Università degli Studi di Padova

FACOLTÀ DI INGEGNERIA

Applicazioni web con tecnologia Java

Relazione finale di Tirocinio

Laureando: Luca Falcon
Relatore: Prof. G.Clemente

Dipartimento di Ingegneria dell'Informazione
Anno Accademico 2009-2010

Indice

INDICE III

INTRODUZIONE V

1 JEE 1

1.1 La piattaforma di sviluppo JEE 1
1.2 L'Application Server 3
1.3 Servlet 4
1.4 JSP (JavaServer Pages) 7
1.5 JDBC (Java DataBase Connectivity) 9
1.6 JNDI (Java Naming and Directory Interfaces) 11
1.7 JavaBean ed Enterprise JavaBean (EJB) 12
1.8 Struttura delle directory 15

2 APACHE STRUTS 19

2.1 Pattern 19
2.2 Il paradigma Model 2: il pattern architetturale MVC 20
2.3 Introduzione al framework 22
2.4 ActionServlet e RequestProcessor 23
2.5 struts-config.xml 25
2.6 Action, ActionFordward, ActionMessage, ActionForm 27
2.7 Localizzazione 30
2.8 Tag Library 31
2.9 Alternative 34

3 PROGETTO "STARTUP" 37

3.1 Requisiti 37
3.2 Progettazione 38
3.3 Implementazione 41
3.4 Deployment 49

CONCLUSIONI 55

BIBLIOGRAFIA 57

RINGRAZIAMENTI 59

Introduzione

Con questa tesi si è voluto esplorare e approfondire gli aspetti fondamentali dello sviluppo di applicazioni software orientate al web mediante la tecnologia Java. L'attività di tirocinio si è svolta presso l'azienda Aive S.p.A. di Marcon (VE) con la supervisione dell'ing. M. Baraldi. Ha avuto come oggetto la progettazione e l'implementazione di un'applicazione web, all'interno di un progetto denominato "Startup". Questa attività, fin da subito, ha avuto come obbiettivo quello di farmi acquisire sufficienti conoscenze riguardanti l'architettura JEE, paradigmi di programmazione e strumenti utilizzati nell'ambito delle web application. Tali conoscenze mi hanno permesso di implementare completamente un'applicazione accessibile tramite browser web, che consentisse ai propri utenti di inviare delle comunicazioni di consuntivazione ore, aggiornando una base dati.

La piattaforma standard di sviluppo JEE è un'architettura piuttosto complessa ed è composta da un elevato numero di componenti, ognuna delle quali offre uno specifico servizio. Per questo motivo, si sono approfondite solo una parte delle tecnologie di questa piattaforma, necessarie per lo sviluppo del progetto. Le componenti dello standard trattate in questa tesi sono Servlet, JSP, JNDI, JDBC e JavaBeans.

L'azienda ospitante, nelle proprie scelte progettuali, è piuttosto consona nell'utilizzo di alcuni, classici, paradigmi di progettazione di applicazioni web. Questi pattern essendo ormai molto diffusi, sono implementati da numerosi framework che aiutano progettisti e sviluppatori ed apportando vari benefici. Tra i framework disponibili si è valutato che Apache Struts 1 si adattasse bene allo specifico problema in esame. A tale proposito, nel secondo capitolo di questa relazione è descritta la logica di funzionamento e le specifiche tecniche.

Infine, nel terzo capitolo, sono discussi i dettagli progettuali e implementativi che hanno riguardato il progetto, dando un aspetto pratico alle nozioni teoriche apprese durante l'attività di tirocinio.

1 JEE

In questo capitolo verrà fornita un'introduzione alle tecnologie dello standard JEE che sono alla base dell'applicazione sviluppata durante il tirocinio. Il progetto "Startup" proposto dall'azienda ha richiesto un approfondimento del funzionamento delle specifiche Java Servlet, JSP, JDBC, JNDI, JavaBean ed alcuni elementi di base riguardanti gli Application Server.

