenze presenti nel classpath;
- server embedded (***Tomcat, Jetty, Undertow***) per lo sviluppo e il testing rapidi;
- metriche, stato dell'applicazione e monitoraggio integrati;
- supporto per il packaging delle applicazioni come jar eseguibili;
- ...e una vasta gamma di starter per semplificare l'aggiunta di dipendenze e la configurazione.

Facilitare lo sviluppo e il deployment

Spring Boot e i microservizi

Perché è popolare per i microservizi?

Spring Boot è estremamente popolare nello sviluppo di microservizi grazie alla sua capacità di **creare servizi leggeri**, **indipendenti ed eseguibili**, che si avviano rapidamente e richiedono una configurazione minima.

La sua struttura modulare **supporta** l'architettura dei **microservizi**, consentendo agli sviluppatori di sviluppare, distribuire e scalare i servizi in modo indipendente.

Favorire l'architettura dei microservizi

Spring Boot Auto-configuration

Come funziona la configurazione automatica?

La configurazione automatica di **Spring Boot tenta di configurare automaticamente** la tua applicazione Spring in base alle dipendenze aggiunte al progetto.

Ad esempio, se Spring Boot rileva una dipendenza ***H2*** nel classpath, configurerà automaticamente un database in memoria.

Questo approccio elimina la necessità di definire esplicitamente ogni aspetto della configurazione, permettendo di concentrarsi maggiormente sulla logica di business dell'applicazione.

Ridurre la configurazione manuale

Installazione di Eclipse

Preparare l'ambiente di sviluppo

Per iniziare a sviluppare applicazioni Spring Boot in Eclipse, è necessario avere Eclipse IDE installato.

Assicurati di scaricare la versione di Eclipse IDE per sviluppatori **Java EE** o **Java SE + Plugin STS4** dal sito ufficiale di Eclipse (Marketplace).

Questa versione include il supporto per Maven e altri strumenti utili per lo sviluppo di applicazioni enterprise.

Scaricare e installare Eclipse IDE per Java EE o installare il plugin STS4 per Java SE

Installazione del Plugin Spring Tools 4

Aggiungere il supporto per Spring Boot

Per facilitare lo sviluppo di applicazioni Spring Boot in Eclipse, installa il plugin **Spring Tools 4** (noto anche come STS4).

Vai a ***Help -> Eclipse Marketplace***, cerca 'Spring Tools 4' e installa il plugin.

Questo aggiungerà funzionalità specifiche per Spring, come la creazione di nuovi progetti Spring Boot, il supporto per l'auto-completamento e l'integrazione con Spring Initializr direttamente in Eclipse.

Installare Spring Tools 4 per una migliore esperienza di sviluppo con Spring

Spring Initializr: Il punto di partenza

Come iniziare con Spring Boot

Spring Initializr è uno strumento online che fornisce un'interfaccia utente per generare rapidamente il boilerplate del progetto basato su Spring Boot: https://**start.spring.io**

Consente di selezionare le dipendenze, la versione di Spring Boot, il linguaggio (***Java***, Kotlin, Groovy), il sistema di build (***Maven*** o Gradle) e altre opzioni di configurazione, generando un progetto pronto per l'uso che si può importare nell'IDE preferito (**`Eclipse`**).

Generare rapidamente progetti Spring Boot

Creazione di un nuovo progetto Spring Boot

Utilizzare Spring Initializr

Con il plugin Spring Tools 4 installato, puoi creare facilmente un nuovo progetto Spring Boot. Vai a ***File -> New -> Other -> Spring -> Spring Starter Project***.

Compila i dettagli del progetto come il nome, il tipo di packaging (***jar*** è il più comune), la versione di Java, e seleziona le dipendenze iniziali che desideri includere nel tuo progetto.

Dopo aver compilato i dettagli, clicca su **Finish** per generare il progetto.

Generare un nuovo progetto Spring Boot con STS4

Struttura del Progetto Spring Boot

Comprendere la struttura del progetto

Dopo aver creato il tuo progetto, Eclipse mostrerà la struttura del progetto in conformità con le convenzioni Maven e Spring Boot.

Troverai le cartelle standard di Maven come ***src/main/java*** per il codice sorgente e ***src/main/resources*** per le risorse e i file di configurazione.

Spring Boot utilizza un file ***application.properties*** o ***application.yml*** in ***src/main/resources*** per la configurazione dell'applicazione.

Navigare la struttura del progetto Spring Boot

Esecuzione dell'applicazione Spring Boot

Avviare l'applicazione da Eclipse

Per eseguire l'applicazione Spring Boot:

1. individua la classe principale generata da Spring Initializr, che include il metodo main e l'annotazione ***@SpringBootApplication***.
2. Fai clic destro sulla classe e seleziona ***Run As -> Spring Boot App***.
3. Eclipse utilizzerà Maven per costruire il progetto e avviare l'applicazione.
4. Potrai vedere i log di avvio in console e accedere all'applicazione tramite il browser all'indirizzo ***http://localhost:8080*** *(*se è stata aggiunta la dependency *Spring Web* al Progetto*)*

Eseguire e testare l'applicazione localmente

Java SQL

Quarta parte: **Spring Boot**

Giorno 24: 10.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Spring MVC

Introduzione al Pattern MVC

Cos'è il Model-View-Controller?

Il Model-View-Controller (MVC) è un pattern **architetturale** utilizzato nello sviluppo di applicazioni software, in particolare per le applicazioni web.

Il pattern separa l'applicazione in tre componenti principali:

- **Model** (il cuore dell'applicazione, dove vengono gestiti i dati e la logica di business)
- **View** (la presentazione dei dati, ossia ciò che l'utente vede)
- **Controller** (il mediatore tra Model e View, gestisce l'input dell'utente, elabora le richieste e restituisce la risposta).

Un pattern per organizzare le applicazioni in modo logico

Benefici del Pattern MVC

Perché usare MVC?

L'utilizzo del pattern MVC offre numerosi vantaggi, tra cui:

- **separazione delle competenze**, facilitando così la manutenibilità e l'espandibilità dell'applicazione;
- promozione del **riutilizzo del codice**;
- facilitazione del **test** dei componenti **in modo isolato**;
- miglioramento della gestione dei team di sviluppo, consentendo a designer e sviluppatori di **lavorare in modo indipendente** sui loro componenti senza interferire l'uno con l'altro.

