



# JAVA INTERMEDIATE CON SPRING BOOT



# Java Intermediate con Spring Boot

---

- Architettura
  - Multi-tier application
  - Enterprise application
    - Relazione e confronto tra Java EE e Spring Framework
- Spring Framework
  - Introduzione a Spring Core
- Layer Dati
  - JPA/Hibernate
  - Spring Data
- Layer Applicativo
  - Richiami di OOP in Java – Cenni di Design Pattern
  - Beans in Spring Boot
- Layer Utente
  - Il pattern MVC
  - Introduzione a Spring MVC
  - Spring Form e validazione lato server
    - Cenni di Validazione lato client
  - Servire i dati
    - Tecnologia RESTful
    - Spring RESTful web services
- Configurazione di Applicazioni
  - Spring Security

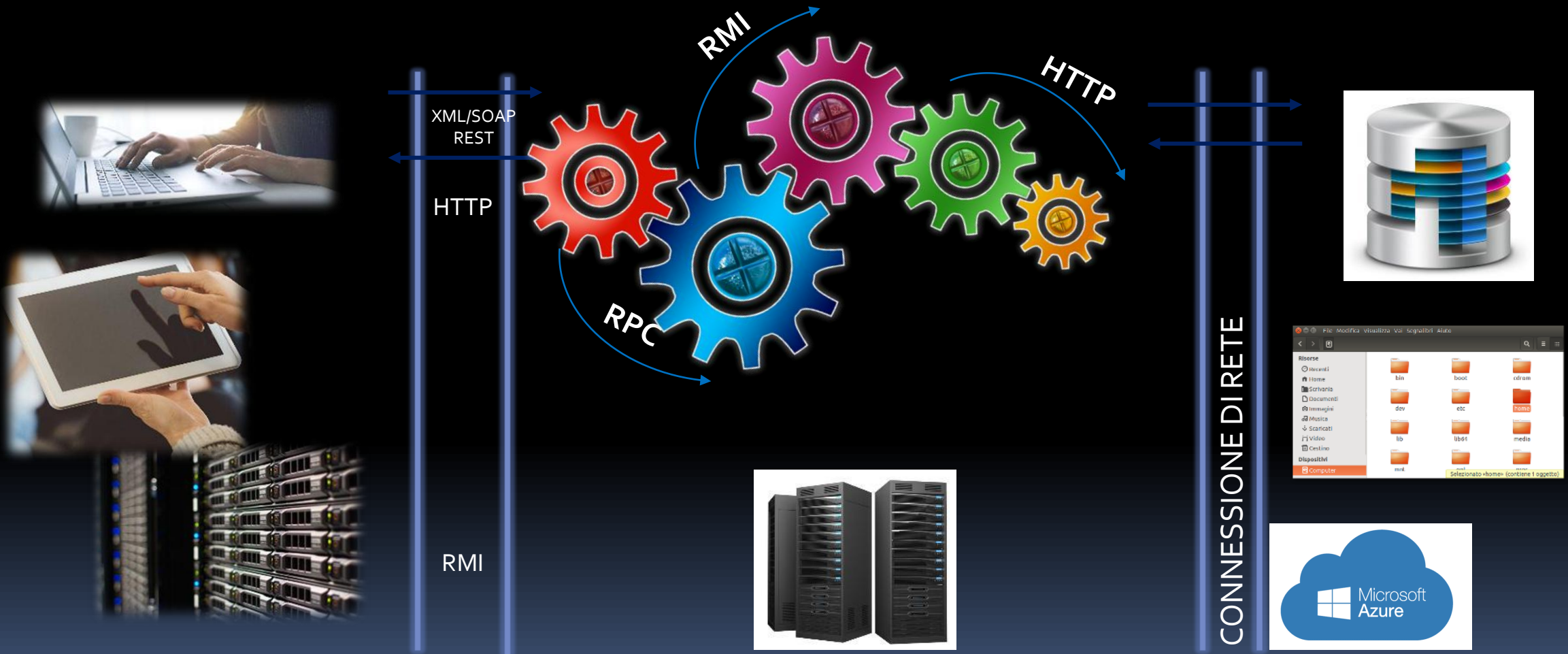




Architettura e Primi Passi in Spring

# MODULO 1

# Architettura



# Architettura

- Suddivisione Logica di un'architettura 3-Tier
  - User (Presentation) Layer / Client
  - Application Layer
  - Data Layer
- In generale le applicazioni sono suddivise in n-tier ognuno dei quali ha le proprie specifiche competenze

# Spring Framework

- Spring è uno dei framework più popolari per lo sviluppo di applicazioni nell'ecosistema Java
- Esso è utilizzato perché di ampia portata, stabilità e maturità
- Spring, insieme a Boot, è un framework raffinato, moderno e altamente espressivo



# Spring Framework

- Spring Framework fornisce un modello di programmazione e configurazione completo per le moderne applicazioni aziendali basate su Java
  - su qualsiasi tipo di piattaforma di distribuzione.
- Un elemento chiave di Spring è il supporto infrastrutturale a livello applicativo
- Spring si concentra sul "plumbing" delle applicazioni aziendali
  - in modo che i team possano concentrarsi sulla logica aziendale a livello applicativo,
  - senza inutili legami con ambienti di distribuzione specifici

# || Differenze tra Java EE e Spring

- Java Platform in Edizione Enterprise
  - è una piattaforma per lo sviluppo di applicazione business basata sul linguaggio Java
  - Java EE centralizza la gestione degli algoritmi di business per la gestione del flusso dei dati dall'utente al database e viceversa
- Spring Framework è un framework
  - basato sul linguaggio Java e sui concetti alla base del paradigma di IoC



# Differenze tra Java EE e Spring

- Java EE comprende
  - la tecnologia di programmazione
  - la definizione di ambienti di esecuzione
  - la possibilità di eseguire suite di test
- Le sue parti fondamentali sono
  - Enterprise JavaBeans (EJBs)
  - JavaServer Pages (JSP) e servlet Java
  - e un assortimento di interfacce per la connessione alle risorse di dati nell'impresa
- È nata nell'ottobre 2002 a opera di Rod Johnson
  - nel suo libro Expert One-to-One J2EE Design and Development
- Successivamente si è avuto il suo primo rilascio con licenza Apache 2.0 nel giugno 2003

# Differenze tra Java EE e Spring

- Le interfacce Java EE comprendono
  - JDBC per database
  - JNDI per i registri
  - JTA per gli scambi
  - JMS per la messaggistica (asincrona)
  - JavaMail per i framework di posta elettronica
  - JavaIDL per la comunicazione di rete tramite CORBA
- EE sta per Enterprise Edition
  - è una tecnologia in cui sono definite le interfacce / API standard per la creazione di applicazioni di business
  - Queste applicazioni sono costituite da moduli o "parti" che utilizzano "compartimenti" Java EE come framework di runtime

# ||| Differenze tra Java EE e Spring

- Abstraction over Concretation
- Il principio alla base di Java EE (ma anche di tutti gli altri framework enterprise) è il disaccoppiamento tra i servizi che l'applicazione deve rendere disponibili e la piattaforma dove essi vengono effettivamente eseguiti:
  - Un'applicazione dipende dalla astrazione e non dalla effettiva implementazione

# ||| Differenze tra Java EE e Spring

- Spring è un framework modulare
  - in cui ogni modulo (circa 20 di base) è contenuto all'interno del container core
  - in cui non c'è particolare dipendenza da uno specifico IDE o compilatore
  - a fronte di una semplificazione dei processi di configurazione ed implementazione è sicuramente più lento di Java EE

# Differenze tra Java EE e Spring

Java EE vs Spring	Java EE	Spring
<b>Architettura</b>	Basato su un framework architettonico tridimensionale, ad esempio livelli logici, livelli client e livelli di presentazione	Si basa su un'architettura a strati che include molti moduli. Questi moduli sono realizzati sopra al suo contenitore centrale.
<b>Linguaggio</b>	Utilizza un linguaggio di alto livello orientato agli oggetti che ha un certo stile e sintassi	Non ha un certo modello di programmazione
<b>Interfaccia</b>	In genere ha un'interfaccia utente grafica creata dalle API Project Swing o Abstract Window Toolkit	La sintassi è la stessa ovunque, indipendentemente da un IDE o da un compilatore
<b>Iniezione delle Dipendenze</b>	Usa l'iniezione delle dipendenze	Usa l'iniezione delle dipendenze
<b>Struttura</b>	Può essere basato sul web o non web	Basato su quasi 20 moduli
<b>Velocità</b>	Molto performante	Più lento di Java EE

# Differenze tra Java EE e Spring

## ■ Java EE

- È destinata ad organizzazioni e aziende che richiedono un ampio quadro diffuso adattabile per la creazione di applicazioni di massa
- Incorpora librerie extra per l'accesso al database (JDBC, JPA), tecniche di comunicazione remote (RMI), messaggistica (JMS), ambienti Web, gestione XML e propone API standard per JavaBean Enterprise, portlet, servlet e così via via
- Nella prima implementazione era noto come Java 2 Platform Enterprise Edition o J2EE, mentre da qualche anno è distribuito con il nome di Java EE
- Inizialmente è stato creato per la gestione di applicazioni con componenti fortemente disaccoppiati attraverso moduli
- L'obiettivo fondamentale di Java EE è quello di districare i problemi di base osservati dai progettisti per quanto riguarda la creazione di applicazioni attuali attraverso varie API
- Alcune cospicue API che accompagnano Java EE incorporano Servlet, Java Server Pages (JSP), Java Persistence API (JPA), Enterprise JavaBeans (EJB), JSP Standard Tag Library (JSTL), possessori di Java EE, ecc.

# Differenze tra Java EE e Spring

- Spring è un sistema open source per le grandi aziende Java
  - I punti salienti centrali di Spring Framework possono essere utilizzati nella creazione di qualsiasi applicazione Java, ma ci sono espansioni per la creazione di applicazioni Web sullo stage Java EE
  - La struttura è realizzata per semplificare la gestione di applicazioni in ambiente Java EE usando un modello di programmazione basato su POJO
  - Punta fortemente sul disaccoppiamento e sulla dependency injection
  - Spring MVC è una delle numerose parti di Spring ed è un sistema web che sfrutta i punti salienti generali di Spring
  - È un sistema abbastanza esclusivo in quanto è eccezionalmente configurabile
    - è possibile, ad esempio, utilizzare diversi livelli DB



# Installazione dell'ambiente di lavoro


- Integrated Development Environment
  - A scelta Eclipse o IntelliJ
- Server di Database
  - A scelta PostgreSQL o MySQL







# Esercitazione GUIDATA

1. Installazione dell'IDE (ove necessario)
  2. Installazione del server di database
  3. Installazione di Spring Boot Tools
  4. Gestione delle configurazioni dei progetti con l'IDE
- 



# **RICHIAMI DI JAVA OOP**

# Richiami di Java Essenziali

- Di seguito gli aspetti di Java essenziali per una corretta progettazione di un'applicazione enterprise
- Basi di OOP
  - Incapsulamento, astrazione, ereditarietà
  - Polimorfismo
    - Interfacce funzionali
      - Espressioni lambda

# Richiami Java Essenziali

- È indispensabile una conoscenza di base del concetto di Design Pattern
  - Creazionali
  - Strutturali
  - Comportamentali
- Si rimanda alla letteratura classica (ad es. si cerchi GOF su Wikipedia) per un approfondimento se necessario

# |||| Esercitazione 1

- Esercitazione

- Occorre implementare un algoritmo per la gestione di figure geometriche rappresentabili su un canovaccio (incapsulamento - ereditarietà - polimorfismo)

- *Nota: si implementi una versione del canovaccio che presenti le diverse figure su console a caratteri*



# Esercitazione 2

- Esercitazione
  - Occorre implementare un algoritmo per la distribuzione di un mazzo di carte da gioco (programmazione generica e polimorfismo)



# Esercitazione 3

- Esercitazione

- Dato un file in formato .csv contenente la classificazione statistica dei comuni italiani (fonte ISTAT: <https://www.istat.it/storage/codici-unita-amministrative/Elenco-comuni-italiani.xlsx>) si effettui un'analisi dello stesso (java nio2) costruendo un set (collection framework) contenente istanze di classi (POJO) appositamente create per effettuare operazioni di ricerca, estrazione e trasformazione dei dati letti (Stream / lambda)



Business Layer

# MODULO 2



# Inversion of Control

- Inversion of Control è un principio nell'ingegneria del software che trasferisce il controllo di oggetti o parti di un programma a un contenitore o framework
  - In contrasto con la programmazione tradizionale, in cui il nostro codice personalizzato effettua chiamate verso una libreria, IoC consente a un framework di prendere il controllo del flusso di un programma ed effettuare chiamate al nostro codice personalizzato
  - Per abilitarlo, i framework usano astrazioni con un comportamento aggiuntivo integrato. Se vogliamo aggiungere il nostro comportamento, dobbiamo estendere le classi del framework o plugin delle nostre classi.
  - I vantaggi di questa architettura sono:
    - disaccoppiare l'esecuzione di un compito dalla sua attuazione
    - facilitando il passaggio tra diverse implementazioni
    - maggiore modularità di un programma
    - maggiore facilità nel testare un programma isolando un componente o non considerando le sue dipendenze

# Dependency Injection

- L'iniezione delle dipendenze è alla base della implementazione dell'IoC in Spring
  - Constructor based
  - Setter based
  - Field based

```
@Service
public class BlogServiceImpl implements BlogService {
    private final Logger log = LoggerFactory.getLogger(BlogServiceImpl.class);

    @Autowired
    private ArticlesRepository articlesRepository;
    @Autowired
    private CommentsRepository commentsRepository;
    @Autowired
    private UsersRepository usersRepository;
```



# Log

```
@RestController
public class LoggingController {

    Logger logger = LoggerFactory.getLogger(LoggingController.class);

    @RequestMapping("/")
    public String index() {
        logger.trace("A TRACE Message");
        logger.debug("A DEBUG Message");
        logger.info("An INFO Message");
        logger.warn("A WARN Message");
        logger.error("An ERROR Message");

        return "Howdy! Check out the Logs to see the output...";
    }
}
```

# Annotations Standard per i Bean più Comuni

## Annotazione

## Utilizzo

**@Component**

*Stereotipo per tutti i component gestiti nel modulo IoC*

**@Repository**

*Stereotipo per il layer di persistenza*

**@Service**

*Stereotipo per il layer di business*

**@Controller**

*Stereotipo per il layer utente*

# Bean in Spring

- Gli oggetti che costituiscono la spina dorsale dell'applicazione
  - gestiti dal contenitore Spring IoC
  - sono definiti come beans
- Un bean è un oggetto che viene istanziato, assemblato e gestito da un contenitore **Spring IoC**

