Java Servlets
Concetti e Programmazione di Base

Giuseppe Della Penna
Università degli Studi di L'Aquila

dellapenna@univaq.it
http://www.di.univaq.it/gdellape

Introduzione alle Servlet

- Le servlet sono particolari classi Java che vengono eseguite in server web opportunamente predisposti (**servlet containers**).
- Le servlet sono esposte all'esterno come risorse web standard (hanno, cioè, una URL che le caratterizza).
- Le servlet agiscono nella tradizionale modalità *request/response*, tipica del *server side scripting*: quando l'utente ne fa richiesta tramite la URL associata, il server attiva la servlet, la esegue, e ne restituisce il risultato come contenuto della risorsa.
- Le **servlet API** di Java permettono di programmare in maniera completamente indipendente dalle caratteristiche del server, del client e del protocollo di trasferimento.
- Il codice è normale codice Java, che può far uso di tutte le librerie e le utilità del linguaggio, tra cui la connessione a tutti i DBMS tramite JDBC, l'uso di XML tramite JAXP, ecc.
- Le servlet sostituiscono i classici CGI fornendo un elevatissimo grado di sicurezza, versatilità e astrazione al programmatore.

Dove e Come Eseguire una Servlet

- Per eseguire una servlet è necessario disporre di un server che possa fungere da **container**, fornendo adeguato supporto alla loro attivazione ed esecuzione.
- Nel mondo free, il server più utlizzato a questo scopo è **Tomcat** (Apache group).
- Il tradizionale server Apache non è adatto a contenere servlet, tuttavia è possibile integrarlo se necessario ad una installazione di Tomcat tramite un opportuno *connector*.
- Tomcat è un container *leggero*, ed **implementa solo le tecnologie di base** per lo sviluppo delle web applications (*Servlet*, *JSP*). Il **JEE Web profile** completo è disponibile in server più complessi, come *TomEE*, basato su Tomcat, o *Glassfish*.

Configurazione di Apache Tomcat
Installazione

- Apache Tomcat, disponibile per tutte le piattaforme (è esso stesso un programma Java) può essere scaricato dal sito http://tomcat.apache.org/.
- L'istallazione su Windows e Unix è semplificata dall'uso di script di installazione completamente automatici.
  - Su entrambe le piattaforme, è possibile scegliere di avviare il server come **servizio** (windows) o **demone** (unix), cioè in automatico, oppure **manualmente**.
  - E' sempre necessario che l'istallazione di Java (o meglio del JRE) sia individuabile dallo script di istallazione di Tomcat. A questo scopo, verificare che la variabile di ambiente **JAVA_HOME** sia correttamente impostata.
- Una volta mandato in esecuzione, Tomcat risponde di default alla porta **8080**.
- Connettendosi alla url http://localhost:8080/manager/ è possibile configurare il server tramite un'applicazione web e monitorare lo stato delle web application in esecuzione sul server.
- Per poter accedere all'amministrazione, è prima di tutto necessario **creare un utente con privilegi amministrativi**, aggiungendo al file **conf/tomcat-users.xml** una riga come la seguente

```
<user username="admin" password="adminadmin" roles="admin,manager"/>
```

Configurazione di Apache Tomcat

Creazione di un nuovo contesto

- Le applicazioni web sono eseguite in **contesti**. Ogni contesto, in generale, corrisponde a una particolare directory che viene configurata nel server e associata a una URL specifica.
- Per creare manualmente un nuovo contesto è sufficiente creare una sottodirectory nella directory **webapps** di Tomcat. Il nome del contesto sarà quello della directory creata.
- A questo punto, per testare il nuovo contesto, è possibile inserire un file html nella directory indicata e provare a caricarlo con la URL http://localhost:8080/PATH/NOMEFILE, dove path è il nome del contesto. Ad esempio http://localhost/progetto/index.html
- All'interno della directory creata, è necessario approntare una particolare struttura di sottodirectory e files per rendere pienamente funzionale la web application. I principali elementi di questa struttura verranno illustrati di seguito.
- Tuttavia, **il metodo di installazione consigliato per le web application è quello di usare un IDE (ad esempio Netbeans) per realizzare l'applicazione ed effettuarne il packaging in un file war (Web ARchive), che potrà poi essere copiato direttamente nella directory webapps di Tomcat.**

Configurazione di Apache Tomcat

Struttura di un contesto

- Le directory corrispondenti ad una web application hanno una struttura base particolare che permette al server di accedere alle risorse dinamiche (servlet, jsp) e statiche (html, css, immagini, ecc.). In particolare:
  - La sottodirectory WEB-INF, contiene alcuni files di configurazione, tra cui il *web application deployment descriptor* (web.xml).
  - La sottodirectory WEB-INF/classes, contiene le classi Java dell'applicazione, comprese le servlet.
    - (!) Secondo le convenzioni Java, le singole classi andranno poste all'interno di un albero di directory corrispondente al loro package.
  - La sottodirectory WEB-INF/lib, contiene le librerie JAR necessarie all'applicazione, comprese quelle di terze parti, come i driver JDBC.
  - Tutti le altre sottodirectory del contesto, compresa la stessa root directory, conterranno normali files come pagine html, fogli di stile, immagini o pagine JSP.

