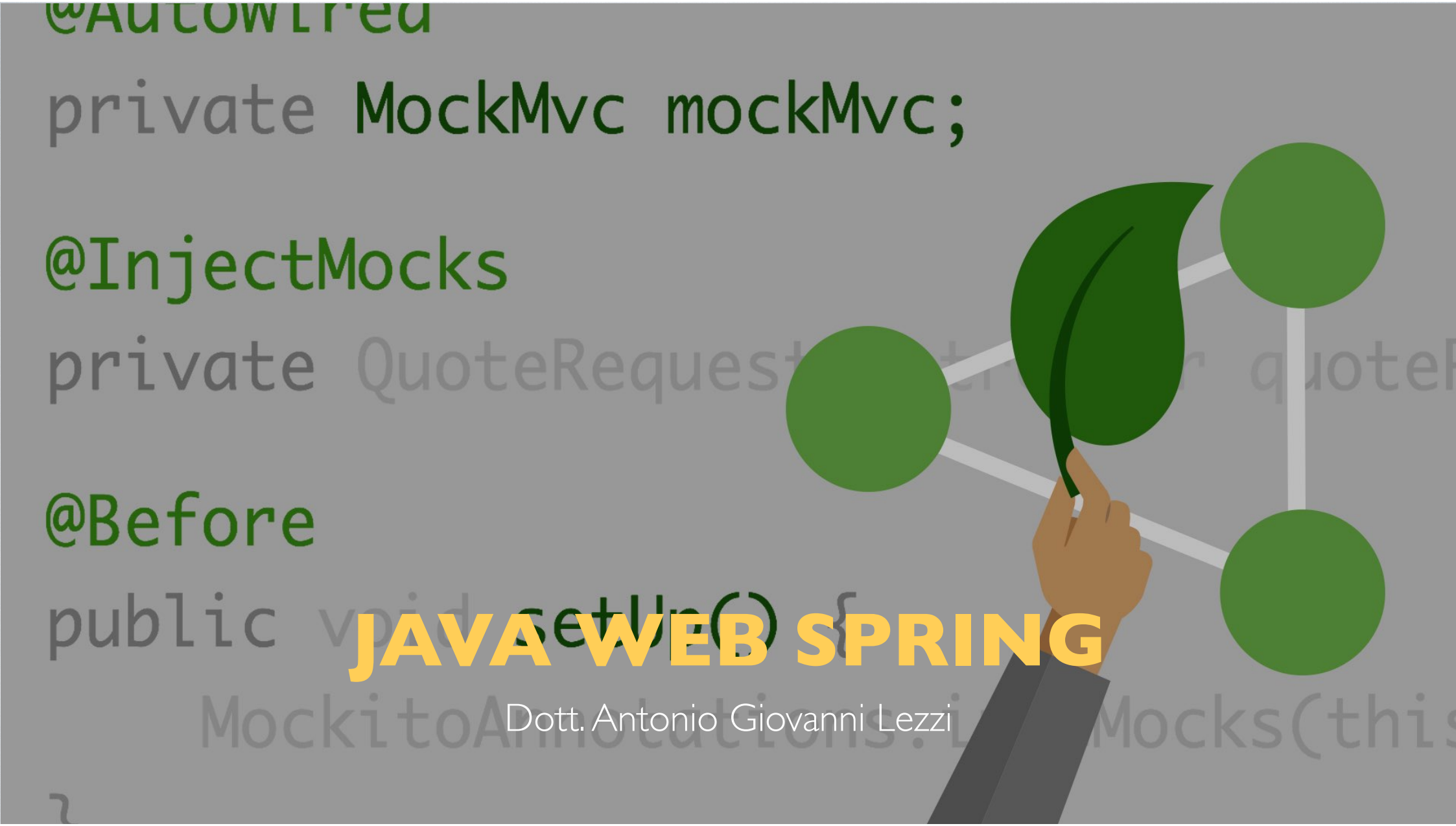
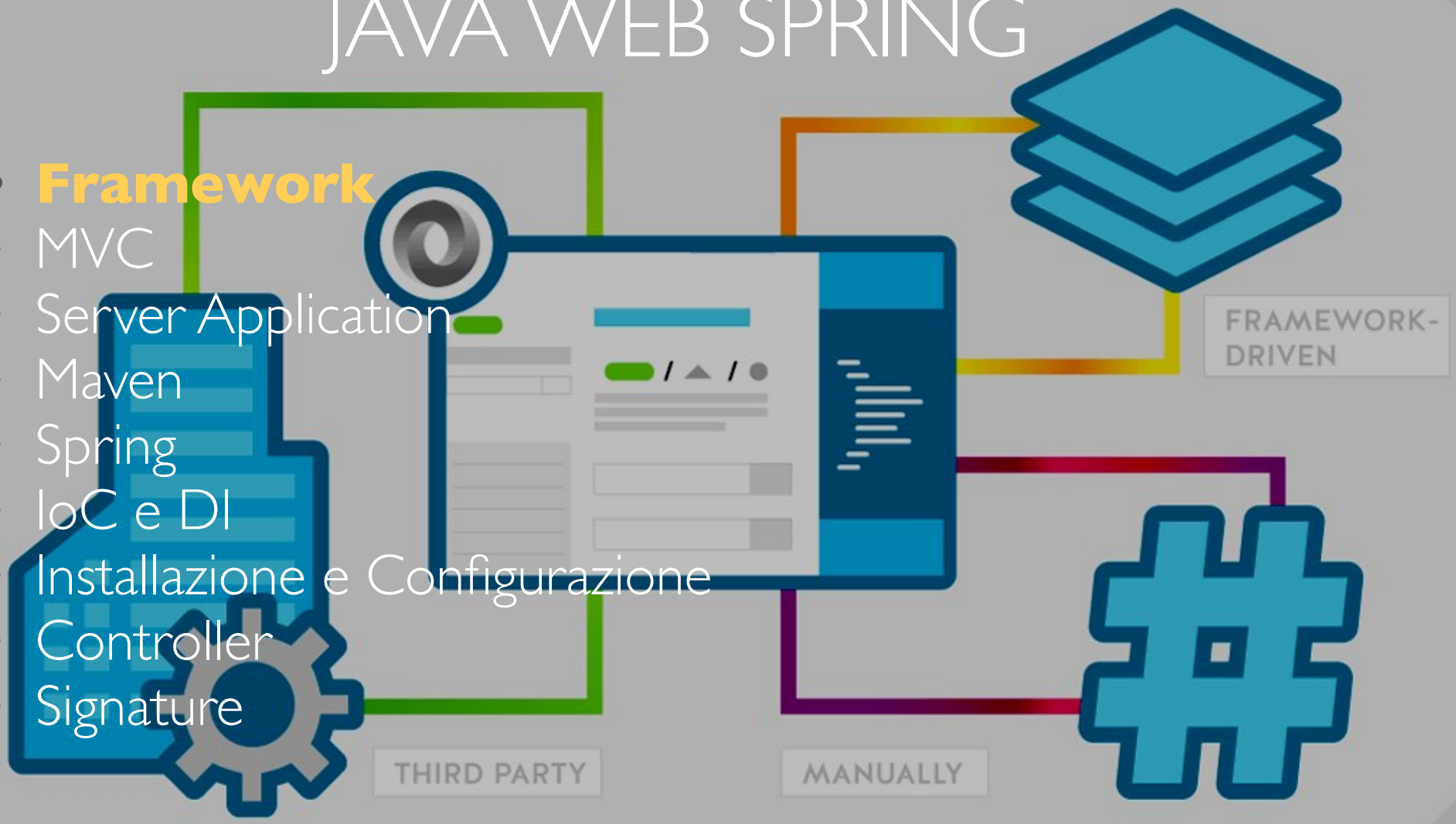


JAVA WEB SPRING

Dott. Antonio Giovanni Lezzi

JAVA WEB SPRING

- **Framework**
- MVC
- Server Application
- Maven
- Spring
- IoC e DI
- Installazione e Configurazione
- Controller
- Signature



FRAMEWORK

- Un **framework**, può essere tradotto come intelaiatura o struttura.
- In informatica e specificatamente nello sviluppo software, è un'**architettura logica di supporto** (spesso un'implementazione logica di un particolare design pattern) su cui un **software può essere progettato e realizzato**, spesso **facilitandone lo sviluppo** da parte del programmatore.
- Un framework è definito da un **insieme di classi astratte e dalle relazioni tra esse**.

CARATTERISTICHE

- Utilizzare un framework significa fornire un'implementazione delle classi astratte. L'insieme delle **classi concrete**, definite ereditando il framework, **eredita le relazioni tra le classi**; si ottiene in questo modo un insieme di classi concrete con un insieme di relazioni tra classi.
- Alla base di un framework c'è sempre **una serie di librerie di codice utilizzabili in fase di linking con uno o più linguaggi di programmazione**, spesso **corredate da una serie di strumenti di supporto allo sviluppo del software**, come ad esempio un IDE, un debugger o altri strumenti ideati per aumentare la velocità di sviluppo del prodotto finito.
- L'utilizzo di un framework impone dunque al programmatore una precisa metodologia di sviluppo del software.

SCOPO

- La sua funzione è quella di **offrire una infrastruttura generale, lasciando al programmatore il contenuto vero e proprio dell'applicazione**.
- Lo scopo di un framework è infatti quello di **risparmiare allo sviluppatore la riscrittura di codice già scritto in precedenza per compiti simili**.
- Questa circostanza si è presentata sempre più spesso man mano che le interfacce utente sono diventate sempre più complesse, o, più in generale, man mano che è aumentata la quantità di software con funzionalità secondarie simili.

ESEMPIO

- Ad esempio, il tipo di interazione con l'utente offerto da un menu a tendina sarà sempre la stessa indipendentemente dall'applicazione cui il menu appartiene (o almeno questo è ciò che l'utente si aspetta);
- in casi come questo un framework, che permette di aggiungere la funzionalità di una finestra con un menu a tendina con poche righe di codice sorgente a carico del programmatore, o magari permettendogli di disegnare comodamente il tutto in un ambiente di sviluppo, permetterà al programmatore di concentrarsi sulle vere funzionalità dell'applicazione, senza doversi far carico di scrivere codice "di contorno".

JAVA WEB SPRING

- Framework
- **MVC**
- Server Application
- Maven
- Spring
- IoC e DI
- Installazione e Configurazione
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

PAGINE WEB

- Il **World Wide Web** è una delle maggiori **fonti di informazione** a cui oggi possiamo attingere: avendo a disposizione un collegamento a Internet ed un **browser Web**, un software ormai comune su qualsiasi computer, con la possibilità di **consultare** un patrimonio di centinaia di milioni di pagine riguardanti praticamente ogni **argomento**.

- Spesso queste pagine **non sono documenti statici**, ma vengono creati dinamicamente quando noi li richiediamo e **le informazioni che contengono vengono estratte da un database**.

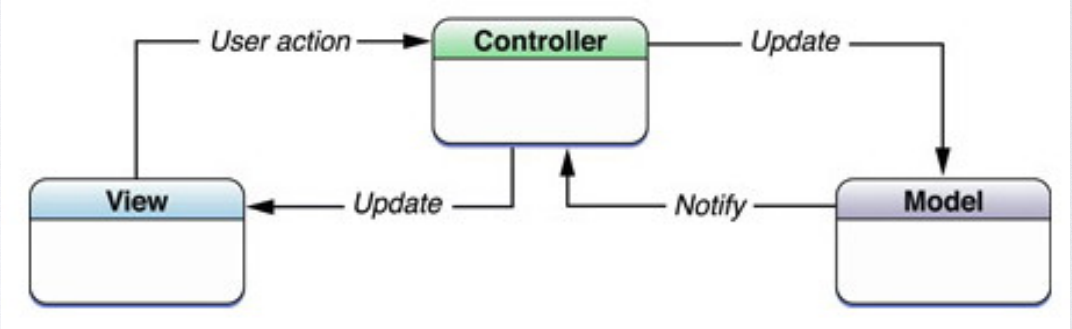
PARADIGMA MVC

- Il design pattern **Model-View-Controller** (MVC) è l'architettura più utilizzata nello sviluppo di applicazioni Web/Mobile ecc.
- È necessario introdurre il design pattern MVC in quanto rappresenta uno dei concetti fondamentali della programmazione ad oggetti e **permette di strutturare l'applicazione in maniera molto efficiente** (massima coesione minima conoscenza).
- Nel design pattern MVC **ogni singolo oggetto può ricoprire solo uno dei seguenti ruoli: modello, vista o controllore**.
- Questi ruoli rappresentano una sorta di classificazione dell'oggetto che stiamo utilizzando che risultano dunque essere **elementi logicamente separati ai quali però è consentita, ovviamente, una stretta comunicazione**.

PARADIGMA MVC

- Dunque, per dare una definizione un po' più rigorosa, **il pattern non solo stabilisce che ruolo deve avere un determinato oggetto** all'interno dell'applicazione**, ma anche il modo in cui gli oggetti comunicano** tra di loro.