1.1 La piattaforma di sviluppo JEE

JEE (Java Enterprise Edition), nella fattispecie, rappresenta una base solida per lo sviluppo di un'applicazione, dalla più semplice alla più complessa realtà enterprise aziendale. La piattaforma JEE, proposta da Sun Microsystem nella prima versione nel dicembre del 1999, fornisce un'ampia serie di specifiche e interfacce, Application Programming Interface (API), che definiscono le componenti tecnologiche in cui è strutturata e la modalità di interazione tra di esse. Nota anche come *"J2EE"* fino alla versione *1.4*, Sun ha deciso un rebrand con il nome *"JEE"* dalla versione *1.5*. Alcune delle specifiche JEE sono comprese dalle API J2SE (Java 2 Standard Edition), come ad esempio JDBC e RMI.

Un'applicazione che implementa le specifiche JEE è un'applicazione distribuita multi-tier, schematizzabile in quattro layer (fig. 1.1):

1) il *Client Tier*, il quale presenta all'utente finale i risultati dell'elaborazione del server. Spesso è rappresentato da un browser web, ma può talvolta essere costituito da client specifici, in grado di interagire direttamente con il Business Tier o il Data Tier;
2) il *Web Tier,* in funzione nel server JEE (o semplicemente in un Web Server), comprende una serie di componenti che riguardano il lato front-end dell'applicazione, mediando richieste e risposte del protocollo HTTP tra client e server;

3) il *Business Tier,* in funzione nel server JEE, dove viene implementata la logica di business dell'applicazione, organizzando i dati e l'accesso ad essi ed interagendo, ad esempio, con un DBMS. Nelle applicazioni più semplici viene accorpato al Web Tier.
4) L'*EIS Tier* è spesso rappresentato da un DBMS (Database Management System) o più in generale da un EIS (Enterprise Information System).

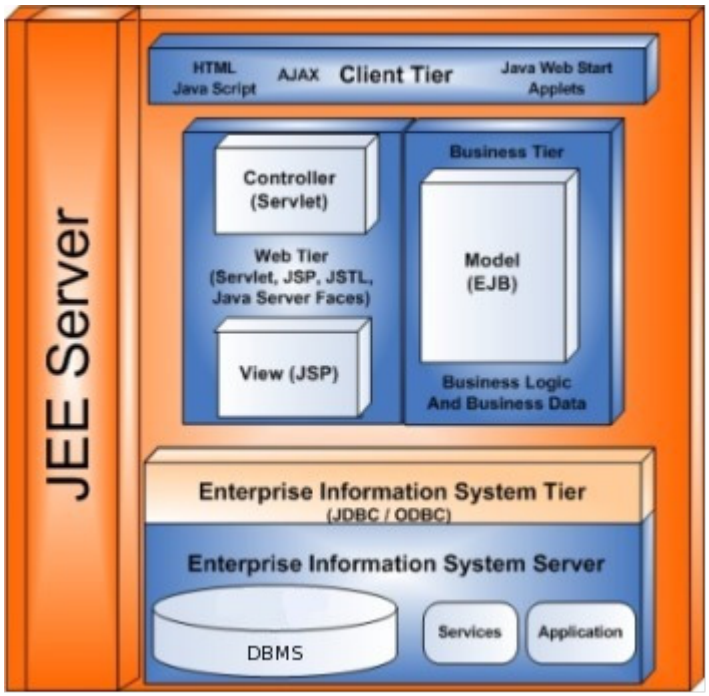

Figura 1.1 Architettura di una generica applicazione JEE

I principali vantaggi derivanti dall'uso delle specifiche JEE per lo sviluppo di applicazioni web sono fondamentalmente quattro:

- Scalabilità: è possibile aumentare le funzionalità di un software in continua evoluzione, grazie anche alla peculiare proprietà di distribuibilità della tecnologia Java.
- Portabilità: esigenza fondamentale dell'attività di sviluppo software, permette di utilizzare una stessa applicazione JEE in Application Server diversi purché questi implementino le specifiche JEE.
- Efficienza: una strutturazione così intensa facilita la gestione di progetti anche molto complessi. Inoltre, grazie all'utilizzo del multi-threading di Java è possibile ottenere elevate performance per l'interazione tra Client e Server
- Sicurezza: assicurata dall'elevata stratificazione dell'architettura e dalle proprietà intrinseche della tecnologia Java.

