Introduzione all'Application Framework

Un **application framework** è un insieme di librerie, strumenti e convenzioni pensate per facilitare lo sviluppo del software, quindi un framework è volto a facilitare l'attività dello sviluppatore. Il framework consente allo sviluppatore di concentrarsi soltanto sulla logica applicativa e di non preoccuparsi di tutto il resto e dell'infrastruttura comune. L'application framework riduce di fatto lo sforzo e il tempo necessario per creare una applicazione. Uno dei principali vantaggi di usare questi strumenti è che consentono agli sviluppatori di seguire una struttura organizzativa predefinita e una metodologia di sviluppo coerente. Il codice sarà più ordinato e leggibile che è un vantaggio non da poco se più persone lavorano allo stesso progetto. Quindi il framework facilita quello che è lo sviluppo di lavoro in team, l'iterazione con i databases, la gestioen ad eventi, l'autenticazione, l'I/O dei dispositivi ecc...

L'application framework offre anche strumenti e risorse come gli IDE, simulatori e documentazione...

Alcuni dei framework più popolari includono Django per lo sviluppo Python, Ruby on rails per Ruby, Angular per Typescript e ovviamente Spring Framework per lo sviluppo di applicazioni in Java.

Non è necessario conoscere interamente un framework, ma soltanto le parti essenziali che possono tornare utili alla attività di sviluppo dell'applicazione.

Introduzione al framework Spring

Lo spring framework è un ecosistema di framework basato sul linguaggio Java e offre una vastissima gamma di strumenti e funzionalità per semplificare lo sviluppo di applicazini aziendali robuste e scalabili.

Principali caratteristiche di Spring Framework:

- Inversion of control e dependency injection (concetti che vengono usati per la creazione e la gestione degli oggetti nelle applicazioni, attraverso la dependency injection le dipendenze tra oggetti si risolvono con il container di Spring)
- Spring container (è il componente centrale del framework che gestisce e controlla tutti gli oggetti creati all'interno di una applicazione Spring, il container crea gli oggetti, gestisce le loro dipendenze e li fornisce quando richiesto)
- Spring MVC (offre un framework per lo sviluppo di applicazioni basate sul modello MVC con una robusta gestione delle richieste http e gestione della view e della logica di business)
- Spring Security (fornisce le funzionalità per la gestione dell'autenticazione e delle autorizzazioni e della protezione delle applicazioni da attacchi comuni quali il crosso site scripting XSS o il cross site request forgery CSRF
- Persistence (supporta la Java Persistence Api JPA, Hibernate e MyBatis e offre il supporto per il database transaction manager con il modulo Spring Transaction Management)
- AOP (aspect oriented programming → integra il concetto di aspect oriented programming che consente di separare le funzionalità trasversali dalla logica di business di una applicazione, ciò consente di implementare in modo pulito e modulare funzionalità come il logging, il caching e la transazione.
- Testing → il framework spring supporta il testing delle applicazioni, fornisce una ambiente di test integrato che facilita la scrittura degli unit test.
- Gestione delle transazioni → Spring offre un modulo per la gestione delle transazioni, che semplifica l'implementazione delle funzionalità di transazione nelle applicazioni.

- Modularità → altamente modulare, consente allo sviluppatore di usare solo i moduli che ha bisogno, è flessibile e si integra con altri framework
- Facilità di integrazione
- Gestione delle eccezioni → semplifica la gestione delle eccezioni
- Internazionalizzazione i18n e localizzazione I10n
- Spring Boot → semplifica la creazione di applicazioni Spring stand-alone
- Spring Cloud → permette e semplifica la creazione di applicazioni cloud-native, inclusi servizi di registrazione, load balancing di carico e configurazioni esterne
- Forte comunità attiva e grande documentazione

Spring è il framework Java più usato al momento.

Introduzione al framework Spring Boot 3

Si tratta di un complesso di elementi pronti all'uso per lo sviluppo di applicazioni con Java (vedremo che possiamo usare anche altri 2 tipi di linguaggi – Kotlin e Groovy). Questo specifico framework fa parte di un ecosistema molto più ampio che è Spring. Il suo scopo è quello di semplificare il processo di sviluppo e deploy delle applicazioni basate su Java fornendo una struttura preconfigurata e automatizzata permettendo allo sviluppatore di concentrarsi soltanto sulla logica di business. Quindi focus solamente per la logica di business da parte dello sviluppatore. Spring boot prevede un sistema di avvio rapido (bootstrap) che configura tutto l'ambiente in cui si esegue l'applicazione. Spring Boot consente la creazione di applicazioni stand-alone, non c'è bisogno di installare nessun web server, c'è l'embedded server, cui posso sceglierne di più tipi, quello di default è il Tomcat. Spring boot supporta anche un sistema di gestione delle dipendenze basate su due tool in particolare:

- Maven
- Gradle

Ciò facilità il processo di creazione dell'applicazione. Spring boot prevede una eccellente integrazione con altre tecnologie:

- Timeleaf (per le viste)
- Spring Data (database)
- Spring Security

Concetti chiave dello spring boot

- **Convenzione di configurazione** → il framework fornisce delle impostazioni predefinite intelligenti e una ampia configurazione automatica.
- **Autoconfigurazione** → potenti funzionalità di autoconfigurazione

Ciò permette allo sviluppatore di iniziare subito con una applicazione che funziona da subito. Ciò riduce il tempo di configurazione iniziale.

- **Embedded Server** → Spring boot fornisce un server web incorporato (Tomcat oppure Jetty), non c'è nessun bisogno di configurare un server esterno, tutto ciò che server è un semplice eseguibile JAR.
- **Spring Boot Starter** → è un concetto chiave che semplifica la gestione delle dipendenze della nostra applicazione, gli starter sono serie di dipendenze preconfigurate che vengono fornite in bundle per supportare specifiche funzionalità.
- **Actuator** → sono funzionalità di monitoraggio e stato dell'applicazione.

In sintesi → Spring boot è quindi un framework Java molto efficiente per la creazioni di applicazioni web altamente performanti e di qualità, che semplifica il processo di sviluppo ed evita del tutto la complessità delle configurazioni manuali iniziali. Inoltre la sua vasta community e la sua ampia documentazione lo rendono uno strumento altamente accessibile e viene considerato come uno dei migliori framework disponibili per lo sviluppo di applicazioni Java.

Modi per creare un progetto con Spring Boot

- Usare uno specifico sito web → Spring Initializr
- Usare Eclipse (attraverso il pacchetto Spring Tool Suite)

Usando lo Spring Tool suite non c'è bisogno di scompattare il progetto altrove per poterlo usare subito.

Trattandosi di progetti Spring Boot 3 la versione di Java di partenza è per forza di cose la versione 17.

Le versioni di Spring Boot cambiano abbastanza velocemente e può succedere che da una versione all'altra non funzioni più un determinato tipo di codice, purtroppo non c'è una compatibilità verso il basso che viene mantenuta sempre.

Dipendenze → elementi centrali del framework che permettono alla nostra web app di funzionare.

Analisi di un progetto Spring Boot

- **/src/main/java** → è la cartella di base dove sono presenti tutti i packages del codice sorgente Java.
- **Classe Application** → classe di entry point della nostra applicazione (possiamo modificare per convenzione il nome di questa classe e mettere soltanto Application.java)
- **/src/main/resources** → sono le risorse che servono per far funzionare il nostro progetto.
- **/src/main/resources/static** → inizialmente vuote conterrà le risorse statiche come i js o i css della mia applicazione
- **/src/main/resources/templates** → inizialmente vuote conterrà i files dei template html del frontend dell'applicazione
- **application.properties** → contiene le properties del nostro intero progetto è un file molto importante che posso convertire anche in application.yml
- **src/test/java** → contiene il codice sorgente dei junit test
- **JRE System Library** → è il riferimento alla nostra versione di Java
- **Maven Dependencies** → è l'elenco complete delle dipendenze del progetto importate dal pom.xml
- **Pom.xml** → siccome si è selezionato il tool Maven, viene configurato questo file, che contiene tutte le dipendenze, il file viene coompilato dal tool del Maven, è un file .XML, che riguardano i metadati che abbiamo selezionato. Nel gruppo <dependencies> vengono racchiuse le nostre dipendenze selezionate. Alcune vengono messe in automatico dal tool. Il gruppo <build> riguarda la parte di build del nostro progetto.