- Andiamo adesso a descrivere, in maniera più approfondita, i tre tipi di oggetti che il pattern definisce.



MODELLO

- L'oggetto che ricopre il ruolo di modello è un **oggetto contenente i dati specifici** di un'applicazione e si occupa di definire tutti **le varie procedure** che effettuano **la manipolazione dei dati stessi**.

- Data la sua natura, un modello non dovrebbe avere **nessun tipo di connessione diretta con un oggetto di tipo vista** (che vedremo tra poco), in quanto il modello **ha il compito di possedere solamente dei dati** che però non devono essere legati ad un particolar tipo di visualizzazione.

VISTA

- Il ruolo principale di un oggetto vista è quello di **presentare all'utente i dati contenuti all'interno di un modello**. Esiste subito una netta differenza concettuale tra modello e vista: il primo è un oggetto non concreto, mentre il secondo **è un oggetto concreto e con il quale l'utente può interagire** andando, per esempio, a modificare i dati contenuti.

- Quello che offre **la vista** è dunque una **realizzazione di un oggetto non concreto e mette a disposizione un'interfaccia per la modifica dei dati contenuti nel modello**.

- Dato che anche un **oggetto di tipo vista non dovrebbe avere un riferimento esplicito ad un oggetto di tipo modello** è necessario introdurre il terzo concetto di questo patter: l'oggetto **controllore**.

CONTROLLORE

- L'oggetto che ricopre il ruolo di controllore svolge la funzione di **intermediario tra oggetti di tipo vista ed oggetti di tipo modello**.
- Il numero di relazioni tra un singolo controllore ed oggetti di tipo modelli e vista è arbitraria: può essere una relazione uno ad uno come una relazione molti a molti.
- Il controllore, data la sua natura di mediatore, **si occuperà di inizializzare la vista con i dati contenenti nel modello, informare la vista che i dati all'interno del modello hanno subito modifiche e rendere le modifiche effettuate dall'utente** (tramite la vista) permanenti all'interno del modello.

JAVA WEB SPRING

- Framework
- MVC
- **Server Application**
- Maven
- Spring
- IoC e DI
- Installazione e Configurazione
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

STRUTTURA DEL WEB

Le applicazioni Web sono composte da 3 elementi:

- **URL**: i nomi delle risorse nel Web
- **HTTP**: il modo in cui vengono trasferite le risorse nel Web
- **HTML**: il linguaggio delle risorse più importanti del Web

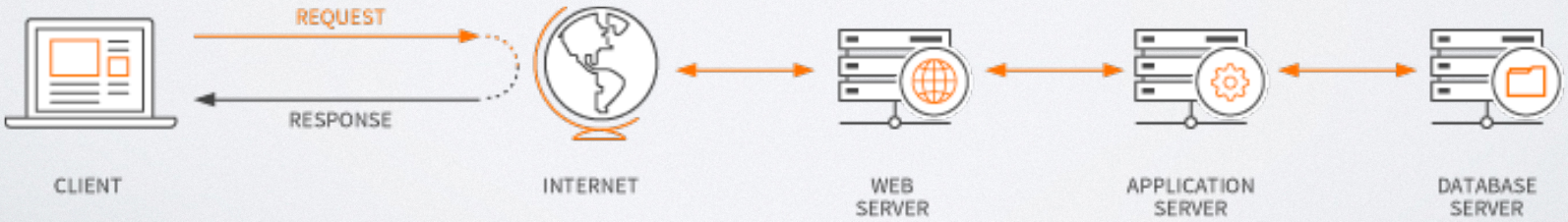
URL

- Gli Uniform Resource Locator ci permettono di individuare in maniera univoca una risorsa nel Web. Corrispondono all'indirizzo che mettiamo nelle buste delle lettere: senza indirizzo non sappiamo dove inviare fisicamente la lettera.
- Un Url è composto dalle seguenti parti:
  - **protocollo**: normalmente http, altri valori sono ftp, telnet, mailto;
  - **user e password (opzionali)**: sono le credenziali per accedere al Web Server;
  - **l'indirizzo del Web Server**: può essere specificato tramite un I.P. address (157.138.20.15) o un nome (www.antonio.it);
  - **una porta (opzionale)**: indica il numero di porta TCP alla quale connettersi;
  - **il path della risorsa** (file normalmente) cercata;
  - **i parametri della richiesta (opzionali)**: importanti per le applicazioni Web permettono di scambiare utili parametri tra il browser e l'applicazione tramite ad esempio i FORM HTML.

HTTP

- L'Hyper Text Transfer Protocol **stabilisce delle regole per il trasferimento delle informazioni**, Il protocollo HTTP prevede le seguenti fasi:

  1. il browser (Internet Explorer/Safari) apre una connessione col Web server specificato nel URL;
  2. il browser invia la richiesta al Web server attraverso la connessione;
  3. il Web server invia la risposta al browser attraverso la connessione;
  4. il server chiude la connessione.



HTTP

- Le richieste inviate al server normalmente sono richieste di lettura di file (file HTML, testo, immagini etc.).
- Le richieste di lettura di file avvengono tramite il richieste HTTP di tipo GET.
- Talvolta le richieste fatte al Web server riguardano l'invio di dati dal Browser al Server, queste richieste avvengono tramite richieste HTTP di tipo POST.

HTML

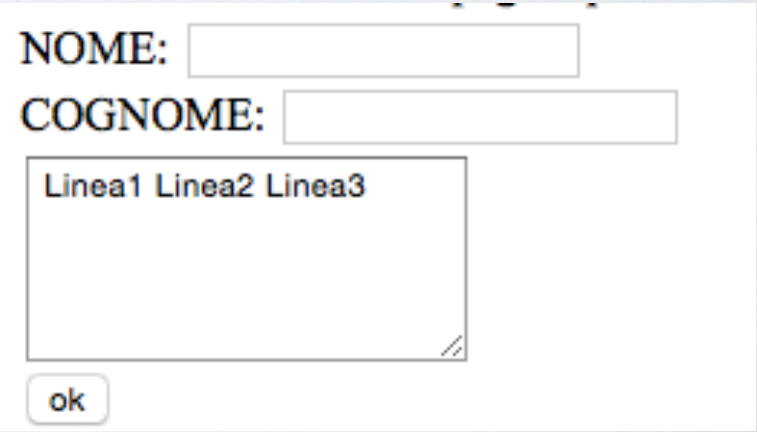
- L'HyperText Markup Language è il linguaggio con il quale scrivere i documenti scambiati nel Web.
- Il file HTML **specifica al browser l'aspetto grafico e alcuni aspetti semantici della pagina da visualizzare**. Il linguaggio HTML ha una sintassi molto semplice che prevede i seguenti TAG principali:
  - **<html>** è il tag di apertura che indica l'inizio del contenuto HTML del file;
  - **<body>** è il tag di inizio della parte "visibile" del file HTML;
  - **<a href="URL">** è il tag principale per specificare gli aspetti semantici di un file html, cioè i collegamenti allo stesso o altri file. Questi collegamenti vengono specificati tramite il loro "nome" ovvero l'Url;
  - **<form action="URL">** è il tag che indica l'inizio di un modulo dove l'utente può inserire o selezionare dei dati da inviare dal browser al Web server.
- I tag HTML possono avere un corrispondente tag di chiusura composta nel seguente modo: </tag>. Ad esempio i tag di chiusura per i tag elencati prima sono: </html>,</body></a></form>.

HTML FORM

- I form sono usati principalmente per trasferire dati dal client al server. All'interno della coppia di tag di apertura e chiusura del form possono comparire una varietà di altri tag quali:
- ***<INPUT NAME=nome TYPE=tipo VALUE=valore>*** che serve per introdurre un testo, può avere uno dei seguenti tipi:
- **TEXT**: per introdurre un testo libero di una riga;
- **SUBMIT**: visualizza un pulsante per la sottomissione;
- **PASSWORD**: per introdurre un testo nascosto;
- **FILE**: per introdurre un file.
- *<TEXTAREA NAME=nome ROWS=n>* per introdurre più linee di testo.

ESEMPIO HTML

```html
<html>
<body>
    <form method="GET" action="testform.jsp">
        NOME:
        <input type="text" name="nome" value="">
        <BR>
        COGNOME:
        <input type="text" name="cognome" value="">
        <BR>
        <textarea name="linee" rows=5>
            Linea1
            Linea2
            Linea3
        </textarea>
        <BR>
        <input type="submit" value="ok">
    </form>
</body>
</html>
```



APPLICAZIONI NEL BROWSER

- Analizzando il funzionamento del protocollo HTTP possiamo vedere alcune sue particolarità che condizionano il funzionamento delle applicazioni Web.
- **Ogni richiesta di risorsa richiede una nuova connessione dal browser al Web server**, questo comporta un rallentamento dovuto al fatto che la gestione della connessione TCP ne incrementa le prestazioni con l'uso della connessione stessa.
- Sempre perché dopo **l'invio della risorsa il server chiude la connessione, non c'è nessuna interazione tra browser e server fino alla prossima richiesta da parte del client** (PULL).
- **Inoltre il protocollo HTTP è senza stato** e questo ci complica le cose quando dobbiamo mantenere delle informazioni riguardo le richieste precedenti.

APPLICAZIONI NEL BROWSER

- Altro problema è rappresentato dal fatto che il **Web server ritorna file HTML e i tipi di funzionalità**, è quella permessa dal linguaggio HTML all'inizio ideato solo per facilitare la navigazione, ora invece si tende ad usare il browser per implementare le funzionalità di un generico client.

- Le funzionalità aggiunte al linguaggio HTML è **Javascritpt** un linguaggio di **scripting in grado di programmare il funzionamento del browser**. Javascript permette di aumentare l'interattività del browser senza bisogno di interagire col server

SERVER HTTP

- Il server HTTP o server Web (ad esempio Apache o IIS), **è un programma sempre in esecuzione che aspetta richieste dai client e ritorna le risorse specificate nelle richieste**. Ci sono due tipi fondamentali di risorse disponibili nel Web server:

  1. **risorse statiche**: le risorse esistono prima della richiesta e/o il loro contenuto è fissato;

  2. **risorse dinamiche**: le risorse vengono create al momento della richiesta e in maniera automatica con programma.

SERVER HTTP

- In generale le risorse che un Web server gestisce sono principalmente **pagine html, file di testo, formati per lo scambio di documento di testo** (Adobe Portable Document Format, PostScript)**, immagini** (GIF, JPEG, PNG)**, animazioni e altri contenuti multimediali**. Tutti queste risorse possono essere statiche o dinamiche, anche se generalmente vengono generate pagine html

- Per la creazione delle pagine html ma non solo, sono stati predisposti dei meccanismi da **agganciare ai Web server in modo da aggiungere la generazione delle pagine dinamiche** (PHP, Perl etc, le servlet e le JSP)

INTRODUZIONE JAVA EE

- La tecnologia Java 2 Enterprise Edition (J2EE) è diventata negli anni sinonimo di sviluppo di applicazioni aziendali robuste, sicure ed efficienti
- Il successo è dovuto al linguaggio object oriented Java e a come attraverso quest'ultimo è stato creato. È un continuo lavoro delle più importanti aziende di information technology: Sun, IBM, Oracle, BEA, ecc
- La tecnologia JEE può facilitare la creazione di modelli B2B (Business to Business) e B2C (Business to Consumer) e quindi permettere all'azienda lo sviluppo di nuovi servizi.
- Attraverso il modello di sviluppo proposto, rende facile l'accesso ai dati e la sua rappresentazione in diverse forme (un browser web, un applet, un dispositivo mobile, un sistema esterno, ecc).