Queste proprietà rappresentano le esigenze dell'azienda che lavora con ambienti composti da tipologie di risorse differenti e distribuite, assicurando una buona robu-

stezza software. Inoltre, attraverso il modello proposto, si rende facile l'accesso ai dati e la loro rappresentazione in diverse forme (un browser web, un applet, un dispositivo mobile, un sistema esterno, ecc).

1.2 L'Application Server

Tra i più importanti elementi di un'applicazione web, vi è l'*Application Server,* il quale, implementando le specifiche JEE, rappresenta uno strato software che fornisce i servizi e le interfacce per la comunicazione tra tutte le componenti e permette all'applicazione di funzionare. Molti degli Application Server presenti sul mercato, implementano sia il livello Web-Tier che il Business-Tier. Le applicazioni più semplici, quelle cioè che non possiedono una vera e propria logica di business, dovrebbero utilizzare Application Server con solo la proprietà di *Web-Container* (che ha la funzione principale di presentazione dei dati): sono chiamati Web Server e sono consigliati in queste circostanze in quanto, le risorse richieste, sono di molto inferiori. Tra i principali componenti Java gestiti da un Web-Container vi sono *Servlet, JSP (Java Server Pages), JDBC (Java DataBase Connectivity)* e *JNDI (Java Naming and Directory Interface)*.

Un Application Server può fornire, inoltre, un ambiente detto *EJB-Container (Enterprise Java Beans Container)* che contiene la logica di business dell'applicazione. Elementi fondamentali di questo ambiente sono gli Enterprise JavaBeans, le cui specifiche sono state introdotte per la prima volta da IBM nel 1997 e successivamente accorpate nelle API standard di Sun Microsystem. Il compito di un EJB-container è quello di gestire sicurezza, dettagli di persistenza e integrità delle transazioni, lasciando agli sviluppatori maggior concentrazione riguardo i problemi *core business*. La scelta di quale Application Server utilizzare a sostegno delle proprie applicazioni è influenzata da numerosi fattori tra cui:

- Tecnologie preesistenti: alcuni case software dei principali DBMS commerciali propongono i propri Application Server. Optare per tale soluzione, solitamente, facilita l'integrazione tra le due componenti, aumentando l'efficienza.
- Costo delle licenze: questo aspetto è chiaramente importante, normalmente però i costi delle licenze dei soli Application Server sono esigui, perché legati ai costi del DBMS (molto più importanti).
- Supporto a pagamento: un fattore importante soprattutto per le grandi aziende, che possono permettersi di sostenerne i costi
- Supporto della rete: Application Server (ma non solo) molto diffusi comportano una maggior quantità di informazioni in rete, con una maggior possibilità di reperire soluzioni a costi irrisori. E' il fattore principale delle piccole aziende.
- Portabilità: uno degli aspetti fondamentali del successo di Java per gli application Server JEE. Questa tecnologia permette di integrare componenti molto diverse tra loro, favorendo economie di scala.

I principali Application Server presenti nel panorama delle applicazioni web e che implementano lo standard JEE (chi completamente e chi in parte) sono:

- JBOSS: pioniere degli Application Server OpenSource, funziona sia da EJB-Container che da Web-Container grazie all'integrazione di Apache Tomcat. JBOSS fa parte di una divisione di RedHat.
- Apache Tomcat: funziona solo da Web-Container. Gestisce Java Servlet e Java Server Pages.
- Sun GlassFish Enterprise Server: essendo gestito direttamente da Sun implementa nel modo più completo e fedele le specifiche dello standard JEE; le versioni seguono di pari passo i nuovi rilasci delle specifiche JEE.
- BEA Weblogic: soluzione commerciale proposta da Oracle, supporta al momento (Marzo 2010) lo standard JEE 1.5.
- IBM WebSphere