Migliorare struttura e manutenibilità delle applicazioni web

Il Modello in MVC

Gestione dei dati e della logica di business

Il Model rappresenta la struttura dei **dati**, **la logica di business** e le **regole** per modificare e gestire questi dati.

È completamente **indipendente** dalla **View** e dal **Controller**, il che significa che non ha conoscenza diretta delle interfacce utente.

Questo approccio permette di modificare e testare la logica di business senza influenzare la presentazione dei dati all'utente.

Cuore dell'applicazione, gestisce dati e logica

La Vista in MVC

Presentazione dei dati

La View è responsabile della **presentazione dei dati** provenienti dal Model all'utente.

Può esistere in diverse forme (ad esempio, come pagine web, interfaccia grafica desktop) e può essere implementata attraverso vari tecnologie come ***HTML, CSS e JavaScript*** per le applicazioni web.

La View si aggiorna in base ai cambiamenti dei dati nel Model, spesso attraverso meccanismi di binding o observers.

Mostrare i dati all'utente in modo efficace

Il Controller in MVC

Intermediazione tra Model e View

Il Controller agisce come **intermediario** tra **Model** e **View**, gestendo l'input dell'utente.

Quando l'utente interagisce con la View, il Controller interpreta l'input, richiedendo al Model di cambiare stato o di aggiornarsi e quindi aggiorna la View.

Il Controller decodifica le richieste dell'utente, determina l'azione da eseguire, individua il servizio del model da chiamare e ne restituisce la risposta.

Collega input utente, elaborazione e output

Introduzione a Spring Framework

Che cos'è Spring?

Spring Framework è un framework open source per lo sviluppo di applicazioni enterprise su piattaforma Java.

Creato per semplificare lo sviluppo di applicazioni Java, offre supporto completo per lo sviluppo basato su modelli di programmazione come la **Dependency Injection** (DI) e l'**aspect-oriented programming** (AOP).

Spring facilita la gestione dei componenti dell'applicazione attraverso il suo container di **inversione di controllo** (IoC), permettendo agli sviluppatori di concentrarsi sulla logica di business.

Un framework Java per semplificare lo sviluppo enterprise

Perché usare Spring?

Benefici del framework

Spring Framework offre numerosi vantaggi, tra cui:

- semplificazione della **configurazione** e dell'integrazione;
- supporto per **transazioni database**, gestione eccezioni, e accesso ai dati;
- framework **web MVC** per applicazioni web;
- supporto per la **sicurezza**, test, e gestione delle transazioni;

Questi strumenti e funzionalità rendono Spring una scelta popolare tra gli sviluppatori per creare applicazioni robuste, scalabili ed efficienti.

Motivi per scegliere Spring Framework

Componenti principali di Spring

Panoramica dei moduli

Spring è composto da diversi moduli che offrono una vasta gamma di funzionalità:

- Spring ***Core*** Container (gestione dei bean e DI);
- Spring *AOP* (programmazione orientata agli aspetti);
- Spring ***MVC*** (model-view-controller per applicazioni web);
- Spring ***Data* Access**/Integration (supporto JDBC, ORM, JMS);
- Spring *Web Flow*;
- Spring ***Security***.
- etc . . .

Questi moduli possono essere utilizzati insieme o separatamente, a seconda delle necessità dello sviluppatore.

I moduli che compongono Spring Framework sono diversi

Integrazione con l'ecosistema Spring

Come si inserisce Spring Boot in Spring?

- **Spring Boot non sostituisce il framework Spring**, piuttosto, lo **arricchisce** fornendo una nuova modalità di sviluppo.

- Offre un'esperienza integrata che sfrutta **Spring MVC**, **Spring Security**, **Spring Data**, e altri progetti Spring, può integrare altri progetti come Jetty, **Tomcat** su richiesta, riducendo drasticamente la quantità di configurazione richiesta.

- Spring Boot è costruito **sulla** fondazione di Spring Framework, estendendone le funzionalità per offrire uno sviluppo più rapido e una configurazione semplificata.

Spring Boot in pratica è un bundle di progetti spring, altri progetti e configurazione semplificata

Spring MVC per applicazioni web

Struttura MVC in Spring

Spring MVC è un framework **model-view-controller** che fornisce un'architettura per sviluppare **applicazioni web** flessibili e ben strutturate.

Utilizza funzionalità di Spring come la **Dependency Injection** per collegare oggetti di modello, controller e vista, facilitando lo sviluppo di interfacce utente dinamiche e reattive.

Spring MVC supporta anche la validazione, il binding dei dati e la gestione delle eccezioni, rendendolo uno strumento potente per lo sviluppo web.

Con Spring MVC è possibile creare applicazioni web

Spring MVC

Implementazione di MVC in Spring

Spring MVC è l'implementazione del pattern MVC di **Spring Framework**.
Offre un potente meccanismo di **routing**, elaborazione delle richieste, binding dei dati, validazione e molto altro.

Utilizza le **annotazioni** per definire i controller e i model e supporta una vasta gamma di tecnologie per le view, inclusi ***JSP, Thymeleaf e FreeMarker***.

Spring MVC facilita lo sviluppo di applicazioni web seguendo il pattern MVC, promuovendo la separazione delle competenze e migliorando la testabilità.

Spring MVC facilita lo sviluppo web

Spring MVC: Architettura e Componenti

Comprendere l'architettura di Spring MVC

Spring MVC è un framework basato sul modello Model-View-Controller che semplifica lo sviluppo di applicazioni web.

Utilizza il pattern **Front Controller** tramite il ***DispatcherServlet***, che riceve tutte le richieste e le indirizza ai controller specifici.

Il DispatcherServlet funge da Front Controller per tutte le richieste verso l'applicazione, decidendo quali controller devono gestire le richieste basandosi sulle mappature definite.
È configurato nel file ***web.xml*** o tramite Java Config.

Il DispatcherServlet è il cuore di Spring MVC, coordinando il flusso delle richieste.

Configurazione del DispatcherServlet

Introduzione al file web.xml

Il file ***web.xml*** è utilizzato in applicazioni web Java per configurare componenti quali servlet e filtri.