INTRODUZIONE JAVA EE

- Dal punto di vista tecnologico tutto ciò è realizzato con una **struttura tecnologica a livelli, dove ogni livello implementa uno specifico servizio**, a partire dal quale può essere implementato il processo aziendale, curando quindi la sua evoluzione senza preoccuparsi delle operazioni di base.
- J2EE è un raccoglitore di tecnologie che facilitano lo sviluppo di **software web based distribuito**
- Per vedere solo una parte di queste tecnologie non è sufficiente un corso di qualche mese

IL RUOLO DELL'APPLICATION SERVER

- Un Application Server è uno strato **software residente su una macchina server che fornisce una serie di servizi**, fornisce tutti i servizi che la tecnologia Java Enterprise espone
- Diciamo subito che di application server ce ne sono tantissimi. Come Bea Weblogic, IBM Websphere, Sun Application Server, Pramati, ecc ed alcuni prodotti opensource molto noti come JBoss
- In generale **la principale differenza tra i diversi application server risiede in come alcune particolari operazioni (deploy, clustering, ecc) sono realizzate**, in quanto, come abbiamo detto, la specifica Java Enterprise deve essere rispettata dando così vita ad uno sviluppo che è **server-independent** (a beneficio di chi si occupa di sviluppo)

IL RUOLO DELL'APPLICATION SERVER

- Un **application server si installa e si avvia come un web server**. Infatti un web server è uno dei tanti servizi che l'application server ha in sé. Presenta un pannello di amministrazione accessibile da postazione remota attraverso il quale possono essere definiti i diversi servizi.
- Possono essere installate le applicazioni che andremo a sviluppare, **possono essere definiti servizi accessori come le transazioni o l'accesso a sorgenti di dati, i web services o le code di messaggi**.
- **La connessione ai servizi avviene mediante i protocolli** http, rmi, iiop, in base ai tipi di servizi richiesti
- Un application server assolverà ai tipici compiti di web server, per la gestione delle interfacce di presentazione dinamiche, a quelli di gestione delle componenti che contengono la logica di business (quindi la parte applicativa) e delle componenti che realizzano l'accesso ai dati

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- **Maven**
- Spring
- IoC e DI
- Installazione e Configurazione
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

MAVEN

- È uno strumento di **gestione di progetti software basati su Java e build automation** e può essere usato anche in progetti scritti in C#, Ruby, Scala e altri linguaggi.
- Maven usa un costrutto conosciuto come **Project Object Model (POM); un file XML che descrive le dipendenze fra il progetto e le varie versioni di librerie necessarie** nonché le dipendenze fra di esse.
- In questo modo si separano le librerie dalla directory di progetto utilizzando questo file descrittivo per definirne le relazioni.

MAVEN

- Maven **effettua** automaticamente il **download di librerie Java e plug-in Maven dai vari repository definiti scaricandoli in locale** o in un repository centralizzato lato sviluppo.

- Questo permette di recuperare in modo uniforme i vari file JAR e di poter spostare il progetto indipendentemente da un ambiente all'altro avendo la sicurezza di utilizzare sempre le stesse versioni delle librerie.

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- Maven
- **Spring**
- IoC e DI
- Installazione e Configurazione
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

INTRODUZIONE

- **Spring** è un **framework open source** per lo sviluppo di applicazioni Java, in particolare per **applicazioni J2EE**.
- La prima versione del framework fu **rilasciata nel 2002** da Rod Johnson, **subendo poi molti cambiamenti** che hanno portato questo framework ad essere molto apprezzato dalla comunità internazionale ed a vincere nel 2006 un Jolt productivity award e un JAX Innovation Award.

PERCHÉ USARE SPRING

- **Modulare**: essendo ampio e modulare si può scegliere di integrare solo alcuni dei suoi moduli all'interno del nostro progetto.
- **Lightweight**: le dipendenze dal framework all'interno della nostra business logic praticamente nulle e facilmente isolabili.
- **Integrabile**: non ci sono package di logging, connection pool o O\R mapping, poiché tutte queste funzionalità sono fornite da altri framework open source.

PERCHÉ USARE SPRING

- **Portabile**: può essere trasferita senza problemi da un Application Server all'altro, come WebLogic, Tomcat, JBoss e WebSphere.
- **POJO-based programming enabled**: permette di sviluppare una completa applicazione J2EE usando solo POJO, dotandoli di funzionalità enterprise in modo non invasivo.
- **Test driven enabled**: permette di scrivere software facile da testare, ideale per i software che sono scritti basandosi sul Test Driven Development.

INTRODUZIONE SPRING

- Spring MVC è un framework per realizzare **applicazioni web basate sul modello MVC** sfruttando i punti di forza offerti dal framework Spring come l'**inversion of control** (tramite **dependency injection**) e la **aspect oriented programming**.

- Esso si occupa di mappare metodi e classi Java con determinati url, di gestire differenti tipologie di "viste" restituite al client, di realizzare applicazioni internazionalizzate e di gestire i cosiddetti temi per personalizzare al massimo l'esperienza utente.

SPRING MVC

- Spring MVC implementa questo approccio mantenendo sia i concetti che la nomenclatura del pattern:
  - i **Model** sono rappresentati dalle classi che a loro volta rappresentano gli oggetti gestiti e le classi di accesso al database;
  - le **View** sono rappresentate dai vari file JSP (che vengono compilati in HTML) e da eventuali classi per l'esportazione in formati diversi da HTML (PDF, XLS, CSV…);
  - i **Controller** sono rappresentati da classi (chiamate appositamente Controller) che rimangono "in ascolto" su un determinato URL e, grazie ai Model e alle View, si occupano di gestire la richiesta dell'utente.

SPRING MVC

- Spring MVC presenta inoltre molti altri vantaggi oltre alla netta separazione tra le funzionalità:
  - è **adattabile, flessibile** e non intrusivo grazie alla presenza di comode e chiare Java Annotations;
  - permette di scrivere codice **riusabile**;
  - possibilità di essere **esteso** tramite adattatori e validatori scritti ad hoc per le nostre esigenze;
  - **url dinamici**, SEO-friendly e personalizzabili;
  - gestione integrata **dell'internazionalizzazione** e dei **temi**;
  - libreria JSP sviluppata ad hoc per facilitare alcune operazioni ripetitive;
  - nuovi scope per i bean (request e session) che permettono di adattare i container base di Spring anche al mondo web.

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- Maven
- Spring
- **IoC e DI**
- Installazione e Configurazione
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

INVERSION OF CONTROL

- L'Inversion of Control (IoC) è una caratteristica tipica dei framework, **spostano il controllo del flusso dall'applicativo al framework stesso**.
- Se si crea una applicazione con GUI, il programmatore deve solo preoccuparsi di implementare gli handler degli eventi scatenati dall'utente, quali la pressione di un tasto o l'inserimento di un testo.
- **Al framework viene lasciato il controllo del flusso applicativo**, come ad esempio la chiusura e l'apertura delle finestre.
- In Spring tutte **le applicazioni sono formate da oggetti che per raggiungere l'obiettivo prefissato devono collaborare gli uni con gli altri**, e quindi inevitabilmente si vengono a formare delle dipendenze.

INVERSION OF CONTROL

- Se IoC viene usato per invertire il controllo sulla configurazione delle dipendenze, il termine Dependecy Injection (DI), coniato da Martin Fowler, erroneamente viene usato come sinonimo id IoC.
- Il concetto di base della **DI consiste nell'avere un oggetto separato che si occupi della risoluzione delle dipendenze** e della loro inizializzazione.
- Ci sono tre tipi di Dependecy Injection: Interface injection, Setter injection, Constructor injection
- In Spring, sono implementate solo la Setter e la Constructor Injection, ma c'è anche la possibilità di inizializzare e creare oggetti attraverso dei Factory Method e dei Factory Object

IOC CONTAINER

- L'**IoC container** è la parte di Spring che si occupa di **instanziare e configurare gli oggetti** che vengono inseriti in esso, che **prendono il nome di beans**.
- I beans vengono **configurati attraverso dei metadati** che possono essere dei file xml o delle Java annotations.
- Di default i metadati vengono letti solo dagli xml, per abilitare l'uso delle annotations c'è bisogno di configurare l'ApplicationContext.
- La parte di IoC e di DI viene implementata attraverso la BeanFactory e l'ApplicationContext. Dato che l'ApplicationContext è un superset della BeanFactory, se ne consiglia l'uso, e da qui in avanti faremo riferimento solo all'ApplicationContext.

IOC CONTAINER

- Ci sono vari tipi di ApplicationContext forniti da Spring, a seconda dell'applicazione che si deve sviluppare.
- Per esempio per applicazioni stand alone ci sono il ClassPathXmlApplicationContext e il FileSystemXmlApplicationContext, mentre per le applicazioni enterprise c'è il WebApplicationContext, che viene instanziato attraverso un serlvet listener.
- Tutti gli ApplicationContext hanno bisogno dei metadati di configurazione e quindi per instanziarne uno abbiamo bisogno prima di scrivere un file xml.

IOC CONTAINER

- Facendo riferimento al seguente esempio usiamo il FileSystemXmlApplicationContext:

```
ApplicationContext applicationContext = new FileSystemXmlApplicationContext("/path/
applicationContext.xml");
```

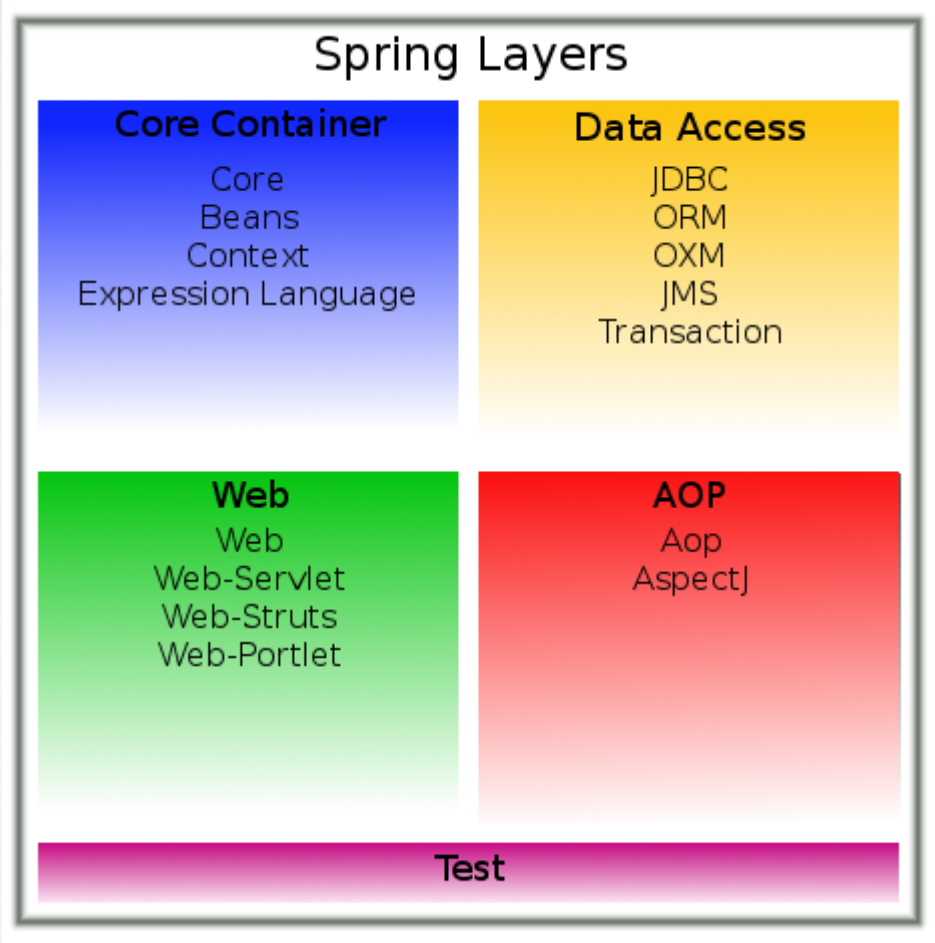
- Invece per usare il WebApplicationContext in un'applicazione enterprise, nel nostro web.xml bisogna aggiungere:

```
<!-- SPRING -->
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath*:applicationContext.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

MODULI

- Uno dei vantaggi di Spring è quello di essere modulare e i suoi layer principali sono:
  - Core Container
  - Data Access
  - Web
  - Aop
  - Test



MODULI

- Il **layer Core Container** è formato dai moduli: Core, Beans, Context, e Expression Language.
  - **I moduli Core e Beans formano la struttura principale del framework,** includendo le funzionalità di IoC e DI, che sono implementate tramite la BeanFactory, che è una implementazione sofisticata del Factory Method Pattern. La BeanFactory consente il disaccoppiamento tra definizione e configurazione delle dipendenze e la logica applicativa.
  - Il modulo Context, basato sul modulo Beans, funge da tramite per l'accesso agli oggetti attraverso la sua intefaccia principale: ApplicationContext. **L'ApplicationContext eredita tutti le funzionalità della BeanFactory**, aggiungendone altre quali: supporto per l'internazionalizzazione, propagazione di eventi, caricamento di risorse, il supporto per J2EE.
  - **Il layer Expression Language** offre un potente expression language **per la manipolazione degli oggetti** a runtime.

MODULI

- **Il layer Data Access/Integration** è formato dai sottomoduli: JDBC, ORM, OXM, JMS e Transaction.
  - **Il modulo JDBC fornisce una astrazione delle Api JDBC**, creata per rendere meno tediosa la programmazione attraverso le stesse. Il modulo ORM fornisce un livello di integrazione con i popolari framework di O\R mapping, come JPA, Hibernate, iBatis. Spring permette l'uso di tutte le potenzialità di questi framework, dando la possibilità di intergrarli con le proprie peculiarità, come ad esempio la gestione delle transazioni.
  - **Il layer OXM fornisce un livello di astrazione per l'Object/XML mapping per JAXB, XMLBeans e altri**. Il modulo JMS contiene funzionalità per l'invio e la ricezione di messaggi, mentre il modulo Transaction offre un gestore delle transazione per tutti i tipi di classi.

MODULI

- **Il layer Web** è formato dai moduli: Web, Web-Servlet, Web-Struts e Web-Portlet.
  - **Il modulo Web contiene le funzionalità di base per creare applicazioni enterprise**, come ad esempio il supporto per l'upload dei file, offrendo anche l'inizializzazione del container IoC attraverso un Servlet Listener.
  - Il **modulo Web-Servlet** contiene la parte di MVC del framework
  - il **modulo Web-Struts** contiene classi per l'integrazione con Struts2, che però è sconsigliata nella versione 3 di Spring.
  - Infine il **modulo Web-Portlet** offre funzionalità identiche al modulo Web-Servlet, ma orientato ad applicazioni che fanno uso di portlet.

MODULI

- Il **layer Aop** fornisce un'implementazione aderente alla AOP Alliance, permettendo una programmazione aspect-oriented.
- Questo layer **consente di creare codice che deve essere disaccoppiato dalle altre funzionalità, anche se legato al flusso generale dell'applicativo**. Questo layer ha anche un modulo Aspects per l'integrazione con AspectJ, ed un modulo Instrumentantions che consente il monitoraggio delle classi, offrendo anche un ClassLoader che può essere usato in alcuni Application Server.
- Il **layer Test** offre la possibilità di testare i componenti di Spring, usando JUnit o TestNG. Questo modulo fornisce la possibilità di inizializzare e cachare ApplicationContexts per scopi di test, fornendo anche la possibilità di usare oggetti di mock per testare il codice in isolation.

WEB APPLICATION CONTEXT

- Per ciascuna servlet utilizzata viene istanziato un WebApplicationContext all'interno del quale vivono tutti i bean definiti.
- Il contesto viene agganciato allo standard ServletContext in modo da essere recuperato facilmente.
- All'interno di questo speciale contesto sono presenti alcuni bean che configurano in maniera particolare Spring MVC.
- Alcuni bean sono già presenti di default ma ovviamente si possono personalizzare sempre tenendo in mente la logica del convention over configuration.

Bean	**Descrizione**
HandlerMapping	Bean che si occupa di mappare le url invocate dal client verso un particolare metodo o classe; il bean di default utilizza le java annotations
HandlerExceptionResolver	Bean che gestisce l'output al client in caso di eccezioni
ViewResolver	Bean che si occupa di identificare quale view caricare sulla base del nome in formato stringa
LocaleResolver	Bean che gestisce l'internazionalizzazione

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- Maven
- Spring
- IoC e DI
- **Installazione e Configurazione**
- Controller
- Signature

Dott. Antonio Giovanni Lezzi

INSTALLAZIONE E CONFIGURAZIONE

- Il miglior modo per apprendere questo nuovo framework è partire da un progetto nuovo in modo da concentrarsi esclusivamente sulle funzionalità di Spring MVC.

- L'installazione delle librerie avviene tramite Maven con un pacchetto per Spring MVC, infatti nel file pom.xml inseriamo la direttiva dependency:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>3.2.4.RELEASE</version>
</dependency>
```