Configurazione di Apache Tomcat

Inserimento di una servlet in un contesto

- Per rendere una classe Java disponibile come risorsa di tipo servlet, è necessario configurarne le caratteristiche tramite un file detto **web application deployment descriptor**. Questo file, denominato **web.xml**, deve essere posto nella directory WEB-INF dell'applicazione.
- Un semplice esempio di descrittore è il seguente:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems,
Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
 <display-name>Progetto IW</display-name>
 <description>Progetto X</description>
 <servlet>
  <servlet-name>Servlet1</servlet-name>
  <description>Questa servlet implementa la funzione Y
</description>
  <servlet-class>org.iw.project.class1</servlet-class>
 </servlet>
 <servlet-mapping>
  <servlet-name>Servlet1</servlet-name>
  <url-pattern>/funzione1</url-pattern>
 </servlet-mapping>
 <session-config>
  <session-timeout>30</session-timeout>
 </session-config>
</web-app>
```

- Ciascuna servlet è configurata in un elemento **<servlet>** distinto. L'elemento **<servlet-class>** deve contenere il nome completo della classe che implementa la servlet.
- E' necessario poi mappare ciascuna servlet su una URL tramite l'elemento **<servlet-mapping>**. L'**<url-pattern>** specificato andrà a comporre la URL per la servlet nel seguente modo:

http://[indirizzo server]/[contesto]/[url pattern]

Le Classi Base per le Servlet

- Alla base dell'implementazione di una servlet c'è l'interfaccia **Servlet**, che è implementata da una serie di classi base come **HttpServlet**. Tutte le servlet che creeremo saranno derivate (*extends*) da quest'ultima.
- Le altre due classi base per la creazione di una servlet sono **ServletRequest** e **ServletResponse**.
- Un'istanza di **ServletRequest** viene passata dal contesto alla servlet quando questa viene invocata, e contiene tutte le informazioni inerenti la richiesta effettuata dal client: queste comprendono, ad esempio, i parametri GET e POST inviati dal client, le variabili di ambiente del server, gli *header* e il *payload* della richiesta HTTP.
- Un'istanza di **ServletResponse** viene passata alla servlet quando le si richiede di restituire del contenuto da inviare al client. I metodi di questa classe permettono di scrivere su uno *stream* che verrà poi indirizzato al client, modificare gli *header* della risposta HTTP, ecc.
- Altre classi comprese nelle servlet API, di cui non ci occuperemo qui, permettono ad esempio di gestore le sessioni in maniera automatica.

Il Ciclo di Vita di una Servlet

- Il ciclo di vita di una servlet è scandito da una serie di chiamate, effettuate dal container, a particolari metodi dell'interfaccia Servlet.
  1. **Inizializzazione**. Quando il container carica la servlet, chiama il suo metodo *init*. Tipicamente questo metodo viene usato per stabilire connessioni a database e preparare il contesto per le richieste successive. A seconda dell'impostazione del contesto e/o del contenitore, la servlet può essere caricata immediatamente all'avvio del server, ad ogni richiesta, ecc.
  2. **Servizio**. Le richieste dei client vengono gestite dal container tramite chiamate al metodo *service*. Richieste concorrenti corrispondono a esecuzioni di questo metodo in thread distinti. L'implementazione dovrebbe essere quindi thread-safe. Il metodo service riceve le richieste dell'utente sotto forma di una ServletRequest e invia la risposta tramite una ServletResponse.
  3. **Finalizzazione**. Quando il container decide di rimuovere la servlet, chiama il suo metodo *destroy*. Quest'ultimo viene solitamente utilizzato per chiudere connessioni a database, o scaricare altre risorse persistenti attivate dal metodo init.
- La classe **HttpServlet** specializza questo sistema per la comunicazione HTTP, fornendo in particolare due metodi: *doGet* e *doPost*, corrispondenti alle due request più comuni in HTTP. Il metodo service delle classe HTTPServlet provvede automaticamente a smistare le richieste al metodo opportuno.

Scrivere una Classe Servlet

- Per scrivere una semplice servlet è necessario creare una classe che estende **javax.servlet.http.HttpServlet**.
- La logica di funzionamento della servlet va codificata nei metodi corrispondenti al verbo HTTP cui devono rispondere.
  - **doGet** viene chiamata in risposta a richieste GET e HEAD
  - **doPost** viene chiamata in risposta a richieste POST
  - **doPut** viene chiamata in risposta a richieste PUT
  - **doDelete** viene chiamata in risposta a richieste DELETE
- Tutti questi metodi hanno la stessa *signature*: prendono cioè come argomenti la coppia *(HttpServletRequest, HttpServletResponse)* e restituiscono *void*.
  - (i) Le classi **HttpServletRequest** e **HttpServletResponse** sono specializzazioni di ServletRequest e ServletResponse specifiche per il protocollo HTTP.
- (i) La classe HttpServlet fornisce un'implementazione di default di tutti i metodi appena descritti, che si limita a generare un errore 400 (BAD REQUEST)
- (i) La classe servlet contiene altri metodi utili, come *getServletContext*, tramite il quale si possono leggere molte informazioni sul contesto di esecuzione della servlet stessa.
- (!) Per compilare una servlet, è necessario che nel **CLASSPATH** sia inclusa la libreria che definsce il package **javax.servlet**. Una copia di questa libreria, chiamata *servlet-api.jar*, è presente nella directory common/lib di Tomcat.

Scrivere una Classe Servlet

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;

public class class1 extends HttpServlet {

  public void doGet(HttpServletRequest in,
  HttpServletResponse out) {
  //…
  }
}
```