Per Spring MVC, è qui che si configura il DispatcherServlet, indicando a Spring il **punto di ingresso per le richieste HTTP**.

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Il file web.xml gioca un ruolo chiave nella configurazione delle applicazioni web tradizionali Java.

I Componenti Chiave di Spring MVC

Controller, View Resolver, e Model

In Spring MVC: i **Controller** gestiscono le richieste utente mappandole a metodi specifici con annotazioni come @RequestMapping.

Il **View Resolver** determina quale vista deve essere mostrata in risposta, mentre il **Model** trasporta i dati tra controller e vista.

```
@Controller
public class MyController {
    @RequestMapping("/hello")
    public String sayHello(Model param) {
        param = service.metodo(param); // viene chiamato metodo del service (modulo del model di MVC)
        return "hello";
    }
}
```

I controller sono responsabili della logica di gestione delle richieste, interagendo con modelli e scegliendo le viste.

Gestione delle Richieste in Spring MVC

Da DispatcherServlet ai Controller

Il flusso delle richieste inizia con il **DispatcherServlet** che riceve la richiesta HTTP. Basandosi sulle mappature dei controller, **delega la richiesta al controller** appropriato. Il controller elabora la richiesta, popola il modello e restituisce il nome della vista.

Il View Resolver poi risolve la vista e la restituisce al client.

```
@RequestMapping("/user")
public class UserController {
    // Logica del controller
}
```

Il flusso delle richieste in Spring MVC è un processo coordinato che coinvolge diversi componenti per rispondere efficacemente alle richieste utente.

Creare Controller con Spring MVC

Introduzione ai Controller

I controller in Spring MVC gestiscono le richieste degli utenti.

Utilizzando l'annotazione **@Controller**, possiamo definire una **classe come controller** che gestisce le richieste HTTP.

```
@Controller
public class MyController {
    @RequestMapping("/home")
    public String home() {
        return "home";
    }
}
```

I Controller sono componenti centrali in Spring MVC per gestire le richieste e le risposte.

Passare Dati ai View

Comunicazione tra Controller e View

I controller possono passare dati ai view utilizzando l'oggetto Model.

Questo è utile per trasmettere informazioni raccolte o elaborate nel controller alla vista che verrà renderizzata.

```
@GetMapping("/showUser")
public String showUser(Model model) {
    User user = // ottenere l'utente
    model.addAttribute("user", user);
    return "showUser";
}
```

L'oggetto Model in Spring MVC facilita il passaggio di dati dai controller ai view, permettendo una visualizzazione dinamica dei contenuti.

Passare dati alla view in RestController

Uso di rest controller per passare direttamente I dati

I **RestController** trasformano il dato ritornato strutturato, in formato **JSON**, oppure in stringa se non strutturato.

Utilizzando l'annotazione **@RestController**, possiamo definire una classe come controller che ritorna una stringa (eventualmente in formato JSON)

```
@RestController
public class MyController {
	@GetMapping("/studenti")
	public List<Studente> getStudenti() {
		// qui andrebbe chiamato il model che ritorna l'array di studenti
		return studenti;
	}
}
```

I Restcontroller sono componenti centrali in Spring MVC per gestire le richieste e le risposte REST.

Gestire le Richieste con @RequestMapping

Mappatura delle Richieste

L'annotazione @**RequestMapping** è utilizzata per mappare le richieste web ai metodi dei controller.

Può essere applicata a livello di classe o di metodo per catturare diverse tipologie di richieste HTTP.

```
@RequestMapping(value = "/users", method = RequestMethod.GET)
public String getUsers(Model model) {
    // Logica per ottenere gli utenti
    return "users";
}
```

@RequestMapping consente di definire in modo flessibile il modo in cui le richieste HTTP vengono gestite dai controller.

Semplificare la Mappatura

Mappature Specifiche per Metodo

Per una semplificazione, Spring MVC offre **@GetMapping** e **@PostMapping**, annotazioni specifiche per gestire rispettivamente le richieste **GET** e **POST**.

```
@GetMapping("/addUser")    // equivalente a @RequestMapping(value = "/addUser", method = RequestMethod.GET)
public String addUserForm(Model model) {
    // Aggiungere attributi al modello se necessario
    return "addUser";
}

@PostMapping("/addUser")   // equivalente a @RequestMapping(value = "/addUser", method = RequestMethod.POST)
public String addUserSubmit(@ModelAttribute User user, Model model) {
    // Logica per gestire la sottomissione
    return "userAdded";
}
```

@GetMapping e @PostMapping rendono più leggibile e organizzato il codice dei controller.

Java SQL

Quarta parte: **Spring Boot**

Giorno 25: 11.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Annotazioni Comuni in Spring

Panoramica sulle annotazioni

Spring offre diverse annotazioni per semplificare la configurazione e la gestione di classi dell'applicazione, che vengono automaticamente istanziate

- **@Component,**
- **@Service,**
- **@Repository,**
- **@Controller/@RestController.**

Queste classi sono **component** che estendono la classe **Component** base per la ***Dependency Injection***

Queste annotazioni aiutano a definire il ruolo dei componenti nell'applicazione.

Dependency Injection (DI) in Spring

Cos'è la Dependency Injection?

La **Dependency Injection** è un pattern di progettazione chiave utilizzato in Spring per realizzare l'inversione di controllo (IoC).

Consente di **iniettare le dipendenze (istanze) nelle classi** invece di farle creare direttamente dalle classi stesse, promuovendo un alto livello di disaccoppiamento tra componenti.

```
public class BookService {
    private BookRepository bookRepository;

    @Autowired
    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }
}
```

La DI migliora la modularità e la testabilità delle applicazioni.

Model: l'annotazione @Service

Layer di Servizio

@Service identifica una classe del **Model** come un servizio in Spring, indicata per eseguire operazioni di business, transazioni o chiamate al database.
Questo layer astrae la logica di business dall'accesso ai dati e dalla presentazione.

```
@Service
public class StudentService {
    private final StudentRepository repository;

    @Autowired
    public StudentService(StudentRepository repository) {
        this.repository = repository;
    }
}
```

*@Service è fondamentale per la logica di business delle applicazioni ed è lo **strato superficiale del Model**.*

Model: l'annotazione @Repository

Accesso ai Dati

@Repository è utilizzata per marcare una classe come repository, ovvero il layer di accesso ai dati del **Model**.

