**Strutturazione di un Progetto**

**Frontend:** JSP/Servlet  
**Backend** lo si può dividere in due con : mybatis, Dao, Mapper, Model  
**RestController:** Classi java POJO  
  
In futuro:

**Creare un progetto con servlet:**  
  
Si parte da

1. Dynamic Web Project e si crea un nuovo server, si sceglie la 3.1 dopo aver scelto wildfly 12
2. Si utilizza la JRE 1.8 con Java 1.8, next >
3. Si sceglie di creare un file web.xml!
4. Si crea un Java project e si sceglia la JRE 1.8 (terza opzione)
5. Cliccare con tasto destro sul progetto BackEnd e controllare la facets se corrisponde alla versione Java dichiarata
6. Si crea l’EAR e si selezionano BE e FE
7. Dentro la cartella BE si crea un package it.exolab.model (Dentro si mettono le classi) (Ricorda che quando si crea il metodo equals bisogna lasciare solo l’ISBN e non tutto)
8. Dentro la cartella BE si crea un package it.exolab.dao e si crea una classe con il concetto del singleton.

Avremo quindi la creazione del DAO

1. **Creazione DAO**

|  |
| --- |
| **package** it.exolab.dao;  ​  **import** java.util.ArrayList;  **import** java.util.List;  ​  **import** it.exolab.model.Libro;  ​  **public** **class** LibroDAO {  ​  **private** List<Libro> dati = **new** ArrayList<Libro>();  ​  **private** **static** LibroDAO instance = null;  ​  **private** LibroDAO() {  ​  }  ​  **public** **static** LibroDAO getInstance() {  **if** (instance == null) {  instance = **new** LibroDAO();  }  **return** instance;  }  ​  **public** void insert(Libro model) {  dati.add(model);  } // Metodo per inserire  ​  **public** void update(Libro model) {  ​  int pos = dati.indexOf(model);  dati.set(pos, model);  ​  } // Metodo per aggiornare, dando la posizione di model  ​  **public** void delete(String isbn) {  ​  Libro model = **new** Libro();  model.setIsbn(isbn);  dati.remove(model);  ​  } // Metodo per rimuovere un libro dandogli l'isbn!  ​  **public** List<Libro> findAll() {  ​  **return** dati;  ​  }  ​  **public** Libro findByIsbn(String isbn) {  Libro model = **new** Libro();  model.setIsbn(isbn);  **return** dati.get(dati.indexOf(model));  ​  }  ​  }  ​ |

1. *Crea una lista chiamata dati che memorizza oggetti di tipo Libro.*

**private** List<Libro> dati = **new** ArrayList<Libro>();

1. *Dichiara una variabile statica per implementare il pattern Singleton. Permette di avere un'unica istanza di LibroDAO.*

**private** **static** LibroDAO instance = null;

1. *Costruttore privato: impedisce la creazione di istanze esterne alla classe.*

**private** LibroDAO() {

}

1. *Metodo SINGLETON statico per ottenere l'unica istanza di LibroDAO. Se non esiste ancora, viene creata.*

**public** **static** LibroDAO getInstance() {

**if** (instance == null) {

instance = **new** LibroDAO();

}

**return** instance;

}

1. *Aggiunge un oggetto Libro alla lista dati.*

**public** void insert(Libro model) {

dati.add(model);

}

1. *Trova la posizione di un libro nella lista tramite il metodo indexOf, poi lo sostituisce con l'oggetto passato come parametro.*

**public** void update(Libro model) {

int pos = dati.indexOf(model);

dati.set(pos, model);

}

1. *Crea un oggetto Libro con l'ISBN passato come parametro. Cerca e rimuove l'oggetto corrispondente nella lista.*

**public** void delete(String isbn) {

Libro model = **new** Libro();

model.setIsbn(isbn);

dati.remove(model);

}

1. *Crea un oggetto Libro con l'ISBN passato come parametro. Cerca e restituisce il libro corrispondente nella lista.*

**public** void delete(String isbn) {

Libro model = **new** Libro();

model.setIsbn(isbn);

dati.remove(model);

}

**Questa classe implementa un semplice sistema per gestire una collezione di oggetti Libro. Utilizza una lista in memoria per archiviare, aggiornare, eliminare e recuperare libri. Il pattern Singleton assicura che esista una sola istanza di LibroDAO.**