- Una servlet di base estende la classe javax.servlet.http.HttpServlet
- Per gestire le richieste HTTP, si sovrascrivono opportunamente i metodi corrispondenti: in questo esempio, il metodo doGet verrà chiamato per gestire le richieste HTTP GET.

Fornire Informazioni sulla Servlet

- E' possibile, anche se non richiesto, fornire delle informazioni sulla servlet che possono essere utilizzate dal container.
- Le informazioni potrebbero ad esempio includere l'autore e la versione della servlet stessa.
- A questo scopo è sufficiente sovrascrivere il metodo *getServletInfo()* dell'interfaccia Servlet, che nella sua implementazione di default in HttpServlet si limita a resitutire *null*.
- Il metodo non prende alcun argomento e deve restituire una stringa.

Fornire Informazioni sulla Servlet

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;

public class class1 extends HttpServlet {

  public String getServletInfo() {
    return "Servlet di esempio, versione 1.0";
  }

  public void doGet(HttpServletRequest in,
  HttpServletResponse out) {
  //…
  }
}
```

- La stringa restituita da **getServletInfo** verrà usata dal container per fornire informazioni sulla servlet.

Inizializzare e Finalizzare la Servlet

- L'inizializzazione di una servlet si effettua nel suo metodo **init**, che ha come parametro un oggetto *ServletConfig*.
  - La prima cosa che il metodo deve fare è chiamare **super.init()** passandogli il suo parametro ServletConfig.
  - Successivamente è possibile eseguire tutto il codice di inizializzazione necessario, eventualmente impostando campi della classe con dati che andranno utilizzati dai metodi di servizio.
  - Se la servlet è stata attivata con dei parametri di inizializzazione esterni (che vengono specificati in una maniera dipendente dal container), è possibile leggerli tramite il metodo *getInitParameter* di HttpServlet. Questo metodo prende come argomento il nome del parametro e restituisce una stringa.
  - Se l'inizializzazione incontra dei problemi, è possibile generare un'eccezione (*ServletException*) per segnalarlo al contanier.
- La finalizzazione di una servlet si effettua nel suo metodo **destroy**.
  - E' necessario sovrascrivere questo metodo solo se esistono effettivamente operazioni da fare prima della distruzione della servlet.

Inizializzare e Finalizzare la Servlet

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
I
public class class1 extends HttpServlet {
  private int parameter1;

  public void init(ServletConfig c)
   throws ServletException {
   super.init(c);
   parameter1 = 1;
  }

  public String getServletInfo() {
    return "Servlet di esempio, versione 1.0";
  }
  public void doGet(HttpServletRequest in,
  HttpServletResponse out) {
  //…
  }
}
```

- Il metodo **init**, dopo aver chiamato lo stesso metodo della classe superiore, può procedere con l'inizializzazione della servlet.
- Se ci sono problemi di inizializzazione, viene generata una ServletException.

Scrivere la Risposta

- L'oggetto **HttpServletResponse** fornito a tutti i metodi di servizio permette di costruire la risposta da inviare al client.
  - Il metodo *setContentType(String)* permette di dichiarare il tipo restituito (ad esempio "text/xml").
  - Il metodo *setHeader(String,String)* permette di aggiungere *headers* alla richiesta.
  - Il metodo *sendRedirect(String)* permette di ridirezionare il browser verso una nuova URL.
  - I metodi *getWriter()* e *getOutputStream()* permettono di aprire un canale, testuale o binario, su cui scrivere il contenuto della risposta. Vanno chiamati dopo aver (eventualmente) impostato il *content type* e gli altri *headers*.
  - Altri metodi (che non tratteremo in questa sede) permettono di gestire, ad esempio, i cookie.