Favorisce l'astrazione dell'accesso ai dati e la gestione delle eccezioni specifiche della persistenza.

```
@Repository
public class BookRepository {

    // Accesso ai dati dei libri

}
```

@Repository facilita l'integrazione con il database e gestisce le eccezioni legate alla persistenza.

@RequestParam

Lettura dei Parametri per il Controller

@RequestParam è usato per estrarre i parametri della query dall'URL della richiesta.

È utile per **parametri obbligatori** che devono essere **sempre presenti** nel percorso URL.

```
@GetMapping("/students")
public List<Student> getStudentsByAge(@RequestParam(value = "age", required = true) Integer age) {
    return studentService.findByAge(age);
}
```

Estrazione di parametri della query con RequestParam.

@RequestBody

Leggere il Corpo della Richiesta

@RequestBody è usato per leggere il **body della richiesta** HTTP e deserializzarlo **in un oggetto Java.**

@RequestBody facilita la creazione o l'aggiornamento di risorse.

```
@PostMapping("/users")
public User createUser(@RequestBody User user){}

@PostMapping("/students")
public Student addStudent(@RequestBody Student student) {
    return studentService.save(student);
}
```

Leggere il corpo della richiesta per creare risorse con RequestBody.

Spring Data JPA

Introduzione a Spring Data JPA

Cos'è la Persistenza?

La **persistenza** si riferisce alla caratteristica dei dati di rimanere esistenti oltre l'esecuzione del programma che li ha creati.

Nel contesto delle applicazioni web, ciò significa, per esempio, salvare i dati in un database persistente come ***MySQL***, così che possano essere recuperati e utilizzati in futuro.

Definizione del concetto di persistenza e della sua importanza.

Che Cos'è JPA?

Java Persistence API (JPA)

JPA, **Java Persistence API**, è una specifica Java standard per il mapping oggetti-relazionali (**ORM**) per la gestione della persistenza e dell'accesso ai dati in database relazionali.

Fornisce un modo per mappare oggetti Java a tabelle di database, facilitando operazioni **CRUD**, query e transazioni.

Introduzione a JPA come standard ORM in Java.

Spring Data JPA

Semplificazione dell'accesso ai dati

Spring Data JPA **estende JPA**, fornendo un layer **astratto** che riduce la quantità di codice *boilerplate* necessario per implementare il layer di persistenza.

Vantaggi:

- automatizza l'implementazione dei repository
- supporta la generazione dinamica di query
- migliora significativamente l'efficienza dello sviluppo.

Vantaggi dell'utilizzo di Spring Data JPA per l'accesso ai dati.

Repository in Spring Data JPA

Astrazione dell'accesso ai dati

Il cuore di Spring Data JPA è il concetto di **repository**, **interfacce** che forniscono metodi CRUD per entità **senza richiedere implementazioni manuali**.

Spring Data JPA può **generare automaticamente queste implementazioni** al runtime, permettendo agli sviluppatori di concentrarsi sulla logica di business.

```
public interface StudenteRepository extends JpaRepository<Studente, Long> {}
```

Come i repository semplificano l'accesso ai dati.

Annotazioni Fondamentali di JPA

Definire entità e relazioni

Le annotazioni come ***@Entity, @Table, @Id, @GeneratedValue, @ManyToOne, @OneToMany***, ecc., sono utilizzate per **mappare classi** Java a **tabelle** di database e configurare relazioni tra entità.

```
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    // Altri campi
}
```

Panoramica delle annotazioni JPA per il mapping oggetti-database.

Configurazione di JPA in Spring Boot

Integrazione di JPA

Spring Boot facilita la configurazione di JPA fornendo **auto-configurazione** per la maggior parte delle impostazioni. È possibile personalizzare la configurazione di JPA nel file **application.properties** o application.yml di Spring Boot.

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

Imposta la connessione al database e controlla il comportamento di JPA con le proprietà Spring.

Impostazioni JPA Avanzate in Spring Boot

Personalizzazione di JPA

Spring Boot permette di personalizzare ulteriormente le impostazioni JPA, come:

- la strategia di generazione dello schema
- la visualizzazione delle query SQL.

```
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Configurare comportamenti specifici di JPA.

Definizione delle Entità JPA

Creazione di entità JPA

Le entità JPA rappresentano le tabelle del database:

- Annotare le classi Java con ***@Entity***
- definire la chiave primaria con ***@Id***.

N.B. ci deve essere **corrispondenza** fra i nomi nella tabella e nella classe

```
@Entity
public class Studente {
  @Id
  private Long id;
  // ...
}
```

Le entità JPA sono il ponte tra il tuo database e il codice Java.

Mappatura avanzata di Entità

Mappare le entità

Le annotazioni JPA come ***@Entity, @Table, @Id, e @GeneratedValue*** permettono di definire come gli oggetti Java si mappano alle tabelle e colonne del database.

```
@Entity
@Table(name="studente")      // è preferibile evitarlo
public class Student {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)   //id autoincrementale
  private Long id;
}
```

Definire la struttura delle entità JPA.

Repository JPA

Accesso ai dati astratto

Spring Data JPA usa **repository** per fornire un accesso astratto ai dati.

Estendere ***JpaRepository*** per sfruttare metodi **CRUD** predefiniti o definire query personalizzate .

```
public interface StudenteRepository extends JpaRepository<Studente, Long> { }
```

I repository di Spring Data JPA semplificano l'accesso ai dati e le query.

Query Methods in Repositories

Definizione di query attraverso metodi

Spring Data JPA permette di definire query personalizzate semplicemente dichiarando i metodi nel repository.

La **convenzione** di nomenclatura di Spring Data **costruisce** automaticamente la query.