1. Adesso si passa al **FRONTEND**
2. Dentro a Java Resources creerò un package it.exolab.controller (contiene la servlet) lascio tutto di default
3. Modifico il file XML ed aggiungo quanto segue:

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd" id="WebApp\_ID" version="3.1">   <display-name>libreriaFE</display-name>     <servlet>   <servlet-name>libro</servlet-name>   <servlet-class>it.exolab.controller.libreriaController</servlet-class>   </servlet>    <servlet-mapping>  <servlet-name>libro</servlet-name>  <url-pattern>\*.do</url-pattern>  </servlet-mapping>     <welcome-file-list>     <welcome-file>index.html</welcome-file>     <welcome-file>index.jsp</welcome-file>     <welcome-file>index.htm</welcome-file>     <welcome-file>default.html</welcome-file>     <welcome-file>default.jsp</welcome-file>     <welcome-file>default.htm</welcome-file>   </welcome-file-list>  </web-app> |

1. **<servlet-name>**: Assegna un nome al servlet (libro).
2. **<servlet-class>**: Specifica la classe Java che implementa il servlet (it.exolab.controller.libreriaController).

|  |
| --- |
| <servlet>   <servlet-name>libro</servlet-name>   <servlet-class>it.exolab.controller.libreriaController</servlet-class>  </servlet>  ​ |

1. **<servlet-name>:** Fa riferimento al nome del servlet definito sopra (libro).
2. **<url-pattern>:** Indica che tutte le richieste con suffisso \*.do saranno gestite dal servlet libreriaController. Ad esempio:
3. **http://localhost:8080/libreriaFE/addLibro.do** sarà indirizzato al servlet.

|  |
| --- |
| <servlet-mapping>   <servlet-name>libro</servlet-name>   <url-pattern>\*.do</url-pattern>  </servlet-mapping> |

Lui memorizzerà il contest del FE, quindi l’url sarà:

[*localhost:8080/libreriaFE/index.do*](http://localhost:8080/libreriaFE/index.do)

Adesso:

1. **CREAZIONE WEBAPP**
2. Dobbiamo creare delle pagine ***dentro*** **webapp** (o WebContent) ovvero delle cartelle presenti all’interno di libreriaFE > src > webapp
3. Pagina index.jsp (elenco dei libri)
4. Pagina new.jsp (inserimento del libro)
5. Pagina edit.jsp (pagina che aggiorna)
6. Pagina view.jsp (che presenta le pagine di un libro)

Tienamocela pronte ed andiamo a:  
  
Immettere la logica dentro LibreriaController (La servlet) ed abbiamo

1. Nel DOGET: Vedere dettaglio di un libro
2. Nel DOGET: Vedere tutti i libri
3. Nel DOGET: Voglio cancellare un libro
4. Nel DOGET: Voglio caricare la pagina new.jsp
5. Nel DOGET: Voglio caricare la pagina edit.jsp
6. UNO SWITCH
7. Nel DOPOST: Voglio inserire un libro
8. Nel DOPOST: Voglio modificare un libro
9. UNO SWITCH

In pratica io devo tenere bene a mente cosa voglio fare in questi metodi e scrivermelo.

|  |
| --- |
| **protected** void doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  ​  String action = request.getParameter("action");  **switch** (action) {  **case** "view":  // Voglio vedere il commento  **break**;  ​  **case** "index":  // Voglio vedere tutti i libri  **break**;  ​  **case** "delete":  // Voglio cancellare un libro  **break**;  }  ​  } |

Cosa devo pensare adesso?

Pensiamo ad index, che abbiamo la index.jsp e quindi partiamo da questa ovvero il case “index” includiamo dei metodi.

Alla index.JSP però dovrò passare l’elenco dei libri, perché altrimenti non riuscirebbe a passare niente:

|  |
| --- |
| **case** "index":  // Voglio vedere tutti i libri  ​  List<Libro> model = LibroDAO.getInstance().findAll();  request.setAttribute("model", model);  request.getRequestDispatcher("/index.jsp").include(request, response);  **break**;  ​  **case** "delete":  // Voglio cancellare un libro  **break**;  } |

Eccolo qui, il CASE INDEX è completato, si scrive così.  
 **SI LAVORA PER COMPARTIMENTI STAGNI, PRENDO UN CASO E LO FINISCO**