Il modello MVC in Spring

È un design pattern che consente di strutturare una web app in modo equilibrato e strutturato. L'MVC si compone di:

- **Model** → il modello dei dati, la business logic della nostra applicazione e contiene i dati e le operazioni che possiamo eseguire sopra di essi, nel caso di Spring il model viene rappresentato da speciali classi di dominio note come Entity annotate come **@Entity** o semplici classi POJO.
- **Vista** → sono template/viste basate su JSP o Thymeleaf e presentano i dati al client
- **Controller** → funge da intermediario tra vista e modello. Interroga il modello e li passa alla vista, nel caso di Spring i Controller sono gestiti dal modulo Spring MVC e annotati con l'annotation **@Controller**.

Quando una richiesta arriva al server, viene attivato il controller appropriato per gestire la richiesta.

In spring boot la classe Controller è componente chiave per la creazione delle nostre web app. Rappresenta la logica della gestione delle richieste http. Quindi in Spring la classe Controller contiene la logica per la rappresentazione delle richieste http, consentendo di definire la comunicazione tra il client e il server. Nella pratica il compito del controller è quello di esporre una serie di endpoint REST http definendo la logica di business per esaudire al richiesta.

Una **classe controller** si compone di:

- Annotation @Controller, una classe con questa annotation indica a Spring che si tratta di una classe Controller
- Metodi annotati con l'annotation @RequestMapping, oppure @GetMapping ecc... che indicano quale metodo http si vuole gestire
- Parametri di metodi con annotations @PathVariable, @RequestParam, @RequestBody ...
- Restituzione dei dati con @ResponseBody per restituire dati direttamente nel corpo della risposta.
- Possono esserci diversi metodi che implementano tanti controllers

Per comodità e ordine del codice creiamo sempre le nostre classi nuove dentro dei packages!

Primo esempio della prima classe Controller in Spring Boot:

- Attraverso l'annotation **@Controller** informiamo Spring che si tratta di un tipo di classe controller che quindi sarà in grado di gestire degli specifici endpoint
- Attraverso l'annotation **@GetMapping** indichiamo esplicitamente di quale tipo di endpoint http richiesta si tratta.
- **Model** è una classe che produce dei tipi di oggetti che ci permettono di passare dei dati alle nostre viste.

```
-   @Controller
-   public class IndexController {
-
-         public String saluti = "Salve, sono la tua prima applicazione web
    creata in Spring Boot 3";
-
-         @GetMapping(value = "/")
-         public String getWelcome(Model model) {
-               model.addAttribute("intestazione", "Benvenuti nella root page
    della webapp Alphashop");
-               model.addAttribute("saluti", saluti);
-               return "index";
-         }
-   }
```

Attivare le viste

Abbiamo 2 alternative:

- Pagine JSP
- Thymeleaf

Pagine JSP

Java Server Pages sono una tecnologia per creare delle pagine web in maniera dinamica. Sono pagine che uniscono html con java per creare dinamismo nelle pagine. Le pagine jsp sono come pagine html, ma contengono codice java delimitato dal tag "<% .... %>".

Il codice viene quindi letto ed eseguito dal server web quando la pagina viene richiesta dal client, ma quindi questo significa che il client non sa quale parte del codice della pagina jsp è generato e quale sia quello statico. Esiste tutta una classe di tag jstl per rendere dinamiche le nostre pagine.

Ma come si attivano dentro spring boot 3? Il primo passo è aprire il pom.xml e nelle dipendenze aggiungere la dipendenza:

```
<!-- https://mvnrepository.com/artifact/org.apache.tomcat.embed/tomcat-embed-
jasper -->
            <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-jasper</artifactId>
            </dependency>
```

Poi dobbiamo andare ad inserire la nostra prima vista nel percorso:

- Sotto il percorso delle **/resources** e per la precisione dentro il percorso che creiamo **/META-INF/resources/WEB-INF/jsp** e qui dentro creiamo un file **index.jsp** che si deve chiamare esattamente come il nome della stringa che ritorniamo nel nostro controller, cioè return "index"

Questa pagina contiene dello standard HTML, con in più le porzioni di codice che richiamano la parti dinamiche definite nel nostro Controller, ad esempio per richiamare le parti dinamiche scriviamo così:

```
<h1 style="text-
align:center;display:block;color:red;margin:0px;">${intestazione}</h1>
            <h3 style="text-
align:center;display:block;color:green;margin:0px;">${saluti}</h3>
```

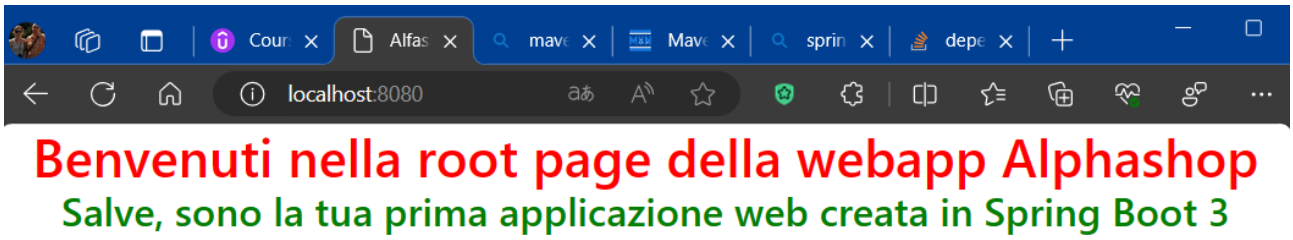
Le parti "intestazione" e "saluti" sono state dinamicizzate nel nostro controller come detto sopra.

E poi dobbiamo agganciare il **view resolver**, quindi andiamo a configurarlo nel file application.yaml in questo modo:

```
spring:
  application:
    name: alphashop
  mvc:
    view:
      prefix: /WEB-INF/jsp/
      suffix: .jsp
```

A questo punto non ci resta che avviare il server tomcat e andare all'url http://localhost:8080 dove sarà richiamata e visualizzata la nostra pagina di root del nostro progetto.

Ed ecco quello che dovrei vedere:



N.B. Si definisce **Whitelabel Error Page** la pagina generica dello Spring Boot quando si tenta di accedere a delle risorse che un controller non è in grado di gestire. L'errore che viene ritornato è di solito un 404 Not Found in quanto non abbiamo gestito la risorsa da ritornare.

Creazione di una pagina di Login (esercizio)

- Creare una classe LoginController.java in grado di gestire l'endpoint di login
- Creare la relativa pagina login.jsp che presenta il form di login

Introduzione alla annotazione @RequestParam

Questa annotazione dello Spring Framework si utilizza per recuperare i parametri di una richiesta http (da form o da url) e rimapparli come parametri di metodo nel controller. Quando con Spring definiamo un controller possiamo quindi servirci dell'annotations per dichiarare i parametri che si desidera estrarre dalla richiesta http. Questa annotazione permette a Spring di passare i parametri e leggerli al controller.

Esempio pratico (passaggio di un parametro nella URL con RequestParam):

```
@GetMapping(value = "/index")
	public String getWelcome2(Model model, @RequestParam("name") String name)
{
		model.addAttribute("intestazione", String.format("Benvenuto %s nella
index page della webapp Alphashop", name));
		model.addAttribute("saluti", saluti);
```

```
            return "index";
        }
```

- In RequestParam non è obbligatorio specificare l'attributo "name", che deve essere uguale alla variabile
- Per usare il parametro name basta richiamarlo nel codice
- Se andiamo all'url http://localhost:8080/index ci verrà ritornato un bad request 400 in quanto non stiamo passando il parametro, dobbiamo usare la query string per far funzionare la pagina e il passaggio di parametri ad esempio→ http://localhost:8080/index?name=Roberto

Esempio pratico (passaggio di parametri in un FORM con RequestParam):

```
@PostMapping(value = "/login")
    public String goToWelcomePage(
            @RequestParam("name") String name,
            @RequestParam("password") String password,
            ModelMap model
            ) {
        return null;
    }
```

- Il ModelMap è una classe dell'ecosistema dello Spring Framework usata per condividere dati tra le componenti del Controller e la vista, quindi per noi è da considerarsi come un contenitore di dati.
- Sia la classe Model che ModelMap sono classi usate per passare dati alla vista dal controller.
- ModelMap è una classe concreta che estende Model aggiungendo più metodi. ModelMap è più ricca di funzionalità.