```
List<Student> findByLastName(String lastName);
```

Verificare la mappatura delle query sul sito ufficiale Spring:
https://docs.spring.io/spring-data/jpa/reference/jpa/query-methods.html

La convenzione di nomenclatura di Spring Data consente di definire query senza SQL.

Java SQL

Quarta parte: **Spring Boot**

Giorno 26: 14.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Spring REST API

Introduzione alle API REST

Cos'è REST e perché è importante?

REST (Representational State Transfer) è un insieme di principi architetturali per la creazione di servizi web.

Utilizza il protocollo **HTTP** esistente per comunicare dati in formati leggibili come **JSON** o **XML**.

È importante perché facilita l'interoperabilità e la scalabilità delle applicazioni web, consentendo servizi loosely coupled e altamente efficienti.

Panoramica di REST: principi e vantaggi.

Principi Fondamentali di REST

I sei vincoli di REST - Parte 1

REST si basa su sei **vincoli** fondamentali:

1) **Interfaccia uniforme**: assicura che l'API sia consistente e prevedibile.
2) **Stateless**: ogni richiesta da client a server deve contenere tutte le informazioni necessarie per comprenderla, senza tenere conto dello stato del client.
3) **Cacheable**: le risposte devono, ove possibile, essere cacheabili per migliorare le prestazioni.

I primi tre vincoli architetturali di REST: Interfaccia uniforme, Stateless, Cacheable

Principi Fondamentali di REST

I sei vincoli di REST - Parte 2

4) **Sistema a strati**: l'architettura può essere strutturata in strati che incapsulano le funzionalità del server, facilitando la scalabilità e la sicurezza.
5) **Codice su richiesta** (opzionale): permette al client di scaricare ed eseguire codice sotto forma di applet o script, estendendo le funzionalità del client.
6) **Client-server**: separazione delle responsabilità, migliorando la portabilità dell'interfaccia utente e scalabilità.

I restanti vincoli architetturali di REST: Sistema a strati, codice su richiesta, client-server

Spring Boot e le API REST

Perché Spring Boot per le API REST?

Spring Boot semplifica notevolmente lo sviluppo di API **RESTful** grazie alla sua capacità di auto-configurazione e all'ampia gamma di starter, annotazioni e configurazioni predefinite.

Offre integrazioni semplici con database, validazione dei dati, sicurezza e molto altro, consentendo agli sviluppatori di **concentrarsi sulla logica di business** piuttosto che sulla configurazione dell'infrastruttura.

Vantaggi nell'utilizzo di Spring Boot per API RESTful.

Uso delle Annotazioni Spring per le API REST

Introduzione alle Annotazioni Spring

Spring fornisce diverse **annotazioni** per facilitare la creazione di API REST.

Queste annotazioni aiutano a:

- definire i **controller**
- gestire le **richieste/risposte**
- **mappare** le operazioni HTTP.

Esistono diverse annotazioni in spring per definire un servizio rest

@RestController

Definire i Controller REST

@**RestController** è una specializzazione di @Controller.

Indica che la classe gestisce le richieste web e ogni metodo ritorna direttamente il corpo della risposta.

```
@RestController
public class MyController {}
```

Utilizzo di @RestController per API REST.

@RequestMapping

Mappatura delle Richieste

@**RequestMapping** è l'annotazione generale per mappare le richieste web su metodi nei controller Spring.

Può essere usato a livello di classe o metodo.

```
@RequestMapping(value="/users", method=RequestMethod.GET)
public List<User> getUsers(){}
```

Definire il percorso e il metodo HTTP con RequestMapping.

@GetMapping, @PostMapping

Metodi Specifici per le Richieste

Spring 4.3 ha introdotto annotazioni composte che agiscono come **scorciatoie per *@RequestMapping*,**
tra cui:

- ***@GetMapping*** per gestire le richieste **GET**
- ***@PostMapping*** per gestire le richieste **POST**.

```
@GetMapping("/users")
public List<User> getUsers(){}

@PostMapping("/users")
public User addUser(@RequestBody User user){}
```

Annotazioni per gestire *GET* e POST.

@PutMapping, @DeleteMapping

Gestire Aggiornamenti ed Eliminazioni

Similmente a **@GetMapping** e **@PostMapping**, @PutMapping e @DeleteMapping sono utilizzati per gestire:

- ***@PutMapping*** per gestire le richieste **PUT**
- ***@DeleteMapping*** per gestire le richieste **DELETE**.

```
@PutMapping("/users/{id}")
public User updateUser(@PathVariable Long id, @RequestBody User user){}

@DeleteMapping("/users/{id}")
public void deleteUser(@PathVariable Long id){}
```

Annotazioni per PUT e DELETE: PutMapping, DeleteMapping.

@PathVariable

Catturare le Variabili del Percorso

@***PathVariable*** permette di leggere le variabili di percorso dall'URL della richiesta.

Ad esempio per identificare risorse specifiche

```
@GetMapping("/students/{id}")
public Student getStudent(@PathVariable Long id) {
    return studentService.findById(id).orElse(null);
}
```

Utilizzare variabili di percorso nelle API: PathVariable.

@ResponseBody e @ResponseStatus

Personalizzare le Risposte

- ***@ResponseBody*** indica che il valore di **ritorno** di un metodo deve essere legato al **corpo della risposta web**
- ***@ResponseStatus*** permette di specificare il **codice di stato HTTP** della risposta.

```
@GetMapping("/students/{id}")
@ResponseBody
public User getUserById(@PathVariable Long id){}

@ResponseStatus(HttpStatus.CREATED)
@PostMapping("/students")
public User createUser(@RequestBody User user){}
```

Controllare corpo della risposta e stato HTTP con ResponseBody e ResponseStatus.

POST GET PUT DELETE

Quando usarle affinché si abbia una API RESTful :

- **POST** crea una nuova risorsa in una collezione specifica e l'URI della nuova risorsa è assegnato dal server
- **GET** è usato per recuperare una risorsa o una collezione di risorse senza modificarle
- **PUT** aggiorna una risorsa esistente all'URI specificato (*o la crea se non esiste*).
- **DELETE** rimuove la risorsa specificata dall'URI.

Usare GET per operazioni di lettura, POST per creare nuove risorse quando l'URI finale non è noto, PUT per aggiornare risorse quando l'URI è noto, e DELETE per eliminare risorse.

POST, GET, PUT e DELETE sono le basi per le operazioni CRUD

Implementazione di Operazioni CRUD

Introduzione alle Operazioni CRUD

Le operazioni CRUD (**Create, Read, Update, Delete**) formano la base dell'interazione tra un'API RESTful e il database.

Spring Boot facilita l'implementazione di queste operazioni attraverso l'uso di Spring Data JPA e varie annotazioni.

Fondamenti delle operazioni CRUD in API REST.

Implementazione di Operazioni CRUD

Creazione di Risorse (Create)

L'operazione **Create** può essere implementata utilizzando ***@PostMapping***.

Questo metodo del controller prenderà un oggetto dalla richiesta, lo salverà nel database e restituirà l'oggetto salvato.