1. **INIZIALIZZAZIONE INDEX.JSP**

I compartimenti verranno presentati in tabellati:

|  |
| --- |
| <table>  <thead>  <th>ISBN</th>  <th>Titolo</th>  <th>Autore</th>  <th></th>  </thead>  <tbody>  <tr>  <td>ISBN</td>  <td>Titolo</td>  <td>Autore</td>  <td></td>  </tr>  </tbody>  </table> |

Guarda bene, I bottoni effettivi sono dentro TH, in <td> ho le stesse parole, ma scritte solo per ricordarmi cosa ci devo inserire dopo, per scrivere delle righe dinamiche si usano i JSTL.  
Non è che non sopravviviamo senza JSTL, però ci da comandi utili, scrivere dentro la JSP diventa laborioso, la JSTL ci viene incontro.

**Ma come si fa?**

Si utilizza e quindi si incolla la medesima taglib dentro la JSP:   
***<%@ taglib uri = "http://java.sun.com/jsp/jstl/core" prefix = "c" %>***

***(N.B, la “c” è di default, potrei anche scriverci francescototti, basta riportarlo quando faccio una funzione).***

|  |
| --- |
| <c:forEach items="${model}" var="item">  ​  <tr>  <td>${item.isbn}</td>  <td>${item.titolo}</td>  <td>${item.autore}</td>  <td></td>  </tr>  </tbody>  </table> |

A questo punto Andiamo ad incollare questo <c:foreach, che abbiamo grazie all’utilizzo della tagList, ed utilizziamo un linguaggio chiamato expression language, sai come si fa?  
  
Semplice:  
Si definisce “${model}” *(****Lui va a pescare model negli attributi passati dalla request****)  
essendo model presente nella index.jsp va a pescare il model del case index, ovviamente.*

*[request.setAttribut(“model”, model);] > DENTRO IL CONTROLLER* ***>*** *NEL DOGET* ***>*** *NEL CASE INDEX*  
  
 perché ricordiamo che noi dobbiamo passare un libro, ed una variabile chiamata in questo caso item, ma potrebbe chiamarsi anche francescototti, quindi si modificano i parametri tappabuchi di prima dentro i td con questa sintassi ${item.isbn}!

1. **La INDEX.JSP sarà quindi così:**

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=ISO-8859-1"     pageEncoding="ISO-8859-1"%>  <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>  ​  <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Lista Libri</title>  </head>  <body>  ​     <table>         <thead>             <tr>                 <th>ISBN</th>                 <th>Titolo</th>                 <th>Autore</th>                 <th>Azioni</th>             </tr>         </thead>         <tbody>             <c:forEach items="${model}" var="item">                 <tr>                     <td>${item.isbn}</td>                     <td>${item.titolo}</td>                     <td>${item.autore}</td>                     <td><button>Dettagli</button></td>                 </tr>             </c:forEach>         </tbody>     </table>  ​  </body>  </html>  ​ |

​  
  
Arrivati a questo punto bisogna creare un libro

1. **Creazione di un libro e visualizzazione nell'applicazione**

Per aggiungere un libro al sistema e verificarne la presenza nella lista, segui i passaggi descritti:

1. **Aggiungi il libro nella classe LibroDAO**:  
   Vai nel costruttore privato LibroDAO() all'interno della classe LibroDAO. Qui puoi inizializzare la lista di libri (dati) aggiungendo manualmente un libro predefinito.  
   Modifica il costruttore come segue:

|  |
| --- |
| private LibroDAO() {     dati.add(**new** Libro("12345", "Il Signore degli Anelli", "J.R.R. Tolkien"));     dati.add(**new** Libro("67890", "1984", "George Orwell"));  }  ​ |

In questo esempio, sono stati aggiunti due libri con valori predefiniti di **ISBN**, **titolo**, e **autore**. Sostituisci questi dati con quelli desiderati.

1. **Avvia l'applicazione**:  
   Esegui il progetto e assicurati che il server sia attivo e configurato correttamente. La configurazione del server dovrebbe includere il collegamento al progetto frontend (libreriaFE) e backend.
2. **Accedi alla lista dei libri**:  
   Apri il browser e visita il seguente URL per vedere l'elenco completo dei libri attualmente presenti nel sistema:

**http://localhost:8080/libreriaFE/index.do?action=index**

Questo URL invia una richiesta al LibreriaController, che processa l'azione index e restituisce la lista dei libri visualizzata nella pagina index.jsp.