# Esempio di Bean in Spring Boot

```
@Configuration
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(auth -> auth.anyRequest().permitAll()) // ac
            .cors(c -> c.disable()) // CORS disabilitato
            .csrf(c -> c.disable()) // CSRF disabilitato
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthe

        return http.build();
    }

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```



# Bean in Spring

- In Spring esistono diversi tipi di scope (ambiti) in cui sono definiti i bean:
    - singleton
    - prototype
    - request
    - session
    - application
    - websocket
- 

# Singleton Scope

- Il container crea una singola istanza per il bean
- Tutte le richieste restituiscono lo stesso oggetto che è messo in cache
- Ogni modifica all'oggetto è replicata in ogni reference all'oggetto stesso
- Questo è l'ambito predefinito





# Prototype Scope

- Un bean definito in questo ambito restituisce una istanza diversa ad ogni richiesta



# Request Scope

- In questo ambito viene creata una istanza del bean per ogni richiesta HTTP



# Session Scope

- In questo ambito viene creata una istanza del bean per ogni sessione

# Application Scope

- In questo ambito viene creata una istanza del bean per tutto il lifecycle del Servlet Context
- A differenza del Singleton, una istanza di questo tipo viene condivisa attraverso diverse applicazioni servlet-based che vengono eseguite nello stesso Servlet Context



## WebSocket Scope

- Viene creato un bean per ogni sessione WebSocket
- Il bean ha un comportamento da Singleton, ma limitato alla sessione del WebSocket al quale è associato



Data Layer

# MODULO 3

# Java Persistence Framework

- Java Persistence API è una specifica relativa alla persistenza
  - in senso generale indica la capacità di oggetti Java di sopravvivere oltre il ciclo di vita dell'applicazione che li ha generati
- Non tutti gli oggetti Java devono essere resi persistenti
  - ma la maggior parte delle applicazioni enterprise necessitano che i loro oggetti principali vengano salvati
- La specifica JPA consente
  - di definire quali oggetti debbano essere resi persistenti e come la loro persistenza debba essere gestita all'interno delle applicazioni, tali oggetti vengono chiamati Entities
- Rispetto alla persistenza implementata integrando manualmente i costrutti SQL all'interno del codice Java, JPA offre un approccio molto più avanzato ed efficace.

# Java Persistence Framework

- JPA non è uno strumento o un framework che possa essere utilizzato direttamente
  - ma è piuttosto un insieme di concetti (espressi da interfacce) che possono essere implementati da tools o frameworks
    - Il framework più diffuso che implementa le specifiche JPA è Hibernate (<https://hibernate.org/>)
    - rilasciato per la prima volta nel 2002 e si è evoluto nel tempo per supportare sempre maggiori funzioni, seguendo lo sviluppo delle specifiche JPA
    - Hibernate, così come tutte le altre implementazioni di JPA, definisce uno strato **ORM** (Object Relational Mapping).



# Java Persistence Framework

- L'Object Relational Mapping si concentra sulla necessità di trovare una corrispondenza tra la struttura delle classi Java
  - costruite secondo il paradigma Object Oriented
- e la struttura dei database relazionali
  - realizzati per mezzo di tabelle (ovvero relazioni)
- Lo strato ORM (JPA layer) è responsabile della conversione degli oggetti Java in tabelle e colonne del database e viceversa
  - Poiché tale conversione è molto importante per la realizzazione di applicazioni enterprise con dati persistenti, l'impiego di uno strumento che possa effettuarla in modo automatico ed efficiente è utilissimo per gli sviluppatori

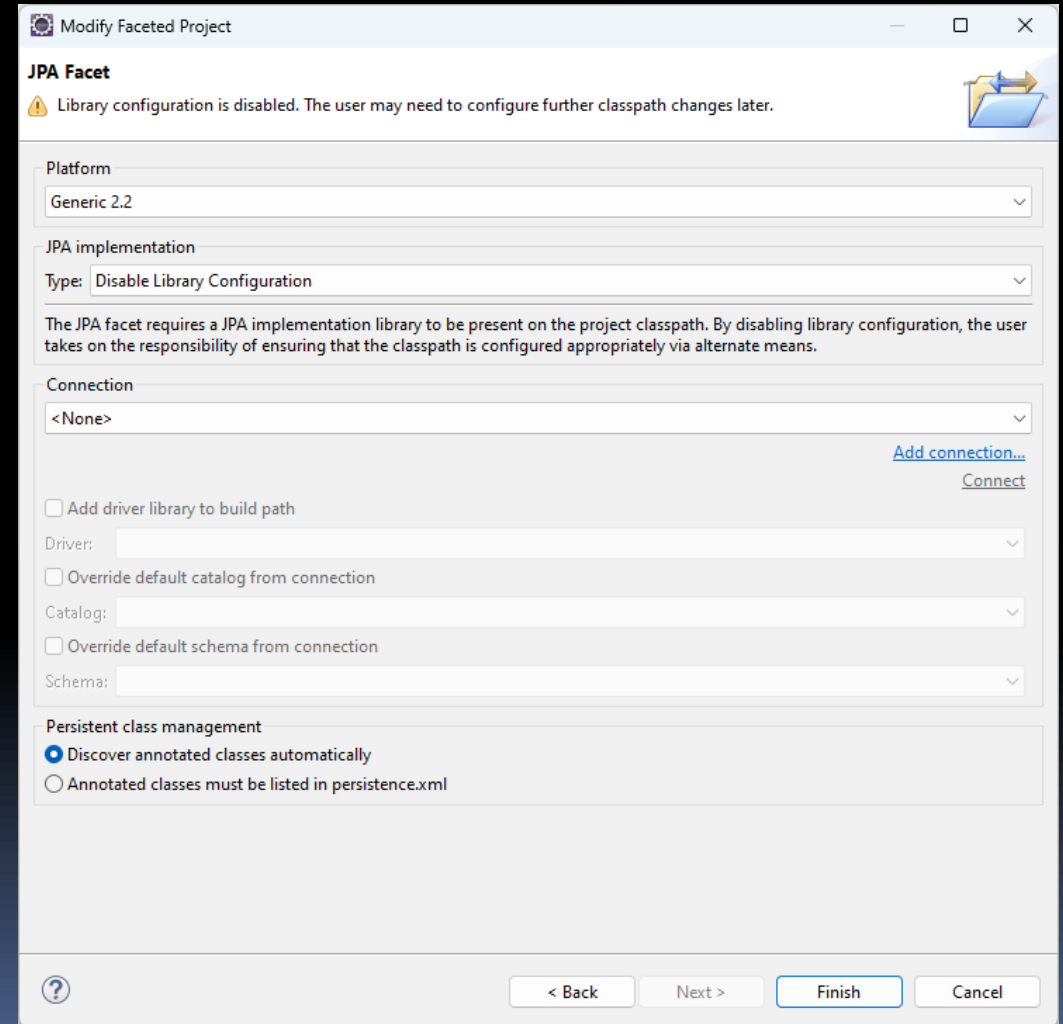
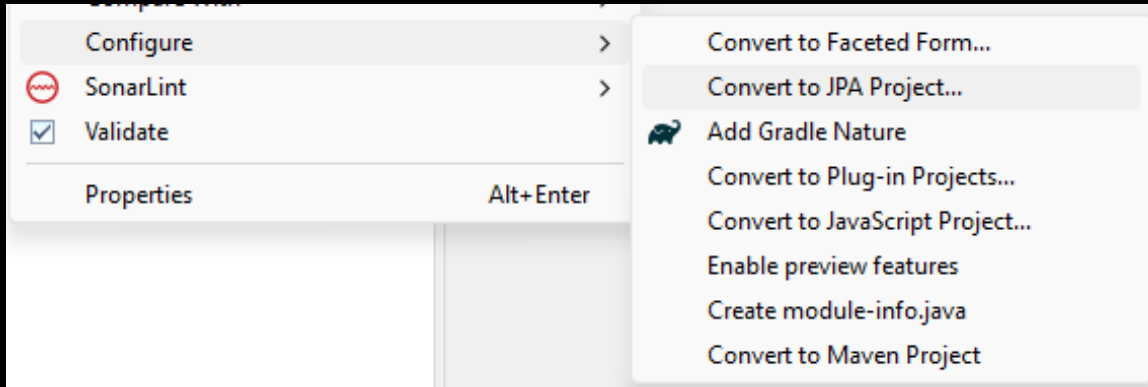
# Java Persistence Framework

- Il JPA nasconde ed automatizza le operazioni di sincronizzazione tra i campi delle queries e le proprietà degli oggetti che intende salvare in modo che lo sviluppatore possa concentrarsi sull'implementazione della logica applicativa
  - attraverso una serie di annotazioni sulle classi persistenti
    - che istruisce lo strato JPA su come gestire le operazioni di mapping

# Java Persistence Framework

- Per poter utilizzare la tecnologia JPA è necessario effettuare le seguenti operazioni:
  - Configurare il progetto per utilizzare JPA
  - Configurare le impostazioni JPA (indirizzo del server, username, password...)
  - Effettuare il mapping delle entities utilizzando le apposite annotations
  - Effettuare le operazioni di interazione con il db impiegando gli strumenti JPA come l'EntityManager
- L'impiego del JPA porta i seguenti vantaggi:
  - Lo sviluppatore, una volta configurato il sistema, non deve più curarsi della logica SQL
  - È possibile effettuare in modo immediato il passaggio ad altri DBMS senza dover modificare il codice SQL per renderlo compatibile con il nuovo sistema (es. da PostgreSQL a MySQL)
  - Nel caso di applicazioni sviluppate ex-novo si può far generare automaticamente lo schema SQL necessario a garantire la persistenza

# Java Persistence Framework



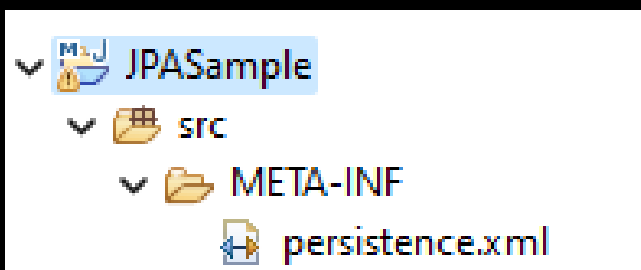
# Java Persistence Framework

- Dopo aver abilitato JPA tra i facets del progetto, occorre aggiungere nel pom.xml la dipendenza dalle librerie Hibernate, oltre alle librerie già previste, tra cui i driver del DBMS che si intende usare:

```
<dependencies>
  <!-- https://mvnrepository.com/artifact/com.mysql/mysql-connector-j -->
  <dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <version>8.0.33</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager -->
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>5.6.15.Final</version>
  </dependency>
</dependencies>
```

# Java Persistence Framework

- Per funzionare correttamente, il sistema necessita di alcuni parametri di configurazione che vengono inseriti nel file META-INF/persistence.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2"
  xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence http://xmlns.jcp.org/xml/ns/persistence/persistence.xml"
  <persistence-unit name="JPASample">
    <properties>
      <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost/javaspring" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.MariaDBDialect" />
      <property name="hibernate.default_schema" value="javaspring" />
      <property name="hibernate.hbm2ddl.auto" value="create" />
    </properties>
  </persistence-unit>
</persistence>
```

# Java Persistence Framework

- Una volta configurati i parametri dell'applicazione è necessario effettuare la mappatura delle classi del modello che devono essere rese persistenti e diventare quindi delle Entities.
- Per effettuare il mapping è possibile utilizzare apposite annotations da inserire nel codice sorgente delle classi, che appartengono al package `javax.persistence`.
- Le annotazioni principali sono:
  - `@Entity` – indica che la classe deve essere gestita come entity
  - `@Table` – indica il nome della tabella DB corrispondente alla classe
  - `@Column` – indica la colonna della tabella a cui far corrispondere un attributo
  - `@Id` – indica l'attributo che deve essere utilizzato come chiave primaria dell'elemento
  - `@Enumerated` – Indica come deve essere mappato un campo corrispondente ad una proprietà di tipo Enum

# Java Persistence Framework

```
@Entity
@Table(name = "sample")
public class SampleEntity {
    private int id;
    private String text;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() {
        return id;
    }

    @Column(name = "content", columnDefinition = "VARCHAR(50)", nullable = false)
    public String getText() {
        return text;
    }
}
```

[illegible]



# Java Persistence Framework

- Quando si implementa una applicazione con entità persistenti possono presentarsi due scenari:
  1. Le tabelle relative alle entities sono già presenti nel database
    - In questo caso occorre utilizzare le annotation in modo da far corrispondere classi e tabelle usando l'annotation @Table e attributi con colonne usando @Column
    - Nel caso in cui non si specifichino tali attributi, JPA cerca una corrispondenza esatta tra nome della classe persistente e tabella e attributo e colonna
  2. Le tabelle relative alle entities non sono ancora state create
    - Questo può avvenire sia per nuove applicazioni che per applicazioni pre-esistenti a cui si aggiungono classi e funzionalità
    - In questo caso, per velocizzare lo sviluppo, si può impiegare la funzione di autogenerazione dei DDL
- La funzione si attiva inserendo nel persistence.xml un'istruzione che indica al sistema di effettuare un controllo sulle tabelle del db e creare o aggiornare le tabelle corrispondenti alle entities in caso di necessità:

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
```

# Java Persistence Framework

- Quando si lavora con le entities in JPA, si stanno manipolando oggetti che possono rappresentare elementi già presenti in un db, oppure elementi che, se salvati, verranno inseriti nel db, passando da uno stato di persistenza all'altro
  - Tali passaggi rappresentano il ciclo di vita (lifecycle) delle entities
- L'insieme di tutti gli oggetti in stato managed rappresenta il **Persistence Context**
  - Se si cerca di caricare dal db una entity che risulta già presente nel Persistence Context, esso la restituirà senza accedere al db, garantendo che non ci sia in memoria più di una istanza della stessa entity collegata ad un persistence context



# Java Persistence Framework

- L'EntityManager è la classe JPA che permette di eseguire le interazioni con il DB
  - viene configurato per mezzo dei parametri presenti nel file persistence.xml e può essere istanziato a partire dalla classe EntityManagerFactory
- Tra i metodi che l'EntityManager espone vi sono:
  - find() – recupera una entity per mezzo della sua chiave primaria
  - createQuery() – crea un'istanza di Query che può essere impiegata per recuperare entities dal DB
  - getTransaction() – definisce una istanza di EntityTransaction per effettuare interazioni con il database. Ha lo stesso funzionamento delle transazioni viste con il JDBC, permettendo di eseguire gruppi di istruzioni in modo atomico e offrendo la possibilità di effettuarne il rollback
  - persist() – aggiunge una entità al persistence context, in modo che quando la transazione verrà committata, l'entità sarà salvata sul db.
  - refresh() – aggiorna lo stato dell'entity allineandolo a quello del db
  - flush() – sincronizza lo stato del persistence context con il database. Viene invocato automaticamente alla commit.
- Ogni EntityManager è associato ad un proprio Persistence Context

# Java Persistence Framework

```
public static void main1(String[] args) {  
    var emf = Persistence.createEntityManagerFactory("JPASample");  
    var em = emf.createEntityManager();  
    try {  
        var data = new SampleEntity().setText("Esempio");  
        var trans = em.getTransaction();  
        trans.begin();  
        em.persist(data);  
        trans.commit();  
    } catch (Exception e) {  
        System.err.println("ERRORE: " + e.getMessage());  
    } finally {  
        em.close();  
        emf.close();  
    }  
}
```

# Java Persistence Framework

```
public static void main(String[] args) {  
    var emf = Persistence.createEntityManagerFactory("JPASample");  
    var em = emf.createEntityManager();  
    try {  
        var data = em.find(SampleEntity.class, 1);  
        System.out.println(data.getText());  
    } catch (Exception e) {  
        System.err.println("ERROR: " + e.getMessage());  
    } finally {  
        em.close();  
        emf.close();  
    }  
}
```

# Java Persistence Framework

- Quando si deve salvare un nuovo elemento è necessario fornire una primary key diversa (chiave di integrità referenziale)
- La generazione delle chiavi primarie, quando esse non vengono scelte tra i campi significativi di un oggetto (*es. il numero di targa o il codice fiscale*) può essere demandata a dei meccanismi automatici, in base al database impiegato (chiavi surrogate)
  - Tali meccanismi permettono al sistema di generare in modo autonomo una chiave univoca per salvare l'oggetto
    - In questo caso, oltre a sollevare lo sviluppatore da un compito, potremo gestire in modo molto più efficiente il salvataggio di oggetti in relazione tra loro

# Java Persistence Framework

- Per generare automaticamente una chiave primaria si può ricorrere a diverse strategie
- scelte in base alle proprie esigenze e soprattutto alla tipologia e versione del DBMS utilizzata (alcuni DBMS offrono funzioni non supportate da altri):
  - Chiavi generate autonomamente dall'applicazione
  - Campi delle tabelle configurati per valorizzarsi con un valore incrementato automaticamente ad ogni inserimento
  - Sequenze
    - Oggetti del db che possono venire interrogati con una apposita query e forniscono un valore sequenziale diverso ad ogni interrogazione
  - Tabelle ad-hoc
    - Il sistema gestisce una tabella in cui memorizza l'ultimo valore precedentemente assegnato agli oggetti e ha la responsabilità di incrementarlo

# Java Persistence Framework

- Tramite l'annotazione `@GeneratedValue`, utilizzata in congiunzione con l'annotazione `@Id` usata per identificare il campo che deve fungere da chiave primaria dell'entità è possibile indicare al sistema che deve gestire in modo automatico la generazione della chiave primaria.
- Se non si fornisce alcun parametro aggiuntivo all'annotazione, il JPA selezionerà una strategia che permetta di avere un identificativo che sia univoco nell'ambito dell'intero database, generando una sequence chiamata `hibernate_sequence` che verrà usata per tutte le entities del database.

```
@Entity
@Table(name = "sample")
public class SampleEntity {
    private int id;
    private String text;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    public int getId() {
        return id;
    }

    @Column(name = "content", columnDefinition = "VARCHAR(50)", nullable = false)
    public String getText() {
        return text;
    }
}
```



# Java Persistence Framework

- L'annotazione `@GeneratedValue` permette di definire una strategia di generazione, che indica al JPA in che modo gestire la valorizzazione della chiave univoca dell'entità e che supporta le possibili modalità di gestione introdotte precedentemente :

- AUTO
- IDENTITY
- SEQUENCE
- TABLE

```
@Entity
@Table(name = "automobile")
public class Automobile {
    @Id
    @Column(name = "id")
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(name = "targa")
    private String targa;

    //...
}
```

# Java Persistence Framework

- Mentre risulta sempre conveniente definire una chiave primaria separata dalle proprietà di modello di una entity, a volte si deve lavorare con database preesistenti o che presentino particolari vincoli che obbligano ad utilizzare chiavi composte, formate cioè da più campi, in cui è la combinazione dei valori nei campi ad essere univoca nel database.
- In questo caso è necessario creare una nuova classe che mappa la combinazione dei campi.