Esempio (metodo completo):

```
@PostMapping(value = "/login")
    public String goToWelcomePage(
            @RequestParam("name") String name,
            @RequestParam("password") String password,
            ModelMap model
            )
    {
        if (authenticationService.auth(name, password))
{
            model.put("name", name);
            return "welcome";
        }
        else return "login";
    }
```

Per l'autenticazione ho usato una classe Component:

```
@Component
public class AuthenticationService {

	public boolean auth(String username, String password) {
		boolean isValidUsername = username.equalsIgnoreCase("Roberto");
		boolean isValidPassword = password.equalsIgnoreCase("1234");

		return isValidUsername && isValidPassword;
	}
}
```

e poi per usare questo componente nel controller ho 2 modi per usare l'injection del componente:

- Usare @Autowired
- Usare un costruttore

Esempio:

```
@Autowired
	private AuthenticationService authenticationService;
```

opppure:

```
public LoginController(AuthenticationService authenticationService) {
		this.authenticationService = authenticationService;
	}
```

N.B. → non esiste una teoria su cosa sia meglio o peggio, però ultimamente si consiglia di usare l'injection da costruttore e non usare più l'autowired.

Introduzione alla annotations @SessionAttributes

È una annotations che gestisce gli attributi di sessione all'interno delle applicazioni web. Viene usata per collegare un attributo dell'oggetto http session con un parametro del metodo di un controller in modo da poterli riutilizzare.

Esempio:

```
@SessionAttributes("name")
public class LoginController {
```

Messo in testa alla classe controller, in questo modo persistiamo il parametro che dobbiamo passare.

Introduzione allo Strato di servizio (il service layer)

Lo strato di servizio, normalmente è lo strato intermedio, tra lo strato della base dati (di persistenza dei dati, dove si salvano i dati sul database) e lo strato delle classi controller (dove stanno i passaggi verso le viste). Lo strato di servizio può di fatto intervenire e modellare i dati e applicare della logica sui dati. Questo strato è stato introdotto dal momento che le applicazioni sono diventate più complesse, quando si è iniziato ad introdurre una base di dati e le viste e i nostri modelli sono diventati più complessi, allora si è vista l'esigenza di introdurre un livello in più per ridurre la complessità delle applicazioni.

Per prima cosa conviene sempre creare un nuovo package dedicato allo strato del servizio, questo package dovrà ospitare:

- Interfacce dello strato di servizio
- Classi ed implementazioni dello strato di servizio

Esempio:

- com.demo.example.services

creiamo inoltre l'interfaccia dentro chiamata: ArticoliService.java

```
package com.demo.example.services;

import java.util.List;
import com.demo.example.entities.Articoli;

public interface ArticoliService {
    public List<Articoli> selAll();
}
```

In questa interfaccia per ora definiamo un metodo finto (perché non abbiamo ancora il database) che è in grado di simulare la selezione completa di una lista dei nostri articoli. In più, ora, ci tocca andare a creare un nuovo package in modo da ospitare le nostre "entità" che arriveranno direttamente dal modello dei dati (il database) e quindi definiamo un nuovo package:

- com.demo.example.entities

e dentro ci mettiamo la classe Articoli.java, dove grazie alle potenzialità della lireria Lombok possiamo definire facilmente i metodi getter/setter e i relativi costruttori.

```
@Data
public class Articoli {
    private String codArt;
    private String descrizione;
    private String um;
    private int pzcart;
    private double pesonetto;
    private double prezzo;
}
```

Il progetto Lombok

Si tratta di un java tool che semplifica lo sviluppo delle applicazioni gestendo automaticamente la generazione del codice boilerplate, come i getter, i setter, costruttori, metodi toString(), equals() e altri più comuni metodi delle classi Java. Spring Boot è compatibile con il Lombok e può sfruttarne tutte le

potenzialità per ridurre la quantità di codice che viene scritto. Lombok fornisce una serie di importanti annotations che conviene distinguere:

- @Getter e @Setter generano automaticamente i metodi getter e setter di ogni classe
- @NoArgsConstrutor genera il costruttore senza argomenti
- @AllArgsConstructor genera il costruttore con tutti i parametri
- @Data che combina tutte le annotations getter/setter/EqualsAndHashCode/RequiredArgsConstructor in una sola annotations
- @Builder genera il pattern builder per la creazione di oggetti immutabili

Vedere documentazione qui: Project Lombok

Per usare Lombok nel nostro progetto Spring Boot basta importarne la dipendenza nel nostro pom.xml:

```
<dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
            <scope>provided</scope>
        </dependency>
```

Vantaggi:

- Meno codice ripetitivo
- Migliore manutenzione
- Migliore leggibilità
- Risparmio di tempo nello sviluppo

In Eclipse lo devo installare prima di poterlo usare) devo installere il JAR file) altrimenti non funzionano i richiami delle annotations nel codice. Su IntelliJ non è necessario basta attivarne il plugin.

- java -jar nomeDelLombokJar

e seguire le info di installazione.

Lo Stato del servizio (continua)

Adesso andiamo a creare la classe di implementazione dello strato di servizio, la classe ArticoliServiceImpl.java.

Esempio:

```
@Service
public class ArticoliServiceImpl implements
ArticoliService {
}
```

Dal codice si nota che:

- essendo un servizio deve portare l'annotations @Service
- deve implementare subito l'interfaccia ArticoliService

Questa classe è il vero strato del servizio!

In più, per poter compilare correttamente questa classe, dobbiamo fare l'override del metodo presente nell'interfaccia ArticoliService, poiché la implementiamo:

```
@Override
	public List<Articoli> selAll() {
		return new ArrayList<>();
	}
```

Quindi → lo strato di servizio si compone sempre di:

- una classe di interfaccia
- una classe che implementa l'interfaccia e porta l'annotation @Service

Simulazione dello strato del servizio

Andiamo ad ottenere, per ora, una lista di articoli creando una lista fittizia, senza passare dal database, a scopo soltanto di apprendimento, in seguito vedremo come passare dal database, con il livello della persistenza per ottenere i nostri dati.

```
@Service
public class ArticoliServiceImpl implements ArticoliService {

	private static List<Articoli> getArticoli() {
		List<Articoli> articoli = new ArrayList<>();

		articoli.add(new Articoli("014600301", "BARILLA FARINA 1 KG", "PZ",
24, 1, 1.09));
		articoli.add(new Articoli("013600450", "BARILLA PASTA GR. 500 N.70
1/2 PENNE", "PZ", 30, 0.5, 1.39));
		articoli.add(new Articoli("014600234", "FINDUS FIOR DI NASELLO 30
GR.", "PZ", 8, 0.3, 4.59));
		articoli.add(new Articoli("014600235", "FINDUS CROCCOLE 500 GR.",
"PZ", 12, 0.4, 3.99));

		return articoli;
	}

	@Override
	public List<Articoli> selAll() {
		return getArticoli();
	}

}
```

Ora questo strato di servizio bisogna cercare di usarlo nmello strato di Presentazione, quindi dobbiamo andare a modificare la classe ArticoliController.java, per prima dobbiamo fare injection dello strato di servizio nel nostro controller, adiamo a creare una nuova variabile privata e la injettiamo nel costruttore (facciamo injection di costruttore).

```java
@Controller
public class ArticoliController {
	private ArticoliService articoliService;

	public ArticoliController(ArticoliService articoliservice) {
		this.articoliService = articoliService;
	}

	@GetMapping(value = "/articoli")
	public String getArticoli(ModelMap modelMap) {

		List<Articoli> articoli = articoliService.selAll();
		modelMap.addAttribute("articoli", articoli);

		return "articoli";
	}
}
```

Però adesso dobbiamo visualizzare nello nostra vista una lista di articoli e in che modo possiamo farlo? Tramite il JSTL.