**Ricorda:** *Queste sono tecnologie vecchie, difficilmente verranno fatte domande sulle servlet, però è la base di quello che viene dopo, ovvero, tecnologie di alto livello.*  
  
**7) Creazione di un HREF:**

1. All’interno della index.jsp, sotto la chiusura del </tbody> aggiungiamo:

|  |
| --- |
| </tbody>  <tfoot>  <tr>  <tfoot>  <tr>  <td><a href=index.do?action=new”>Inserisci</a>  <td>  </tr>  </tfoot>  </table>  ​  </body>  </html> |

**Sai a cosa serve questa cosa? Semplice:**

Viene aggiunto un elemento HTML **<a>** (un collegamento ipertestuale).

Questo link punta all'URL **index.do?action=new**, che è una richiesta inviata al server con un parametro action=new.

**Scopo**: Questo link consente all'utente di accedere a una nuova funzionalità, in questo caso "Inserisci", che potrebbe essere per aggiungere un nuovo elemento **(ad esempio, un nuovo record o elemento nel sistema).**

**ANDIAMO AVANTI:**

Adesso all’interno del nostro controller includiamo la /new.jsp:

1. Includiamo anche la edit.jsp allo stesso modo:

|  |
| --- |
| **case** "new":  request.getRequestDispatcher("/new.jsp").include(request, response);  **break**;  ​  **case** "edit":  request.getRequestDispatcher("/edit.jsp").include(request, response);  **break**;  } |

1. Siamo arrivati nel momento in cui il collegamento visto in precedenza porta alla pagina “new.jsp”, questa JSP al momento sappiamo che è vuota...  
   Bene, è arrivato il momento di andare a modifcarla.

Faccio corrispondere I name delle caselle di testo ai nomi degli attributi del modello:

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=ISO-8859-1"     pageEncoding="ISO-8859-1"%>  <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Insert title here</title>  </head>  <body>  ​  <form>  <label>ISBN</label><br>  <input type = "text" name = "isbn" /><br>  ​  <label>Titolo</label><br>  <input type = "text" name = "titolo" /><br>  ​  <label>Autore</label><br>  <input type = "text" name = "autore" /><br>  <br>  <input type= "submit" value= "Salva"/>  ​  ​  </form>  ​  </body>  </html> |

1. Adesso modifichiamo il form in questo modo, perché non deve finire in doPost, ma in doGet:

|  |
| --- |
| <form method="post"> |

In questo momento però, l’indirizzo associato è (visibile nel browser) >

index.do?action=new

Lui però dovrà sottomettere ad un indirizzo che io voglio e devo definire subito dopo, così:

|  |
| --- |
| <form method="post" action="index.do?action=insert"> |

QUINDI!  
  
1) L'utente clicca su "Inserisci" nella pagina index.jsp, accedendo a new.jsp.

2) In new.jsp, compila i campi e clicca su "Salva".

3) Il modulo invia una richiesta POST a index.do?action=insert.

4) Il controller servlet analizza l'azione insert e salva i dati ricevuti

**Ma quindi perché la modifica?** Perché vedere nel posto e perché inserire l’indirizzamento ad insert?

**Cosa cambia:**

1. **Metodo POST:**
   * Il modulo invia i dati con il metodo POST invece di GET.
   * **Vantaggio:** Con POST, i dati non sono visibili nell'URL e si possono inviare informazioni più grandi.
2. **Action:**
   * La richiesta viene inviata a index.do?action=insert invece che a index.do?action=new.
   * Questo indirizza la richiesta al controller servlet, che gestirà l'azione "insert" per salvare i dati.

**Risultato:**

Quando l'utente clicca su **Salva**, i dati vengono inviati al server con:

* URL: index.do?action=insert
* Metodo: POST
* Dati: isbn, titolo, autore come coppie chiave-valore

***QUINDI QUESTA MODIFICA SOTTOMETTE IL PULSANTE A MANDARMI ALL’INSERT PRESENTE NELLO SWITCH DENTRO AL DOPOST!***

**8) Adesso modifichiamo gli switch nel doPost (Ciclo virtuoso)**

|  |
| --- |
| **case** "insert":  String isbn = request.getParameter("isbn"); |

*(N.B, I nomi delle variabili dentro gli apici devono chiamarsi in maniera identica a quelli dentro l’insert, altrimenti non funziona nulla)*