```
@Entity
@Table(name = "automobile")
@IdClass(AutomobileIdClass.class)
public class AutomobileComposite {
    @Id
    @Column(name = "targa")
    private String targa;

    @Id
    @Column(name = "nazione")
    private String nazione;

    // ..
}
```

```
public class AutomobileIdClass implements Serializable {

    protected String targa;
    protected String nazione;

    public AutomobileIdClass() {}

    public AutomobileIdClass(String targa,
                               String nazione) {
        this.targa = targa;
        this.nazione = nazione;
    }
    // equals, hashCode

}
```

# Java Persistence Framework

Sia nei database relazionali che nel paradigma Object Oriented è possibile stabilire relazioni tra entità

Esistono varie tipologie di relazioni, definite dal numero di elementi presenti ad ogni capo della relazione stessa:

- ❖ One-to-One (uno-a-uno)
- ❖ One-to-Many (uno-a-molti)
- ❖ Many-to-Many (molti-a-molti)

JPA permette di mappare queste relazioni automatizzandone la gestione

# Java Persistence Framework

- Le relazioni uno-a-uno, in Hibernate, possono essere di due tipi:
  - Component o embedded - due oggetti risiedono su una sola tabella (es. l'indirizzo di una azienda)
    - quando un'entità è interamente racchiusa all'interno di un'altra entità, ossia è un componente dell'altra entità
  - Classiche - due tabelle corrispondono a due oggetti
    - si usa nei casi in cui è preferibile avere due tabelle che rappresentano due aspetti differenti di una certa entità che devono essere gestiti differentemente, o che devono evolvere in modo indipendente (Es. Profilo Autenticazione e Anagrafiche)

# Java Persistence Framework

```
public class User {  
    // ...  
    @Embedded  
    public Address address;  
}  
  
@Embeddable  
public class Address{  
    private String via;  
    private String civico;  
    private String localita;  
    // ...  
}
```

```
public class User {  
    // ...  
    @OneToOne  
    private Address address;  
}  
@Entity  
public class Address{  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String via;  
    private String civico;  
    private String localita;  
    // ...  
}
```

# Java Persistence Framework

- L'annotazione `@OneToOne` può avere i seguenti attributi:
  - `cascade`: indica se le operazioni su una entità devono propagarsi alle altre della relazione
  - `fetch`: indica il tipo di acquisizione, "eager" (carica tutti gli oggetti del grafo delle relazioni) o "lazy" (carica gli oggetti correlati solo se servono)
  - `optional`: serve per stabilire se ci possono essere NULL
- La relazione `OneToOne` è bidirezionale quando l'oggetto A contiene un riferimento all'oggetto B, e l'oggetto B contiene un riferimento
  - all'oggetto A (Es. `User.getAddress()` e `Address.getUser()`)
- In questo caso uno dei due oggetti deve essere considerato principale: quello la cui tabella contiene la chiave primaria dell'altro come riferimento
- L'oggetto secondario dovrà usare l'attributo "mappedBy" nella sua annotazione `@OneToOne`

```
@Entity
public class User {
    //User è l'oggetto "principale"
    @OneToOne
    private Address address;


    // ...
}

@Entity
public class Address{
    //Address è l'oggetto "secondario"
    //il "mappedBy" si riferisce alla proprietà dell'oggetto principale
    @OneToOne(mappedBy=address)
    public User user;

    // ...
}
```



# Java Persistence Framework

- Per default, Hibernate cercherà come foreign key un campo della tabella con queste caratteristiche:
    - Inizia con il nome della proprietà dell'oggetto principale (User ha getAddress(), quindi “address”)
    - Concatenato con underscore “\_”
    - Concatenato con il nome della chiave primaria dell'oggetto secondario (Es, Address ha getId(), quindi “id”)
  - È possibile sovrascrivere questo default con due tipi di annotazioni:
    - @PrimaryKeyJoinColumn: usato assieme a @OneToOne sull'oggetto principale, indica che la foreign key è la stessa chiave primaria (ovvero, le due tabelle possiedono la stessa chiave primaria)
    - @JoinColumn(name=“address\_fk”): usato assieme a @OneToOne sull'oggetto principale, indica che la foreign key sta in una colonna con un nome preciso (in questo caso “address\_fk”)
- 

# Java Persistence Framework

- Hibernate consente di propagare le operazioni (es. salvataggio o eliminazione) effettuate su una entità alle entità ad essa collegate, tramite il meccanismo del cascading
- Hibernate definisce diversi tipi di cascading, definiti nelle relazioni:
  - CascadeType.MERGE: propaga gli UPDATE
  - CascadeType.PERSIST: propaga il primo inserimento (INSERT)
  - CascadeType.REFRESH: propaga l'aggiornamento dal database verso gli oggetti (SELECT)
  - CascadeType.DETACH: propaga la rimozione dell'oggetto dalla persistenza
  - CascadeType.REMOVE: propaga la rimozione dei dati dell'oggetto (DELETE)
  - CascadeType.ALL: tutti i precedenti
- Se non si specifica niente, per default non si ha cascading
- Esso può essere impiegato su sia su relazioni @OneToOne che su relazioni di altro tipo descritte



# Java Persistence Framework

- La relazione uno-a-molti può essere vista da due “prospettive” diverse:
  - L'oggetto A ha una relazione con molti oggetti B
  - Ogni oggetto B ha una relazione con uno e un solo oggetto A
- Hibernate mappa queste due prospettive usando @OneToMany e @ManyToOne:
  - @OneToMany va sull'oggetto che contiene
  - @ManyToOne va sull'oggetto contenuto
- JPA considera, per convenzione, “oggetto principale” quello che ha @ManyToOne e quindi si dovrà specificare un mappedBy sulla @OneToMany

```
@Entity
public class Telephone{
    //Telephone è l'oggetto "principale"
    @ManyToOne
    public User user;
    // ...
}

@Entity
public class User {
    //User è l'oggetto "secondario"
    @OneToMany(mappedBy="user")
    public Set<Telephone> listaTelefono;

    // ...
}
```

# Java Persistence Framework

- Anche nel caso della relazione One-to-many valgono le stesse considerazioni di JoinColumn fatte nel caso One-to-One: nell'esempio Hibernate cercherà una colonna chiamata "user\_id" nella tabella "telephone" (oggetto principale). Possiamo però specificare un'altra colonna utilizzando @JoinColumn insieme a @ManyToOne nell'oggetto principale

```
@Entity
public class Telephone{
    //Telephone è l'oggetto "principale"
    @ManyToOne
    @JoinColumn(name = "id_user")
    public User user;
    // ...
}
```

# Java Persistence Framework

- Quando vengono mappate delle Collection, è possibile definire una colonna sulla quale queste verranno ordinate
- Si può aggiungere a @OneToMany l'annotazione @OrderBy, che permette di specificare quale proprietà dell'entità collegata usare per l'ordinamento

```
@Entity
public class User {
    //User è l'oggetto "secondario"
    @OneToMany(mappedBy="user")
    @OrderBy("prefisso ASC")
    public List<Telephone> listaTelefono;

    // ...
}
```

# Java Persistence Framework

- In una relazione Many-to-Many entrambi gli oggetti coinvolti utilizzeranno @ManyToMany: uno dei due deve essere l'oggetto principale, l'altro conterrà l'attributo "mappedBy"
- Una relazione multi-a-molti nel modello relazionale ha bisogno di una tabella ausiliaria che contiene le foreign key di entrambe le tabelle
- La tabella ausiliaria ha, per default, il nome Tabellaprincipale\_Tabellasecondaria e chiavi Tabellaprincipale\_id e Tabellasecondaria\_id. Questi default possono essere sovrascritti con l'annotazione @JoinTable

```
@Entity
public class Studente {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany(mappedBy = "studenti")
    private Set<Corso> corsiSeguiti;

    // ...
}

@Entity
public class Corso {
    @Id
    @GeneratedValue
    private Long id;

    @ManyToMany
    private Set<Studente> studenti;

    // ...
}
```

# Java Persistence Framework

- Il linguaggio utilizzato nella scrittura di query JPA, pur essendo molto simile nella struttura e nella sintassi all'SQL, è strettamente legato al modello di oggetti definito dal paradigma Object Oriented
- Gli elementi a cui si fa riferimento quando si scrivono queries in JPQL sono classi e proprietà, invece di tabelle e colonne
- È lo strato ORM che si occupa della traduzione automatica delle queries in formato SQL standard, utilizzando i metadati forniti dalle annotazioni JPA e ottimizzando le strutture in base al tipo di DBMS utilizzato

# Java Persistence Framework

- Una path expression è una variabile identificatrice seguita dall'operatore di navigazione "." e da un campo persistente o un campo associazione, quest'ultimo può contenere un singolo oggetto o una collezione
- Il campo associazione che rappresenta associazioni uno a molto o molto a molti è tipo collection
- Per esempio se si ha una relazione molti a molti fra Associazione e Persona, si può scrivere la seguente query:
  - `SELECT distinct p FROM Persona p WHERE p.associazioni is NOT EMPTY`
- In questo caso p.associazioni è un tipo collection quindi l'espressione è detta collection-value expression
- Se l'associazione fosse stata uno a uno o molti a uno allora l'espressione sarebbe stata una single-value path expression
- È possibile navigare attraverso campi persistenti o associazioni usando single-value path expression:
  - `p.info.info1`
- Non è possibile usare path expression per navigare attraverso tipi collection

# Java Persistence Framework

- Una condizione espressa nella clausola WHERE che filtra i risultati della query è detta espressione condizionale
- È possibile costruire espressioni condizionali usando path expression e operatori supportati dal linguaggio
- JPQL può valutare un'espressione con numeri, stringhe, booleani usando operatori relazionali. Un esempio di espressione condizionale è il seguente:
  - `c.nome = 'libri'`
- È possibile usare l'operatore BETWEEN in espressioni aritmetiche per confrontare una variabile con un range di valori
- L'operatore IN invece permette di verificare se il valore di una path expression appartiene o meno ad una lista di valori:
  - `path_expression [NOT] IN (List_of_values)`

# Java Persistence Framework

- L'operatore LIKE consente di determinare se una single-value path expression corrisponde ad una pattern string. La sintassi dell'operatore è la seguente:
  - `string_value_path_expression [NOT] LIKE pattern_value`
- Il `pattern_value` può contenere i valori `_` e `%`: `_` sta per un singolo carattere mentre `%` rappresenta un numero qualsiasi di caratteri
- È possibile usare `IS NULL` o `IS NOT NULL` per stabilire se una single-value path expression contiene valori null o non null:
  - `WHERE c.campo IS NOT NULL`



# Java Persistence Framework

- Nel caso di un collection type, non è possibile usare IS NULL ma JPQL fornisce gli operatori IS EMPTY o IS NOT EMPTY:
  - WHERE c.articoli IS EMPTY
- È possibile usare l'operatore MEMBER OF per testare se una variabile identificatore, una single-value path expression o un parametro di input appartiene ad una collection-value path expression:
  - entity\_expression [NOT] MEMBER [OF] collection\_value\_path\_expression

# Java Persistence Framework

- Le aggregazioni sono utili quando si scrivono query con l'obiettivo di collezionare entità
- JPQL fornisce le funzioni di aggregazione AVG, COUNT, MAX e MIN che possono essere usate solo su campi persistenti
- Se si desidera ad esempio trovare il valore massimo per il campo prezzo fra tutti gli articoli si può scrivere:
  - `SELECT MAX(a.prezzo) FROM Articolo a`
- Se si vuole conoscere il numero di categorie:
  - `SELECT COUNT(c) FROM Categoria c`

# Java Persistence Framework

- In una applicazione si potrebbe avere la necessità di raggruppare i dati in base a qualche dato persistente usando la clausola GROUP BY
- Assumendo che esista una relazione uno a molti tra Articolo e Categoria, la seguente query genererà un report del numero di categorie create da ogni utente:
  - `SELECT a.categoria, COUNT(a.id) FROM Articolo a GROUP BY a.categoria`
- È possibile raggruppare per single value path expression che sono persistenti o campi associazione, inoltre quando si usa GROUP BY si possono usare solo funzioni di aggregazione
- È possibile inoltre filtrare ulteriormente i risultati mediante la clausola HAVING
- Si supponga ad esempio di voler recuperare gli Utenti che hanno creato più di 5 categorie:
  - `SELECT c.utente, COUNT(c.id) FROM Categoria c GROUP BY c.utente HAVING COUNT(c.id) > 5`

# Java Persistence Framework

- È possibile controllare l'ordinamento dei valori mediante la clausola ORDER BY:
  - ORDER BY path\_expression1 [ASC | DESC], ... path\_expressionN [ASC | DESC]
- Se ad esempio si desidera recuperare tutte le entità Categoria e ordinarle alfabeticamente per nome:
  - SELECT c FROM Categoria c ORDER BY c.nome ASC
- Specificando ASC si indica di ordinare i risultati in ordine ascendente, mentre DESC specifica l'opposto
- È possibile utilizzare più campi per effettuare l'ordinamento a parità di valore:
  - SELECT c FROM Categoria c ORDER BY c.nome ASC, c.titolo DESC
- La clausola SELECT deve contenere le path expression usate dalla clausola ORDER BY

# Java Persistence Framework

- Una sottoquery è una query dentro una query
- È possibile usare sottoquery con le clausole WHERE o HAVING per filtrare i risultati, ma diversamente da SQL non è possibile usarle nella clausola FROM. Se si ha una sottoquery questa viene valutata per prima e quindi la query principale viene valutata su questa
- La sintassi generale di una sottoquery è:
  - [NOT] IN / [NOT] EXISTS / ALL / ANY / SOME (subquery)
- IN si usa per valutare se una single-value path expression appartiene ad una lista di valori:  

```
SELECT a FROM Articolo a WHERE a.utente IN (SELECT c.utente FROM Categoria c WHERE c.nome LIKE :nome)
```

  
EXISTS verifica se la sottoquery contiene qualche valore restituendo true o false altrimenti:  