Scrivere la Risposta

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
  //…
  public void doGet(HttpServletRequest in,
  HttpServletResponse out) {
    out.setContentType("text/xml");
    try {
      Writer w = out.getWriter();
      w.write("pippo");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

- Nel caso comune in cui si debba restituire dell'XHTML al client, è sufficiente prelevare il **Writer** della response tramite il metodo *getWriter()* e scrivervi sopra tutto il testo da restituire dal browser.
- Qualunque altro parametro della risposta (in questo caso, il tipo) deve essere impostato *prima* di richiedere il canale di output.

Leggere la Richiesta

- L'oggetto **HttpServletRequest** fornito a tutti i metodi di servizio contiene tutte le informazioni sulla richiesta del client.
  - Il metodo *getParameter(String)* restituisce il valore di un parametro inserito nella richiesta HTTP. Se il parametro può avere più di un valore, utilizzare *getParameterValues(String)*, che restituisce *l'array* di tutti i valori assegnati al parametro dato.
    - **Questi metodi non sono validi nel caso in cui si utilizzi la codifica *multipart/form-data* (upload di files)!**
  - Per le richieste GET, il metodo *getQueryString()* permette di leggere l'intera stringa di query (non interpretata).
  - Per le richieste POST, i metodi *getReader()* e *getInputStream()* permettono di aprire uno stream (testuale o binario) e leggere direttamente il payload della richiesta.
  - Il metodo *getHeader()* permette di leggere il contenuto degli header della richiesta HTTP.
  - Il metodo *getSession()* viene utilizzato per la gestione delle sessioni (si veda più avanti).
  - Altri metodi (che non tratteremo in questa sede) permettono di gestire autenticazione, cookie, ecc.
  - I metodi ereditati dalla classe **ServletRequest**, infine, permettono di leggere informazioni come l'indirizzo della richiesta (*getRemoteAddr*) o il protocollo (*getProtocol*).

Leggere la Richiesta

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in,
 HttpServletResponse out) {
  String p1 = in.getParameter("p1");
 }
 public void doPost(HttpServletRequest in,
 HttpServletResponse out) {
  try {
   Reader r = in.getReader();
   //lettura del payload raw da r…
  } catch(Exception e) {
   e.printStackTrace();
  }
 }
}
```

- I metodi doGet e doPost possono leggere i parametri interpretati della richiesta tramite **getParameter()** e **getParameterValues()**.
- doGet può leggere la stringa di query in maniera raw chiamando **getQueryString()**.
- doPost può leggere il payload del messaggio in maniera raw tramite gli stream restituiti da **getReader()** (testuale) e **getInputStream()** (binario).

Le Sessioni

- Il concetto di sessione è utilizzato ampiamente nella programmazione lato server. La sessione permette di **associare delle informazioni di stato alle richieste di un utente**, permettendo in pratica di aggirare la caratteristica *stateless* del protocollo HTTP.
- Tipicamente, per gestire le sessioni si utilizza un **identificatore unico** (*session identifier*) che viene assegnato all'utente al suo ingresso nell'applicazione web (ad esempio quando esegue la login) e trasmesso insieme ad ogni successiva richiesta http, per poi venire invalidato quando si effettua il logout o dopo un certo periodo di inattività.
- Il session identifier deve essere un numero ragionevolmente unico, e di solito viene generato con algoritmi random basati sulla data/ora corrente.
- Per trasmettere il session identifier con ogni richiesta, si utilizzano di solito due approcci:
  - **Cookies**: i cookie sono stringhe inviate dal server al browser. I browser ritrasmettono automaticamente al server i cookie di sua competenza insieme ad ogni request, quindi non è necessario alcun accorgimento lato client. I cosiddetti *cookie di sessione* vengono cancellati dal browser alla chiusura, e contengono il session identifier.
    I cookie sono la soluzione più semplice e adottata, ma risentono delle politiche di sicurezza del browser (ad esempio la disattivazione dei cookies).
  - **URL rewriting**: si tratta di inserire il session identifier all'interno delle URL, come parte del path o, più comunemente, come parametro GET.
    Si tratta della soluzione più generica e compatibile, che però richiede la generazione dinamica di tutte le pagine (ogni link deve essere generato inserendovi il session identifier corrente).

Gestire le Sessioni con i Cookie

- Nelle servlet, creare ed utilizzare una sessione usando i cookies è molto semplice. La sessione è gestita da oggetti **HttpSession**. Le *variabili di sessione* sono semplici stringhe a cui vengono associati generici valori di tipo Object.
- Per prima cosa, si preleva un riferimento all'oggetto sessione richiedendolo alla **HttpServletRequest** tramite il metodo *getSession(boolean)*.
  - (i) Se il parametro di getSession è true, una nuova sessione verrà creata nel caso non ce ne sia una valida già attiva. In caso contrario, la funzione potrebbe restituire null.
  - (!) la chiamata a questo metodo può modificare gli headers della risposta, per cui va eseguita prima di iniziare l'output della risposta stessa.
- I metodi di HttpSession permettono poi di utilizzare la sessione:
  - *isNew()* restituisce true se la sessione è stata appena creata: in questo caso di solito vengono inizializzate le sue variabili di stato.
  - *getAttribute(String)* restituisce l'oggetto associato al nome dato all'interno della sessione.
  - *setAttribute(String,Object)* associa al nome specificato l'oggetto passato come secondo argomento. In pratica, crea o aggiorna la variabile di stato data dal primo argomento usando il valore contenuto nel secondo argomento.
  - *removeAttribute(String)* rimuove la variabile di stato indicata.
  - *invalidate()* chiude la sessione ed elimina tutte le informazioni di stato associate.

Gestire le Sessioni con i Cookie

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
 //…
 public void doGet(HttpServletRequest in, HttpServletResponse
 out) {
   HttpSession s = in.getSession(true);
   if (s.isNew()) s.setAttribute("pagine",new Integer(1));
   int a = ((Integer)s.getAttribute("pagine")).intValue();
   s.setAttribute("pagine", new Integer(a+1));
   try {
     Writer w = out.getWriter();
     w.write("pagine visitate in questa sessione: "+a);
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
}
```