Però dato che il DAO accetta un modello di tipo libro, non gli posso passare solo queste veriabili, ma devo istanziare un oggetto.

|  |
| --- |
| **protected** void doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  ​  String action = request.getParameter("action");  **switch** (action) {  **case** "insert":  String isbn = request.getParameter("isbn");  String titolo = request.getParameter("titolo");  String autore = request.getParameter("autore");  Libro model = **new** Libro (isbn,titolo,autore);  LibroDAO.getInstance().insert(model);  response.sendRedirect("index.do?action=index");  **break**; |

Ho creato un ciclo virtuoso, inserisco dei parametri,torno indietro ed inserisco,

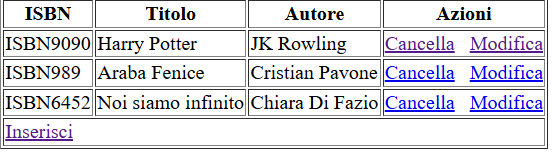
Il ciclo virtuoso si crea perché:

1. L'utente può inserire nuovi dati.
2. Dopo ogni inserimento, viene aggiornato l'elenco, mostrando i dati appena salvati.
3. L'utente può continuare ad aggiungere altri elementi senza interruzioni.

Si passa quasi alla fine di questa creazione, ovvero la creazione dei tasti di cancellazione e modifica

1. **CANCELLAZIONE E MODIFICA**
2. Nella Index aggiungiamo il tasti della cancellazione, ricordiamo che l’azione di cancellazione negli switch si chiama “delete”, quindi, andiamo all’interno della index.jsp.

|  |
| --- |
| <tbody>             <c:forEach items="${model}" var="item">                 <tr>                     <td>${item.isbn}</td>                     <td>${item.titolo}</td>                     <td>${item.autore}</td>                     <td>                         <a href="index.do?action=delete&isbn=${item.isbn}">Cancella</a>                         &nbsp;                         <a href="index.do?action=edit&isbn=${item.isbn}">Modifica</a>                     </td>                 </tr> |

Ecco qui, abbiamo creato dei semplici comandi di cancellazione e modifica, il risultato sarà quindi il seguente:  


1. Dopo aver creato i tasti che permettono il reindirizzamento, dobbiamo creare la logica dietro la cancellazione, vediamo come si fa:

CANCELLAZIONE: La faccio nel doGet,

|  |
| --- |
| **case** "delete":  String isbn = request.getParameter("isbn");  LibroDAO.getInstance().delete(isbn);  response.sendRedirect("index.do?action=index");  **break**; |

Eccolo qui: Come abbiamo fatto in precedenza vediamo che richediamo il parametro “isbn” istanziamo un librodao che avrà il metodo definito inizialmente, ovvero in questo caso quello di rimozione, cosa vogliamo rimuovere? L’isbn, che essendo una chiave valore va ad eliminare tutto il libro.  
successivamente creiamo il ciclo virtuoso che ci reindirizza alla index, in pratica:

1. **Recupero del parametro:** Viene richiesto il parametro isbn dalla richiesta HTTP utilizzando request.getParameter("isbn"), che rappresenta l'identificativo univoco del libro da eliminare.
2. **Eliminazione del libro:** Si utilizza il metodo delete della classe LibroDAO, passando l'ISBN come argomento, per rimuovere il libro corrispondente dal database o dalla struttura dati.
3. **Reindirizzamento:** Dopo aver completato l'operazione di eliminazione, il client viene reindirizzato alla pagina index.do con l'azione index attraverso response.sendRedirect.
4. **Chiusura del case:** Si esce dal blocco case "delete" con l'istruzione break, evitando che vengano eseguiti altri casi del costrutto switch.

|  |
| --- |
| **case** "edit":  isbn = request.getParameter("isbn");  Libro libro = LibroDAO.getInstance().findByIsbn(isbn);  request.setAttribute("model", libro);  request.getRequestDispatcher("/edit.jsp").include(request, response);  **break**;  } |

1. Ecco qui, avevo già delle variabili chiamate isbn, quindi al di sotto della dichiarazione del metodo doGet ho messo String isbn = null;  
   ed ho tolto gli String agli ISBN nel doGet, e faccio un Libro libro semplicemente perché un Libro model già esiste, quindi cambio il nome solo per questo!
2. Ho bisogno di andare nella edit.jsp e modificarla! Mi copio il form, quindi tutto quello che ho scritto nel new.jsp

**Che modifiche sono state fatte qui sotto?** Semplicemente cambiato il “index.do?action=” con l’update ed inseriti i value, ovvero la possibilità di riuscire ad accedere a degli attributi, tramite l’expression language.