```
SELECT a FROM Articolo a WHERE EXISTS (SELECT c FROM Categoria c WHERE c.utente = a.utente)
```
- ANY, ALL e SOME sono simili all'operatore IN e possono essere usati con qualsiasi operatore di confronto numerico come =, >, >=, <, <=, <>:
  - ```
SELECT c FROM Categoria c WHERE c.data >= ALL (SELECT a.data FROM Articolo a WHERE a.utente = c.utente)
```
- L'operatore ALL produce true se tutti i risultati prodotti dalla sottoquery soddisfano la condizione data. SOME e ANY sono equivalenti restituiscono true se almeno un risultato recuperato dalla sottoquery soddisfa la condizione

# Java Persistence Framework

- Esercitazione

- Si implementi un'applicazione Java che possa caricare su un apposito database le entità per la gestione di città e province italiane lette da un file negli esercizi precedenti

# Spring Data

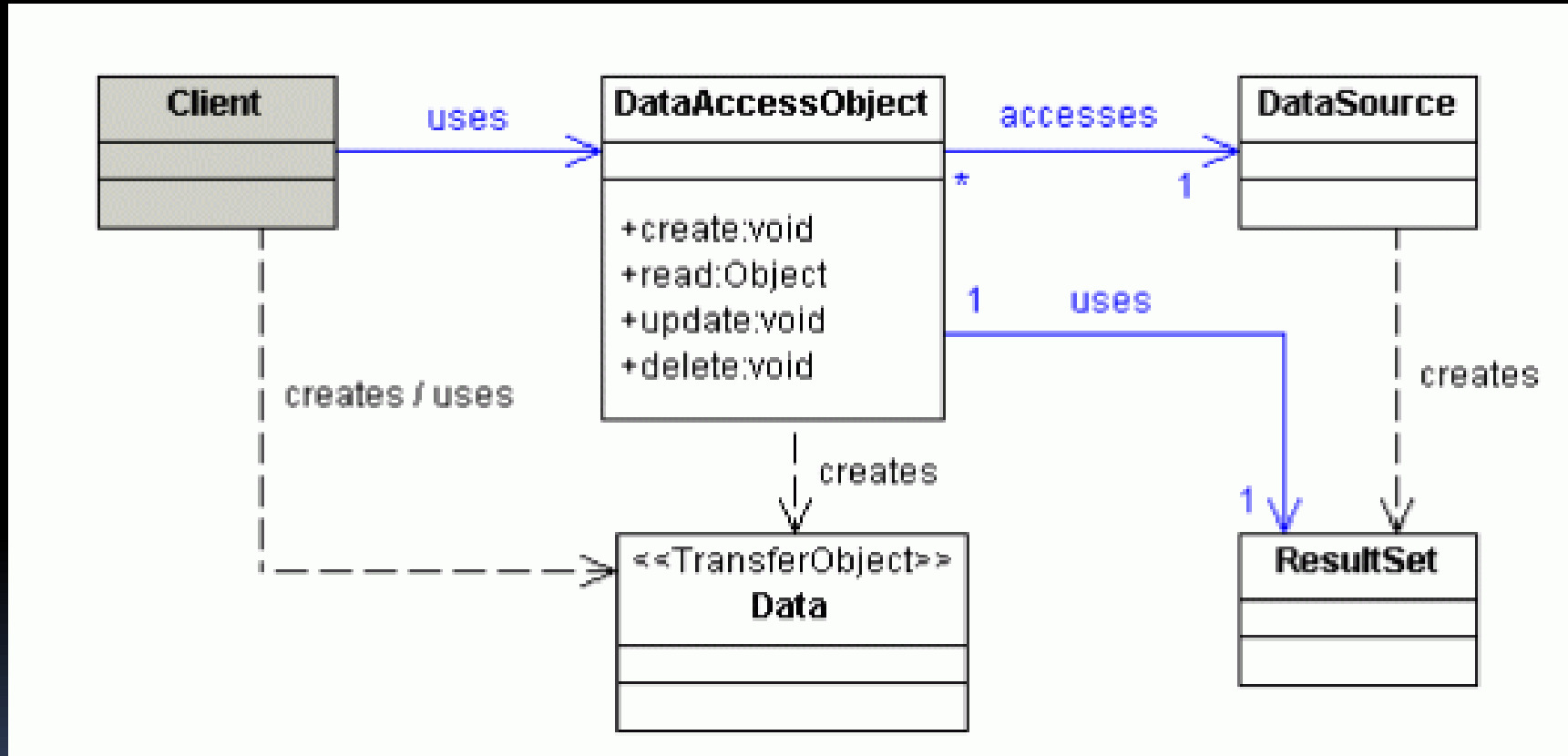
- Uno dei moduli principali offerti dal framework Spring è Spring Data
  - ha il compito di fornire allo sviluppatore tutti gli strumenti necessari ad implementare la gestione della persistenza impiegando i meccanismi e i concetti su cui si basa Spring, come la Dependency Injection
  - In questo modo è possibile realizzare applicazioni con gestione dei dati persistenti in modo molto rapido e standardizzato
    - favorendo la collaborazione di più sviluppatori sui progetti e che aumenta la manutenibilità del codice sorgente.
- Spring Data supporta due modalità principali di interazione con il database:
  - JDBC
  - JPA

# Spring Data

- Per rendere il codice modulare e favorirne il riuso, Spring Data promuove l'impiego del pattern DAO (Data Access Object)
- che ha lo scopo di separare le logiche di business da quelle di accesso e gestione dei dati
- L'adozione del pattern DAO basato su interfacce permette di cambiare i meccanismi di accesso ai dati, passando ad esempio da JDBC a JPA , senza impattare sulla logica applicativa



# Spring Data

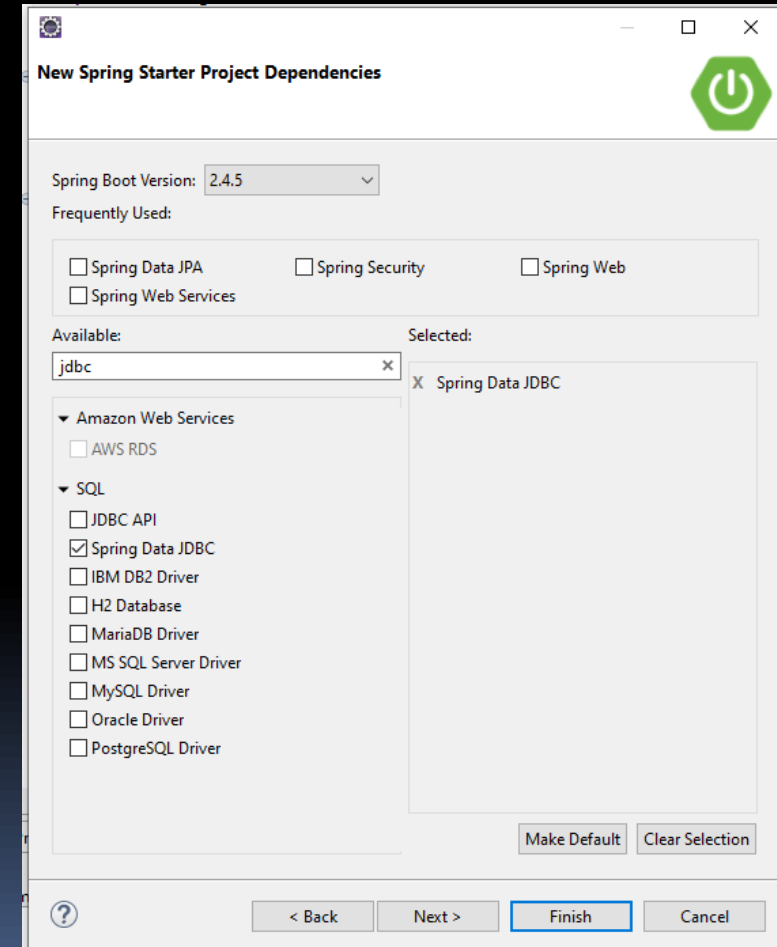


# Spring Data

- La forma più semplice di gestione della persistenza in Java è senz'altro JDBC
  - che permette di effettuare operazioni sui DBMS in modo rapido e concettualmente molto vicino alla modalità di lavoro dei db, utilizzando queries e istruzioni di insert, delete e update
- Per supportare lo sviluppatore nell'implementazione di applicazioni basate su JDBC, Spring Data offre il pacchetto Spring JDBC, che si fa carico della gestione di aspetti come:
  - Apertura della connessione
  - Creazione ed esecuzione dei PreparedStatement
  - Estrazione dei risultati dal ResultSet
  - Gestione delle eccezioni
  - Chiusura della connessione

# Spring Data

- Per creare un nuovo progetto Spring Boot che faccia uso del modulo Spring JDBC è possibile utilizzare lo Spring Initializr selezionando Spring Data JDBC.
- Alternativamente, nel caso di progetti preesistenti, è possibile indicare nel file pom.xml la dipendenza dal modulo spring-boot-starter-data-jdbc.
- Occorre inoltre aggiungere il driver JDBC per il DBMS utilizzato.



# Spring Data

- Una volta impostato il progetto, è necessario definire i parametri per l'accesso al database che verranno utilizzati da Spring JDBC per comunicare con il DBMS.
  - Questi parametri possono essere indicati nel file `application.properties`.
- Una volta completata la configurazione è possibile passare all'implementazione della persistenza.

# Spring Data

- Lo strumento principale del modulo Spring JDBC è il JDBC Template.
- Questa classe si basa sul pattern Template Methods e permette di implementare l'intero processo di accesso ai dati facendo l'override di metodi specifici.
- Il JDBC Template permette di:
  - Aprire e chiudere le connessioni
  - Eseguire statements
  - Iterare sui ResultSet e restituire i risultati delle queries

# Spring Data

```
public class Automobile {  
  
    private Integer id;  
    private String targa;  
    private int annoProduzione;  
    private String colore;  
    private String marca;  
  
    // Setter e gett  
}  
  
public interface AutomobileDao {  
  
    public void insert(Automobile auto);  
    public void update(Automobile auto);  
    public void delete(Long id);  
    public Automobile getById(Long id);  
    public List<Automobile> getByAnnoProduzione(int annoProduzione);  
  
}
```

# Spring Data

```
@Component
public class AutomobileJdbcDao implements AutomobileDao {

    JdbcTemplate jdbcTemplate;

    @Autowired
    public AutomobileJdbcDao(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @Override
    public void insert(Automobile auto) {
        jdbcTemplate.update("insert into automobile.automobile (id, targa, marca, annoproduzione, colore) values (?, ?, ?, ?, ?)",
            auto.getId(), auto.getTarga(), auto.getMarca(), auto.getAnnoProduzione(), auto.getColore());
    }

    @Override
    public void update(Automobile auto) {
        jdbcTemplate.update("update automobile.automobile set targa=?, marca=?, annoproduzione=?, colore=? where id=?",
            auto.getTarga(), auto.getMarca(), auto.getAnnoProduzione(), auto.getColore(), auto.getId());
    }

    @Override
    public void delete(Long id) {
        jdbcTemplate.update("delete from automobile.automobile where id=?", id);
    }

    // ...
}
```

# Spring Data

- Per consentire alla classe Dao di convertire i resultset provenienti dal db in oggetti, Spring permette di implementare apposite classi, definite RowMapper.
- Le classi RowMapper vengono spesso definite come classi interne delle classi Dao in cui vengono utilizzate, ma possono essere anche definite come classi autonome per essere impiegate facilmente in più contesti.

```
public class AutomobileRowMapper implements RowMapper<Automobile> {  
    public Automobile mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Automobile auto = new Automobile();  
        auto.setId(rs.getLong("id"));  
        auto.setTarga(rs.getString("targa"));  
        auto.setAnnoProduzione(rs.getInt("annoproduzione"));  
        auto.setColore(rs.getString("colore"));  
        auto.setMarca(rs.getString("marca"));  
  
        return auto;  
    }  
}
```



# Spring Data

```
@Component
public class AutomobileJdbcDao implements AutomobileDao {

    JdbcTemplate jdbcTemplate;

    @Autowired
    public AutomobileJdbcDao(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

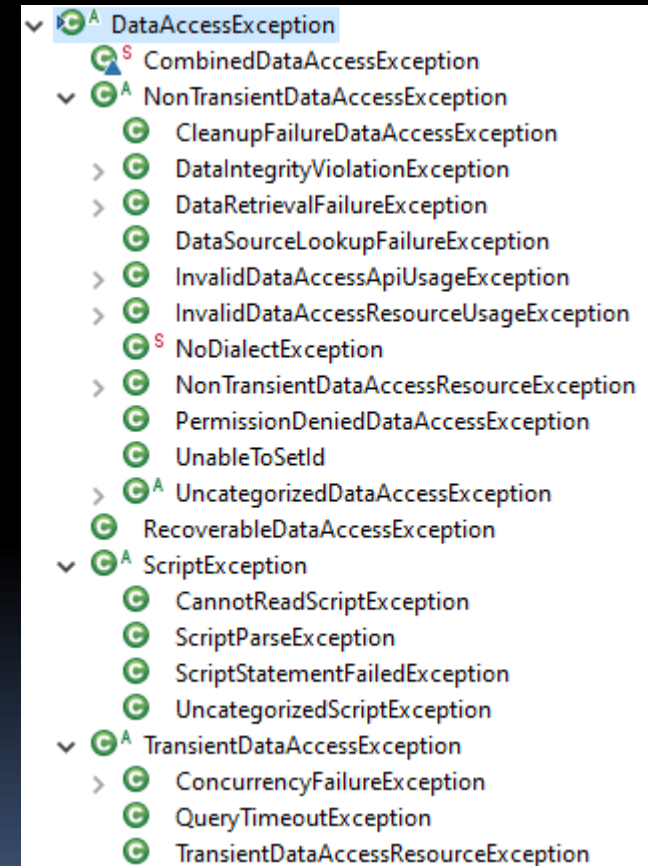
    @Override
    public Automobile getById(Long id) {
        Automobile auto =
            jdbcTemplate.queryForObject("select * from automobile.automobile where id = ?",
                new AutomobileRowMapper(), id);
        return auto;
    }

    @Override
    public List<Automobile> getByAnnoProduzione(int annoProduzione) {
        List<Automobile> result =
            jdbcTemplate.query("select * from automobile.automobile where annoproduzione = ?",
                new AutomobileRowMapper(), annoProduzione);
        return result;
    }

    // Altri metodi della classe Dao
}
```

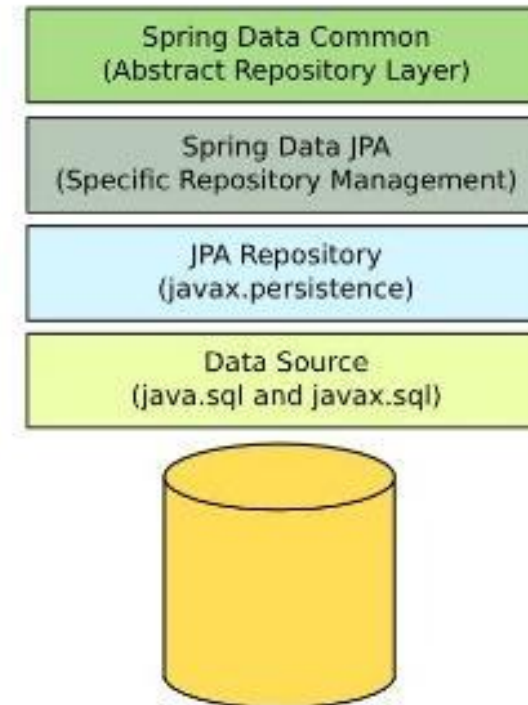
# Spring Data

- Spring fornisce una gerarchia di eccezioni personalizzata per la gestione dei problemi inerenti alla persistenza. Quando viene generata una eccezione, ad esempio per un errore del database, il sistema incapsula automaticamente l'eccezione specifica in una eccezione del framework rendendo più semplice la gestione centralizzata delle eccezioni e permettendo di modificare il DBMS utilizzato senza dover apportare modifiche al codice applicativo.
- L'elemento principale della gerarchia di eccezioni è `DataAccessException`.



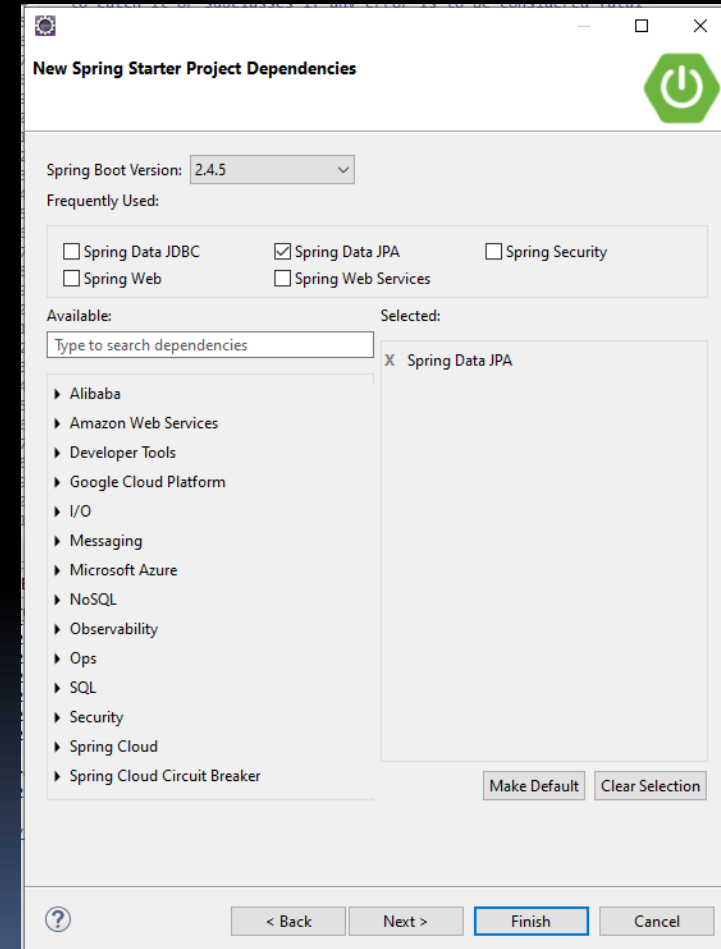
# Spring Data

- Oltre all'impiego della tecnologia JDBC, Spring offre il pieno supporto al JPA per implementare la persistenza con un approccio basato sull'Object Relational Mapping.
- È importante notare che Spring JPA non implementa un JPA Provider, ma si appoggia su implementazioni specifiche come Hibernate, che è il provider di default, ciò consente di utilizzare tutte le funzionalità dell'accoppiata JPA/Hibernate all'interno delle applicazioni Spring Boot.



# Spring Data

- Analogamente a Spring JDBC, per impiegare Spring JPA all'interno di una nuova applicazione è sufficiente selezionare il modulo Spring Data JPA nel wizard dell'Initializer.



# Spring Data

- Anche in questo caso si possono specificare all'interno del file `application.properties` tutti i parametri necessari al collegamento con il DBMS.
- Utilizzando Hibernate come JPA Provider è inoltre possibile indicare nel file anche proprietà specifiche di Hibernate: esse verranno applicate automaticamente al Persistence Context.
- È importante notare che, se si impiegano proprietà di configurazione specifiche di un particolare JPA Provider, nel caso in cui si decida di cambiare il provider, sarà necessario modificare anche le proprietà.

# Spring Data

- Come per numerosi altri aspetti, Spring ed in particolare Spring Boot impiega delle convenzioni per ottimizzare i processi di scrittura del codice sorgente.
- Quando si effettua il mapping JPA delle entità di un'applicazione, il sistema applica una notazione denominata underscore-lowercase (o snake-case) per identificare gli elementi del db associati a classi e proprietà:
  - La classe `StazioneDiServizio` viene mappata sulla tabella `stazione_di_servizio`
  - L'attributo `annoProduzione` viene mappato sulla colonna `anno_produzione`
- È sempre possibile specificare una politica di mapping diversa utilizzando esplicitamente le annotazioni JPA, ma se ci si attiene alla suddetta convenzione si risparmia la scrittura di numerose righe di codice.

# Spring Data