Introduzione al JSTL

Java Server Pages Standard Tag Library → è una libreria di tag per semplificare lo sviluppo delle pagine web, utilizzando le JSP e le Servlet. JSTL è un componente standard fornito dalla J2EE e fa parte della specifica delle JSP.

I tag JSTL danno la possibilità di effettuare iterazioni, condizioni e manipolazioni di stringhe con accesso ad oggetti java, l'uso di JSTL rende più leggibile e mantenibile il codice.

Per poter usare JSTL dentro il nostro progetto dobbiamo prima di tutto aggiungere 2 nuove dipendenze al nostro progetto:

```xml
<!-- https://mvnrepository.com/artifact/org.eclipse.jetty/glassfish-jstl -->
		<dependency>
		<groupId>org.eclipse.jetty</groupId>
		<artifactId>glassfish-jstl</artifactId>
		</dependency>

		<!--
https://mvnrepository.com/artifact/org.glassfish.web/javax.servlet.jsp.jstl -->
		<dependency>
		<groupId>org.glassfish.web</groupId>
		<artifactId>jakarta.servlet.jsp.jstl</artifactId>
		</dependency>
```

A questo punto andiamo a modificare la vista degli Articoli:
copiamo e incolliamo in testa alla vista sopra l'inizio del codice html questo taglib:

```jsp
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

e andiamo ad usarlo dentro la pagina html...

Per poter usare JSTL dentro il nostro codice html:

```
<table>
		<thead>
			<tr>
				<th>Codice</th>
				<th>Descrizione</th>
				<th>UM</th>
				<th>Pezzi</th>
				<th>Peso</th>
				<th>Prezzo</th>
			</tr>
		</thead>
		<tbody>
			<c:forEach items="${articoli}" var="article">
				<tr>
					<td>${article.codArt}</td>
					<td>${article.descrizione}</td>
					<td>${article.um}</td>
					<td>${article.pzcart}</td>
					<td>${article.pesonetto}</td>
					<td>${article.prezzo}</td>
				</tr>
			</c:forEach>
		</tbody>
	</table>
```

Panoramica dei comandi JSTL

Elenco di alcuni dei tags JSTL più usati:

Core Tags

- <c:forEach> iterazioni
- <c:if> <c:choose> <c:when> c:otherwise> condizioni
- <c:set> <c:remove> assegnamento e rimozione di variabili

Format Tags

- <fmt: formatNumber>
- <fmt:formatDate>

XML Tags

- <x:parse>
- <x:transform>

Inoltre prima di poter usare questi comandi bisogna dichiararne l'uso, come detto sopra importando il taglib di riferimento:

- **<%@ taglib prefix="c" uri="">**
- **<%@ taglib prefix="fmt" uri="">**
- **<%@ taglib prefix="x" uri="">**

Introduzione ai devtools

In spring boot i dev tools sono una serie di funzionalità e strumenti atti a migliorare lo sviluppo. Questi strumenti servono a semplificare la gestione dell'applicazione. Tra le funzionalità di questi dev tools troviamo:

- Riavvio automatico della applicazione (riduce il tempo di sviluppo)
- Ho swapping (ricarica parti specifiche del codice senza riavviare l'applicazione)
- Proprietà di configurazione dinamica (senza ricompilare l'applicazione)
- Gestione delle dipendenze (gestione semplificata della dipendenza)

Per attivare i dev tools nella nostra applicazione dobbiamo aprire il file pom.xml e inserire una nuova dipendenza:

```
<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools -->
            <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <version>${devtools.version}</version>
            </dependency>
```

Come includere file statici in un progetto

Configurare Bootstrap senza usare la libreria online, ma solo quella locale:

per prima cosa andare sul sito di Bootstrap e scaricare tutti gli elementi che permettono il funzionamento del framework css. Scarichiamo lo ZIP del framework a questo link Download · Bootstrap v5.3 (getbootstrap.com)

Una volta scaricato tutto il pacchetto ZIP devo andare a localizzare soltanto i file che mi servono a me per attivare Bootstrap:

- Bootstrap.min.css
- Bootstrap.bundle.min.js

Prendo questi due files e li devo andare ad inserire sempre sotto questo percorso:

- **src/main/resources/META-INF/resources/**

creiamo quindi il nuovo percorso /static/css e /static/js

Dentro queste due nuove cartelle ci copiamo il js e il css minificati di Bootstrap direttamente dallo ZIP che abbiamo scaricato. A questo punto apriamo le nostre pagine JSP e andiamo a sostituire i puntamenti ai rispettivi file del Bootstrap.

In questo modo:

- usando le taglib con prefisso c → `<link href="<c:url value="/static/css/bootstrap.min.css" /> rel="stylesheet">`

La stessa identica cosa va fatta anche per la parte Javascript e ovviamente per tutti i files di immagine, fonts ecc...

Uso dei frammenti nelle JSP

Abbiamo la possibilità di iniettare delle porzioni di pagine di codice JSP direttamente sulle nostre pagine, in modo da migliorare la pulizia, la leggibilità e ridurre le porzioni ripetute di codice.

Per prima cosa andiamo a creare una nuova cartella dentro il percorso:

- **src/main/resources/META-INF/resources/WEB-INF/jsp/**

questa cartella la chiamiamo **/common**

e dentro ci creiamo un nuovo file chiamato → navbar.jspf

dentro il file ci copiamo tutto il contenuto della navigation bar della nostra appllicazione. Dopodichè andiamo ad iniettare il codice della navbar direttmente nella pagina dove l'abbiamo rimosso prima in questo modo:

```
<%@ include file="common/navbar.jspf" %>
```

Migliorare l'estetica della pagina di login

Per prima cosa abbiamo diviso in frammenti il contenuto delle nostre jsp come mostrato nella precedente sezione di introduzione ai frammenti, nel percorso /common ora abbiamo altri files:

- **foot.jspf** (contiene il richiamo del javascript del bootstrap, che richiamiamo ovunque)
- **head.jspf** (contiene il frammento contenuto all'interno dei tag <head></head, anche questo componente lo richiamiamo ovunque nelle nostre pagine della applicazione)
- **jumbotron.jspf** (è l'intestazione della pagina di login)
- **navbar.jspf** (la navbar vista nella precedente sezione, che richiamiamo ovunque)

Abbiamo poi richiamato questi frammenti dentro le nostre jsp come nell'esempio della login.jsp:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html lang="en">
	<%@ include file="common/head.jspf" %>
	<body>
		<%@ include file="common/navbar.jspf" %>
		<%@ include file="common/jumbotron.jspf" %>
		<!-- Form Login -->
	<div class="container login-container">
	  <div class="row">
		<div class="col-md-6 login-form">
		    <h3>Login Form</h3>
		    <form method="post">
			<div class="form-group">
			    <input type="text" class="form-control" name="name"
id="name" placeholder="Nome Utente">
			</div>
			<div class="form-group">
			    <input type="password" class="form-control"
name="password" id="password" placeholder="Password">
			</div>
			<div class="form-group">
			     <input class="btnSubmit" type="submit">
			</div>
			<div class="form-group">
```

```
                            <a href="#" class="ForgetPwd">Password
Dimenticata?</a>
                        </div>
                         <c:if test="${param.error != null}">
                              <div class="alert alert-danger">
                                  <p>${errmsg}</p>
                              </div>
                          </c:if>
                    </form>
                </div>
            </div>
        </div>
            <%@ include file="common/foot.jspf" %>
    </body>
</html>
```

Richiamando questi frammenti sempre con l'uso del TAG "include" del JSTL.