- Maven forzerà il download di altri pacchetti da quali esso dipende (tra i quali Spring Core).

INSTALLAZIONE E CONFIGURAZIONE

- Il funzionamento di Spring MVC è semplice: tutto ruota intorno ad una servlet, la DispatcherServlet, che permette di redirigere tutte le chiamate della nostra applicazione verso il mondo di Spring MVC.

- La configurazione iniziale deve partire da questa Servlet: apriamo quindi il nostro file web.xml e inseriamo la seguente direttiva J2EE:

```
<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

- Utilizzando come url-pattern lo slash (/) abbiamo comunicato alla DispatcherServlet che tutte le chiamate web che transitano attraverso la nostra applicazione devono essere convogliate verso Spring MVC. Nulla ci vieta di inserire una direttiva meno globale per gestire diversamente altre chiamate.

INSTALLAZIONE E CONFIGURAZIONE

- Al load della servlet, il framework va alla ricerca di un file all'interno della cartella WEB-INF chiamato {nomeservlet}-servlet.xml (quindi nel nostro caso springmvc-servlet.xml) dove devono essere definiti tutti i bean e le configurazioni da utilizzare in Spring MVC, esattamente come avviene utilizzando Spring al di fuori del contesto web. Al momento creiamo un XML senza nessun bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
</beans>
```

- Ora Spring MVC è ufficialmente installato e pronto per essere utilizzato.

CONFIGURAZIONI AGGIUNTIVE

- Spring MVC si basa sul modello di **sviluppo convention over configuration** che prevede che per avviare una applicazione base siano necessarie poche configurazioni in quanto è preferibile utilizzare le convenzioni previste dagli sviluppatori.
- Questo ovviamente è valido per applicazioni standard, ma non può essere la regola.
- La configurazione basata sulle convenzioni è banale ma è ovviamente possibile personalizzarla ulteriormente per adattarla alle esigenze e preferenze di ogni sviluppatore.
- Per esempio utilizzando Maven è consigliato utilizzare la cartella resources per tutti i file di configurazione dell'applicazione al posto della cartella WEB-INF.

CONFIGURAZIONI AGGIUNTIVE

- Grazie al parametro contextConfigLocation è possibile differenziare il percorso del (o dei) file XML di configurazione del contesto di Spring MVC:

```
<servlet>
<servlet-name>springmvc</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<init-param>
<param-name>contextConfigLocation</param-name>
<param-value>classpath*:META-INF/spring/*.xml</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

- In questo esempio la cartella creata ad hoc all'interno di META-INF è dedicata alle configurazioni di Spring MVC; inoltre con il meta carattere * possiamo differenziare i bean anche in diversi file XML avendo la certezza che il framework li leggerà tutti.

WEB APPLICATION CONTEXT

- Per ciascuna servlet utilizzata viene istanziato un WebApplicationContext all'interno del quale vivono tutti i bean definiti. Questo contesto viene agganciato allo standard ServletContext in modo da essere recuperato facilmente.
- All'interno di questo speciale contesto sono presenti alcuni bean che configurano in maniera particolare Spring MVC.
- Alcuni bean sono già presenti di default ma ovviamente si possono personalizzare sempre tenendo in mente la logica del convention over configuration.
- Per sovrascrivere questi bean basta includere nel contesto un nuovo bean dello stesso tipo che punti ad una classe esterna.

WEB APPLICATION CONTEXT

- I principali bean presenti nel WebApplicationContext sono:

Bean	Descrizione
HandlerMapping	Bean che si occupa di mappare le url invocate dal client verso un particolare metodo o classe; il bean di default utilizza le java annotations
HandlerExceptionResolver	Bean che gestisce l'output al client in caso di eccezioni
ViewResolver	Bean che si occupa di identificare quale view caricare sulla base del nome in formato stringa
LocaleResolver	Bean che gestisce l'internazionalizzazione

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- Maven
- Spring
- IoC e DI
- Installazione e Configurazione
- **Controller**
- Signature

Dott. Antonio Giovanni Lezzi

METODI DA MAPPARE E ANNOTATIONS

- I controller sono il componente che viene invocato direttamente dai client che si occupano delle principali logiche di business e che possono esistere anche senza la presenza di Model e di View.
- Il tipo di bean che si occupa del mapping tra url invocato dall'utente e metodo Java da invocare è l'handlerMapping.
- L'implementazione di default presente in Spring MVC è la classe RequestMappingHandlerMapping che permette di sfruttare le java annotations per identificare i metodi da mappare.

METODI DA MAPPARE E ANNOTATIONS

Annotation	Descrizione
@controller	Da utilizzare a livello di classe per identificarla come controller
@requestMapping	Da utilizzare per evidenziare il metodo e l'url da mappare

CONTROLLER

```
@Controller
public class DemoController {
    @RequestMapping(value="/home")
    public String getIndex() {
        return "index";
    }
}
```

- Si è creato un controller con un metodo getIndex che risponde all'url /home, ma all'interno del WebApplicationContext dobbiamo definire lo scanner che si occupa di trovare le classi annotate con le annotation di Spring e Spring MVC.

CONTROLLER

- Una volta configurato possiamo avviare l'applicazione (possibilmente nella ROOT dell'application server) e puntare i nostri browser verso "http://localhost:8080/home".
- Se tutto è stato configurato alla perfezione vedremo nello standard error un errore che indica che non esiste nessun handler per la pagina /index.
- Dato che non è stato definito nessun ViewResolver, Spring MVC non è ancora in grado di capire cosa fare della nostra stringa "index" e quindi cercherà di eseguire un forward non trovando però nessun handler, fornendo l'errore:
  WARNING: No mapping found for HTTP request with URI [/index] in DispatcherServlet with name'springmvc'

@REQUESTMAPPING

- Una delle cose più interessanti dell'annotation @RequestParam è però la possibilità di creare URL dinamiche con placeholder da utilizzare al posto dei classici parametri GET che non piacciono molto agli spider SEO.

```
@RequestMapping(value="/book/{id}", method=RequestMethod.GET)
public void getBook(@PathVariable int id) {
    System.out.println("Requested book " + id);
}
```

- Si è creato un handler per l'URL /book/{id} e con l'annotation @PathVariable, si è definito la variabile id di tipo int.
- Ad esempio invocando l'url "/book/10" verrà invocato il metodo avendo come parametro "id = 10".
- Invocando /book oppure /book/aaa otterremo un errore.

@REQUESTMAPPING

- Se il nome del placeholder e del parametro differiscono è possibile utilizzare:

```
@RequestMapping(value="/book/{idBook}", method=RequestMethod.GET)
public void getBook(@PathVariable("idBook") int id) {
    // [...]
}
```

- Per recuperare i valori dei parametri inviati alla pagina possiamo utilizzare @RequestParam

```
@RequestMapping(value="/book", method=RequestMethod.GET)
public void getBookByParam(@RequestParam("bookId") int id) {
    System.out.println("Requested book " + id);
}
```

- In questo caso anche invocando l'URL /book?bookId=10 otterremo lo stesso effetto.

JAVA WEB SPRING

- Framework
- MVC
- Server Application
- Maven
- Spring
- IoC e DI
- Installazione e Configurazione
- Controller
- **Signature**

Dott. Antonio Giovanni Lezzi

SIGNATURE DEGLI HANDLER

- C'è la possibilità di personalizzare la signature del metodo (ovvero il numero e il tipo di parametri accettati e il tipo di dato ritornato) e permette di avere una libertà all'interno di una applicazione J2EE.
- In precedenza si sono creati metodi che non accettavano parametri, metodi che accettavano una String e metodi che accettavano un int.
- Il motore interno di Spring MVC identifica quali parametri il nostro metodo richiede e, se possibile, ce li fornisce senza preoccupazioni da parte nostra.
- Lo stesso approccio è valido per il valore ritornato dal metodo stesso.

PARAMETRI POSSIBILI

- Parametri annotati con ***@PathVariable*** recuperati dal pattern.
- Parametri annotati con ***@RequestParam*** ed inviati via HTTP dal client.
- ***HttpServletRequest*** e ***HttpServletResponse*** (e anche ***ServletRequest*** e ***ServletResponse***) per accedere direttamente agli oggetti nativi J2EE.
- ***HttpSession*** per sfruttare la sessione senza doverla recuperare manualmente dalla request.
- ***Locale*** per identificare automaticamente il locale utilizzato.

PARAMETRI POSSIBILI

- ***InputStream*** e ***OutputStream*** per lavorare a basso livello direttamente sugli stream in ingresso e in uscita.
- Parametri annotati con ***@RequestHeader*** per avere un determinato valore tra gli header HTTP ricevuti.
- Parametri annotati con ***@RequestBody*** per avere direttamente il corpo della request.
- ***Map*** (o ***Model***) per passare alla view valori dinamici.
- ***Errors*** (o ***BindingResult***) per gestire la validazione dei form.

SIGNATURE

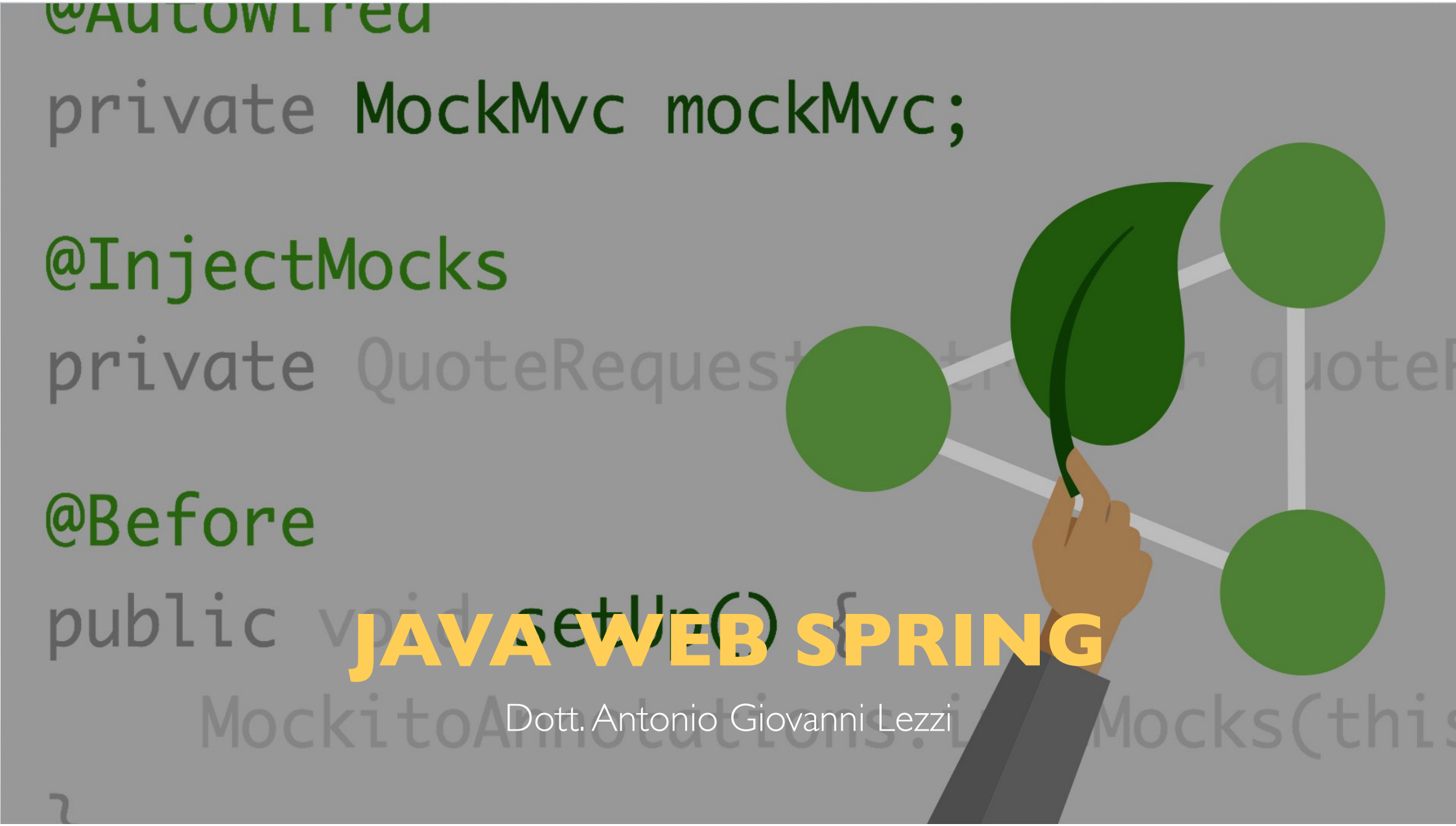
- Tutto questo significa che avere delle signature di questo tipo non solo è possibile, ma è anche raccomandabile:

```
public void withSession(HttpSession session) {
    // TODO...
}
public void withResponseRequest(HttpServletRequest request, HttpServletResponse response) {
    // TODO...
}
public void withLocale(Locale locale) {
    // TODO...
}
```

- Non bisognerà preoccuparsi di come recuperare questi oggetti, basterà inserirli nella signature del metodo e diverranno disponibili.

VALORI DI RITORNO

- È possibile modificare a proprio piacimento il valore ritornato dai nostri handler, seppur con minore variabilità.
  - ***void*** per comunicare a Spring MVC che l'intera gestione del flusso di response verrà gestita manualmente;
  - una stringa per invocare un ***ViewResolver*** che a partire da essa recupererà una particolare vista (JSP per esempio) da utilizzare;
  - una stringa con prefisso ***redirect***: per forzare una redirect verso un'altra url;
  - una stringa con prefisso ***forward***: per forzare il forward verso un altro handler;
  - un ***ModelAndView*** (o un Model) che incapsulano sia il nome della vista da caricare sia l'insieme dei dati da passare ad essa;
  - qualsiasi tipo di oggetto da convertire tramite un ***HttpMessageConverter*** (per esempio in JSON o XML) se il metodo è annotato con ***@ResponseBody***



JAVA WEB SPRING

Dott. Antonio Giovanni Lezzi

JAVA WEB SPRING

- **View, ViewResolver**
- Gestione Form
- Handler
- Eccezioni
- DAO e JDBC
- Hibernate

Dott. Antonio Giovanni Lezzi

VIEW E VIEWRESOLVER

- I controller delegano alle viste il compito di mostrare all'utente i contenuti in maniera presentabile e stilisticamente appropriata.
- Quando un controller ritorna una stringa (che non inizi con redirect o forward) o un oggetto ModelAndView, significa che esso vuole delegare al motore interno di Spring MVC la creazione di una vista identificata dal valore della stringa ritornata (o incapsulata nell'oggetto) sulla base dei parametri inseriti nel Model.
- In sintesi una View identifica una particolare risorsa in grado di mostrare all'utente dei dati (per esempio una classica JSP) mentre un ViewResolver permette di identificare una determinata View sulla base di una stringa.
- Spring MVC non presenta nessun ViewResolver di default, infatti ottenevamo degli errori ritornando delle stringhe nei controller implementati.

INTERNALRESOURCEVIEWRESOLVER

- Il caso di ViewResolver più classico è InternalResourceViewResolver che permette di identificare una View sulla base di risorse interne al progetto come ad esempio JSP e basta creare un bean:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="/WEB-INF/view/"/>
<property name="suffix" value=".jsp"/>
</bean>
```

- Si è definito il ViewResolver che sarà in grado di recuperare la JSP concatenando le stringhe /WEB-INF/view e .jsp al valore della stringa ritornata.

- Il parametro viewClass rappresenta la classe da cui ciascuna vista verrà istanziata. Se per esempio il nostro controller dovesse tornare home il ViewResolver cercherebbe la JSP in /WEB-INF/view/home.jsp.

- Inoltre se nel metodo handler viene istanziato o modificato un oggetto Model (o ModelAndView) esso viene passato alla JSP.

INTERNALRESOURCEVIEWRESOLVER

- Analizziamo un esempio:

```
@RequestMapping(value="/pagina-personale", method=RequestMethod.GET)
    public String paginaPersonale(Model model) {
    model.addAttribute("nome", "Antonio");
    model.addAttribute("cognome", "Lezzi");
    return "pagina-personale";
}
```

- In questo caso i parametri aggiunti all'oggetto Model vengono resi disponibili nella JSP.

```
<h1>Ciao <c:out value="${nome }" /> <c:out value="${cognome}" /></h1>
```

ALTRI VIEWRESOLVER

Resolver	Descrizione
XmlViewResolver	Identifica la view corretta a partire da un file XML di configurazione.
ResourceBoundleViewResolver	Identifica la view corretta a partire da chiavi contenuto nel resource boundle (utile per offrire viste localizzate).
UrlBasedViewResolver	Identifica la view corretta trasformando il nome logico della vista in un URL (l'InternalResourceViewResolver è una sottoclasse di UrlBasedViewResolver).
ContentNavigationViewResolver	Identifica la view sulla base del filename richiesto o in base al parametro Accept della request.
BeanNameViewResolver	Identifica la view sulla base del nome del bean all'interno del contesto di Spring MVC.

CONTENTNAVIGATIONVIEWRESOLVER

- Uno dei risolver più interessanti è ContentNavigationViewResolver che rappresenta una sorta di collezione di sotto-resolver ed è in grado di differenziarne l'invocazione sulla base di due parametri:
  - contenuto dell'header Accept della request
  - estensione del file richiesto
- Dato che difficilmente via browser è possibile gestire il valore di Accept, prenderemo in considerazione la seconda opzione.

SIGNATURE

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="atom" value="application/atom+xml"/>
            <entry key="html" value="text/html"/>
            <entry key="json" value="application/json"/>
        </map>
    </property>
    <property name="viewResolvers">
        <list>
            <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
            <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
                <property name="prefix" value="/WEB-INF/jsp/"/>
                <property name="suffix" value=".jsp"/>
            </bean>
        </list>
    </property>
    <property name="defaultViews">
        <list>
            <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
        </list>
    </property>
</bean>
```

SIGNATURE

- Grazie a questo esempio è possibile registrare dei Content-Type a partire dall'estensione del file (in questo caso atom, html, json) e dei ViewResolver.
- Per ciascuna richiesta il ContentNegotiatingViewResolver identifica il Content-Type e il ViewResolver in grado di gestire la chiamata.
- Nel caso in cui ne trovi uno, delegherà ad esso la gestione, altrimenti si affiderà alla View di default.
- **Nota**: per poter utilizzare questo esempio è necessario importare nel progetto la dipendenza Jackson.

VIEWRESOLVER E VIEW PERSONALIZZATE

- Un'applicazione può presentare diverse tipologie di contenuti, quindi diverse viste e quindi diversi ViewResolver.
- Tramite una configurazione mirata è possibile gestire ViewResolver multipli, organizzandoli in una sorta di serie ordinata dalla quale Spring MVC cercherà il resolver più adatto.
- Data la vastissima estendibilità di Spring, è inoltre possibile definire delle View personalizzate definendo puntualmente quale deve essere il loro compito.
- Nel nostro esempio creeremo una vista che si appoggierà a OpenCSV per generare appunto dei CSV (Dopo aver incluso le librerie necessarie nel pom.xml di Maven, creiamo la classe CSVView)

VIEWRESOLVER E VIEW PERSONALIZZATE

```
@Component
public class CSVView extends AbstractView {
    @Override
    protected void renderMergedOutputModel(Map model, HttpServletRequest request, HttpServletResponse
response) throws Exception {
        CSVWriter writer = new CSVWriter(response.getWriter());
        String[] names = (String[]) model.get("names");
        writer.writeNext(names);
        writer.close();
    }
}
```

- È necessario ricordare che la classe deve essere decorata con l'annotation @Component in modo che Spring la consideri parte del proprio contesto e che deve estendere AbstractView e quindi implementare il metodo void renderMergedOutputModel.

- In questo metodo non facciamo altro che recuperare i dati dal model (che sarà popolato nel controller) e scrivere, grazie alla classe CSVWriter di OpenCSV, il contenuto della response.

VIEWRESOLVER E VIEW PERSONALIZZATE

- Una volta definita la nostra CSVView è necessario includere nel WebApplicationContext un ViewResolver in grado di gestire le nuove classi che verranno istanziate.

- In questo caso utilizziamo un BeanNameViewResolver che assocerà la CSVView al nome logico CSVView (ricordiamo che i bean identificati dall'annotation @Component di default acquisiscono come id il nome stesso della classe).