```
@Entity
public class Automobile {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String targa;
    private int annoProduzione;
    private String colore;
    private String marca;

    @OneToOne
    private Motore motore;

    // Setter e getter
}
```

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

    private String matricola;
    private Integer cilindrata;
    private Integer numeroCilindri;

    // Setter e getter
}
```

# Spring Data

- L'elemento principale per realizzare la gestione della persistenza in Spring JPA è il repository.
- I repository sono classi che possono essere assimilate come responsabilità alle classi DAO e consentono di disaccoppiare completamente le operazioni di gestione della persistenza degli oggetti dai dettagli di implementazione del provider di persistenza sottostante.
- L'interfaccia **JpaRepository** fornisce tutte le funzioni CRUD utili all'esecuzione di scenari di persistenza complessi.
- Spring offre un meccanismo automatico per l'implementazione dei repository Jpa, che viene attivato definendo una interfaccia che estende **JpaRepository**. Di norma, per ogni entità del dominio applicativo occorre realizzare una interfaccia di questo tipo.
- Una volta implementata l'interfaccia, il framework rende disponibili in modo automatico le operazioni di CRUD standard senza la necessità di scrivere codice.
- Si possono inoltre definire metodi aggizionali che forniscono funzioni più specifiche.



# Spring Data

```
public interface AutomobileRepository extends JpaRepository<Automobile, Long> {  
    // L'implementazione può rimanere vuota  
}  
  
public interface MotoreRepository extends JpaRepository<Motore, Long> {  
    // L'implementazione può rimanere vuota  
}
```

# Spring Data

```
@Component
public class JpaTest implements CommandLineRunner {

    @Autowired
    private AutomobileRepository automobileRepository;

    @Override
    public void run(String... args) {
        // Carica tutti gli elementi
        List<Automobile> findAll = automobileRepository.findAll();

        // Carica l'elemento per Id gli elementi
        Optional<Automobile> find = automobileRepository.findById(1L);

        // Conteggia gli elementi
        long count = automobileRepository.count();

        // Salva l'elemento
        Automobile auto = new Automobile();
        // Set delle proprietà dell'oggetto...
        automobileRepository.save(auto);

        // Elimina l'elemento
        automobileRepository.deleteById(12L);

        // Applica le modifiche al database
        automobileRepository.flush();
    }
}
```

# Spring Data

```
public interface AutomobileRepository extends JpaRepository<Automobile, Long> {  
  
    public List<Automobile> findByTarga(String targa);  
  
    public List<Automobile> findFirstByAnnoProduzione(Integer annoProduzione);  
  
    public List<Automobile> findByAnnoProduzioneAndMarca(Integer annoProduzione, String marca);  
  
    public List<Automobile> findByAnnoProduzioneBetween(Integer min, Integer max);  
  
    public List<Automobile> findByMarcaIn(List<String> marcaList);  
  
    public List<Automobile> findByMotoreCilindrata(Integer cilindrata);  
  
    //...  
  
}
```

# Spring Data

```
public interface AutomobileRepository extends JpaRepository<Automobile, Long> {  
  
    @Query("FROM Automobile WHERE marca in ('Alfa Romeo', 'Ferrari', 'Maserati')")  
    public List<Automobile> findAutoItaliane();  
  
    @Query("SELECT targa FROM Automobile WHERE annoProduzione = :annoProduzione")  
    public List<String> findTargaByAnnoProduzione(Integer annoProduzione);  
  
}
```



# Spring Data

- Esercitazione

- Si implementi un'applicazione Java che possa caricare su un apposito database le entità per la gestione di città e province italiane lette da un file negli esercizi precedenti utilizzando Spring Data JDBC e Spring Data JPA

# ■ ■ ■ Ereditarietà in Spring Data JPA

- Cos'è l'ereditarietà?
  - Un meccanismo di riutilizzo del codice per rappresentare gerarchie di classi
- Perché usarla?
  - Riutilizzo del codice
  - Modellazione dei dati più intuitiva

# Ereritarietà in Spring Data JPA

```
@Entity
public class Employee {
    @Id
    private Long id;
    private String name;
}
```

```
@Entity
public class Manager extends Employee {
    private String department;
}
```

# Strategie

- I dati organizzati in gerarchie ereditarie possono essere resi persistenti attraverso diverse strategie
  - Single Table
  - Joined
  - Table per Class



# Single Table

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "type")
public class Product {
    @Id
    private Long id;
    private String name;
}

@Entity
public class Book extends Product {
    private String author;
}
```

# Single Table

- Strategia:
  - Una singola tabella contiene tutte le entità della gerarchia.
- Vantaggi:
  - Performance delle query elevate.
  - Design semplice.
- Svantaggi:
  - Colonne inutilizzate per alcune entità.


# Joined

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Vehicle {
    @Id
    private Long id;
    private String manufacturer;
}

@Entity
public class Car extends Vehicle {
    private int doors;
}
```



# Joined

- Strategia
    - Tabelle separate per ogni entità, collegate tramite join
  - Vantaggi
    - Dati normalizzati
    - Nessuna ridondanza
  - Svantaggi
    - Query più lente a causa dei join
- 

# Table per Class

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Payment {
    @Id
    private Long id;
    private double amount;
}

@Entity
public class CreditCardPayment extends Payment {
    private String cardNumber;
}
```

# Table per Class

- Strategia
  - Ogni classe concreta ha la sua tabella
- Vantaggi
  - Nessuna ridondanza e nessun join
- Svantaggi
  - Query polimorfiche complesse
    - Scompaiono i join ma sono necessarie delle UNION

# MappedSuperclass

- Cos'è
  - Una classe base che fornisce proprietà e comportamento comuni alle sottoclassi, ma che non è una vera e propria entità JPA
  - Non viene mappata a una tabella nel database.
  - Utile per eliminare ridondanze nei campi comuni
    - ad esempio id, createdAt, ecc.

# MappedSuperclass

- Caratteristiche
  - Definita con l'annotazione @MappedSuperclass
  - Le proprietà definite sono ereditate dalle sottoclassi
    - Compresa le caratteristiche di persistenza sul database
  - Le sottoclassi sono entità complete e mappate sul database
- Quando usarla
  - Generalmente quando non si intende mappare la classe base su una tabella



# MappedSuperclass

```
@MappedSuperclass  
public abstract class BaseEntity {
```

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

```
    @Temporal(TemporalType.TIMESTAMP)  
    private Date createdAt;
```

```
    @Temporal(TemporalType.TIMESTAMP)  
    private Date updatedAt;
```

```
@Entity  
public class User extends BaseEntity {  
    private String username;  
    private String email;  
  
    // Other fields, constructors, getters, and setters  
}
```

```
@Entity  
public class Product extends BaseEntity {  
    private String name;  
    private double price;  
  
    // Other fields, constructors, getters, and setters  
}
```

# JPQL

- **Cos'è JPQL?**

- Java Persistence Query Language (JPQL)
- Linguaggio di query orientato agli oggetti simile a SQL
- Utilizzato per interagire con database in applicazioni JPA

- **Perché usare JPQL?**

- Query indipendenti dal database specifico
- Migliore integrazione con gli oggetti Java
- Supporto per Named Queries e Dynamic Queries

# JPQL

```
@Repository
public interface UtenteRepository extends JpaRepository<Utente, Long> {
    @Query("SELECT u FROM Utente u WHERE u.email = :email")
    Utente trovaPerEmail(@Param("email") String email);
}
```

```
@Query("SELECT u FROM Utente u ORDER BY u.nome ASC")
List<Utente> trovaTuttiOrdinatiPerNome();
```

# JPQL

```
@Query("SELECT u.nome, COUNT(o) FROM Utente u JOIN u.ordini o GROUP BY u.nome")  
List<Object[]> contaOrdiniPerUtente();
```

```
@Query("SELECT u FROM Utente u JOIN u.ordini o WHERE o.importo > :importo")  
List<Utente> trovaUtentiConOrdiniSuperiori(@Param("importo") Double importo);
```

```
@Modifying  
@Query("UPDATE Utente u SET u.email = :nuovaEmail WHERE u.id = :id")  
void aggiornaEmail(@Param("id") Long id, @Param("nuovaEmail") String nuovaEmail);
```

```
@Modifying  
@Query("DELETE FROM Utente u WHERE u.id = :id")  
void eliminaUtente(@Param("id") Long id);
```



User Layer

# MODULO 4

# Spring Web

- Utilizzando il modulo Web MVC è possibile realizzare in modo rapido applicazioni web che implementano il pattern Model View Controller
- L'utilizzo di Spring Web MVC per realizzare web application ha vari vantaggi, tra cui:
  - Chiara separazione dei ruoli
  - Configurazione e gestione uniforme sia del framework che delle classi di modello come Javabeans
  - Adattabilità e flessibilità
  - Riutilizzabilità delle logiche di business
  - Gestione automatica del ciclo di vita e dello scope degli oggetti

# Spring Web

- In un'applicazione web vengono create, tra le risorse, anche due cartelle (inizialmente vuote) per i contenuti statici (static) e dinamici (templates) dell'applicazione
- La classe principale per l'applicazione non cambia

# Spring Web

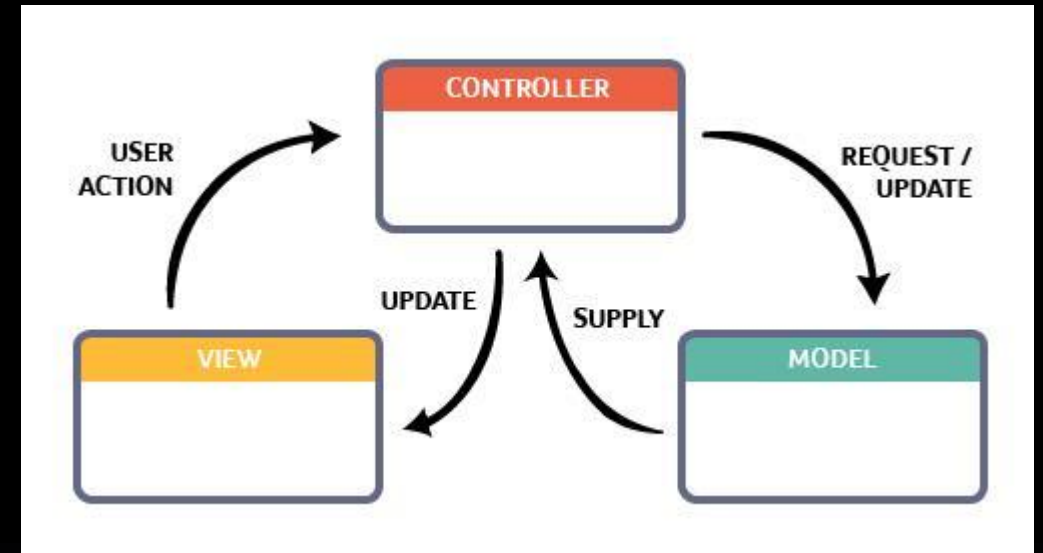
- Il modulo Web MVC prevede che l'applicazione venga fatta funzionare all'interno di un server Tomcat embedded
  - ciò significa che non è necessario associare il progetto ad un server per farlo funzionare, ma è sufficiente lanciarlo come se fosse un semplice eseguibile, il framework si occupa di avviare un server integrato nell'applicazione
- Per impostare i parametri del server embedded, si possono usare delle proprietà all'interno del file application.properties, ad esempio per scegliere la porta su cui il server si metterà in ascolto
  - Il valore di default della porta di ascolto del server embedded è 8080
- È comunque possibile configurare l'applicazione per effettuare il deploy su un server Tomcat stand-alone.

```
# Set the embedded server listening port
server.port=8080
```



# Spring Web

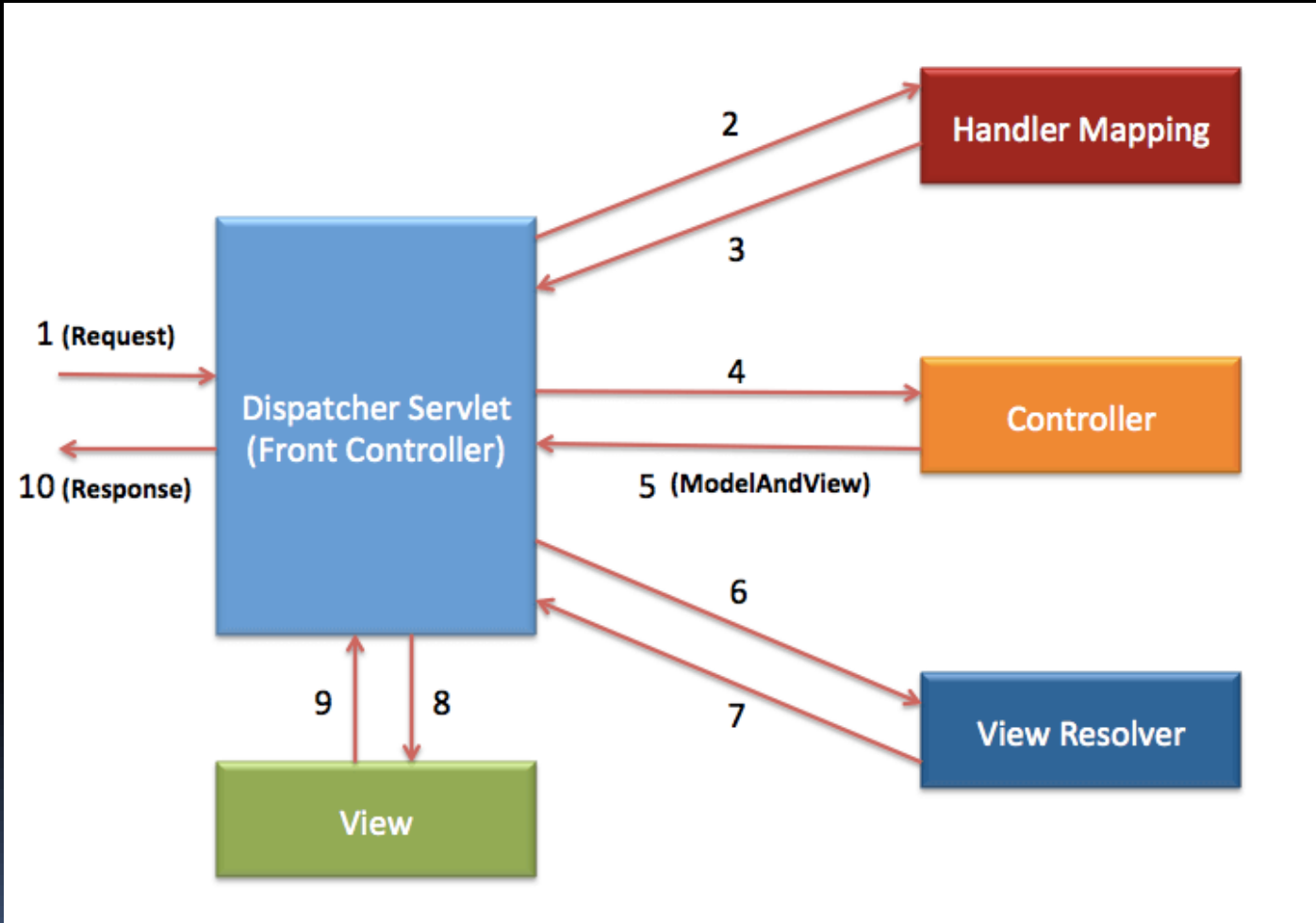
- Il framework Spring Web MVC definisce la struttura e il modello di programmazione delle applicazioni web con Spring – basato sul pattern MVC (Model-View-Controller) e composto da:
  - controller – responsabili di elaborare le richieste web degli utenti
  - modello – responsabile di gestire le informazioni di interesse dell'applicazione
  - viste – responsabili di visualizzare le risposte e le informazioni del modello agli utenti



# Spring Web

- Il modello è una semplice mappa `Map<String, Object>` da attributi a valori – oppure si può usare un oggetto `Model` con operazioni `addAttribute` e `getAttribute`
- Ciascun controller svolge di solito queste attività
  - riceve una richiesta con i suoi parametri
  - elabora la richiesta e popola il modello
  - restituisce il nome della vista da utilizzare per il rendering della risposta
- Ciascuna vista è un template di pagine web
  - le viste possono essere implementate con diverse tecnologie (come JSP e Thymeleaf)
  - il rendering di una vista è di solito basato sulla sostituzione di elementi del template con i valori degli attributi del modello

# Spring Web



# Spring Web

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    public @ResponseBody String hello() {
        return "Hello, welcome!";
    }
}
```

# Spring Web

- L'annotazione `@Controller` indica un tipo di `@Component`, un controller Spring Web MVC, che ha la funzione di ricevere richieste web
- l'annotazione `@GetMapping` associa un metodo del controller web a un'operazione HTTP GET per il path specificato (nell'esempio, `/hello`)
- l'annotazione `@ResponseBody` specifica che il valore restituito dal metodo va interpretato come il contenuto della risposta, altrimenti, in Spring Web MVC, il valore restituito da un metodo di un controller viene interpretato come il nome della vista da visualizzare al termine dell'esecuzione del metodo
- Una richiesta GET `http://localhost:8080/hello` restituisce la stringa Hello, welcome!