Per migliorare l'aspetto della pagina di login si è deciso di inserire un nuovo FORM di login:

- i parametri che vengono passati al form sono gli stessi del precedente (quindi sempre "name" e "password")
- particolarmente interessante l'introduzione del TAG <c:if ...> dove gestiamo la situazione in cui un utente possa inserire le credenziali errate, viene emesso un messaggio di errore, che valorizziamo nella classe LoginController.java

Particolare della classe LoginController.java:

```
@GetMapping(value = "/login")
	public String getLoginPage(Model model) {
		model.addAttribute("intestazione", saluti);
		model.addAttribute("advertising", advertising);
		model.addAttribute("titolo", titolo);
		model.addAttribute("sottotitolo", sottotitolo);
		model.addAttribute("errmsg", errmsg);

		return "login";
	}


	@PostMapping(value = "/login")
	public String goToWelcomePage(
			@RequestParam("name") String name,
			@RequestParam("password") String password,
			ModelMap model
			)
	{
		if (authenticationService.auth(name, password)) {
			model.put("name", name);
			return "welcome";
		}
		else return "redirect:/login?error=nologged";
	}
```

Nel @GetMapping passiamo tramite il Model le variabili con i testi del login form (titolo, sottotitolo, errmsg), mentre nella @PostMapping abbiamo sostituito la return del blocco else, non con il ritorno alla pagina di login e basta, ma con il ritorno alla pagina di login e il passaggio del messaggio di errore "errmsg".

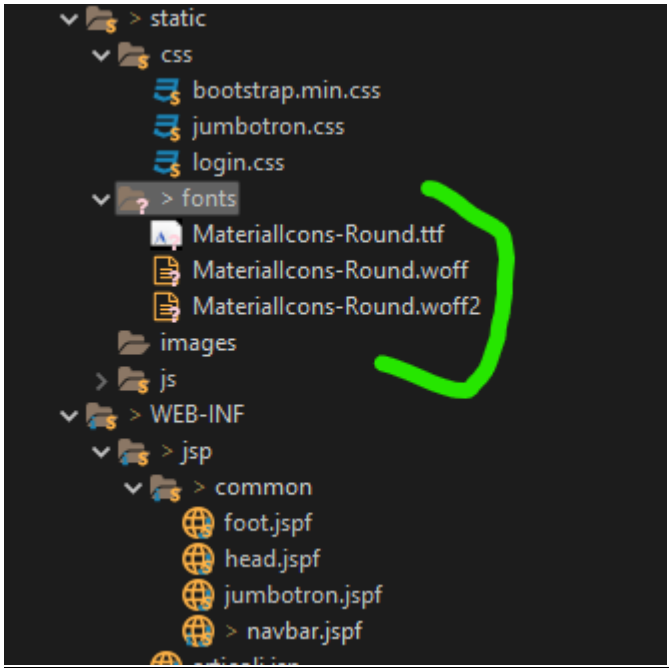
Importante → per popolare i titoli nella nostra pagina di login, sempre nella classe LoginController.java si sono definite queste stringhe:

```
private static final String titolo = "Accesso & autenticazione";
	private static final String sottotitolo = "Procedi ad inserire lo userid e la password";
	private static final String errmsg = "Spiacente, la userid e la password sono errati";
```

e poi passati nei vari frammenti grazie al Model.

Migliorare l'estetica della navigation bar

Usiamo le Material Icons di uso per poter visualizzare le icone sulle voci di menù della nostra web application.



Abbiamo aggiunto 3 files nel percorso /static/fonts per poter usare le Material Icons. Inoltre abbiamo copiato e incollato dal progetto sotto il percorso /static/css due nuovi fogli di stile "round.css" e "navbar.css", ricordandosi sempre di linkarli correttamente nella nostra head:

```
<head>
		<!-- Required meta tags -->
	<meta charset="utf-8">
	<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
	<!-- Bootstrap CSS -->
	<link href="<c:url value="/static/css/bootstrap.min.css" />" rel="stylesheet">
	<link href="<c:url value="/static/css/jumbotron.css" />" rel="stylesheet">
	<link href="<c:url value="/static/css/login.css" />" rel="stylesheet">
	<link href="<c:url value="/static/css/round.css" />" rel="stylesheet">
	<link href="<c:url value="/static/css/navbar.css" />" rel="stylesheet">
		<title>${title}</title>
	</head>
```

Migliorare l'estetica dell'elenco degli articoli

- aggiunti i seguenti file css → main.css e articoli css
- linkati al solito modo dentro il file head.jspf
- modifica della jsp della view degli articoli introducendo nuovo codice html e jstl in particolare → `<fmt:formatNumber value="${article.prezzo}" type="currency" currencyCode="USD" />`

Importante → <fmt:formatNumber ... > è il tag JSTL che possiamo usare per la formattazione, in questo caso della valuta.

Introduzione ai Database

Cosa sono i database → sono dei software che memorizzano e gestiscono quantità di dati. I db strutturano i dati in specifiche entità chiamate tabelle, documenti, permettono l'inserimento di indici in uno o più campi chiave e minimizzano i tempi di ricerca del dato. Permettono di impiegare uno specifico linguaggio (ad esempio: SQL) o l'uso di software creati ad hoc per gestire i dati in maniera strutturata e flessibile. I db sono usati per uno specifico scopo e possono essere di diverse tipologie → relazionali, nosql ...

Obiettivo dei database → ottenere i dati velocemente in maniera strutturata.

Database relazionali → concetto che risale alla fine degli anni 70, ma al momento è il tipo di database più diffuso ed usato, si basa sul modello relazionale, un modello dove i dati sono organizzati all'interno in forma di tabelle. Le tabelle sono composte da righe e colonne. Ogni tabella ha un chiave primaria per identificare i dati, una riga di una tabella rappresenta una singola istanza, mentre le colonne rappresentano gli attributi o campi dei dati. La relazione tra le varie tabelle si realizza attraverso quella che si chiama "chiave esterna", che è un campo che fa riferimento ad un campo chiave di un'altra tabella. L'accesso ai dati avviene tramite il linguaggio SQL. Questo linguaggio permette la manipolazione del dato e l'interrogazione dello stesso.

Limite → una volta deciso come organizzare i dati, e dopo averli inseriti nelle tabelle non c'è modo di avere una struttura dinamica, in quanto se volessi cambiare la definizione dei dati devo riadattarli da capo.

Database NOSQL → quando si ha l'esigenza di avere maggiore flessibilità nella struttura dei dati conviene usare i db nosql, progettati per gestire grandi quantità di dati non strutturati e difficili da organizzare in schemi tabellari.

Introduzione a PostgreSQL

PostgreSQL → è un RDBMS open source e gratuito, che offre una vasta gamma di caratteristiche:

- open source
- uso gratuito
- modifica e distribuzione del software
- basato sull'architettura client/server

Supporto per dati strutturati e non strutturati (json e xml).

Le Transazioni sono ACID → atomicity, consistency, isolation e durablity.

Supporta i vincoli per mantenere l'integrita dei dati, si possono definire chiavi primarie ecc...

Supporta SQL standard, più una estensione al linguaggio per creare funzioni avanzate sul linguaggio SQL.

Postgres è in grado di gestire grandi volumi e scalare in modo efficiente. È sicuro sui dati con autenticazione e autorizzazione basata sui ruoli.

Sito del prodotto → PostgreSQL: The world's most advanced open source database

Installazione Postgres tramite Docker for Windows

Passi da compiere, prima di prelevare il container del Postgres:

- installare Docker for Windows → Docker Desktop: The #1 Containerization Tool for Developers | Docker
- configurare Docker desktop perché si esegue in modo WSL2 → Docker Desktop WSL 2 backend on Windows | Docker Docs
- Verifica del funzionamento di docker → aprire la powershell e dare i seguenti comandi: docker –version e docker-compose –version

Provare a fare la pull del container HelloWorld e vedere se funziona correttamente come proposto dalla documentazione ufficiale sul sito di Docker.