```
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver">
<property name="order" value="1" />
</bean>
```

- Dato che ora nel WebApplicationContext saranno presenti due diversi ViewResolver, è necessario aggiungere in almeno uno di essi la proprietà order per identificarne la priorità.

VIEWRESOLVER E VIEW PERSONALIZZATE

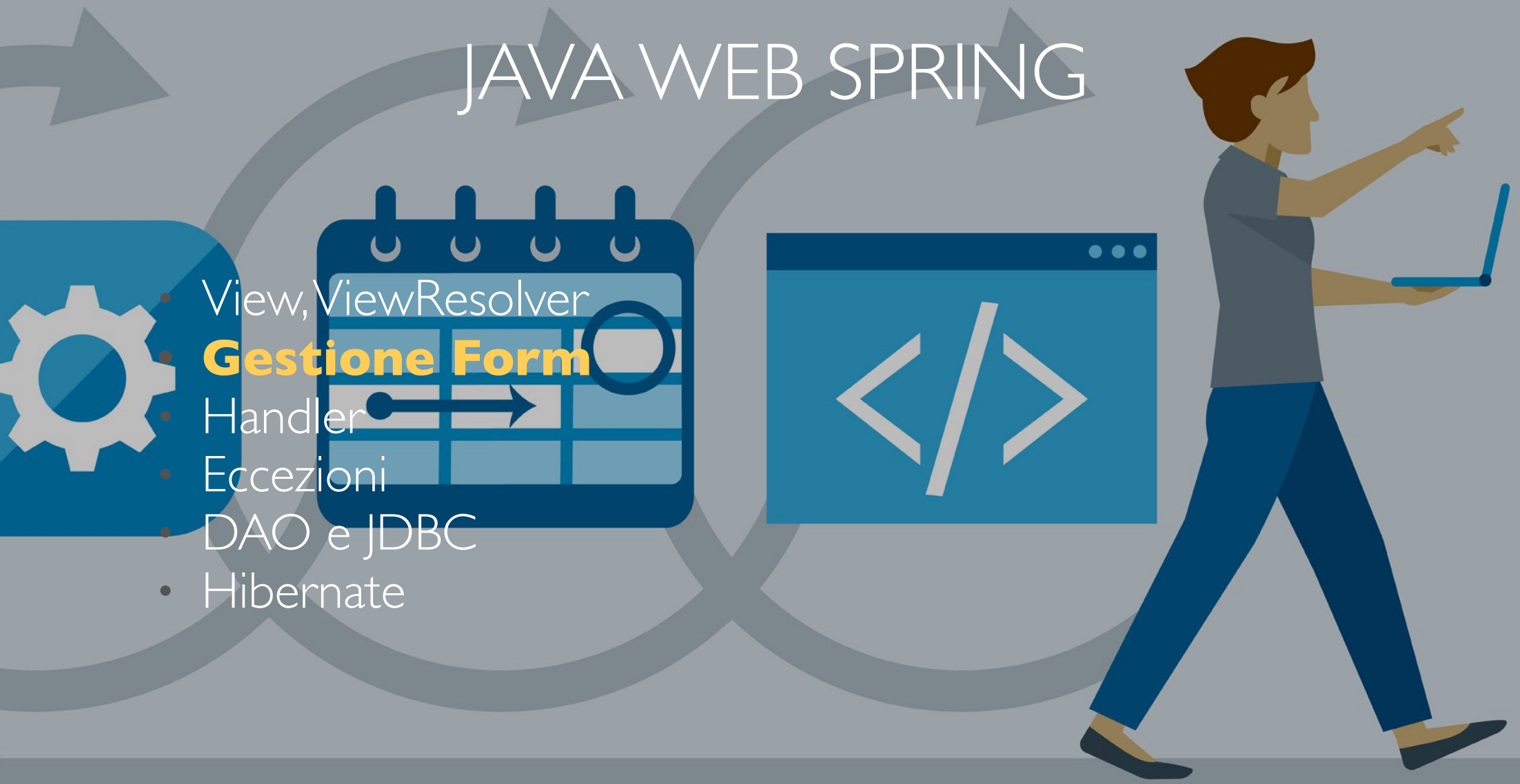
- Impostando "1" come order del nuovo BeanNameViewResolver significherà che Spring MVC cercherà innanzitutto di utilizzare quest'ultimo come resolver e in caso di insuccesso passerà al precedente UrlBasedViewResolver.

- Ora basterà creare un controller che ritornerà CSVView come nome logico della vista:

```
@RequestMapping(value="/csv", method=RequestMethod.GET)
public String getCSV(Model model) {
    String[] names = new String[] { "Alberto", "Luca", "Claudio" };
    model.addAttribute("names", names);
    return "CSVView";
}
```

JAVA WEB SPRING

- View, ViewResolver
- **Gestione Form**
- Handler
- Eccezioni
- DAO e JDBC
- Hibernate



Dott. Antonio Giovanni Lezzi

GESTIONE DEI FORM

- Il concetto di Model è un po' più ampio e privo di confini precisi, infatti si intende qualsiasi oggetto che rappresenta la realtà descritta dall'applicazione Web che funge da connettore tra gli altri componenti.
- L'oggetto Model serve a passare informazioni reali dal Controller alla Vista ma hanno una seconda tipologia di Model: gli oggetti Form.
- Se la dicitura sembra impropria, tale tipologia di oggetti svolge la medesima funzione ma in senso opposto, ovvero permette di raccogliere dati dalle View e di renderli disponibili nei Controller.

PRIMO FORM

- Il miglior modo per realizzare un form Web è quello di partire dalla tipologia di contenuto che esso rappresenta e creare una classe ad hoc per i suoi attributi.

- Consideriamo un form di anagrafica per registrare persone:

```
public class PersonForm {
    private String name;
    private String surname;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSurname() {
        return surname;
    }
    public void setSurname(String surname) {
        this.surname = surname;
    }
}
```

PRIMO FORM

- Definita la classe possiamo occuparci del Controller che, grazie a Spring MVC non dovrà preoccuparsi di Request e parametri vari, ma potrà gestire direttamente un model istanziato dalla classe PersonForm.
- Per gestire un form web sono necessari due step: visualizzazione del form e raccolta dati
- Il miglior modo è quello di mantenere una URL unica per le azioni, differenziandone i metodi HTTP e ovviamente i metodi Java.

PRIMO FORM

```
@RequestMapping(value="/person", method=RequestMethod.GET)
public String savePerson(@ModelAttribute PersonForm personForm, Model model) {
    model.addAttribute("personForm", personForm);
    return "person";
}
@RequestMapping(value="/person", method=RequestMethod.POST)
public String savePersonPost(@ModelAttribute PersonForm personForm) {
    System.out.println(personForm.getName() + " " + personForm.getSurname());
    return "redirect:/person";
}
```

- Entrambi i metodi usano l'annotation @ModelAttribute che associa, se presenti, i parametri ricevuti da un oggetto.
- Nel caso del metodo savePerson esso sarà istanziato senza nessuna proprietà mentre, nel caso del metodo savePersonPost, i parametri verranno iniettati dentro l'oggetto grazie ad un'operazione di binding.

PRIMO FORM

- Il termine binding può essere tradotto come legame o attacco e rappresenta l'operazione, svolta da Spring MVC, di conversione da parametri HTTP (che principalmente sono stringhe) in oggetti composti e definiti dall'utente.

- Per la maggior parte dei casi le operazioni di binding già presenti in Spring MVC sono più che sufficienti, ma in caso contrario sarà possibile definirne dei propri.

PRIMO FORM

- L'ultimo step è la scrittura della pagina person.jsp con un insieme di tag adatto al nostro scopo:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form"%>
<c:url var="url" value="/person" />
<form:form action="${url}" method="post" modelAttribute="personForm">
<label>Nome:</label>
<form:input path="name" />
<label>Cognome:</label>
<form:input path="surname" />
<input type="submit" />
</form:form>
```

PRIMO FORM

- I tag di Spring utilizzano i valori inseriti negli attributi modelAttribute (di form:form) e di path (di form:input) per fare il binding dei valori inseriti dall'utente.

- Ovviamente lo stesso approccio può essere utilizzato anche in caso di form prepopolati, utili in caso di modifica di un'entità:

```
@RequestMapping(value="/updatePerson", method=RequestMethod.GET)
public String updatePerson(@ModelAttribute PersonForm personForm, Model
model) {
    personForm.setName("Nome di default");
    personForm.setSurname("Cognome di default");
    model.addAttribute("personForm", personForm);
    return "person";
}
```

VALIDAZIONE FORM

- Spring MVC utilizza l'API di javax.validation (jsr-303 bean validation) per gestire la validazione degli oggetti form, dopo aver inserito le dipendenze dell'API e di una sua implementazione nel pom.xml, possiamo annotare la nostra classe form in questo modo:

```
public class PersonForm {
    @NotNull
    @Size(min=1)
    private String name;
    @NotNull
    @Size(min=3)
    private String surname;
    // […]
 }
```

- Con queste annotation abbiamo comunicato al framework che entrambi i campi sono obbligatori e che il cognome deve essere lungo almeno 3 caratteri.

- Le specifiche JSR-303 non presentano una valida annotation per gestire le stringhe di lunghezza zero, l'unico meccanismo valido per il controllo è quello di inserire la doppia annotation @NotNull e @Size(min=1).

SIGNATURE

- Il controller che riceverà l'oggetto form:

```
@RequestMapping(value="/person", method=RequestMethod.POST)
public String savePersonPost(@Valid @ModelAttribute PersonForm personForm, BindingResult result,
Model model) {
    if(result.hasErrors()) {
        model.addAttribute("personForm", personForm);
        return "person";
    }
    System.out.println(personForm.getName() + " " + personForm.getSurname());
    return "redirect:/person";
}
```

- L'annotation @Valid all'oggetto form serve per forzarne la validazione automatica e un nuovo parametro di tipo BindingResult alla firma del metodo.

- Quest'ultimo oggetto, disponibile solamente per i form, contiene eventuali incongruenze di binding tra il form e l'oggetto Java e, tramite il metodo .hasErrors(), informazioni su errori di validazione.

SIGNATURE

- In caso di errori restituiamo alla vista person, impostando nel model l'oggetto ricevuto per offrire all'utente i campi già pre-popolati con i valori precedenti.

- Nel JSP possiamo mostrare l'errore utilizzando il tag <form:errors>:

```
<form:errors path="*" />
<label>Nome:</label>
<form:input path="name" />
<form:errors path="name" />
```

- Grazie all'attributo path del tag possiamo scegliere per quale campo visualizzare l'errore; con "*" possiamo dire di mostrarci tutti gli errori.

JAVA WEB SPRING

- View, ViewResolver
- Gestione Form
- **Handler**
- Eccezioni
- DAO e JDBC
- Hibernate

Dott. Antonio Giovanni Lezzi

HANDLER MAPPING

- Gli Handler Mapping sono una serie di classi che hanno come scopo quello di associare un determinato URL ad un Controller in grado di esaudire la richiesta.
- Gli Handler Mapping definiti in una applicazione possono essere molteplici e possono rappresentare classi già disponibili in Spring MVC o possono essere componenti completamente personalizzati.
- Il RequestMappingHandlerMapping è l'Handler Mapping utilizzato di default da Spring MVC in completa sintonia con il paradigma CoC (Convention over Configuration).
- Questo Handler sfrutta, come specificato nel proprio nome, le annotation @RequestMapping per creare un'associazione tra URL e metodo Java da invocare.

SIGNATURE

- La dichiarazione XML che permette di utilizzare questo comportamento di default sarà:

```
<mvc:annotation-driven/>
```

- Non esiste però solamente questo Handler…
  - Il ControllerClassNameHandlerMapping permette di associare un Controller ad un URL sulla base del nome. Il suo comportamento di default è quello di utilizzare il nome della classe in lower case e senza il suffisso "Controller" come URL. Per esempio il controller ProvaController sarà accessibile da /prova e UserListController da /userList.
  - Il BeanNameUrlHandlerMapping utilizza il nome del bean Spring definito nel file XML (che ovviamente deve essere un Controller) come URL di destinazione della chiamata.
  - Infine il SimpleUrlHandlerMapping permette di configurare una mappatura manuale all'interno del file XML di configurazione sfruttando anche i meta-caratteri come l'asterisco.

SIGNATURE

- Come per tutti i componenti descritti fino a questo momento, anche gli Handler Mapping possono essere personalizzati completamente sulla base delle specifiche dell'applicazione.
- Potremmo per esempio aver bisogno di identificare il controller da invocare effettuando una query su database.
- Spring MVC ci viene in aiuto grazie all'interfaccia HandlerMapping che espone tre metodi da implementare nelle nostre classi concrete.
- Una volta definito il nostro HandlerMapping, non resta che annotarlo o inserirlo nella configurazione XML.

INTERCEPTOR

- Gli interceptor arricchiscono un componente molto caro agli sviluppatori J2EE tradizionali: i filtri.
- Gli interceptor servono ad eseguire operazioni indipendenti dai controller in maniera completamente disaccoppiata evitando di ripetere codice nel momento in cui una funzionalità deve essere resa disponibile a più URL.
- Ciascun interceptor può rimanere in ascolto su tre diversi eventi

INTERCEPTOR

Evento	Descrizione
preHandle	prima dell'esecuzione del controller identificato dal Handler Mapping corrente (può anche eventualmente interrompere il flusso originale).
postHandle	dopo l'esecuzione del controller ma prima che la vista corrente venga renderizzata.
afterCompletion	al termine di tutto il flusso Request/Response, spesso utile per rilasciare alcune risorse altrimenti occupate.

- Per implementare il proprio Interceptor, Spring espone sia un'interfaccia (HandlerInterceptor) sia una classe di utilità che implementa, in maniera trasparente i 3 metodi facilitandoci il lavoro nel caso volessimo riscriverne solamente uno (HandlerInterceptorAdapter).

ESEMPIO INTERCEPTOR

```
public class LogHandlerInterceptor implements HandlerInterceptor {
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
Exception exception) throws Exception {
        System.out.println("=== AFTER COMPLETION ===");
        System.out.println("Response ContentType: " + response.getContentType());
    }
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) throws Exception {
        System.out.println("=== POST HANDLE ===");
        System.out.println("View name: " + modelAndView.getViewName());
        System.out.println("Model: " + modelAndView.getModel());
    }
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws
Exception {
        System.out.println("=== PRE HANDLE ===");
        System.out.println("Request URI: " + request.getRequestURI());
        System.out.println("Request Method: " + request.getMethod());
        System.out.println("Request Query String: " + request.getQueryString());
        return true;
    }
}
```

SIGNATURE

- N caso del preHandler stampiamo alcuni parametri contenuti nella Request, nella postHandle il nome della vista che verrà ritornata all'utente e il modello di dati richiesto e infine nella afterCompletion alcuni parametri sulla Response.

- Una volta definita la classe, iniettarla nel modello di Spring è facilissimo:

```
<mvc:interceptors>
    <bean class="com.antoniolezzi.springmvc.utils.LogHandlerInterceptor" />
</mvc:interceptors>
```

- Ovviamente all'interno del tag interceptors possiamo inserire tutta una serie di HandlerInterceptor.