# Spring Web

```
@GetMapping("/hello")
public @ResponseBody String hello() {
    return "Hello, welcome!";
}
```

```
@GetMapping("/hello-name")
public @ResponseBody String hello(@RequestParam String name) {
    return String.format("Hello, %s, welcome!", name);
}
```

```
@GetMapping("/hello-param")
public @ResponseBody String hello2(@PathVariable String name) {
    return String.format("Hello, %s, welcome!", name);
}
```

# Spring Web

- Con l'annotazione `@RequestParam` si indica al framework di iniettare, nel metodo del controller mappato per rispondere a una `GET`, un parametro della query string con nome corrispondente al parametro
  - `http://localhost:8080/hello-name?name=Bob`
- Con l'annotazione `@PathVariable`, usata in congiunzione con il token `{nomeelemento}` nel mapping del metodo, si indica al framework di iniettare nel metodo la parte del path della URL corrispondente al token
  - `http://localhost:8080/hello-param/Bob`

# Spring Web

- In Spring MVC le view possono essere realizzate per mezzo di file HTML o JSP, utilizzando appositi tag EL, in modo molto simile a quanto visto nelle Web application classiche
- Spring mette inoltre a disposizione dell'utente diversi template engines
  - Thymeleaf, scansiona i files HTML, li elabora e produce il contenuto che poi viene inviato al client
    - consente di scrivere pagine dinamiche in un formato più leggibile rispetto alle JSP e molto simile all'HTML classico



# Spring Web

- Quando si usa Thymeleaf, il framework interpreta il valore ritornato dai metodi mappati con `@GetMapping` e `@PostMapping` come il nome della vista Thymeleaf che deve essere restituita al client
- Utilizzando la mappa associata al parametro `model` è possibile condividere con la vista i dati necessari al suo corretto funzionamento.

```
@Controller
public class HelloController {

    @GetMapping("/helloThymeleaf/{name}")
    public String hello(Map<String, Object> model, @PathVariable String name) {
        model.put("name", name);
        return "helloPage";
    }

}
```

# Spring Web

- Le viste devono essere salvate nella cartella `src/main/resources/templates` del progetto e possono essere organizzate in sottocartelle
- Si possono utilizzare dei tag standard per manipolare i dati di interesse all'interno dei template

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Sample Thymeleaf</title>
</head>
<body>
<h1>Welcome!</h1>
<p>Hello, <span th:text="${name}"></span>, you are welcome!</p>
</body>
</html>
```

# Spring Web

- Thymeleaf mette a disposizione dello sviluppatore una libreria di tag completa per la manipolazione e la rappresentazione degli oggetti, analoga a quanto già visto per JSTL.
- È ad esempio possibile iterare su collezioni di oggetti:

```
<div th:each="item: ${model.itemList}">  
    <div th:text="${item.name}"></div>  
    <div th:text="${item.value}"></div>  
</div>
```

# Spring Web

- Una componente essenziale nella realizzazione di applicazioni web interattive è la gestione dei form che permettono di acquisire e processare l'input dell'utente
- Il framework mette a disposizione dello sviluppatore strumenti e meccanismi per gestire i form in modo semplice e flessibile
- Per una panoramica sulle funzioni di gestione del form, si farà riferimento ad un esempio basato sulla gestione degli iscritti ad un corso, rappresentati dal seguente oggetto:

```
public class Alunno {  
    private String nome;  
    private long id;  
    private String email;  
  
    // Setters e getters  
}
```

# Spring Web

```
<!DOCTYPE HTML>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<title>Form Alunno</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
  <h1>Benvenuto, inserisci i dettagli dell'alunno</h1>
  <form action="#" th:action="@{/addAlunno}" th:object="${alunno}"
    method="post">

    <p>
      Nome: <input type="text" th:field="*{nome}" />
    </p>

    <p>
      Id: <input type="text" th:field="*{id}" />
    </p>

    <p>
      Email: <input type="text" th:field="*{email}" />
    </p>

    <p>
      <input type="submit" value="Submit" /> <input type="reset"
        value="Reset" />
    </p>
  </form>
</body>
</html>
```

# Spring Web

```
<!DOCTYPE HTML>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<title>Informazioni alunno inserite</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <h1>Informazioni alunno inserite</h1>
    <p>
        Nome: <span th:text="${nome}"></span>
    </p>

    <p>
        Id: <span th:text="${id}"></span>
    </p>
    <p>
        Email: <span th:text="${email}"></span>
    </p>
</body>
</html>
```

# Spring Web

```
<!DOCTYPE HTML>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<title>Errore</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
  <body>
    <h3>Inserire i dati corretti</h3>
    <table>
      <tr>
        <td><a href="alunno">Riprova</a></td>
      </tr>
    </table>
  </body>
</html>
```

# Spring Web

```
@Controller
public class AlunnoController {

    @RequestMapping(value = "/alunno", method = RequestMethod.GET)
    public ModelAndView showForm() {
        return new ModelAndView("formAlunno", "alunno", new Alunno());
    }

    @RequestMapping(value = "/addAlunno", method = RequestMethod.POST)
    public String submit(@Valid @ModelAttribute("alunno") Alunno alunno, BindingResult result, ModelMap model) {
        if (result.hasErrors()) {
            return "error";
        }
        model.addAttribute("nome", alunno.getNome());
        model.addAttribute("email", alunno.getEmail());
        model.addAttribute("id", alunno.getId());
        return "alunnoView";
    }
}
```



# Spring Web

```
@RequestMapping(value = "/addAlunno", method = RequestMethod.POST)
public ModelAndView submitObj(@Valid @ModelAttribute("alunno") Alunno alunno, BindingResult result) {
    if (result.hasErrors()) {
        return new ModelAndView("error", HttpStatus.INTERNAL_SERVER_ERROR);
    }

    ModelAndView modelAndView = new ModelAndView("alunnoView");
    modelAndView.addObject("alunno", alunno);
    return modelAndView;
}
```

# Spring Web

```
<!DOCTYPE HTML>
<html xmlns:th="https://www.thymeleaf.org">
<head>
<title>Informazioni alunno inserite</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>
<body>
    <h1>Informazioni alunno inserite</h1>
    <p>
        Nome: <span th:text="${alunno.nome}"></span>
    </p>

    <p>
        Id: <span th:text="${alunno.id}"></span>
    </p>
    <p>
        Email: <span th:text="${alunno.email}"></span>
    </p>
</body>
</html>
```

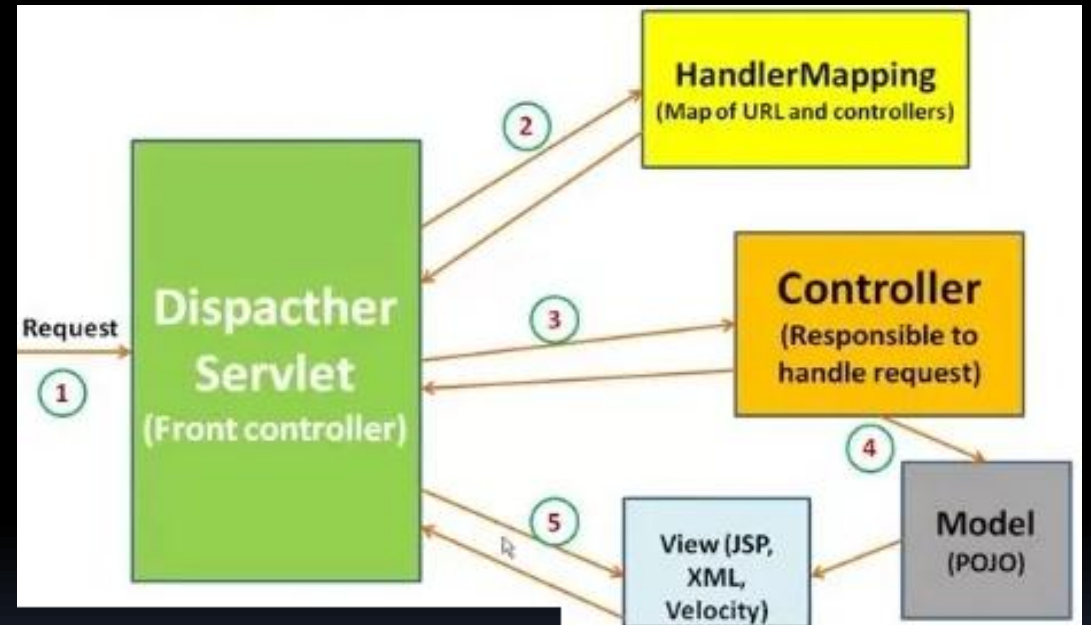


# RESTFUL API



# RESTful API

- L'implementazione di web service REST in Spring Boot ha molte analogie con quanto visto relativamente alla creazione di una web application Spring Boot MVC.
- I servizi web sfruttano infatti la stessa architettura basata su **DispatcherServlet** per instradare le richieste dei client ai componenti configurati per generare la risposta attesa.
- In questo modo è possibile creare Web Services in modo molto rapido scrivendo un numero ridotto di righe di codice: questo ha portato Spring Boot ad essere la piattaforma attualmente più diffusa per realizzare Web Service REST in Java.



# RESTful API

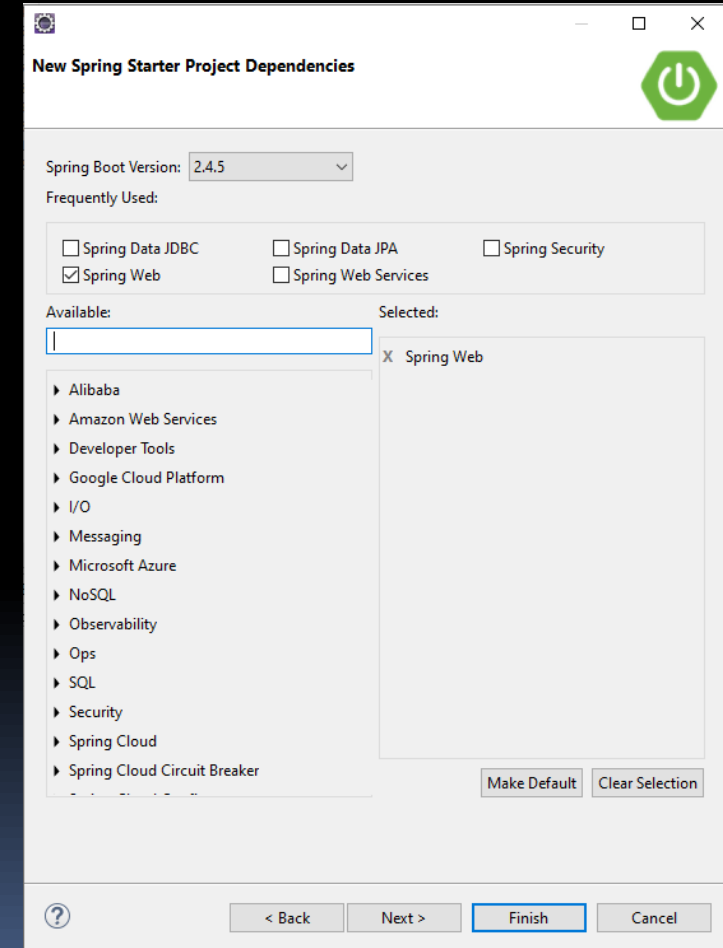
- Per realizzare Web Service REST con Spring Boot è necessario importare il modulo

**spring-boot-starter-web**

- Il modulo

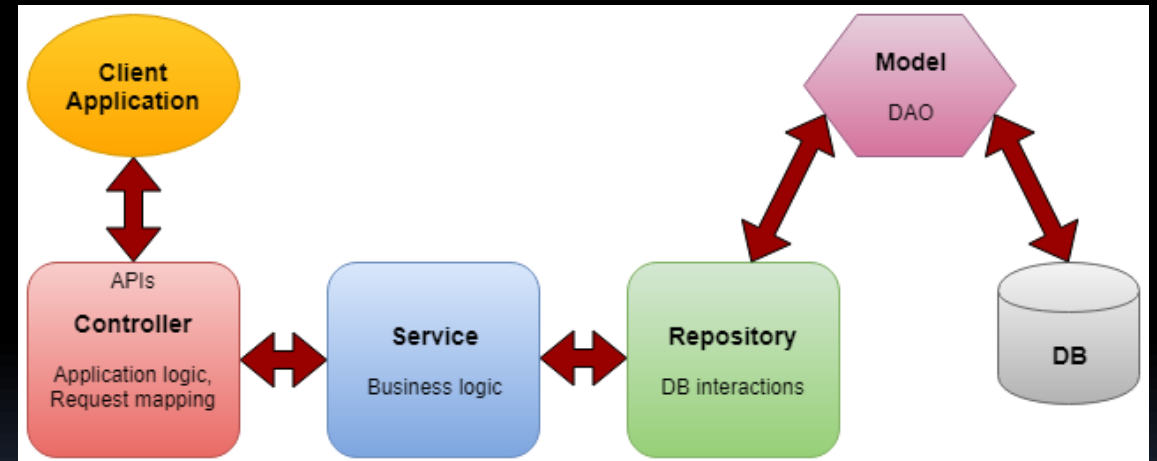
**spring-boot-starter-web-services**

- è invece necessario quando si intende implementare WS in tecnologia SOAP



# RESTful API

- Un pattern architetturale molto diffuso nella realizzazione dei Web Service con Spring Boot è quello che prevede la suddivisione dell'applicazione in quattro strati:
  1. Controller: espone la logica applicativa e mappa le richieste;
  2. Service: implementa la logica di business e fornisce tutte le funzioni di manipolazione degli oggetti del modello necessarie al controller
  3. Repository: si occupa delle interazioni con il database
  4. Model: contiene gli oggetti del dominio applicativo



# RESTful API

- La suddivisione in strati permette di mantenere il codice applicativo ordinato e di definire in modo chiaro le responsabilità di ogni elemento
- Poiché uno degli obiettivi principali dell'architettura REST è offrire funzioni CRUD sugli oggetti del dominio applicativo, generalmente si crea un gruppo di elementi Controller, Service e Repository per ognuno degli elementi principali del dominio stesso
- Questa suddivisione, che si rispecchia nella struttura dei package del progetto, favorisce il lavoro in team e permette di semplificare le attività di sviluppo e manutenzione del codice

# RESTful API

- Quando la richiesta di un client arriva alla DispatcherServlet, quest'ultima analizza il path richiesto e instrada la richiesta verso il controller che dichiara un mapping corrispondente al path. Il framework si occupa di gestire in modo automatico il ciclo di vita del controller e l'iniezione delle dipendenze necessarie al suo corretto funzionamento.
- I controller sono identificati dall'annotation `@Controller` ed un controller può definire uno o più metodi che possono essere mappati su path specifici
- Il mapping dei metodi con i path viene effettuato con le seguenti annotation, che corrispondono ai verbi HTTP:
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`



# RESTful API

- La url corrispondente al metodo si definisce all'interno della notazione di mapping del metodo stesso, ad esempio:
  - `@PostMapping('/path/to/url')`
- Il servizio risponderà alla chiamata `POST http://hostname/path/to/url`
- È inoltre possibile specificare un path di base comune a tutte le chiamate esposte da un controller utilizzando l'annotazione `@RequestMapping` sulla classe Controller:
  - `@RequestMapping("/api")`
- In questo caso il servizio risponderà alla chiamata `http://hostname/api/path/to/url`



# RESTful API

```
@Controller
@RequestMapping("/api")
public class TestController {
    static final Logger logger = LoggerFactory.getLogger(TestController.class);

    @GetMapping("/testGet")
    public @ResponseBody String allAccess() {
        return "Calling GET /testGet";
    }

    @PostMapping("/testPost")
    public @ResponseBody String userAccess() {
        return "Calling POST /testPost";
    }

    @PutMapping("/testPut")
    public @ResponseBody String moderatorAccess() {
        return "Calling PUT /testPut";
    }

    @DeleteMapping("/testDelete")
    public @ResponseBody String adminAccess() {
        return "Calling DELETE /testDelete";
    }
}
```

# RESTful API

- Per semplificare ulteriormente l'implementazione di servizi REST, il framework mette a disposizione dello sviluppatore una ulteriore annotazione per identificare i controller: `@RestController`
- Impiegando questa annotazione non è più necessario utilizzare l'annotazione `@ResponseBody` in corrispondenza dei metodi mappati come chiamate REST, semplificando ulteriormente la scrittura del codice.