- Ad ogni richiesta GET alla servlet, se una sessione non è attiva, ne viene creata una e al suo interno viene inserita una variabile denominata "pagine" inizializzata a 1 (notare l'uso della classe **Integer**)
- Il numero di pagine visitato durante la sessione viene quindi incrementato e stampato in output.

Gestire le Sessioni con i Cookie

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class class1 extends HttpServlet {
  //…
  public void doGet(HttpServletRequest in, HttpServletResponse
  out) {
    HttpSession s = in.getSession(true);
    s.setAttribute("user", in.getParameter("username"));
  }
}
```

- Questa semplice servlet mostra come salvare all'interno della sessione il valore del parametro "username" prelevato dalla request (probabilmente proveniente da una form).
- Si tratta di una tipica operazione effettuata per concludere una procedura di login, tenendo traccia dell'utente attivato nella sessione corrente.

Gestire le Sessioni con le URL

- Le servlet dispongono anche di un sistema semiautomatico per gestire le sessioni tramite la riscrittura delle URL.
- In pratica, oltre al codice di gestione/creazione/uso della sessione visto precedentemente, è necessario far modificare ogni indirizzo internet che punta a una risorsa della nostra applicazione dal metodo *encodeUrl(String)* dell'oggetto **HttpServletResponse**.
- (i) Il metodo encodeURL determina se è necessario inserire il session identifier nella URL: nel caso in cui il metodo dei cookie sia utilizzabile, la URL non viene alterata.

Gestire le Sessioni con le URL

Esempio

```
package org.iw.project;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
I
public class class1 extends HttpServlet {
  //…
  public void doGet(HttpServletRequest in,
  HttpServletResponse out) {
    HttpSession s = in.getSession(true);
    try {
      Writer w = out.getWriter();
      w.write(out.encodeUrl(in.getServletPath()));
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
}
```

- In questo esempio, viene creata (se necessario) una sessione e sulla pagina viene stampata la URL della servlet corrente riscritta da **encodeURL** per includere il session identifier.
- *Se il browser supporta i cookies, la URL non verrà modificata.*

Java e i DBMS: il JDBC

- Una delle operazioni più comuni in un'applicazione web è la **gestione di dati immagazzinati in un database**.
- L'accesso ai dati in Java si effettua usando il package **JDBC** (*Java DataBase Connectivity*). Un uso tipico delle classi JDBC prevede i seguenti passi:
  - Si rende disponibile il **driver JDBC** per il DBMS in uso nel classpath di Java.
  - Si carica il driver facendo riferimento alla classe che lo implementa tramite metodo *Class.forName*
  - Si procede con la creazione dell'oggetto **Connenction** tramite il metodo statico *getConnetion* della classe **DriverManager**.
    - I tre parametri del metodo sono **il nome utente e password** da usare per l'accesso al DBMS e la **stringa di connessione JDBC**, che specifica l'indirizzo del DBMS e il database da selezionare: Questa stringa ha un formato che varia a seconda del DBMS in uso.
  - Si crea un oggetto **Statement** sulla connessione, usando il metodo *createStatement*.
  - Si invia la query SQL, sotto forma di stringa, al DBMS tramite lo **Statement** creato e il suo metodo *executeQuery*.
    - L'oggetto restituito, di tipo **ResultSet**, permette di navigare tra i risultati della query.
    - Se invece si desidera inviare una query che non restituisce risultati, ad esempio di inserimento o aggiornamento, si usa il metodo *executeUpdate*. In questo caso, il valore restituito è un intero che rappresenta il numero di record interessati dalla query stessa.
  - Una volta prelevati i risultati, si libera lo spazio a loro riservato chiamando il metodo *close* dello **Statement.**
  - Infine, terminato l'uso della base di dati, si può chiudere la connessione corrispondente chiamando il metodo *close* della **Connection.**
- Tutte le istruzioni JDBC, in caso di errore, sollevano eccezioni derivate da **SQLException**.

Java e i DBMS: il JDBC

Esempio

