Alma Mater Studiorum
Università di Bologna

Scuola di Ingegneria

Tecnologie Web T

Esercitazione opzionale
Spring – MVC

Home Page del corso: http://www-db.disi.unibo.it/courses/TW/
Versione elettronica: L.10.opt.SpringMVC.pdf
Versione elettronica: L.10.opt.SpringMVC-2p.pdf

1

Modello Pesante vs. Modello Leggero

Due aspetti ortogonali/indipendenti distinguono tra Modello Pesante e Leggero:

- **costo di accesso ai componenti** (il più rilevante)
- granularità moduli container (non sempre soddisfatto)

Modello Pesante

- **tutti gli accessi** ai componenti all'interno del container **vengono gestiti dal container**
- inoltre il container offre le funzionalità in modo monolitico (modello *all-or-nothing*): non è possibile attivare separatamente le singole funzionalità di effettivo interesse

Modello Leggero

- il **container entra in gioco solo per fornire un riferimento** al componente all'interno del container stesso (tipicamente tramite *Dependency Injection*); **poi accesso diretto** al componente, senza alcuna intermediazione da parte del container
- inoltre attivazione di funzionalità solo quando e se necessario (modello *a sotto-componenti*)

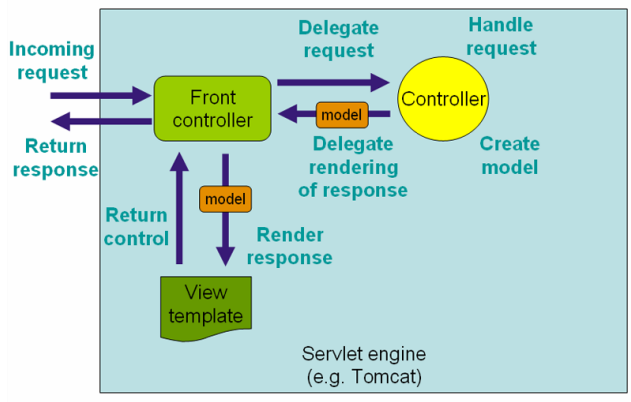
EJB tipico container pesante
Tomcat in associazione a Spring per ottenere molte delle funzionalità offerte da JBoss ma con un approccio più leggero

2

Model-View-Control e Spring

DispatcherServlet è il componente principale di Spring-MVC, che funge da Front Controller interponendosi tra browser e business logic

- punto di accesso all'applicazione
- invoca opportunamente la parte di control relativa alla business logic
- seleziona la view da mostrare



DispatcherServlet

0) Intercetta le HTTP Request in ingresso che giungono al Web container

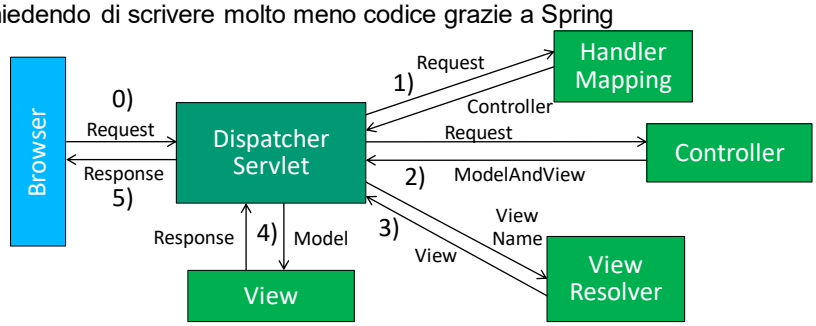
1) Cerca un controller che sappia gestire la richiesta

2) Invoca il controller ricevendo un **model** (l'output della business logic) e una **view** (come l'output deve essere visualizzato)

3, 4) Cerca ed invoca un View opportuno per creare l'HTTP Response

Noi dobbiamo **implementare il Controller** che accede alla business logic

- simile a quanto visto nel caso di MVC con JEE (CalculatorServlet), ma richiedendo di scrivere molto meno codice grazie a Spring



Utilizzare Spring con Eclipse

1) Creare un Dynamic Web Project
- *File → New → Other → Web → Dynamic Web Project*

2) Inserire nella directory *WEB-INF/lib* le librerie Spring utilizzate; ad esempio, per *Spring-MVC* includere:
- commons-logging-1.1.1.jar
- org.springframework.asm-3.1.0.M1.jar
- org.springframework.beans-3.1.0.M1.jar
- org.springframework.context-3.1.0.M1.jar
- org.springframework.core-3.1.0.M1.jar
- org.springframework.expression-3.1.0.M1.jar
- org.springframework.web-3.1.0.M1.jar
- org.springframework.web.servlet-3.1.0.M1.jar

Framework Spring completo (non necessario per questa esercitazione):
- *http://www.springsource.org/download*

Per cominciare

Il file **SpringMVC_TecWeb.zip** contiene due progetti Eclipse

Importare il progetto come visto nelle precedenti esercitazioni, senza esploderne l'archivio su file system (lo farà Eclipse)
- *File → Import → General → Existing Projects into Workspace → Next → Select archive file*

Due progetti Eclipse:
- **opt_TecWeb_SpringMVC_Web**: Dynamic Web Project per Servlet/JSP, risorse statiche, già configurato per Spring-MVC
- **opt_TecWeb_EJB**: EJB Project per Session Bean ed Entity Bean