```
@PostMapping("/students")
    public Student createStudent(@RequestBody Student student) {
        return studentService.saveStudent(student);
    }
```

Aggiungere nuove risorse al database.

Implementazione di Operazioni CRUD

Lettura di Risorse (Read)

L'operazione **Read** viene implementata con ***@GetMapping***.
Questi metodi recuperano risorse dal database, che possono essere un singolo oggetto o una collezione di oggetti, basati su ID o altri criteri.

```
@GetMapping("/students")
public List<Student> getAllStudents() {
    return studentRepository.findAll();
}
@GetMapping("/students/{id}")       // con l'ID è URI : Uniform Resource Identifier
public Student getStudentById(@PathVariable Long id) {
    return studentRepository.findById(id).orElse(null);
}
```

Recuperare risorse dal database.

Implementazione di Operazioni CRUD

Aggiornamento di Risorse (Update)

L'operazione **Update** è gestita da @***PutMapping***.
Richiede l'ID della risorsa da aggiornare e i nuovi dati come input, aggiorna la risorsa nel database e restituisce la risorsa aggiornata.

```
@PutMapping("/students/{id}")
 public Student updateStudent( @PathVariable Long id
                             , @RequestBody Student studentDetails) {
        Student updatedStudent = studentService.updateStudent(id, studentDetails);
        return updatedStudent;
    }
```

Modificare le risorse esistenti.

Implementazione di Operazioni CRUD

Eliminazione di Risorse (Delete)

@***DeleteMapping*** gestisce l'operazione **Delete**.
Questo metodo elimina la risorsa specificata dal database e può restituire uno status per confermare l'eliminazione.

```
@DeleteMapping(“/students/{id}")
public Boolean deleteStudent(@PathVariable Long id) {
    boolean deleted = studentService.deleteStudent(id);
    return deleted;
}
```

Rimuovere risorse dal database.

API REST: Response Entities

Risposte personalizzate

ResponseEntity permette di costruire risposte HTTP dettagliate con status code specifici e personalizzazione del corpo della risposta.

```
return ResponseEntity.status(HttpStatus.CREATED).body(newStudent);
```

Costruire risposte HTTP complesse con ResponseEntity.

Gestione delle Risposte

Personalizzazione delle Risposte HTTP

Per una maggiore flessibilità, puoi utilizzare **ResponseEntity** per costruire risposte HTTP personalizzate, che ti permettono di controllare lo stato, gli header e il corpo della risposta.

```
@GetMapping("/students/{id}")
public ResponseEntity<Student> getStudent(@PathVariable Long id) {
    return studentService.findById(id)
            .map(student -> ResponseEntity.ok().body(student))
            .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

Costruire risposte HTTP dettagliate utilizzando ResponseEntity.

Java SQL

Quarta parte: **Spring Boot**

Giorno 27: 15.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

API REST: Versionamento

Concetti di base

Il versionamento API consente la coesistenza di diverse versioni dell'API per supportare diversi client senza interrompere l'uso di quelli esistenti.

Possibili strategie includono

- Versionamento **URL**
- Versionament **Header** o Media Type.

Il versionamento API per gestire diverse version di metodi e controller

API REST: Versionamento URL

Applicazione pratica

Versionamento tramite URL inserendo un prefisso, come /v1/, nel percorso dell'API.

Semplice da usare e immediatamente visibile ai client API.

```
@RequestMapping("/v1/students")
public class StudentV1Controller {
    // ...
}
```

Implementare il versionamento URL modificando la URL l'annotazione RequestMapping

API REST: Versionamento Header

Strategie avanzate

Versionamento tramite header, come ***Accept-Version***, permette versioni multiple mantenendo la stessa URL.

```
@RequestMapping(value = "/students", headers = "Accept-Version=v1")
public class StudentV1Controller {
    // ...
}
```

Implementare il versionamento URL modificando l'header nell'annotazione RequestMapping

Versionamento URL: quando usarlo

Comprendere la strategie di versionamento URL

Quando usarlo:

- Se è importante che le versioni dell'API siano esplicite e facilmente accessibili.
- Se si vogliono versioni nettamente separate che sono facili da distinguere.
- Per una maggiore visibilità durante lo sviluppo frontend e il testing.

Vantaggi:

- Facile da usare e da capire per gli sviluppatori.
- Può essere navigato direttamente dal browser.
- Facilita la gestione del caching a livello di server (es. proxy, CDN).

Svantaggi:

- Può portare a URL complessi o confusionari con troppi livelli.
- Il cambio di versione richiede la modifica dell'URL

Versionamento Header: quando usarlo

Comprendere la strategie di versionamento URL

Quando usarlo:

- Per mantenere lo stesso URL esternamente ma diverse versioni internamente.
- Per API che cambiano frequentemente e per le quali non si vuole alterare l'URL.
- Quando si desidera una separazione netta nel codice

Vantaggi:

- URL pulito senza modifiche per la versione.
- Flessibilità nel mantenere più versioni dell'API simultaneamente.
- Maggiore incapsulamento della logica di versionamento.

Svantaggi:

- Meno visibile e scopribile rispetto al versionamento URL.
- Richiede che i client debbano impostare correttamente gli headers HTTP.
- Potrebbe essere meno supportato da strumenti di testing e browser.

Documentazione API: Intro

Perché è importante

La documentazione è cruciale per le **API REST** per garantire che siano facilmente utilizzabili e comprensibili da sviluppatori esterni.

OpenAPI e ***Swagger*** offrono strumenti standardizzati per creare documentazione dinamica e interattiva.

Introduzione alla documentazione API con OpenAPI e Swagger.

Documentazione API: OpenAPI

Utilizzo di OpenAPI

OpenAPI è uno standard industriale per la descrizione delle interfacce delle API REST.

Definisce un formato standard per **descrivere la struttura di un'API REST**, consentendo la generazione automatica di documentazione, client SDK e altro ancora.

```
openapi: 3.0.0
info:
  title: Sample API
  description: API documentation example
  version: 1.0.0
paths:
  /students:
    get:
      summary: Lists all students
      responses:
        '200':
          description: A JSON array of students
```

Introdurre le specifiche OpenAPI per la documentazione API.