```
import java.sql.*;

Class.forName ("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/webdb","user", "pass");

Statement stmt1 = con.createStatement();
ResultSet rs = stmt1.executeQuery(
  "SELECT * FROM test");
…
rs.close();
stmt1.close();

Statement stmt2 = con.createStatement();
int rc = stmt.executeUpdate("DELETE FROM test");
stmt2.close();

con.close();
```

- In questo esempio, si crea una connessione a un database *MySQL*.
- La classe del driver JDBC, scaricato dal sito del produttore del DBMS, è *com.mysql.jdbc.Driver*
- La stringa di connessione specifica il tipo di DBMS (*mysql*) il punto di ascolto del DBMS (*localhost*), e il database da selezionare (*webdb*).
- Alla connessione vengono anche passati la username e la password dell'utente con cui autenticarsi nel DBMS.
- Viene eseguita prima una query di selezione tramite *executeQuery* e poi una di cancellamento tramite *executeUpdate*;

Java e i DBMS: il JDBC

I ResultSet

- Tramite il **ResultSet** restituito dal metodo *executeQuery* è possibile leggere le colonne di ciascun record restituito da una query di selezione.
- I record devono essere letti uno alla volta: in ogni momento, il **ResultSet** punta (tramite un *cursore*) a uno dei record restituiti (*record corrente*).
- I valori dei vari campi del record corrente possono essere letti tramite i metodi *getX(nome_colonna)*, dove X è il **tipo base Java** per il dato da estrarre (ad esempio *getString*, *getInt*, …) e nome_colonna è il nome del campo del record da leggere.
- Per spostare il cursore del **RecordSet** al record successivo, si usa il metodo *next*. Il metodo restituisce *false* quando i record sono esauriti.

Java e i DBMS: il JDBC

I ResultSet - Esempio

```
import java.sql.*;

Class.forName ("com.mysql.jdbc.Driver");

Connection con = DriverManager.getConnection(
  "jdbc:mysql://localhost/webdb","user", "pass");

Statement stmt1 = con.createStatement();

ResultSet rs = stmt1.executeQuery(
  "SELECT * FROM test");

while (rs.next()) {
  System.out.println("nome = "+ rs.getString("nome"));
}

rs.close();
stmt1.close();
con.close();
```

- In questo esempio, utilizziamo un ciclo *while* per iterare tra i risultati di una query di selezione.
- Per ogni record viene stampato il valore del campo *nome*, di tipo stringa.

Java e i DBMS: il JDBC

Limiti del metodo d'uso "standard"

- In un'applicazione web *data intensive*, in cui gli accessi al database sono molti e spesso concorrenti (più utenti connessi all'applicazione contemporaneamente), il "normale" pattern d'uso de JDBC presenta notevoli problemi.
- Infatti, **l'apertura di una connessione al database è di solito un'operazione costosa**, in quanto come si è visto richiede
  - Caricamento driver
  - Connessione al DBMS server
  - Autenticazione
- E' necessario limitare l'overhead dovuto a queste operazioni, per rendere la web application più veloce possibile.

Il Connection Pooling

- Il connection pooling è una tecnica che permette di **snellire le procedure di apertura delle connessioni JDBC** utilizzando una **cache di connessioni**, chiamata *connection pool*.
  - Il pool mantiene una serie di connessioni al database **già aperte e inizializzate**.
  - Quando l'applicazione vuole connettersi, **preleva dal pool una connessione già pronta**, sulla quale può operare direttamente.
  - Quando l'applicazione chiude la connessione, **questa in realtà rimane aperta e viene reinserita nel pool**, in attesa di essere riutilizzata per altre richieste.
  - Il pool viene inizialmente riempito con un certo numero di connessioni pronte. Tuttavia, se il pool si svuota (cioè tutte le sue connessioni sono in uso) e c'è richiesta di nuove connessioni, queste vengono create automaticamente, allargando il pool.
  - Le connessioni lasciate inutilizzate nel pool per troppo tempo possono venir chiuse automaticamente per liberate le corrispondenti risorse del DBMS.

Il Connection Pooling

Supporto negli application server

- Il supporto al connection pooling, incorporato nelle più recenti versioni del JDBC, deve ovviamente avere supporto da parte del software (proprio come con i driver JDBC).
- Tutti i più diffusi application server forniscono un'implementazione del connection pooling, ed esistono anche librerie esterne utilizzabili per aggiungere questo supporto al di fuori degli application server.
- Attenzione: **ogni application server fornisce sistemi proprietari per configurare il connection pooling.** Vedremo in particolare come si utilizza il connection pooling su Tomcat.

Il Connection Pooling

Uso con Tomcat

- Per configurare il connection pooling in Tomcat, si procede come segue:
  - Si **configura la connessione al database nel contesto** della web application, agendo sul server.xml (globale) o, meglio, sul context.xml (configurazione specifica dell'applicazione).
  - Si inserisce nel deployment descriptor (web.xml) un riferimento alla connessione, che diventa così **una risorsa dell'applicazione di tipo DataSource**.
  - Nel codice, si preleva un riferimento al DataSource utilizzando lo standard *Java Naming and Directory Interface (JNDI)*.
  - Si crea una normale connessione JDBC attraverso il DataSource.
  - Si chiude la connessione al termine del suo uso.

Il Connection Pooling

Configurazione del contesto dell'applicazione (context.xml) per Tomcat