- Non prendiamo invece sottogamba l'ordine di inserimento dato che un interceptor può eventualmente bloccare la catena di esecuzione.

JAVA WEB SPRING

- View, ViewResolver
- Gestione Form
- Handler
- **Eccezioni**
- DAO e JDBC
- Hibernate

Dott. Antonio Giovanni Lezzi

ECCEZIONI

- Una applicazione Web offre molti "punti di contatto" con gli utenti; per questo motivo è necessario dotarsi di una buona architettura per evitare di incappare in problemi che possono essere dannosi non solo per l'utente ma anche per la stabilità della applicazione.
- SpringMVC offre una serie di possibilità per gestire al meglio le eccezioni che possono essere lanciate durante la vita di un programma. Nel framework esistono due diverse macro strategie per la gestione delle eccezioni:
  - utilizzare un bean di tipo ExceptionResolver configurato genericamente nel contesto dell'applicazione;
  - utilizzare l'annotation @ExceptionHandler per gestire in maniera più puntuale l'errore.

ECCEZIONI

- Non esiste una strategia migliore dell'altra: quelle elencate sono due facce della stessa medaglia che devono essere scelte in base alla tipologia di applicazione che stiamo sviluppando.

- Per essere più precisi, bisogna ricordare che l'utilizzo l'annotation @ExceptionHandler deriva comunque da un ExceptionResolver che, a meno di modifiche, è però attivo di default, offrendo di fatto due diverse scelte architetturali.

EXCEPTIONRESOLVER

- Gli ExceptionResolver non sono nient'altro che delle classi che, configurate opportunamente all'interno del contesto di Spring, si occupano di intercettare e gestire un'eccezione evitando che essa venga mostrata direttamente all'utente.

- Il SimpleMappingExceptionResolver permette di creare una mappatura tra tipologia di eccezione e nome della vista da restituire all'utente. La mappatura può essere fatta direttamente tramite configurazione XML:

```
<bean class="org.springframework.Web.servlet.handler.SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="com.antoniolezzi.Eccezione1">nomevista1</prop>
            <prop key="com.antoniolezzi.Eccezione2">nomevista2</prop>
        </props>
    </property>
    <property name="defaultErrorView" value="defaultvista" />
</bean>
```

EXCEPTIONRESOLVER

- All'interno della proprietà exceptionMappings è possibile definire una mappa chiave/valore che presenta come chiave il nome dell'eccezione e come valore il nome della vista (che verrà risolta con il viewResolver definito).
- È possibile definire una vista di errore di default grazie alla proprietà defaultErrorView.
- Tutti gli ExceptionResolver implementano l'interfaccia HandlerExceptionResolver, permette di definire senza particolari criticità un proprio resolver per offrire all'utente un'esperienza diversa e più completa.

@EXCEPTIONHANDLER

- La seconda possibilità per gestire le eccezioni avviene tramite alcune annotations e in particolare @ExceptionHandler e @ResponseStatus.
- Si basa su alcuni ExceptionResolver che sono abilitati di default, coerentemente con il paradigma Convention Over Configuration.
- L'annotation @ExceptionHandler permette di definire un metodo all'interno dei nostri controller che verrà invocato automaticamente quando verrà lanciata, all'interno del controller specifico, una particolare eccezione (identificata come parametro dell'annotation).

```
@Controller
public class TestExceptionController {
    @ExceptionHandler(MyException.class)
    public String gestoreEccezioni(MyException ex) { return "errorView"; }
    @RequestMapping(value = "/")
    public String handler() {
        throws new MyException("An error occured!"); return null;
    }
}
```

@EXCEPTIONHANDLER

- Avremo un banalissimo controller dotato di un unico entry point (all'url "/"), invocando la URL il controller lancerà una eccezione che però verrà gestita, grazie ad una apposita vista, dal metodo annotato con @ExceptionHandler; attenzione però che le uniche eccezioni gestite saranno quelle istanziate dalla classe MyException all'interno di questo controller.

- L'annotation @ExceptionHandler può essere utilizzata in combinazione con @ResponseStatus per forzare un particolare codice di ritorno (ed un eventuale messaggio di errore):

```
@ExceptionHandler(MyException.class)
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Messaggio di errore
personalizzato!")
public String gestoreEccezioni(MyException ex) {
    return "errorView";
}
```

@EXCEPTIONHANDLER

- **Ultimo suggerimento**: nel caso volessimo usare questa seconda strategia basata su annotation in maniera più globale rispetto alla totalità dell'applicazione abbiamo due possibilità:
  - usare un controller astratto esteso da tutti gli altri che contiene gli @ExceptionHandler;
  - creare un controller vuoto annotato con @ControllerAdvice che permette di creare configurazioni comuni a tutti i futuri controller.

L'IMPORTANZA GESTIONE ECCEZIONI

- La gestione delle eccezioni è un aspetto importante che non deve essere preso sottogamba.
- Una buona organizzazione delle classi e una buona progettazione delle eccezioni, magari utilizzandone di personalizzate, possono facilitare notevolmente lo sviluppo rendendo inoltre più completa la navigazione da parte dagli utenti.
- Non incappare in errori di questo tipo non è per nulla difficile, basterà anche un solo parametro errato in una query string o un link ad un'area riservata.
- Molto del "lavoro sporco" viene svolto senza dubbio da Spring MVC, ma alcune responsabilità rimangono, giustamente, in mano agli sviluppatori.

JAVA WEB SPRING

- View, ViewResolver
- Gestione Form
- Handler
- Eccezioni
- **DAO e JDBC**
- Hibernate

Dott. Antonio Giovanni Lezzi

PERSISTENZA

- Spring mette a disposizione tutte queste tecnologie in un ambiente altamente flessibile, sfruttando appieno le potenzialità offerte dalla Dependency Injection e dall'Aspect Oriented.
  - JDBC
  - Java Persistence API (JPA)
  - Java Data Objects (JDO)
  - Hibernate
  - …

MODELLO PER DB

- Dato il modello da memorizzare nel database è la classe Book contenente le proprietà necessarie per rappresentare un libro e dai relativi metodi accessori.

```
public class Book {
    String isbn;
    String author;
    String title;
    public String getIsbn() { return isbn; }
    public void   setIsbn(String isbn) { this.isbn = isbn; }
    public String getAuthor() { return author; }
    public void   setAuthor(String author) { this.author = author; }
    public String getTitle() { return title; }
    public void   setTitle(String title) { this.title = title; }
  }
```

DATABASE

- La persistenza del modello sarà effettuata nella tabella books del database library.

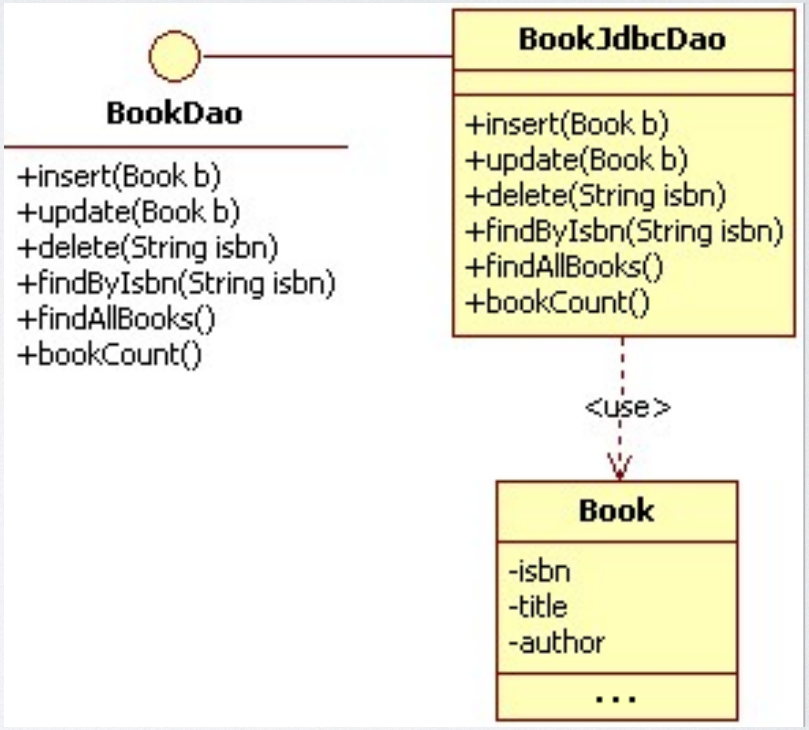
```
CONNECT 'jdbc://localhost/library;create=true';

CREATE TABLE BOOKS ( ISBN VARCHAR(13) NOT NULL,
                     AUTHOR VARCHAR(20),
                     TITLE VARCHAR(20),
                     PRIMARY KEY (ISBN));
```

DATA ACCESS OBJECT

- Spring incoraggia l'utilizzo del DAO (Data Access Object), un pattern architetturale che ha come scopo quello di separare le logiche di business da quelle di accesso ai dati.

- L'idea alla base di questo pattern è quello di descrivere le operazioni necessarie per la persistenza del modello in un'interfaccia e di implementare la logica specifica di accesso ai dati in apposite classi.



DATA ACCESS OBJECT

- La logica di business necessaria per la persistenza del nostro modello è descritta nella classe BookDao, mentre nella classe BookJdbcDao troviamo un'implementazione specifica di questa interfaccia rivolta a gestire le logiche di accesso ai dati mediante tecnologia JDBC.

- Grazie all'utilizzo del pattern DAO sia possibile fornire ulteriori implementazioni della classe BookDao per introdurre nuove logiche di accesso ai dati (ad esempio per l'utilizzo di Hibernate un ipotetico BookHibernateDao).

- L'interfaccia DAO che utilizeremo per i nostri scopi sarà così fatta:

```
public interface BookDao {
    public void insert(Book book);
    public void update(Book book);
    public void delete(String isbn);
    public Book findByISBN(String isbn);
    public List<Book> findAllBooks();
    public int bookCount();
  }
```

SPRING JDBC

- JDBC offre notevoli vantaggi in termini di performances e facilità di utilizzo, richiedendo per contro allo sviluppatore di farsi carico dell'intera gestione dei processi di accesso ai dati, ovvero:
  - Richiesta della connessione al datasource
  - Creazione del PreparedStatement e valorizzazione dei parametri
  - Esecuzione del PreparedStatement
  - Estrazione dei risultati dal ResultSet (in caso di interrogazioni)
  - Gestione delle eccezioni
  - Chiusura della connessione
- Se per applicazioni di piccole dimensioni queste "complicazioni" possono essere accettabili, sicuramente provocano dei grossi disagi in ambito enterprise.

JDBC TEMPLATE

- Spring offre diverse possibilità per la persistenza dei dati mediante JDBC, la principale delle quali è l'utilizzo della classe JDBCTemplate.
- Questa classe implementa l'intero processo di accesso ai dati attraverso template methods, rendendo possibile la personalizzazione di tutte le fasi di tale processo mediante l'ovverride dei metodi specifici.
- L'implementazione specifica per JDBC del BookDao possiede un riferimento alla classe JDBCTemplate, inizializzata attraverso il metodo setDataSource() che provvede a fornire il DataSource necessario.

```
public class BookJdbcDao implements BookDao {
    private JdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
          this.jdbcTemplate = jdbcTemplate;
    }
    // ...
  }
```

AGGIORNAMENTO

- Attraverso il metodo update è possibile eseguire tutte quelle operazioni che comportano una modifica dei dati nel database.

```
public class BookJdbcDao implements BookDao {
    // ...
    //Inserimento
    public void insert(Book book) {
      jdbcTemplate.update("insert into books (isbn, autore, titolo) values (?, ?, ?)",
                          new Object[] { book.getIsbn(), book.getAutore(), book.getTitolo() });
    }
    //Modifica
    public void update(Book book) {
      jdbcTemplate.update("update books set autore = ?, titolo = ? where isbn = ?",
                          new Object[] { book.getIsbn(), book.getAutore(), book.getTitolo() });
    }
    // Eliminazione
    public void delete(String isbn) {
      jdbcTemplate.update("delete from books where isbn = ?”, new Object[] { isbn });
    }
  }
```

RICERCA

- Passando alle interrogazioni, il caso più semplice è la ricerca di un intero.

```
public class BookJdbcDao implements BookDao {
    // ...
    // Query di un intero
    public int bookCount() {
          int rowCount = jdbcTemplate.queryForInt("select count(1) from books");
          return rowCount;
        }
  }
```

- Passando alle interrogazioni di oggetti oltre alla query e i suoi parametri è necessario specificare come deve avvenire il mapping tra i risultati presenti nel ResultSet e le proprietà dell'oggetto.