Documentazione API: Swagger UI

Interfaccia Utente Swagger

Swagger UI è uno strumento che rende la documentazione delle API generata da una specifica OpenAPI interattiva e navigabile da un browser web, facilitando test e esplorazione delle API da parte degli sviluppatori.

```
@Configuration
@EnableSwagger2
public class SwaggerConfig { }

// Accessibile via /swagger-ui.html
```

Implementare Swagger UI per visualizzare la documentazione API.

OpenAPI vs Swagger UI

Comprendere la scelta

OpenAPI è una specifica che definisce un formato standard per descrivere le interfacce delle API REST, consentendo la **generazione automatica** di documentazione, client SDK, e molto altro.

Swagger UI è uno strumento che utilizza queste definizioni OpenAPI per generare una documentazione interattiva e visivamente accattivante che gli sviluppatori possono esplorare e testare direttamente nel browser.

definire le API con OpenAPI per assicurare standardizzazione e interoperabilità, e implementare Swagger UI per migliorare l'accessibilità all documentazione API.

Integrare OpenAPI e Swagger UI per una documentazione API efficace.

Swagger UI: Configurazione in Spring Boot

Configurazione di base

Per integrare Swagger UI, è necessario aggiungere le dipendenze di SpringDoc MVC al progetto Spring Boot (sul ***pom.xml***).

Questo abilita automaticamente **Swagger UI** e la generazione di documentazione **OpenAPI**.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.8.6</version>
</dependency>
```

Aggiungere SpringDoc al progetto Springboot per Swagger UI.

Swagger UI: Documentazione e Test delle API

Interazione con le API

Documentazione

Una volta configurato, Swagger UI può essere accessibile navigando su ***/swagger-ui.html*** nel tuo browser. Questo fornisce una UI interattiva per esplorare le API, i loro endpoint e modelli di dati.

Test

Swagger UI non solo documenta le API, ma permette anche di eseguire richieste direttamente dall'interfaccia utente.
Questo facilita il test delle API e la verifica dei comportamenti e delle risposte.

Testare le API direttamente da Swagger UI.

Gestione degli Errori nelle API REST

Introduzione alla Gestione degli Errori

Una gestione efficace degli **errori** è **cruciale** nelle API REST per informare il client su cosa è andato storto durante l'elaborazione di una richiesta.

Deve fornire feedback chiari e utili, idealmente includendo **codici** di **stato HTTP** appropriati e messaggi di errore.

Importanza della trasparenza negli errori per migliorare l'usabilità dell'API.

Gestione degli Errori nelle API REST

Strutturare la Risposta di Errore

Una buona pratica è strutturare le risposte di errore in modo consistente.

Questo di solito include un codice di stato, un messaggio di errore e, opzionalmente, ulteriori dettagli o un link a più informazioni.

```
{
  "status": 404,
  "error": "Not Found",
  "message": "Student with ID 123 not found.",
  "path": "/students/123"
}
```

Fornire risposte di errore informative e consistenti.

Gestione degli Errori nelle API REST

@ControllerAdvice per la Gestione Globale

@ControllerAdvice consente di definire una classe globale per la gestione delle eccezioni, applicabile a tutti i controller, oppure per uno specific controller:

```
// Adivice gestisce tutti gli errori dei controller spring boot
@ControllerAdvice
public class GlobalExceptionHandler {
    // Definizioni di @ExceptionHandler
}

// Advice che gestisce tutti gli errori di un singolo controller
@ControllerAdvice(assignableTypes = StudentController.class)
public class StudentControllerHandler {
    // Definizioni di @ExceptionHandler
}
```

Centralizzare la gestione degli errori in un'unica classe.

Gestione degli Errori nelle API REST

Uso di @ExceptionHandler

@***ExceptionHandler*** permette di gestire eccezioni specifiche in un controller, consentendo una risposta personalizzata agli errori.

Può essere usato per catturare eccezioni standard o personalizzate.

```
@ControllerAdvice
public class GlobalExceptionHandler {
  @ExceptionHandler(ResourceNotFoundException.class)
  public ResponseEntity<?> resourceNotFoundException(ResourceNotFoundException ex, WebRequest request) {
      ErrorDetails details = new ErrorDetails(new Date(), ex.getMessage(), request.getDescription(false));
      return new ResponseEntity<>(details, HttpStatus.NOT_FOUND);
  }
}
```

Catturare e gestire eccezioni specifiche.

Java SQL

Quarta parte: **Spring Boot**

Giorno 28: 16.04.2025

Vincenzo Errante

Project Manager, Solution Architect, ICT Trainer

Query con Join in Spring Boot

Introduzione ai Join

Le query con join sono essenziali per recuperare dati da più tabelle relazionate.

In Spring Boot, possiamo usare sia **JPQL** (Java Persistence Query Language) che SQL nativo per eseguire queste query.

Inoltre, possiamo utilizzare oggetti **DTO** (Data Transfer Object) per mappare i risultati delle query.

Introduzione ai join in Spring Boot per recuperare dati da più tabelle.

Annotazioni JPA per Relazioni

Vantaggi e svantaggi delle Annotazioni JPA

Le annotazioni JPA offrono vari **vantaggi**:

- Semplicità e convenienza nella definizione delle relazioni.
- Navigazione intuitiva tra entità correlate.
- Gestione automatica delle relazioni e operazioni di cascata.
- Supporto per il lazy loading.

Le annotazioni JPA presentano alcuni **svantaggi**:

- Problemi di performance con relazioni complesse.
- Gestione dei cicli infiniti in relazioni bidirezionali.

Vantaggi e svantaggi delle annotazioni JPA per la gestione delle relazioni.

Annotazioni JPA per Relazioni

Introduzione alle Annotazioni JPA

Le annotazioni `**@OneToMany**` e `**@ManyToMany**` in JPA sono utilizzate per definire le relazioni tra entità nelle applicazioni Java che utilizzano un database relazionale.

Queste annotazioni semplificano la gestione delle relazioni tra entità.

Introduzione alle annotazioni JPA per relazioni tra entità.

Annotazioni JPA per Relazioni

@OneToMany

L'annotazione `@**OneToMany**` viene utilizzata per definire una relazione uno-a-molti tra due entità.

Un esempio tipico è una classe `Department` che ha molti `Employee`.