```
<Context path="/Esempio_Database_Pooling">
<Resource
      name="jdbc/webdb2"
      type="javax.sql.DataSource"
      auth="Container"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost/webdb"
      username="website"
      password="webpass"
      maxActive="10"
      maxIdle="5"
      maxWait="10000"
   />
</Context>
```

- La connessione è configurata all'interno di un elemento **Resource**, che ne contiene tutte le caratteristiche, compreso il driver, la username, la password e la stringa di connessione.
- Il *type* deve essere indicato come **javax.sql.DataSource**. L'attributo *auth* va impostato su "container".
- Gli attributi *maxActive*, *maxIdle* e *maxWait* servono a dimensionare il pool, indicando
  - Quante connessioni al massimo il pool dovrà contenere
  - Quante connessioni inutilizzate sono ammesse nel pool in ogni momento
  - Quanto tempo attendere perché una nuova connessione diventi disponibile
- **Attenzione: il driver indicato andrà copiato nella directory lib di Tomcat. Se lo si copia semplicemente nella WEB-INF/lib dell'applicazione (come parte del deployment), esso sarà invisibile al class loader usato per il pooling!**

Il Connection Pooling

Configurazione del deployment descriptor (web.xml)

```
<web-app version="2.5"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
  instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
...
<resource-ref>
        <res-ref-name>jdbc/webdb2</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
...
</web-app>
```

- Nel deployment descriptor si inserisce un riferimento alla risorsa JNDI con un elemento **resource-ref**.
- Gli attributi *res-type* e *res-auth* rispecchiano quelli *type* e *res* scritti nella definizione della **Resource**
- *res-sharing-scope* dovrebbe generalmente essere posto a "Shareable"
- L'attributo *res-ref-name* indica il nome JNDI utilizzato per la risorsa all'interno del codice. Le risorse di tipo DataSource hanno convenzionalmente un nome che inizia con "jdbc/".

Il Connection Pooling

Uso nel codice Java

```
try {
  //preleviamo un riferimento al naming context
  InitialContext ctx = new InitialContext();
  //e otteniamo un riferimento al DataSource
  DataSource ds =
 (DataSource) ctx.lookup("java:comp/env/jdbc/webdb2");
  //connessione al database locale
  connection = ds.getConnection();
  //…usiamo la connessione…
} catch (NamingException ex) {
  //eccezione sollevata nel caso la risorsa richiesta non
  esista
} catch (SQLException ex) {
  //eccezione standard per le operazione JDBC
} finally {
  //alla fine le connessione DEVE essere chiusa!
  try { connection.close(); } catch (SQLException ex) {}
}
```

- Si crea un *JNDI naming context* (**InitialContext**)
- Si ricerca nel contesto la risorsa necessaria. Notare che al nome configurato nel deployment descriptor va aggiunto sempre il prefisso "java:comp/env/". Può essere utile usare un parametro della web application per configurare questo nome al di fuori del codice.
- Si fa un cast dell'oggetto restituito sul tipo effettivo della risorsa (**DataSource**)
- Si crea la **Connection** JDBC usando il metodo **getConnection()** del DataSource.
- Dopo aver operato sulla connessione, la si chiude normalmente, determinandone il reinserimento nel pool. **Attenzione: se non si chiude, la connessione non rientrerà nel pool!**

Il Connection Pooling

Resource Injection