Attivazione del container del Postgres SQL

Si tratta di un file docker-compose.yml vediamo di capirne un po' le parti del codice. Il docker-compose è un software che fa parte sempre dell'ecosistema di Docker e ci permette di creare delle strutture di reti di Container da usare per lo sviluppo. In particolare in questo file yaml definiamo 2 container:

- Il postgres SQL
- Il PgAdmin (web app per gestire il postgres)

```
-   networks:
-     ntpgsql:
-       driver: bridge
-       ipam:
-         driver: default
-         config:
-           - subnet: 172.21.0.0/24
```

In questo punto si richiede la configurazione di una network docker con la subnet 172.21.0.0/24

```
services:
  pgsql:
    image: postgres:latest
    restart: unless-stopped
    container_name: postgresql
    volumes:
      - psdb-volume:/var/lib/postgresql/data
    networks:
      ntpgsql:
        ipv4_address: 172.21.0.2
    ports:
      - target: 5432
        published: 5432
        protocol: tcp
        mode: host
    environment:
      - POSTGRES_PASSWORD=123_Stella
```

In questo spezzone di codice viene configurato il container del Postgres SQL, viene richiesta l'immagine, l'opzione di start del container, quando viene avviato il Docker, gli viene dato un nome al container, gli viene dato pure un volume su disco dove il programma può mantenere i suoi files. Nella

sezione della rete gli viene assegnato un IP coerente con lo schema della rete definita sopra, poi vengono configurate le porte su cui far ascoltare il container. Se per caso esiste già un servizio sulla porta scelta allora sarà sufficiente modificare la porta "published" dandole un altro numero.

Ultimo elemento importante che si è deciso di configurare è la variabile di ambiente che sarà la password dell'utente amministratore. L'utente amministratore di Postgres è appunto **postgres**.

Il secondo blocco invece:

```
pgadmin:
    image: dpage/pgadmin4
    restart: unless-stopped
    container_name: pgadmin4
    networks:
      ntpgsql:
        ipv4_address: 172.21.0.3
    ports:
      - target: 80
        published: 80
        protocol: tcp
        mode: host
    environment:
      - PGADMIN_CONFIG_SERVER_MODE=True
      - PGADMIN_DEFAULT_EMAIL=nicola@xantrix.it
      - PGADMIN_DEFAULT_PASSWORD=123_Stella
```

Riguarda la configurazione della web app che gestirà il Postgres→PgAdmin. Anche qui viene prelevata una immagine, c'è l'opzione dell'avvio automatico allo start del docker, un nome del container, una network di riferimento, la definizione delle porte e una opzione di protocollo, più alcune variabili di ambiente.

Importante → non ci resta che provare, andiamo sul terminale nel mio caso ho copiato e incollato il file docker-compose.yml nel percorso c:/docker/postgres/, quindi mettiamoci in questo percorso e ci basta digitare il seguente comando → docker-compose up -d

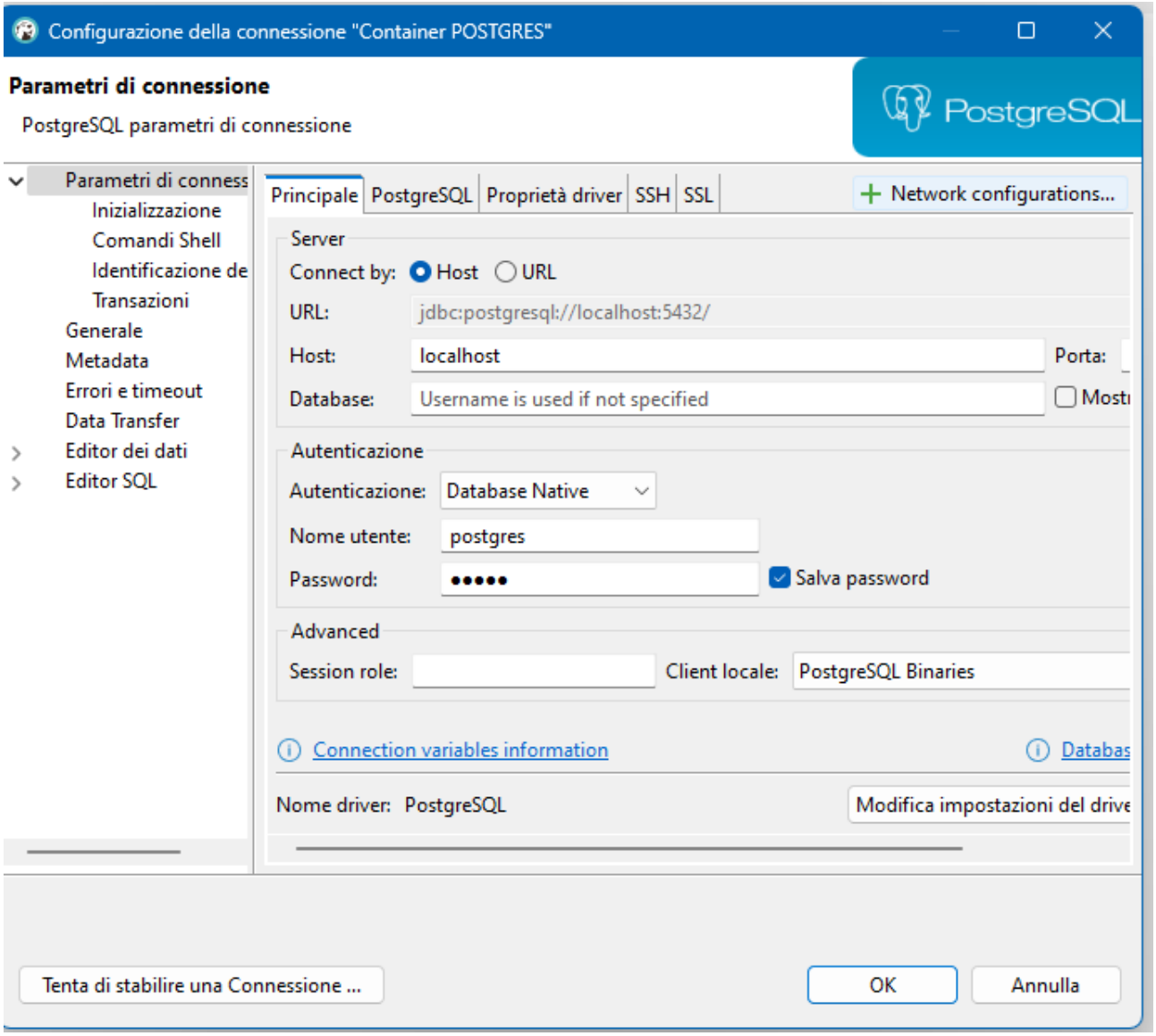
Aspettiamo che finiscano il download dei containers dal docker hub e una volta che sono funzionanti dovremmo vedere scritto il seguente log:

```
2024-06-15 09:51:28 2024-06-15 07:51:28.949 UTC [60] LOG:  checkpoint complete: wrote 44
 average=0.004 s; distance=261 kB, estimate=261 kB; lsn=0/152B660, redo lsn=0/152B628
2024-06-15 09:46:24 2024-06-15 07:46:24.535 UTC [49] LOG:  shutting down
2024-06-15 09:46:24 2024-06-15 07:46:24.541 UTC [49] LOG:  checkpoint starting: shutdown
2024-06-15 09:46:24 2024-06-15 07:46:24.602 UTC [49] LOG:  checkpoint complete: wrote 3 b
verage=0.003 s; distance=0 kB, estimate=0 kB; lsn=0/14EA208, redo lsn=0/14EA208
2024-06-15 09:46:24 2024-06-15 07:46:24.605 UTC [48] LOG:  database system is shut down
2024-06-15 09:46:24  done
2024-06-15 09:46:24 server stopped
2024-06-15 09:46:24
2024-06-15 09:46:24 PostgreSQL init process complete; ready for start up.
2024-06-15 09:46:24
```

Da browser, secondo il mio setup, l'applicazione PgAdmin la si raggiunge tramite l'indirizzo → http://localhost:5435 e poi si mettono le credenziali per accedere definite nella configurazione del docker-compose.yml:

```
- PGADMIN_DEFAULT_EMAIL=xxxxxxxxxxxxxx@xxxxx.it
      - PGADMIN_DEFAULT_PASSWORD=xxxxxx
```

In alternativa è possibile configurare la connessione con strumenti come DBeaver in questo modo→



Introduzione alle classi di Entità

Nello sviluppo di applicazioni basate sui database, in Spring Boot le classi **Entity** sono gli oggetti persistenti che vengono memorizzati nel database.