Effettuare il deploy su JBoss (vedi esercitazione su J2EE) ed accedere all'applicazione: *http://localhost:8080/TecWebSpringMVC_Web/*

Calcolatrice/Accumulatore Spring-MVC

Rispetto a esercitazione su J2EE solo la Servlet è stata modificata (JSP ed EJB identici)

Per supportare gli EJB è ancora necessario l'uso di JBoss; una soluzione completa avrebbe comportato l'uso di Spring anche per EJB (eventualmente lo vedrete in altri corsi di Laurea Magistrale…)

Notate i molti file jar presenti in **TecWeb_SpringMVC_Web/WEB-INF/lib**, tutti necessari per supportare Spring-MVC

Contenuto e Javadoc disponibile al link
http://static.springsource.org/spring/docs/3.0.x/javadoc-api/

DispatcherServlet e web.xml

Innanzitutto è necessario specificare al Web Container tramite il file **/WebContent/WEB-INF/web.xml** che deve instradare le richieste a *org.springframework.web.servlet.DispatcherServlet*

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

L'url-pattern specificato qui sopra comporta l'instradamento di qualunque richiesta a DispatcherServlet

1) Handler Mapping

È necessario definire un **mapping tra richieste HTTP** ricevute da DispatcherServlet e **Controller** che accedono alla business logic

a) Creare un file xml nella directory WEB-INF del progetto Dynamic Web il cui nome dipende da quanto specificato in <servlet-name> di DispatcherServlet (file web.xml)

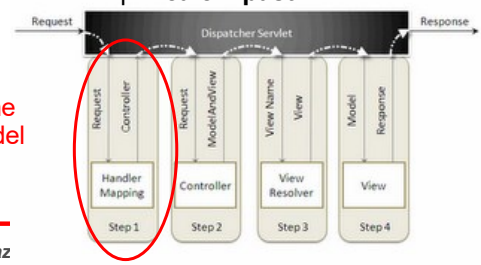
- ad esempio, se <servlet-name>**CalculatorSpringMVC**</servlet-name>, allora il file xml sarà **CalculatorSpringMVC-servlet.xml**

b) Specificare in questo file **la classe che deve gestire le richieste** in base all'url specificato, ad esempio

```
<bean name="/calc*" class="it.unibo.tw.web.controller.CalculatorController"/>
<bean name="/past*" class="it.unibo.tw.web.controller.CalculatorController"/>
```

instrada tutte le richieste con url che inizia per **calc** o **past** alla classe CalculatorController

Dependency Injection: specifichiamo il componente che funge da **Controller** all'interno del file xml di configurazione



2) Controller

Classe che accede alla business logic, l'unica che è necessario implementare

Nel nostro esempio **CalculatorController** implementa l'interfaccia **Controller**, definendo il metodo *public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response);*

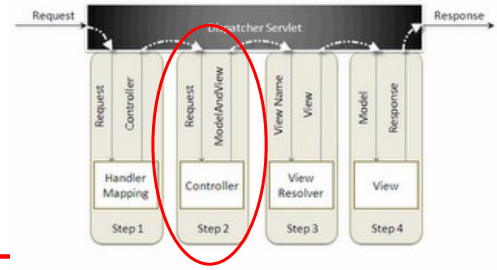
a) invoca opportunamente i Session Bean (che implementano la business logic) in relazione ai parametri della HTTP Request

b) **restituisce un oggetto ModelAndView** tramite cui specifica 1) i **parametri da inserire nella risposta** e 2) il **nome della view** successiva

- vedere package ***org.springframework.web.servlet.mvc*** per una lista completa di possibili Controller

Confrontate CalculatorController con la classe CalculatorServlet vista in passato

Dependency Injection: il Controller indica la View successiva **tramite nome logico**, senza alcun riferimento diretto alla risorsa che la implementa



3) View Resolver

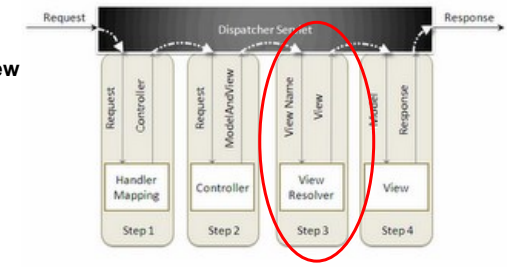
È necessario specificare nel file CalculatorSpringMVC-servlet.xml **come ottenere una View** partendo dal View Name dichiarato nel Controller

Nel nostro caso viene utilizzato un **UrlBasedViewResolver** che identifica la View partendo dal nome della View specificato nel Controller; ad esempio, View Name = calc → View = /calc.jsp

```
<bean name="viewResolver"
  class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass"
value="org.springframework.web.servlet.view.InternalResourceView"/>
  <property name="prefix" value="/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

- vedere package **org.springframework.web.servlet.view** per una lista completa di possibili View Resolver

Dependency Injection: nel file xml di configurazione specifichiamo il mapping tra nome logico di View e risorsa che implementa tale View



4) View

Infine **DispatcherServlet passa il Model creato alla View** che a sua volta genera HTTPResponse finale, ovvero l'output da inviare al client

- nel nostro caso si tratta di **calc.jsp** e **past.jsp**, identiche alle JSP dell'esercitazione scorsa
- questa volta però i parametri utilizzati dalle JSP vengono **passati come Model** (vedi CalculatorController di questa esercitazione) e non come attributi (vedi CalculatorServlet vista in passato per EJB)