```
@RestController
@RequestMapping("/api")
public class TestController2 {
    static final Logger logger =
        LoggerFactory.getLogger(TestController.class);

    @GetMapping("/testGet")
    public String testGET() {
        return "Calling GET /testGet";
    }

    @PostMapping("/testPost")
    public String testPOST() {
        return "Calling POST /testPost";
    }

    @PutMapping("/testPut")
    public String testPUT() {
        return "Calling PUT /testPut";
    }

    @DeleteMapping("/testDelete")
    public String testDELETE() {
        return "Calling DELETE /testDelete";
    }
}
```

# RESTful API

- L'endpoint web service può accettare diverse tipologie di parametri provenienti dalla richiesta effettuata dal client:
  - Parametri da queryString – es. `http://localhost/api/endpoint?param1=value1&param2=value2`
  - Parametri da path – es. `http:// localhost /api/endpoint/pathParam/`
  - Payload della request – es.

```
{  
    "param1": "value 1",  
    "param2": "value 2"  
}
```

- Per ogni tipologia, il framework mette a disposizione un meccanismo per il recupero dei parametri ed il loro successivo impiego nella logica applicativa.

# RESTful API

```
@GetMapping("/testQueryString")
public String testQueryString(@RequestParam String param1, String param2) {

    return "Parametri: " + param1 + " - " + param2;

}

@GetMapping("/testPathParam/{param}")
public String testPathParameter(@PathVariable String param) {

    return "Parametri: " + param;

}
```

# RESTful API

```
@PostMapping("/testBodyString")  
public String testBodyString(@RequestBody String body) {  
    return "Parametri: " + body;  
}
```

# RESTful API

```
@PostMapping(value = "/testBodyPojo", consumes = MediaType.APPLICATION_JSON_VALUE)
public String testBodyPojo(@RequestBody Params params) {

    return "Parametri: " + params.getParam1() + " - " + params.getParam2();

}
```

```
public class Params {
    private String param1;
    private String param2;

    public String getParam1() {
        return param1;
    }

    public void setParam1(String param1) {
        this.param1 = param1;
    }

    public String getParam2() {
        return param2;
    }

    public void setParam2(String param2) {
        this.param2 = param2;
    }
}
```



# RESTful API

```
@GetMapping("/testGet")
@ResponseStatus(HttpStatus.OK)
public String testGET() {
    return "Calling GET /testGet";
}

@GetMapping("/testNonImplementato")
@ResponseStatus(HttpStatus.NOT_IMPLEMENTED)
public String testNonImplementato() {
    // Metodo in corso di implementazione
    return "Work in Progress";
}
```




# RESTful API

- Come già visto in precedenza, gli status code sono una componente del protocollo HTTP che permette di indicare in modo standard l'esito di una richiesta. Spring fornisce vari modi per impostare lo status code di una HTTP response, il più semplice dei quali è l'annotazione `@ResponseStatus`, che si applica sui metodi mappati come chiamate e accetta come argomento un `HttpStatus`. L'enum `HttpStatus` ha molti valori, utili a specificare in modo chiaro per il client l'esito della chiamata, come ad esempio:
  - `HttpStatus.OK`
  - `HttpStatus.CREATED`
  - `HttpStatus.BAD_REQUEST`
  - `HttpStatus.NOT_FOUND`
  - `HttpStatus.NOT_IMPLEMENTED`
- Per i metodi mappati, se non viene specificato altrimenti, il framework ritorna automaticamente
  - `HttpStatus.OK (code 200)`



# RESTful API

- Oltre al body ed al response code, una risposta HTTP è composta anche da intestazioni (headers) che vengono utilizzate per fornire al client metadati aggiuntivi, come tipo del contenuto della risposta , (es. text/plain, charset=UTF-8) lunghezza del contenuto ed altri elementi utili al processing della risposta stessa.
  - Sebbene solitamente la gestione standard degli header effettuata dal framework sia sufficiente, a volte lo sviluppatore ha la necessità di controllare in modo accurato tutti gli elementi della risposta. In questo caso può utilizzare la classe ResponseEntity.
  - La ResponseEntity è un wrapper che permette di specificare al suo interno i tre elementi fondamentali della risposta HTTP:
    - Header
    - Body
    - Response status
- 



# RESTful API

```
@GetMapping("/testResponseEntity")
public ResponseEntity<String> getResponseEntity() {
    HttpHeaders headers = new HttpHeaders();
    headers.add("customHeader", "headervalue");
    String body = "Test della responseEntity";

    ResponseEntity<String> responseEntity = new ResponseEntity<String>(body, headers, HttpStatus.OK);
    return responseEntity;
}
```

# RESTful API

- Il modo più comune ed efficace per inviare dati in una risposta è utilizzare il formato Json.
- Spring supporta pienamente questa funzione permettendo allo sviluppatore di utilizzare solo oggetti POJO nell'implementazione degli endpoint ed occupandosi della conversione degli oggetti in Json, in modo analogo e simmetrico a quanto visto nella gestione delle richieste.
- Poiché questo è l'approccio più diffuso, Spring Boot applica la conversione in Json di default anche quando non viene imposto esplicitamente con l'annotazione: è sufficiente che il metodo mappato restituisca l'istanza di POJO desiderata.

# RESTful API

- Con l'annotazione `@Service` si identificano le classi dello strato Service del pattern architetturale introdotto precedentemente.
- In generale, tutte le interazioni che avvengono tra controller e model dovrebbero essere gestite per mezzo di un componente Service.
- È possibile sfruttare i meccanismi di autowiring e DI di Spring per fornire al controller in modo automatico le istanze di Service necessarie al loro corretto funzionamento.

```
@Service
public class UserService {

    public User getCurrentUser() {
        // ...
    }
}

@RestController
@RequestMapping("/api")
public class UserController {
    @Autowired
    UserService userService;

    @GetMapping("/users/current")
    public ResponseEntity<User> getCurrentUser() {
        User currentUser = userService.getCurrentUser();
        // ...
    }
}
```

# RESTful API

- Per gestire in modo efficace le eccezioni che l'applicazione può generare, ed in particolar modo quelle specifiche definite nel progetto, si può utilizzare il meccanismo basato su `ExceptionHandler`.
- In questo modo lo sviluppatore può implementare un punto centralizzato in cui gestire tutte le eccezioni generate a seguito di una richiesta del client e restituire dei messaggi che contengano tutte le informazioni necessari a comprendere la causa dell'errore: ciò è molto importante per semplificare l'impiego delle API che si realizzano.
- Oltre a questo metodo, Spring offre altri meccanismi per la gestione degli errori, che possono essere utili in contesti o scenari particolari.



# RESTful API

- Utilizzando Junit è possibile implementare dei test che consentono di verificare il corretto funzionamento degli endpoint di un servizio REST. Tali test sono molto importanti perché le API del back-end rappresentano la base su cui vengono costruite le applicazioni client, pertanto degli errori in esse possono portare a blocchi dei sistemi o problematiche di vario tipo.
- La struttura generale dei test degli endpoint è analoga a quella già vista per il test del modello, a cui occorre aggiungere ulteriori elementi per eseguire con successo i test, come configurazione di indirizzi e porte su cui effettuare le chiamate di test ed oggetti in grado di effettuare le chiamate e valutarne il risultato.
- Esistono due modalità principali di test degli endpoint:
  - Esecuzione del servizio sul server: in questo caso durante i test il servizio viene avviato e riceve le chiamate di test in modo identico a quanto avviene in produzione
  - Esecuzione delle chiamate al controller: in questo caso il framework non avvia il servizio ma dialoga direttamente con il controller inviandogli i dati necessari. Questa modalità risulta più veloce nell'esecuzione perché evita l'avvio del server per l'esecuzione dei test

# RESTful API

- La classe `TestRestTemplate` permette di effettuare chiamate alle url su cui sono mappati i servizi e di valutarne la risposta.
- Poiché con questa modalità viene richiesto al sistema di avviare il server, per evitare che ci siano conflitti sulle porte impegnate, si indica di utilizzare una porta casuale ogni volta che viene avviata una istanza di server.

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
public class WsRequestTest {

    @LocalServerPort
    private int port;

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    void testEndpoint() throws Exception {
        assertThat(this.restTemplate.getForObject(
            "http://localhost:" + port + "/api/users/test?param=1",
            String.class)).contains("OK");
    }
}
```



# RESTful API

- La classe MockMvc consente di interagire direttamente con il controller senza la necessità di inizializzare il server.
- Il formato delle chiamate ai metodi del controller mappati sulle URL è identico a quelle effettuate attraverso il server, pertanto è possibile in questo modo realizzare dei test molto più veloci senza perdere efficacia.

```
@SpringBootTest
@AutoConfigureMockMvc
public class ControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void shouldReturnDefaultMessage() throws Exception {
        this.mockMvc.perform(get("/api/users/test?param=1"))
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("OK")));
    }
}
```




# RESTful API

- Esercitazione
- Implementare le CRUD per la gestione delle province italiane sul database





# RESTful API

- Esercitazione
  - Implementare due metodi che forniscano ad una richiesta HTTP l'elenco, ordinato in ordine alfabetico per sigla, di tutte le province italiane e l'elenco, ordinato per denominazione, di tutte le città che si trovano in una determinata provincia.
- 



Configurazione ed Aspetti Avanzati

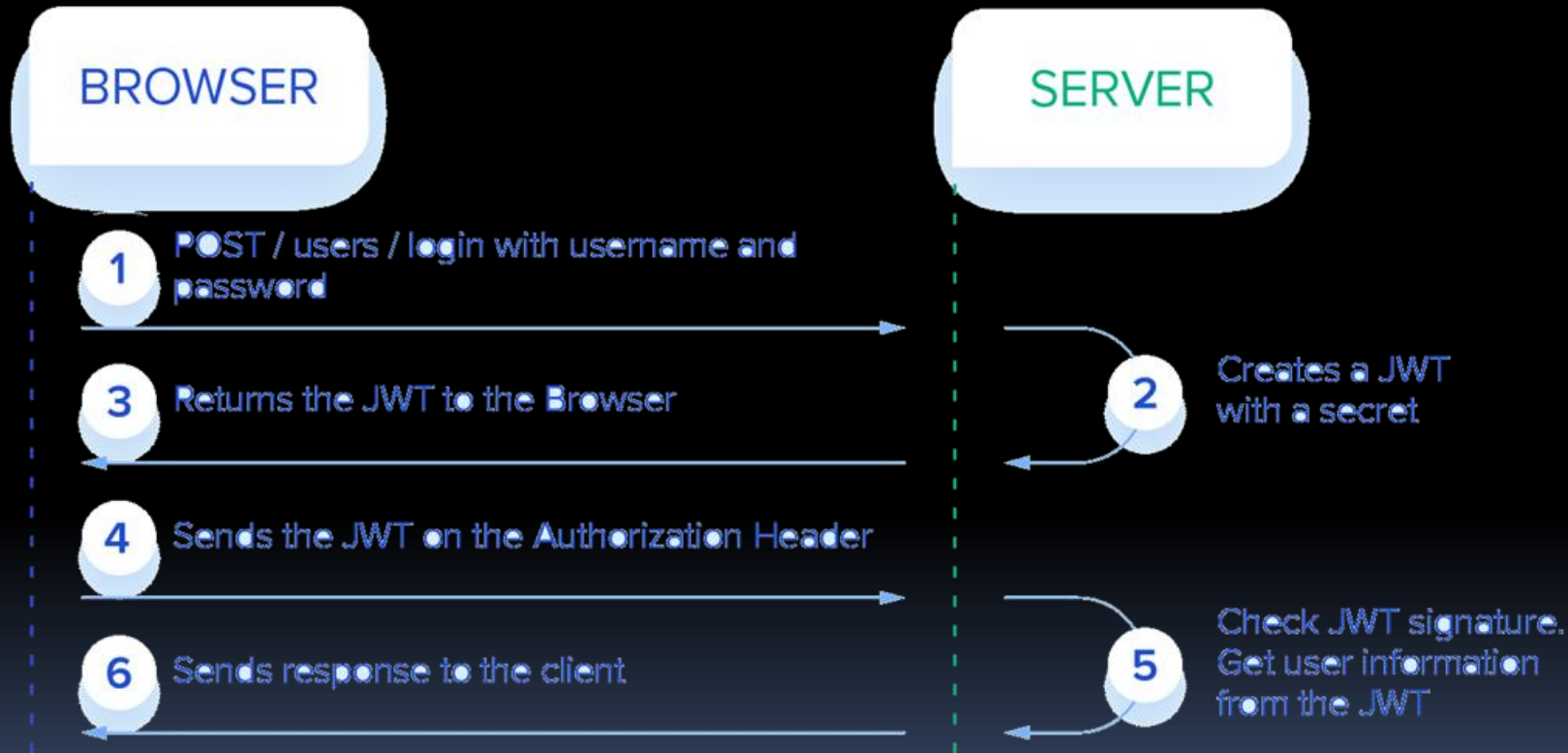
# MODULO 5



# Autenticazione tramite JWT

- Fino a questo momento tutti i web services che sono stati illustrati prevedono un accesso pubblico agli endpoint: chiunque in grado di accedere al server che ospita il Web Service e che sia a conoscenza della URL di un servizio può invocarlo liberamente.
- In genere, soprattutto in ambito enterprise, si necessita di un sistema di sicurezza che permetta di consentire l'accesso agli endpoint dei servizi solo ad utenti autorizzati.
- Inoltre spesso, gli stessi utenti, pur essendo riconosciuti dal sistema, devono poter accedere a risorse differenti in base al proprio ruolo: si pensi ad esempio ad utenti "base" che possono accedere alle funzioni di consultazione dati e ad utenti "avanzati" che invece possono modificare i dati, oppure ad utenti "amministratori" che possono accedere a impostazioni del sistema interdette a tutti gli altri tipi di utente.
- Per rispondere a queste esigenze sono stati sviluppati moltissimi meccanismi di autenticazione e autorizzazione, che permettono di mettere in sicurezza i sistemi.
- Tra i sistemi più diffusi e semplici da utilizzare nell'ambito dei servizi REST c'è senz'altro lo standard JWT (JSON Web Token)

# Autenticazione tramite JWT



# Autenticazione tramite JWT

1. Il client effettua l'accesso inviando le credenziali all'Identity Provider
2. L'Identity Provider verifica le credenziali e, se sono OK, recupera i dati dell'utente, genera un token JWT contenente informazioni sull'utente e permessi a lui associati ed imposta una data di scadenza sul token
3. L'identity Provider firma ed eventualmente cripta il token JWT e lo invia al client come risposta alla richiesta di login
4. Il client memorizza il token per la durata della validità dello stesso
5. Il client invia il token JWT in un apposito Authorization header per ogni successiva richiesta effettuata al servizio
6. Per ogni richiesta, il servizio estrae il token dall'Authorization header, lo decripta e ne verifica la firma. Se il token è valido, estrae i dati utente ed i permessi e verifica se esso ha i permessi necessari per accedere alla funzione chiamata, appoggiandosi eventualmente all'Identity Provider
  - Questo flusso garantisce grande flessibilità e semplicità di implementazione mantenendo al contempo un elevato grado di sicurezza.