Le classi Entity sono classi Java che mappano il database. Mappano i record di una tabella di un database in oggetti Java. Usiamo l'annotazione @Entity per dichiarare la classe come Entity. Questa annotazione indica a Spring che la classe è una tabella del database. Abbiamo poi l'annotazione **@Table** usata per specificare il nome della tabella nel database cui fare riferimento con la nostra classe Entity. Se il nome della classe coincide con la tabella questa annotazione si può tralasciare.

Attributi della entity → rappresentano le colonne della tabella nel database. Ogni attributo deve essere annotato con l'annotazione appropriata per indicare il suo mapping nel database. Ad esempio l'annotazione **@Id** si usa per gli attributi che sono chiavi primarie, mentre l'annotazione **@Column** la usiamo per specificare il nome della colonna della tabella da mappare.

Getter e setter → le classi entity devono fornire metodi getter e setter per tutti gli attributi, in modo che Spring possa accedere in modifica, possiamo autogenerarli grazie all'uso del Lombok.

Relazioni tra le entity → possiamo creare relazioni tra le classi entity allo stesso modo in cui vengono realizzate le relazioni tra le tabelle di un datanase, abbiamo relazioni uno-uno, uno-molti e molti-molti, per questo in Spring vengono usate le seguenti annotazioni per definirle:

- @OneToOne
- @OneToMany
- @ManyToMany
- @ManyToOne

Inoltre nelle nostre classi entity possiamo specificare diverse informazioni aggiuntive:

- La dimensione dei campi
- Informazioni di validità dei campi

Importante → le classi Entity in Spring Boot consentono di mappare oggetti Java direttamente alle tabelle del database e di usare le funzionalità ORM, fornite da Spring per manipolare i dati nel database in modo semplice ed intuitivo.

Usare le entity nel nostro progetto del corso

Per prima cosa introdurre 2 nuove dipendenze:

```
<dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
            <version>${spring-jpa.version}</version>
            </dependency>
<dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <scope>runtime</scope>
            <version>${postgresql.version}</version>
            </dependency>
```

E poi adesso andiamo ad inserire la nostra prima classe entità:

```
@Entity
@Table(name = "articoli")
public class Articoli {
	@Id
	@Column(name = "codart")
	private String codArt;

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

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

	@Column(name = "codstat")
	private String codStat;

	@Column(name = "pzcart")
	private Integer pzCart;

	@Column(name = "pesonetto")
	private double pesoNetto;

	@Column(name = "idstatoart")
	private double idStatoArt;

	@Temporal(TemporalType.DATE)
	@Column(name = "datacreazione")
	private Date dataCreaz;
}
```

In questa classe mancano le chiavi esterne, ma le andiamo a mettere più avanti quando avremo completato le restanti tabelle del progetto, che permettono la relazione con la tabella degli articoli.

Notare l'uso delle annotazioni→@Entity, @Table, @Id, @Column e @Temporal sugli atributi della entity.

Creazione delle classi entità

```
@Entity
@Table(name = "barcode")
public class Barcode {

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

	@Column(name = "idtipoart")
	private String idTipoArt;
}
```

```
@Entity
@Table(name = "famassort")
public class FamAssort {

	@Id
	@Column(name = "id")
	private Integer id;

	@Column(name = "descrizione")
	private String descrizione;
}
```

```java
@Entity
@Table(name = "ingredienti")
public class Ingredienti {

	@Id
	@Column(name = "codart")
	private String codArt;

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

}
```

```java
@Entity
@Table(name = "iva")
public class Iva {
	@Id
	@Column(name = "idiva")
	private Integer idIva;

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

	@Column(name = "aliquota")
	private Integer aliquota;
}
```

Introduzione alle relazioni tra le classi di entità

Relazioni che intercorrono tra le entity:

- Articoli → Barcode è una relazione 1 a N, ad un articolo possono corrispondere più Barcode

La relazione esiste per via del campo codart dentro la tabella degli articoli.

Concetto di Relazione in Spring Boot → Il framework dà la possibilità di creare delle relazioni tra le entità utilizzando le annotazioni di persistenza JPA (Java Persistence Api), le relazioni possono essere di diversi tipologie, come visto in precedenza, uno a uno, uno a molti e molti a molti.

La notazione **@OneToMany** → viene usata per stabilire una relazione uno-a-molti tra due entità, questa relazione permette di definire una associazione in cui una entità ha una relazione "uno" con un'altra entità che ha una relazione "molti". Nel dettaglio ciò che dobbiamo fare ad esempio:

- Prima di tutto definire le entità in gioco, le due entità coinvolte, dove l'entità "uno" è quella principale, mentre l'entità "molti" viene classificata come entità figlia.
- Nell'entità principale si usa la notazione **@OneToMAny** per definire la relazione, ciò indica che un'istanza della entità principale può essere associata a molteplici istanze dell'entità figlia.
- Nell'entità figlia si usa invece la notazione **@ManyToOne** per definire la relazione inversa. Questa notazione indica che molteplici istanze dell'entità figlia possono essere associate a un'unica istanza dell'entità principale.

Parametri della notazione @OneToMany:

- **mappedBy** → è un parametro che specifica il nome del campo della classe figlia che mappa la relazione, viene usato per creare il collegamento tra le due entità. Il valore di mappedBy dovrebbe corrispondere al nome del campo nella classe figlia.
- **cascade** → parametro che specifica le operazioni di cascade che si deve applicare alla relazione, ad esempio se si desidera che le operazioni di salvataggio o eleminazione sulla classe principale si propaghino automaticamente alla classe figlia si può specificare CascadeType.ALL o altre opzioni di combinazioni associate
- altri parametri usati sono fetch (FetchType.LAZY), orphanRemoval (rimuove le entità figlie) ecc…

Esempio da progetto del corso:

dentro la classe Articoli.java

```
@OneToMany(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy =
"articolo", orphanRemoval = true)
	private Set<Barcode> barcode = new HashSet<>();
```

e poi dentro la classe Barcode.java

```
@ManyToOne
	@JoinColumn(name = "codart", referencedColumnName = "codArt")
	private Articoli articolo;
```

Approfondimento del parametro Fetch e Cascade

Quando usiamo il **fetch = FetchType.LAZY** l'associazione viene considerata come un "proxy" o un "riferimento" all'entità associata. L'associazione non viene caricata immediatamente quando si recupera l'entità principale del database. L'associazione viene caricata soltanto quando si accede veramente ad essa, ad esempio con un getter per l'associazione. Questa configurazione permette di ridurre il traffico di rete e migliorare le prestazioni dell'applicazione, perché solo le associazioni necessarie verranno recuperate dal database.

Quando invece si usa **fetch = FetchType.EAGER**, il caricamento è immediato e insieme all'entità principale quando viene recuperata dal database, ciò vuol dire che le associazioni contrassegnate con eager vengono recuperate anche se non sono effettivamente usate. Ciò provoca un carico aggiuntivo, soprattutto se ci sono tante associazioni.

La scelta della configurazione Lazy o Eager dipende sempre dall'applicazione. Di norma si preferisce il Lazy. Eager si usa soltanto se si sa che l'associazione sarà sempre necessaria.

Cascade → specifica il comportamento di cascata delle operazioni di persistenza. Ciò vuol dire che lo uso quando ad esempio si vuole propagare la persistenza e l'aggiornamento alle entità collegate alla principale. Nel contesto di Spring Boot si usa con le annotazioni JPA, come @OneToMany, @ManyToOne ecc…

Valori di cascade:

- CascadeType.ALL → ciò significa che tutte le operazioni di persistenza sul padre vengono propagate anche alle entità figlie
- CascadeType.PERSIST → ciò vuol dire che l'operazione di persistenza (inserimento) eseguita sul padre deve essere propagata alle entità figlie correlate
- CascadeType.MERGE → ciò vuol dire che l'operazione di merge(aggiornamento) eseguita sul padre deve essere propagata alle entità figlie correlate

- CascadeType.REMOVE → ciò vuol dire che l'operazione di rimozione eseguita sull'entità padre deve essere propagata alle entità figlie correlate.
- CascadeType.REFRESH
- CascadeType.DETACH
- CascadeType.REPLICATE…

Introduzione alla annotazione @OneToOne

Esempio: relazione Articoli → Ingredienti è del tipo uno-a-uno. Entrambi campi di join sono chiavi primarie. Per replicare lo stesso comportamento lato Entity, ci serve usare la notazione spring **@OneToOne** → questa annotazione si usa in spring per stabilire una relazione uno-a-uno tra 2 entità. Si usa insieme alle altre notazioni JPA per definire il mappaggio del database. L'annotazione viene applicata ad un campo o un metodo getter nella classe di entità che rappresenta il lato proprietario della relazione. L'annotazione di può usare con o senza l'annotazione @JoinColumn. Quando invece si usa insieme Join Column allora si possono specificare i dettagli sulla colonna della chiave esterna. @JoinColumn si usa quindi per specificare il nome della colonna della chiave esterna nella tabella dell'entità proprietaria.