- Per fare questo si utilizza l'interfaccia RowMapper, nel nostro codice implementata dalla classe BookRowMapper, contiene le regole di mapping tra una riga del ResultSet e classe Book ed è utilizzata dal jdbcTemplate per ritornare i valori della query.

SPRING JDBC

```
public class BookJdbcDao implements BookDao {
    ...
    public Book findByISBN(String isbn) { //Query di un singolo oggetto
      return (Book) jdbcTemplate.queryForObject("select * from books where isbn = ?",
                                                 new Object[] { isbn },
                                                 new BookRowMapper());
   }
    public List<Book> findAllBooks() { // Query di una lista
      List<Book> books = (List<Book>) jdbcTemplate.query("select * from books”, new BookRowMapper());
      return books;
    }
}

public class BookRowMapper implements RowMapper {   // Classe RowMapper
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Book book= new Book();
        book.setIsbn(rs.getString("isbn"));
        book.setAutore(rs.getString("autore"));
        book.setTitolo(rs.getString("titolo"));
        return book;
    }
}
```

JDBC TEMPLATE INJECTION

- Grazie a IoC è possibile iniettare nella nostra classe un oggetto di tipo JdbcTemplate già inizializzato con un datasource:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url" value="jdbc:derby://localhost:1527/library;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="bookDao" class="it.html.spring.book.BookJdbcDao">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
</beans>
```

JDBCDAOSUPPORT

- La classe JdbcDaoSupport permette di agevolare l'utilizzo di JdbcTemplate implementando direttamente i metodi setDatasource() e setJdbcTemplate().

- Estendendo questa classe ai propri DAO sara sufficente iniettare un datasource per poter utilizzare un JdbcTemplate attraverso il metodo getJdbcTemplate() (ereditato anch'esso da JdbcDaoSupport).

```
public class BookJdbcDao extends JdbcDaoSupport implements BookDao {
    ...
   public void insert(Book book) {
      getJdbcTemplate().update("insert into book (isbn, autore, titolo) values (?, ?, ?)",
                               new Object[] { book.getIsbn(), book.getAuthor(), book.getTitle() });
    }
}
```

SIMPLEJDBCTEMPLATE

- La classe SimpleJdbcTemplate aggiunge alcune funzionalità offerte da Java 1.5 alla classe base JdbcTemplate, semplificando ulteriormente le procedure di accesso ai dati.

- Grazie al supporto dei parametri a lunghezza variabile è possibile passare direttamente i valori ai metodi di update senza dover utilizzare un array di oggetti.

```
public class BookJdbcDaoSupport extends SimpleJdbcDaoSupport implements BookDao {
    public void insert(Book book) {
      getSimpleJdbcTemplate().update("insert into book (isbn, autore, titolo) values (?, ?, ?)",
                                     book.getIsbn(), book.getAutore(), book.getTitolo());
      ...
}
```

- Con l'utilizzo dei generics e dell'autoboxing anche le query vengono notevolmente semplificate.

SIMPLEJDBCTEMPLATE

- Spring fornisce la classe ParameterizedBeanPropertyRowMapper, un'implementazione dell'interfaccia RowMapper che si occupa di effettuare le operazioni di mapping viste in precedenza in maniera transparente allo sviluppatore.

```
//Query di un singolo oggetto
public Book findByISBN(String isbn) {
    Book book = getSimpleJdbcTemplate().queryForObject("select * from books where isbn = ?",
                                ParameterizedBeanPropertyRowMapper.newInstance(Book.class), isbn);
    return book;
}
// Query di una lista di oggetti
public List findAllBooks() {
    List books = (List) getJdbcTemplate().query( "select * from books",
                                  ParameterizedBeanPropertyRowMapper.newInstance(Book.class));
    return books;
}
```

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- Spring aggiunge alla programmazione JDBC il supporto ai parametri nominali, consentendo l'utilizzo di label come segnalibri al posto del classico punto interrogativo (?).
- Da un lato favorisce la leggibilità delle query e dall'altro elimina possibili complicazioni dovute alla posizionalità dei parametri.
- In Spring questa funzionalità viene offerta sia attraverso l'utilizzo diretto della classe NamedParameterJdbcTemplate, che grazie al supporto del SimpleJdbcTemplate visto in precedenza.

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- Il binding tra i parametri identificati dalle label e i relativi valori avviene tramite la classe SqlParameterSource, la cui implementazione base è la MapSqlParameterSource.

```
public class BookSimpleJdbcDaoSupportNamedValue extends
SimpleJdbcDaoSupport {
    public void update(Book book) {
        MapSqlParameterSource parameters = new MapSqlParameterSource();
        parameters.addValue("isbn", book.getIsbn());
          parameters.addValue("author", book.getAuthor());
          parameters.addValue("title", book.getTitle());
        getSimpleJdbcTemplate().update("update books set author = :author,
title = :title where isbn = :isbn", parameters);
      }
    // ...
  }
```

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- SqlParameterSource offre anche un'altra implementazione in grado di effettuare un mapping automatico dei parametri, la classe BeanPropertySqlParameterSource.

```
public class BookSimpleJdbcDaoSupportNamedValue extends SimpleJdbcDaoSupport {
  public void insert(Book book) {
    // Binding automatico dei parametri
    SqlParameterSource parameters = new BeanPropertySqlParameterSource(book);
    //Inserimento
    getSimpleJdbcTemplate().update("insert into books (isbn, author, title)
values (:isbn, :author, :title)", parameters);
  }
  ...
}
```

JAVA WEB SPRING

- View, ViewResolver
- Gestione Form
- Handler
- Eccezioni
- DAO e JDBC
- **Hibernate**

Dott. Antonio Giovanni Lezzi

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- Un ORM, acronimo di Object Relation Mapping, è una tecnologia che nasce per semplificare la persistenza di oggetti in un database relazionale, generando automaticamente il codice SQL necessario.
- Nel mondo J2E esiste una moltitudine di ORM e, dal canto suo, Spring offre loro un grande supporto che include tra gli altri Hibernate, Oracle TopLink, iBATIS SQL Maps, JDO e JPA.
- Chi fosse al primo approccio con questa tecnologia, può leggere questo articolo.
- Perché un ORM sia in grado di generare codice SQL è necessario specificare il mapping tra gli oggetti da persistere e le tabelle nel database.

MAPPING CON XML

- In accordo con le specifiche Hibernate, uno dei metodi per effettuare questa operazione è utilizzare un file XML che, per convenzione, ha nome:
  <classe>.hbm.xml

- Il file book.hbm.xml responsabile del mapping tra la classe Book vista in precedenza e la relativa tabella nella base dati.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://
hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="it.html.spring.book">
  <class name="Book" table="BOOK">
    <id name="isbn" type="string" column="ISBN"></id>
    <property name="titolo" type="string" column="TITOLO"></property>
    <property name="autore" type="string" column="AUTORE"></property>
  </class>
</hibernate-mapping>
```

- La gestione del mapping per mezzo dei file XML è relativamente semplice, ma può risultare difficile da gestire, sopratutto quando il numero delle classi da persistere è alto.

MAPPING CON ANNOTATIONS

- Un'alternativa è data dall'adozione delle JPA annotations introdotte da Sun a partire da Java 5 per permettere di specificare il mapping all'interno delle stesse classi da persistere. Le JPA annotations possono essere utilizzate da tutti gli ORM JPA-compliant come Hibernate.

- Di seguito è mostrata la classe Book con aggiunte le annotations necessarie per la sua persistenza.

```
@Entity
@Table(name = "BOOK")
public class Book {
  @Id
  @Column(name = "ISBN")
  String isbn;
  @Column(name = "AUTORE")
  String autore;
  @Column(name = "TITOLO")
  String titolo;
  public String getIsbn() { return isbn; }
  public void setIsbn(String isbn) { this.isbn = isbn; }
  public String getAutore() { return autore; }
  public void setAutore(String autore) { this.autore = autore; }
  public String getTitolo() { return titolo; }
  public void setTitolo(String titolo) { this.titolo = titolo; }
  public String toString() {
    return isbn + " - " + autore + " - " + titolo;
  }
}
```

MAPPING CON ANNOTATIONS

- Dal codice risultante si nota subito come l'utilizzo di queste annotations renda le operazioni di mapping più semplici e manutenibili rispetto ai rispettivi file XML. Per questa ragione, da ora in poi, prenderemo in considerazione questa alternativa.

MAPPING CON ANNOTATIONS

- Proseguiamo ora creando un'implementazione specifica per Hibernate del nostro BookDao. Come per JDBC, Spring ci viene in aiuto fornendo HibernateTemplate: uno specifico template che si fa carico delle operazioni di gestione, tra le quali:
    - Richiesta della connessione alla factory
    - Apertura della transazione
    - Gestione delle eccezioni
    - Commit/Rollback della transazione
    - Chiusura della connessione
- Unitamente a HibernateTemplate Spring mette a disposizione la classe di supporto HibernateDaoSupport per facilitarne l'injection all'interno dei DAO. Estendendo questa classe sarà sufficiente richiamare il metodo getHibernateTemplate() per poter avere un hibernate template da utilizzare.

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

```
public class BookHibernateDaoSupport extends HibernateDaoSupport implements BookDao {
    @Transactional
    public int bookCount() {
      return findAllBooks().size();
    }
    @Transactional
    public void delete(String isbn) {
      Book book = (Book) getHibernateTemplate().get(Book.class, isbn);
      getHibernateTemplate().delete(book);
    }
    @Transactional(readOnly = true)
    public List findAllBooks() {
      return getHibernateTemplate().find("from Book");
    }
    @Transactional(readOnly = true)
    public Book findByISBN(String isbn) {
      return (Book) getHibernateTemplate().get(Book.class, isbn);
    }
    @Transactional
    public void insert(Book book) {
      getHibernateTemplate().saveOrUpdate(book);
    }
    public void update(Book book) {
      getHibernateTemplate().saveOrUpdate(book);
    }
  }
```

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- Fino ad ora non abbiamo ancora parlato del problema di transazioni, supponendo che ogni operazione effettuata dal DAO sia di per sé univoca. Nella maggior parte dei casi invece quello che si vuole ottenere è che le operazioni effettuare dai metodi del DAO siano transazionali, cioè abbiano un senso compiuto nella loro interezza.

PARAMETRI NOMINALI JDBC TEMPLATE INJECTION

- Questo comportamento è reso possibile dall'utilizzo dell'apposita annotation @Trasactional con la quale è possibile richiedere al framework che tutte le chiamate ai metodi coinvolti siano trattati nella stessa transazione.

DATASOURCE

- Con HibernateTemplate è possibile implementare un DAO per la persistenza attraverso Hibernate.
- I componenti che entrano in gioco per permettere a HibernateTemplate di svolgere il proprio lavoro sono molteplici, anche se non direttamente visibili, perchè mascherati dall'utilizzo dell'Inversion of Control, semplificando notevolmente le operazioni di persistenza.
- Il primo bean da configurare è quello responsabile della connessione verso la sorgente dati.
- Come per gli esempi relativi a JDBC utilizzeremo il DriverManagerDataSource un semplice datasource in grado di aprire una connessione ogni volta che questa viene richiesta.
- Questo modo di operare non è certamente dei migliori quanto a gestione delle risorse; per questo motivo, in ambienti più complessi, si suggerisce l'utilizzo di datasource più efficienti come il SingleConnectionDataSource.

SESSION FACTORY

- Per far sì che Hibernate svolga il proprio lavoro di persistenza sono necessarie alcune configurazioni che indicano al framework quali parametri utilizzare per la crezione del SessionFactory, l'oggetto responsabile dell'apertura delle sessioni verso il database. Tali parametri includono:
    - Datasource da utilizzare per la connessione con il database
    - Dialetto SQL utilizzato da Hibernate per l'ottimizzazione delle query
    - Lista degli eventuali file di mapping
    - Altre informazioni accessorie
- Nell'utilizzo classico di hibernate queste configurazioni possono essere espresse sotto forma di file xml (utilizzando il file hibernate.cfg.xml) oppure tramite annotations. A queste modalità Spring aggiunge una terza alternativa sfruttando il meccanismo dell'Inversion of Control per la creazione di SessionFactory e la relativa injection dei parametri necessari.
- Di seguito sono mostrate due possibili configurazioni per il bean sessionFactory da impiegare nell'applicazione presa in esempio. In particolare la prima in caso di utilizzo delle annotations JPA per il mapping delle classi da persistere, la seconda in caso di adozione di file XML.

SESSION FACTORY

```
<!-- beans ... -->
  <!-- Session Factory da utilizzare per mapping attraverso JPA Annotations -->
  <bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
      <list>
        <value>it.html.spring.book.Book</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
      </props>
    </property>
  </bean>
  <!-- Session Factory da utilizzare per mapping attraverso file xml-->
  <!-- bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
        <value>book.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
      </props>
    </property>
  </bean -->
  <!-- ... -->
</beans>
```

Per effettuare delle prove è possibile scaricare l'esempio completo (progetto Eclipse).