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=ISO-8859-1"     pageEncoding="ISO-8859-1"%>  <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Modifica libro</title>  </head>  <body>  ​  <form method="post" action="index.do?action=update">  <label>ISBN</label><br>  <input type = "text" name = "isbn" value = "${model.isbn}" /><br>  ​  <label>Titolo</label><br>  <input type = "text" name = "titolo" value =" ${model.titolo}" /><br>  ​  <label>Autore</label><br>  <input type = "text" name = "autore" value =" ${model.autore}" /><br>  <br>  <input type= "submit" value= "Salva"/>  ​  ​  </form>  ​  </body>  </html> |

**Adesso? SI MODIFICA E QUINDI S’IMPLEMENTA L’UPDATE:**

Ritorniamo nel controller, nel suo doPost e implementiamo l’update, identico all’insert, cambia solo la chiamata al metodo nel DAO:

|  |
| --- |
| **case** "update":  isbn = request.getParameter("isbn");  titolo = request.getParameter("titolo");  autore = request.getParameter("autore");  model = **new** Libro(isbn, titolo, autore);  LibroDAO.getInstance().update(model);  response.sendRedirect("index.do?action=index");  **break**;  } |

N.B: Piccola novità che non c’entra tanto con questa cosa, ma al di sotto del doGet invece viene aggiunto questo:

|  |
| --- |
| **protected** void doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  ​  String isbn = null;  ​  String action = request.getParameter("action");    **if** (action == null)  action = "index"; |

**Quell’ IF sai a cosa sta a significare?**  
  
Significa semplicemente che se l’azione è null di default vengono reindirizzato all’index, comodo no?

1. **ULTIMA COSA, LA VIEW**

|  |
| --- |
| <%@ page language="java" contentType="text/html; charset=ISO-8859-1"  pageEncoding="ISO-8859-1"%>  <!DOCTYPE html>  <html>  <head>  <meta charset="ISO-8859-1">  <title>Dettaglio libro</title>  </head>  <body>  ​  <label>ISBN</label>  <br> ${model.isbn}  <br>  ​  <label>Titolo</label>  <br> ${model.titolo}  <br>  ​  <label>Autore</label>  <br> ${model.autore}  <br>  <br>  ​  ​  <a href="index.do">Indietro</a>  ​  </body>  </html> |

Ecco la view.jsp, come vedi vengono passati i parametri e basta, niente di più, nel controller avrò questo:

|  |
| --- |
| **case** "view":  isbn = request.getParameter("isbn");  Libro libroDaMostrare = LibroDAO.getInstance().findByIsbn(isbn);  request.setAttribute("model", libroDaMostrare);  request.getRequestDispatcher("/view.jsp").include(request, response);  **break**; |

*(FILTRI DA MODIFICARE, non guardarli)*

I filtri: Sono una specie di servlet, vengono invocati prima di qualsiasi servlet.  
Il filtro può o negare l’accesso o aggiungere alla richiesta delle cose.

Un filtro si configura con webxml ma anche con annotazione.  
  
dentro il package si fa new filtro, ed implementa una particolare interfaccia chiamata filter.  
  
ed il metodo che dobbiamo fare l’override è il doFilter, ed assomiglia al doGet e doPost, richiesta e risposta sono i genitori della https servlet request e response, i filtri in realtà agiscono non solo su protocolli http, ma anche protocolli diversi.  
  
Le servlet vengono invocate sulla base di un url giusto? Anche in questo caso di filtri avviene così e quindi si definiscono alla stessa maniera.  
  
se scrivo \* agisce su tutto, ed \*.do su quelle che terminano per .do.  
  
se cho due filtri associati alla stessa url, devono elaborarli a tutti e due, cosa succede se sono associati alla stessa url quindi? (dimmelo tu...)

i filtri vengono elaborati tutti quanti ma solo se scriverà chain.doFilter, finquando non lo riceve non elabora il prossimo filtro.  
  
Vengono utilizzati per esempio se finisci in un pagina ma non sei loggato ti butta sulla login, e può negare all’accesso ad una risorsa.  
Chiaramente una negazione posso farla da servlet, ma i filtri nascono per questo.