Dentro Articoli.java scriveremo quindi:

```
@OneToOne(cascade = CascadeType.ALL, mappedBy = "articolo", orphanRemoval =
true)
	private Ingredienti ingredienti;
```

Il fetch type nel caso del one-to-one non ha ragione di esistere.

Mentre dentro Ingredienti.java scriviamo questo:

```
@OneToOne
	@PrimaryKeyJoinColumn
	private Articoli articolo;
```

Con l'annotazione @PrimaryKeyJoinColumn stiamo dicendo che la relazione avviene attraverso una chiave primaria.

In questo modo la relazione one-to-one tra le 2 entity è di fatto completa.

Conclusione della creazione delle relazioni per le restanti entità del progetto

Mancano due relazioni:

- Articoli → Iva
- Articoli → Famassort

In queste relazioni sono Iva e Famassort ad essere le tabelle padre e no Articoli, come invece era nelle precedenti configurazioni.

Ad un record dell'iva corrispondono più articoli. E anche ad una famiglia assortimento di tipo "alimentari" corrisponderanno decine di articoli e così via…

Quindi sono relazioni uno-molti ma nel senso contrario rispetto a come abbiamo incontrato prima.

Iva e Famassort inoltre sono tabelle "dizionario", cioè tabelle che non vengono alterate nella fase di inserimento degli articoli! Questo aspetto comporta una precisa configurazione lato parametri della entity che andremo a realizzare.

Quindi nella classe Iva.java avremo una notazione uno-a-molti, il motivo per cui la tabella Iva non verrà mai coinvolta in nessuna modifica quando verranno inseriti gli articoli è dato dal fatto che non introduciamo nessuno attributo per il comportamento del cascade:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy
="iva")
    private Set<Articoli> articoli = new
HashSet<>();
```

e nella classe FamAssort.java;

```
@OneToMany(fetch = FetchType.LAZY, mappedBy =
"famAssort")
    private Set<Articoli> articoli = new
HashSet<>();
```

Nella classe entity Articoli.java invece:

```
@ManyToOne
    @JoinColumn(name = "idiva",
referencedColumnName = "idiva")
    private Iva iva;

    @ManyToOne
    @JoinColumn(name = "idfamass",
referencedColumnName = "id")
    private FamAssort famAssort;
```

In questo modo mappiamo tutte le relazioni in merito, ma non abbiamo ancora finito ci mancano ancor alcuni dettagli su queste classi entity.

Aggiunta della notazione di LOMBOK

Oltre ad implementare le configurazioni delle relazioni tra le varie tabelle, dobbiamo anche implementare @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor...

Per le nostre classi di entity inseriamo tutte queste annotazioni:

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class Iva {
    @Id
    @Column(name = "idiva")
    private Integer idIva;
```

Introduzione allo strato della persistenza

Strato di persistenza → è lo strato di accesso ai dati che si occupa di interagire con il database, in un progetto Spring Boot di norma viene implementato usando un ORM (object relational mapping), in cui le classi Java vengono mappate su tabelle relazionali (le nostre entity). Spring Boot fornisce diverse opzioni:

- Spring Data JPA
- Hibernate
- JPA (java persistence api)

Componenti principali che costituiscono lo strato di persistenza di un progetto Spring Boot sono:

- **Le classi entity** (entità)
- **I repository** → interfacce che gestiscono metodi per l'accesso e la gestione dei dati, vengono annotati con l'annotation @Repository, queste classi speciali hanno metodi di basi propri come save(), delete() o findById(), oppure possono ospitare metodi custom definiti dall'utente.
- **JPA** → è una specifica standard per la gestione della persistenza dei dati in JAVA. Spring Boot integra JPA e fornisce Hibernate per l'inplementazione del mapping ORM. JPA definsice un set di annotazioni importanti come: @Entity, @Id, @GeneratedValue …
- **Configurazione del database** → spring boot semplifica la configurazione del database attraverso file di properties come application.properties o application.yml, qui posso definire il driver, gli accessi ecc..
- **Spring Data JPA** → è un modulo di spring framework che semplifica lo sviluppo di applicazioni basate su JPA per l'accesso ai dati. Offre una astrazione aggiuntiva sopra il JPA cercando di ridurre il codice boilerplate necessario per interagire con il database e ne semplifica le operazioni di persistenza con i dati. Principali caratteristiche sono una interfaccia repository generale che offre operazioni CRUD di base e operazioni di query, questa interfaccia può essere estesa per creare repository specifici. Ci sono poi Query Methods, cioè una convenzione basata sul nome (es. findByUsername(String username)), e anche Query annotation attraverso @Query per definire query con il linguaggio JPQL, supporto alla paginazione, supporto alla transazione con la annotazione @Transactional → questa annotazione che posso usare a livello di classe o metodo mi permette di definrie il comportamento delle operazioni di repository. Grazie a questa annotazione possiamo attivare la gestione delle transazioni a livello dello strato di persistenza.

Le interfacce di Spring Data JPA:

- Repository<T, ID> è l'interfaccia di base
- CrudRepository<T, ID> → implementa operazioni di base CRUD ed estende l'interfaccia base
- PagingAndSortingRepository<T, ID> → estende la crud interfaccia e aggiunge operazioni di persistenza avanzate e soprattutto il supporto alla paginazione e all'ordinamento sui dati ritornati
- JPARepository<T, ID> → aggiunge funzionalità specifiche di JPA
- JpaSpecificationExecutor<T>
- QueryByExampleExecutor<T>
- JpaRepositoryImplementation<T, ID>

Queste si possono estendere o combinare per creare repository personalizzati.

Esempio di creazione dello strato di persistenza di una applicazione Spring Boot

1) Creazione di un nuovo package → com.demo.example.repositories questo conterrà tutte le classi/interfacce dello strato di persistenza. Adesso creiamo una nuova interfaccia
2) Creiamo l'interfaccia ArticoliRepository che estendiamo a jpa repository

```
3) package com.demo.example.repositories;
4)
5) import
   org.springframework.data.jpa.repository.JpaRepository;
6) import org.springframework.stereotype.Repository;
7) import com.demo.example.entities.Articoli;
8)
9) @Repository
10)    public interface ArticoliRepository extends
   JpaRepository<Articoli, String> {
11)
12)        Articoli findByCodArt(String codArt);
13)    }
```

La nostra interfaccia repository ha un query method e per la documentazione guardare qui → JPA Query Methods :: Spring Data JPA

Ecco qui sotto la nostra interfaccia repository completa di tutti i metodi per filtrare i dati dal database dove abbiamo utilizzato svariate modalità offerte dal framework per interrogare i dati:

```
@Repository
public interface ArticoliRepository extends JpaRepository<Articoli, String> {
        // query method example
        Articoli findByCodArt(String codArt);
        // query method example
        List<Articoli> findByDescrizioneLike(String descrizione, Pageable pageable);
        // query method example
        List<Articoli> findByCodStatOrderByDescrizione(String codStat);
        // JPQL method example
        @Query(value = "SELECT a FROM Articoli a JOIN a.barcode b WHERE b.barcode IN (:ean)")
        Articoli selByEan(@Param("ean") String ean);
        // native query method example
        @Query(value = "SELECT COUNT(*) FROM ARTICOLI WHERE DESCRIZIONE LIKE :desArt", nativeQuery = true)
        int countRecords(@Param("desArt") String descrizione);
}
```

Creazione dello strato di servizio

Lo strato di servizio di una applicazione Spring →