# Autenticazione tramite JWT

- Spring mette a disposizione numerosi strumenti per la gestione degli aspetti relativi alla sicurezza delle applicazioni. Tra questi strumenti c'è il supporto completo allo standard JWT. Il modulo dedicato alla sicurezza è denominato Spring Security ed ha le seguenti caratteristiche:
- Supporto completo ed espandibile a Autenticazione ed Autorizzazione
- Protezione da attacchi di tipo session fixation, clickjacking, cross site request forgery (CSRF)...
- Integrazione con Spring Web MVC
- Per aggiungere il supporto a Spring Security e JWT in un progetto Spring Boot è come al solito sufficiente aggiungere al pom.xml le seguenti dipendenze:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-api -->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-impl -->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt-jackson -->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
```



# Autenticazione tramite JWT

- Le funzioni di firma e cifratura del token JWT necessitano di una chiave segreta per poter essere utilizzate.
- La chiave, che può essere condivisa tra più applicazioni che hanno in comune lo stesso Entity Provider, viene impiegata dal sistema per applicare gli algoritmi di firma, cifratura e successiva decodifica del token. Essa viene combinata con l'header ed il payload del token per generare un hash univoco.
- È possibile specificare la chiave segreta all'interno del file application.properties:

```
# JWT
jwt.secret=Neque porro quisquam est qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit...
jwt.expirationms=864000000
```

- Oltre alla chiave, è utile inserire in configurazione la durata prevista per la validità dei token, trascorsa la quale un token non sarà più considerato valido.

# Autenticazione tramite JWT

- Il principio alla base dell'implementazione dell'autenticazione JWT in Spring che verrà illustrato è il seguente:
  1. L'applicazione memorizza al proprio interno le informazioni relative agli utenti, in particolare username e password, ed ai ruoli ad essi associati
  2. Per ogni metodo esposto dal web service vengono definiti i ruoli autorizzati a chiamarlo
  3. L'applicazione espone un nuovo endpoint di login che il client invoca passando come argomenti username e password e che restituisce un token JWT contenente i dati dell'utente e i suoi ruoli
  4. Ogni volta che viene invocato un endpoint, il sistema cerca il token nella request e se lo trova lo decripta e ne verifica la validità
  5. Prima di invocare l'endpoint richiesto, il sistema confronta i ruoli dell'utente dichiarati nel token con quelli necessari all'invocazione
  6. Se i dati corrispondono, il sistema invoca l'endpoint normalmente, altrimenti restituisce un messaggio di errore ad-hoc.

# Autenticazione tramite JWT

- Per implementare un sistema di autenticazione ed autorizzazione è necessario prevedere un modello atto a supportare le funzionalità.
- Il modello tipico per la realizzazione di un sistema di autenticazione dinamico è una classe che mappa l'utente, comprensiva di username e password associata ad una classe che mappa i ruoli che sono assegnati all'utente.
- In questo modo, ogni volta che un client autenticato effettua una chiamata ad un endpoint, il sistema può verificare se i ruoli associati all'utente gli consentono di invocare l'endpoint.
- Questo modello di base può essere esteso e migliorato per fornire una granularità dei permessi maggiore, qualora sia richiesto dall'applicazione.
- A titolo di esempio, a partire dal modello dello User introdotto precedentemente, è possibile aggiungere le caratteristiche descritte.

# Autenticazione tramite JWT

```
@Getter
@Setter
@ToString
@EqualsAndHashCode(callSuper = true, onlyExplicitlyIncluded = true)

@Entity
@Table(name = "roles")
public class Role extends BaseEntity {
    @Column(length = 12, nullable = false, unique = true)
    private String name;

    @ManyToMany(mappedBy = "roles", fetch = FetchType.EAGER)
    public final Set<User> users = new HashSet<User>();
}
```

# Autenticazione tramite JWT

```
@Getter
@Setter
@ToString
@EqualsAndHashCode(callSuper = true, onlyExplicitlyIncluded = true)
@NoArgsConstructor

@Entity
@Table(name = "users")
public class User extends BaseEntity {
    @Column(length = 80, nullable = false, unique = true)
    private String email;
    @Column(length = 255, nullable = false)
    private String password;
    @Column(length = 25, nullable = false, unique = true)
    private String username;
    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "users_roles", joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id"))
    private Set<Role> roles = new HashSet<Role>();

    @Builder(setterPrefix = "with")
    public User(int id, LocalDateTime createdAt, String email, String password, String username) {
        super(id, createdAt);
        this.email = email;
        this.password = password;
        this.username = username;
    }
}
```

# Autenticazione tramite JWT

- Per disaccoppiare l'implementazione del modello applicativo dalle meccaniche di autenticazione, è opportuno implementare due classi:
  - **UserDetailsImpl**
    - che implementa l'interfaccia UserDetails fornita dal modulo Spring Security
  - **UserDetailsServiceImpl**
    - che implementa l'interfaccia **UserDetailsService** fornita dal modulo Spring Security
- ❖ Queste classi permettono di acquisire le informazioni di autenticazione e autorizzazione dal modello applicativo, adattandone le interfacce ai requisiti dell'infrastruttura di autenticazione

# Autenticazione tramite JWT

```
@Data
@AllArgsConstructor
public class UserDetailsImpl implements UserDetails {
    private static final long serialVersionUID = 1L;

    private int id;
    private String username;
    private String email;
    @JsonIgnore
    private String password;
    private boolean accountNonLocked = true;
    private boolean accountNonExpired = true;
    private boolean credentialsNonExpired = true;
    private boolean enabled = true;
    private Date expirationTime;
    private Collection<? extends GrantedAuthority> authorities;

    public static UserDetailsImpl from(User user) {
        var authorities = user.getRoles().stream()
            .map(r -> new SimpleGrantedAuthority(r.getName())).toList();
        return new UserDetailsImpl(
            user.getId(), user.getUsername(), user.getEmail(), user.getPassword(), true, true,
            true, true, new Date(), authorities);
    }
}
```

# Autenticazione tramite JWT

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    private UsersRepository users;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
        var user = users.findByUsername(username).orElseThrow();
        return UserDetailsImpl.from(user);
    }
}
```



# Autenticazione tramite JWT

- È possibile strutturare la gestione del token all'interno di una classe di utilità

```
@Value("${jwt.secret}")
private String jwtSecret;

@Value("${jwt.expirationms}")
private long jwtExpirationMs;

public String generateJwtToken(Authentication authentication) {
    var principal = (UserDetailsImpl) authentication.getPrincipal();
    var now = new Date();
    var exp = new Date(now.getTime() + jwtExpirationMs);
    principal.setExpirationTime(exp);
    var key = Keys.hmacShaKeyFor(jwtSecret.getBytes());
    return Jwts.builder() //
        .setSubject(principal.getUsername()) //
        .setIssuedAt(now) //
        .setExpiration(exp) //
        .signWith(key) //
        .compact();
}
```

# Autenticazione tramite JWT

- Per gestire l'autenticazione tramite token è possibile sfruttare la catena di gestione delle request

```
@Component
public class JwtAuthenticationFilter extends OncePerRequestFilter {
    private static final Logger Log = LoggerFactory.getLogger(JwtAuthenticationFilter.class);

    private static final String HEADER = "Authorization";
    private static final String BEARER = "Bearer ";

    @Autowired
    private JwtUtils jwtUtils;

    @Autowired
    private UserDetailsService detailService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)
        throws ServletException, IOException {
        try {
            var jwt = parseJwt(request);
            if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
                var username = jwtUtils.getUsernameFromJwtToken(jwt);
                var details = detailService.loadUserByUsername(username);
                var authentication = new UsernamePasswordAuthenticationToken(details, null,
                    details.getAuthorities());
                authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
                SecurityContextHolder.getContext().setAuthentication(authentication);
            }
        } catch (Exception e) {
            Log.error("Exception authenticating from token", e);
        }
        filterChain.doFilter(request, response);
    }
}
```

# Autenticazione tramite JWT

- Inoltre può essere previsto un entry point per utenti non autenticati

```
@Component
public class AuthEntryPointUnauthorizedJwt implements AuthenticationEntryPoint {
    private static final Logger Log = LoggerFactory.getLogger(AuthEntryPointUnauthorizedJwt.class);

    @Override
    public void commence(HttpServletRequest request, HttpServletResponse response,
        AuthenticationException authException) throws IOException, ServletException {
        Log.error("Unauthorized error: {}", authException.getMessage());
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized");
    }
}
```

# Autenticazione tramite JWT

- La classe identificata dalle annotazioni `@Configuration` e `@EnableWebSecurity` ha lo scopo di abilitare le funzioni di sicurezza del framework e di configurarne i parametri di funzionamento.
  - Istruisce il sistema sul servizio da impiegare per manipolare i dati dello user
  - Registra il filtro che ha il compito di estrarre e processare il token JWT dalle request
  - Definisce il sistema di encoding delle password

# Autenticazione tramite JWT

```
@Configuration
public class SecurityConfig {

    @Autowired
    private JwtAuthenticationFilter jwtAuthenticationFilter;

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests(auth -> auth.anyRequest().permitAll()) // accesso a tutte le risorse
            .cors(c -> c.disable()) // CORS disabilitato
            .csrf(c -> c.disable()) // CSRF disabilitato
            .addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }

    @Bean
    PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

# Autenticazione tramite JWT

- Per permettere al client di effettuare l'autenticazione ed ottenere il token JWT che dovrà poi essere inviato nelle successive chiamate al servizio, è necessario implementare un endpoint di login. Questo endpoint accetta dal client username e password e restituisce una serie di informazioni:
  - Username
  - User id
  - Email
  - Ruoli associati
  - Data di scadenza del token
  - Token
  - Tipo del token – per i nostri scopi sarà sempre di tipo **Bearer**

# Autenticazione tramite JWT

```
@Data
public class LoginModel {
    private String email;
    private String password;
}
```

```
@Data
@Builder(setterPrefix = "with")
public class LoginResponseModel {
    private String username;
    private String email;
    private String token;
    @Builder.Default
    private List<String> roles = new ArrayList<String>();
}
```

# Autenticazione tramite JWT

```
@RestController
@RequestMapping("/api/users")
public class UsersController {

    @Autowired
    private UserService service;

    @PostMapping("register")
    public ResponseEntity<ApplicationResponse<RegisteredUserModel>> registerUser(@RequestBody RegisterUserDto userDto) {
        try {
            var user = service.registerUser(userDto).orElseThrow();
            return ResponseEntity
                .ok(new ApplicationResponse<RegisteredUserModel>(new RegisteredUserModel(user.getUsername(),
                    user.getEmail(), user.getRoles().stream().map(Role::getName).toList())));
        } catch (Exception e) {
            return ResponseEntity.badRequest()
                .body(new ApplicationResponse<RegisteredUserModel>(new ApiError(e.getMessage())));
        }
    }

    @PostMapping("login")
    public ResponseEntity<ApplicationResponse<LoginResponseModel>> loginUser(@RequestBody LoginModel login) {
        try {
            var response = service.login(login.getEmail(), login.getPassword()).orElseThrow();
            return ResponseEntity.ok(new ApplicationResponse<LoginResponseModel>(response));
        } catch (Exception e) {
            return ResponseEntity.badRequest()
                .body(new ApplicationResponse<LoginResponseModel>(new ApiError("Login failed")));
        }
    }
}
```



# Autenticazione tramite JWT

- Per indicare al sistema che l'accesso ad un endpoint deve essere sottoposto ad autenticazione, si impiega l'annotazione `@PreAuthorize` in corrispondenza del metodo mappato. L'annotazione accetta dei parametri che possono essere espressi per mezzo delle Spring Security Expressions (SSE).
- SSE permette di dichiarare in modo semplice le regole che devono essere verificate dal framework per autorizzare l'accesso ad una funzione. Alcune delle istruzioni supportate da SSE sono:
  - `hasRole, hasAnyRole`
  - `hasAuthority, hasAnyAuthority`
  - `permitAll, denyAll`
  - `isAnonymous, isAuthenticated, isFullyAuthenticated`
  - `hasPermission`

# Autenticazione tramite JWT

```
@PostMapping("publish")
@PreAuthorize("isAuthenticated()")
public ResponseEntity<ApplicationResponse<Optional<ArticleDto>>> publishArticle(@RequestBody ArticleModel model) {
    try {
        var user = (UserDetails) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        var articleDto = service.publishArticle(user, ArticleDto.builder() //
            .withAuthor(user.getUsername()) //
            .withContent(model.getContent()) //
            .withTitle(model.getTitle()) //
            .build());
        return ResponseEntity.ok(new ApplicationResponse<Optional<ArticleDto>>(articleDto));
    } catch (Exception e) {
        return ResponseEntity.badRequest()
            .body(new ApplicationResponse<Optional<ArticleDto>>(new ApiError(e.getMessage())));
    }
}
```

111



# Autenticazione tramite JWT

- Una volta ottenuto il token JWT, in tutte le richieste successive il client dovrà aggiungere un header "Authorization" contenente la keyword "Bearer" seguita dal token.
- In caso di scadenza del token o di sua revoca da parte del server, il client dovrà effettuare nuovamente il login per ottenere un nuovo token valido.

The screenshot shows a Postman interface for a GET request to `http://localhost:8080/api/users/1`. The request is configured with the following headers:

Key	Value	Description
Host	<calculated when request is sent>	
User-Agent	PostmanRuntime/7.28.0	
Accept	*/*	
Accept-Encoding	gzip, deflate, br	
Connection	keep-alive	
Authorization	Bearer eyJhbGciOiJIUzUxMiJ9.eyJ...	

The response is a 200 OK status with a response time of 33 ms and a body size of 668 B. The response body is displayed in JSON format:

```
1 {
2   "id": 1,
3   "username": "test",
4   "firstName": "Mario",
5   "lastName": "Rossi",
6   "birthDate": "1980-12-11T23:00:00.000+00:00",
7   "active": true,
8   "password": "test",
9   "email": null,
10  "roles": [
11    {
12      "id": 99999,
13      "roleType": "ROLE_USER"
14    }
15  ]
16 }
```

# Autenticazione tramite JWT

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:8080/api/users/1
- Headers (7):**
  - ☒ Host: <calculated when request is sent>
  - ☒ User-Agent: PostmanRuntime/7.28.0
  - ☒ Accept: \*/\*
  - ☒ Accept-Encoding: gzip, deflate, br
  - ☒ Connection: keep-alive
  - ☐ Authorization: Bearer eyJhbGciOiJIUzUxMiJ9.eyJ...
- Body:** Pretty view showing a JSON response:

```
1 {
2   "timestamp": "2021-05-12T04:40:49.906+00:00",
3   "status": 401,
4   "error": "Unauthorized",
5   "message": "",
6   "path": "/api/users/1"
7 }
```
- Status:** 401 Unauthorized, 25 ms, 561 B
- Buttons:** Send, Save Response

# ■ Crittografia delle Password

- Uno degli aspetti più importanti da tenere in considerazione quando si sviluppa e si mette in opera una applicazione, soprattutto se essa è esposta su internet è la sicurezza dei dati
- Il numero di attacchi e violazioni ai sistemi software aumenta in modo considerevole ogni anno ed è pertanto necessario prendere le misure opportune per limitare al massimo la possibilità che i dati gestiti dalle applicazioni possano essere acquisiti da persone non autorizzate
- Le azioni necessarie per la messa in sicurezza delle applicazioni coinvolgono varie figure del team IT, come sviluppatori, sistemisti ed esperti di networking, che devono lavorare in sinergia per contrastare i rischi e garantire l'integrità dei sistemi



# ■ Crittografia delle Password

- Per ovviare al problema della memorizzazione della password in chiaro, e più in generale della memorizzazione delle informazioni sensibili, l'applicazione deve applicare una codifica che la renda illeggibile ma che permetta comunque di verificarne la correttezza agli accessi successivi
- I due approcci più diffusi per risolvere il problema sono:
  - Cifratura della password
  - Salt & Hashing

# Spring Security

- Il modulo Spring Security mette a disposizione dello sviluppatore gli strumenti per implementare in modo molto semplice le funzionalità di Salt & Hashing delle informazioni. In particolare esso offre diversi algoritmi di hashing, tra cui:
  - Bcrypt
  - PBKDF2
  - Argon2
- L'algoritmo più adatto in questo caso è il Bcrypt



# Spring Intermediate

Un esempio che cerca di illustrare quanto esposto è disponibile all'indirizzo:

<https://github.com/NelloRizzo/WebAPIServer/>

Grazie per l'attenzione!