```
class DatabaseService {
 @Resource(name ="java:comp/env/jdbc/webdb2")
 private DataSource ds;
 //…
 public void dbMethod() {
  try {
   connection = ds.getConnection();
   //…usiamo la connessione…
  } catch (SQLException ex) {
   //eccezione standard per le operazione JDBC
  } finally {
    //alla fine le connessione DEVE essere chiusa!
    try { connection.close(); } catch (SQLException ex) {}
  }
 }
}
```

- La **Resource injection** permette di evitare il complesso sistema di lookup delle risorse, e fare in modo che sia Java stesso a iniettare un riferimento al *DataSource* in una variabile utente.
- L'injection si effettua di solito su un campo della classe, che **deve essere del tipo corretto** (*DataSource*).
- Basta far precedere la dichiarazione del campo dall'annotazione **@Resource**, con parametro ***name*** uguale al nome della risorsa da prelevare.

ServletContextListener

- Un *context listener*, è un elemento utile in generale per eseguire ogni operazione di "inizializzazione globale" (non specifica per una particolare servlet) della web application.
- Oggetti che implementano l'interfaccia **ServletContextListener** possono essere associati all'applicazione dichiarandoli opportunamente nel deployment descriptor (web.xml).
- I due metodi **contextInitialized** e **contextDestroyed** di questi oggetti vengono chiamati rispettivamente all'avvio e alla chiusura della web application.
- I metodi possono, tra l'altro, leggere modificare il **ServletContext** che verrà poi passato a tutte le servlet in fase di esecuzione.

ServletContextListener

Esempio

```
public class ContextInitializer implements
  ServletContextListener {

  public void contextInitialized(ServletContextEvent sce) {
    //inizializziamo qualche variabile globale (di contesto)..
    sce.getServletContext().setAttribute("appID", 1);
  }

  public void contextDestroyed(ServletContextEvent sce)
  {

  }
}
```

```
<listener>
  <listener-class>
    it.univaq.f4i.iw.examples.ContextInitializer
  </listener-class>
</listener>
```

- Questo context listener esegue inizializza alcune variabili di contesto all'avvio della web application, inserendole come attributi nel ServletContext.
- Le singole servlet potranno prelevarlo semplicemente con un'istruzione del tipo **getServletContext().getAttribute("appID")**
- Perché il listener sia elaborato, è sufficiente **aggiungere il frammento in basso, che ne specifica la classe, nel web.xml.**

Filter

- Un *filtro* è un elemento delle applicazioni web che permette di **modificare «al volo» le informazioni in input** (*HTTPServletRequest*) **o in output** (*HTTPServletResponse*) di una o più servlet.
- Oggetti che implementano l'interfaccia **Filter** possono essere associati all'applicazione dichiarandoli opportunamente nel deployment descriptor (web.xml).
- I metodi **init**, **destroy** di questi oggetti vengono chiamati rispettivamente all'avvio e alla chiusura della web application.
- Il metodo **doFilter** viene invece invocato **a ogni richiesta** sulle servlet per le quali il filtro è configurato, e riceve gli oggetti request, response e una *FilterChain*.
  - E' possibile **sostituire gli oggetti request e/o response** con implementazioni speciali per poter controllare l'input/output.
  - E' necessario **chiamare il metodo doFilter della FilterChain.**

Filter

Esempio

```
public class EmailObfuscatorFilter implements Filter {
private FilterConfig config = null;
public void init(FilterConfig filterConfig) throws
  ServletException {
 this.config = filterConfig;
}
public void doFilter(ServletRequest request,
  ServletResponse response, FilterChain chain) throws
  IOException, ServletException {
 chain.doFilter(request, response);
public void destroy() {
config = null;
}
```

```
<filter>
 <filter-name>emailfilter</filter-name>
 <filter-class>EmailObfuscatorFilter</filter-class>
</filter>
<filter-mapping>
 <filter-name>emailfilter</filter-name>
 <url-pattern>*</url-pattern>
 <dispatcher>REQUEST</dispatcher>
</filter-mapping>
```

- Questo filtro non fa assolutamente nulla: si limita a chiamare il *doFilter* sulla *FilterChain*.
- Perché il filtro sia elaborato, è sufficiente **aggiungere il frammento in basso, che ne specifica la classe e gli url pattern associati, nel web.xml.**

Riferimenti

- **Servlet API**
  https://docs.oracle.com/javaee/7/api/javax/servlet/package-summary.html
- **Servlet Tutorial**
  https://docs.oracle.com/javaee/7/tutorial/servlets.htm
- **JDBC Tutorial**
  http://docs.oracle.com/javase/tutorial/jdbc