```
@Entity
public class Department {
   @Id
   @GeneratedValue(strategy =
          GenerationType.IDENTITY)
   private Long id;
   private String name;

   @OneToMany(mappedBy = 'department')
   private Set<Employee> employees;

   // getters and setters
}
```

```
@Entity
public class Employee {
   @Id
   @GeneratedValue(strategy =
       GenerationType.IDENTITY)
   private Long id; private String name;

   @ManyToOne
   @JoinColumn(name = 'department_id')
   private Department department;
   // getters and setters
}
```

Definizione di una relazione uno-a-molti con `@OneToMany`.

Relazioni Molti a Molti in Spring Boot

Introduzione alle Relazioni Molti a Molti

Una relazione molti a molti è una relazione in cui un record in una tabella può essere associato a più record in un'altra tabella e viceversa.

Per rappresentare questa relazione in un database relazionale, utilizziamo una tabella intermedia di join che contiene le chiavi primarie delle due tabelle in relazione.

Introduzione alle relazioni molti a molti in Spring Boot.

Annotazioni JPA per Relazioni

@ManyToMany

L'annotazione `@**ManyToMany**` viene utilizzata per definire una relazione molti-a-molti tra due entità. Un esempio tipico è una classe Student che può essere associata a molti Course e viceversa

```
@Entity
public class Student {
  @Id @GeneratedValue(strategy =
GenerationType.IDENTITY)
  private Long id;
  private String name;
  @ManyToMany @JoinTable( name = 'student_course’,
   joinColumns = @JoinColumn(name = 'student_id’),
   inverseJoinColumns = @JoinColumn(name =
                                    'course_id’))
   private Set<Course> courses;
  // getters and setters
}
```

```
@Entity
public class Course {
   @Id @GeneratedValue(strategy =
GenerationType.IDENTITY)
   private Long id;
   private String name;

  @ManyToMany(mappedBy = ‘courses’)
  private Set<Student> students;

  // getters and setters

}
```

Definizione di una relazione molti-a-molti con `@ManyToMany`.

Query con Join in Spring Boot

JPQL Join

Per eseguire un join con JPQL, utilizziamo l'annotazione **@Query** nei repository.

Esempio: Questo recupera tutti i dipendenti di un dipartimento specifico.

```
@Query('SELECT e FROM Employee e JOIN e.department d WHERE d.name = :departmentName’)

 List<Employee> findByDepartmentName(@Param('departmentName') String departmentName);
```

Esempio di join con JPQL in un repository Spring Boot.

Query con Join in Spring Boot

SQL nativo Join

Per eseguire un join con SQL nativo, utilizziamo l'annotazione **@Query** con **nativeQuery=true**.

```
@Query(value = 'SELECT e.*, d.name AS department_name FROM employees e JOIN departments d ON
e.department_id = d.id WHERE d.name = :departmentName', nativeQuery = true)

List<Object[]> findByDepartmentNameNative(@Param('departmentName') String departmentName);
```

Esempio di join con SQL nativo in un repository Spring Boot.

Query con Join in Spring Boot con DTO

DTO (Data Transfer Object)

Un ***DTO*** è un oggetto che trasporta dati tra processi.

Utilizzare un DTO nelle query con join aiuta a mappare i risultati della query su oggetti specifici.

Esempio di DTO:

```
public class EmployeeDTO {
      private Long id; private String name;
      private String departmentName;

     // getters and setters
}
```

Introduzione ai DTO per mappare i risultati delle query con join.

Query con Join in Spring Boot con DTO

Utilizzo dei DTO

Per utilizzare un DTO in una query con join, specifichiamo il **costruttore** del DTO nella query JPQL.

```
@Query('SELECT new com.example.EmployeeDTO(e.id, e.name, d.name)
          FROM Employee e JOIN e.department d WHERE d.name = :departmentName’)

List<EmployeeDTO> findByDepartmentNameDTO(
                      @Param('departmentName’) String departmentName
                  );
```

Esempio di join con JPQL e DTO in un repository Spring Boot.

Approccio con DTO

Vantaggi e svantaggi dell'Approccio con DTO

Utilizzando i DTO nelle query, possiamo mappare i risultati su oggetti specifici, migliorando la gestione dei dati recuperati.

Vantaggi:

- Performance ottimizzata recuperando solo i dati necessari.
- Separazione delle preoccupazioni migliorando la manutenibilità.
- Controllo granulare sui dati recuperati.
- Maggiore sicurezza evitando di esporre direttamente le entità del dominio.

Svantaggi:

- Richiede più codice per definire e mappare i DTO.
- Aggiunge un livello di complessità che deve essere gestito e mantenuto.

Vantaggi e svantaggi dell'approccio con DTO.

Query con Join in Spring Boot con DTO

SQL nativo con DTO

In SQL nativo, mappiamo manualmente i risultati della query al DTO.

Esempio:

// poi mappiamo manualmente i risultati al DTO in un servizio o controller.

```
@Query(value =
           'SELECT e.id, e.name, d.name AS department_name
              FROM employees e JOIN departments d ON e.department_id = d.id
             WHERE d.name = :departmentName', nativeQuery = true)
       List<Object[]>


findByDepartmentNameNative(@Param('departmentName') String departmentName);
```

Esempio di join con SQL nativo e DTO in un repository Spring Boot.

Annotazioni JPA vs DTO

Quando Usare Quale Approccio

La scelta tra annotazioni JPA e DTO **dipende dal contesto**:

- **Annotazioni JPA:** Sono preferibili quando la semplicità e la convenienza sono più importanti, e quando le relazioni tra entità sono semplici e ben definite. Questo approccio è ideale per applicazioni CRUD di base e per progetti che non richiedono ottimizzazioni di performance significative.
- **DTO:** Sono preferibili quando è necessario un controllo granulare sui dati recuperati, quando si lavora con relazioni complesse o quando le performance sono una priorità. Questo approccio è ideale per applicazioni che richiedono operazioni complesse sui dati e ottimizzazioni delle query.

Considerazioni per scegliere tra annotazioni JPA e DTO.