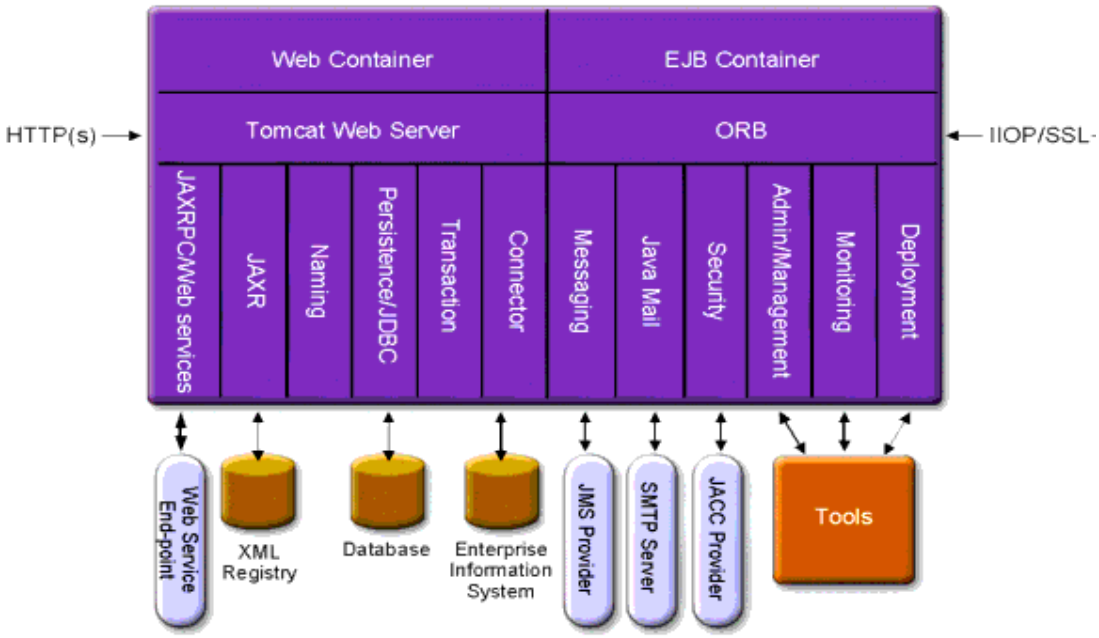

Figura 1.2 Architettura di un Application Server JEE

Nell'azienda Aive la conoscenza di tecnologie e software diversi è importante anche al fine di poter integrare i propri software nelle più diverse realtà aziendali dei propri clienti.

1.3 Servlet

Questa tecnologia è utilizzata all'interno del Web-Container. Una Servlet ha il compito di gestire le richieste dell'utente, effettuare eventuali elaborazioni e fornire una

risposta al Client, spesso mediante il protocollo di rete HTTP. La struttura di base di una Servlet è rappresentata di seguito:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Servlet extends HttpServlet
    {
    public void doGet(HttpServletRequest request,HttpServletResponse
    response) throws ServletException, IOException
    {//Overriding del metodo doGet
    }
    public void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {//Overriding del metodo doPost
    }
}
```

I package *javax.servlet* e *javax.servlet.http* forniscono le classi e le interfacce rispettivamente per scrivere una Servlet generica e una Servlet per il protocollo http; la propria Servlet, in quanto tale, deve implementare l'interfaccia *Servlet* o per lo meno estendere javax.servlet.GenericServlet. Nell'implementazione è possibile utilizzare (ed in questo caso estendere) una classe fornita dalle API di Java ovvero la classe *HttpServlet*, che fornisce due importanti metodi per gestire servizi HTTP: *doGet()* e *doPost()*. Questi ultimi rappresentano i metodi GET e POST del protocollo http; richiedono come parametri gli oggetti *HttpServletRequest* e *HttpServletResponse* e rappresentano rispettivamente la richiesta del client (che contiene ad esempio i dati di input per l'elaborazione) e la risposta del server (che può contenere dati di output come risultati di elaborazione).

Descriviamo ora il ciclo di vita di una Servlet. Queste fasi sono controllate dal Web-Container, in particolare, quando una richiesta viene indirizzata verso una servlet:

1. Se non esiste un'istanza della Servlet il Web-container
    a) Carica la classe Servlet
    b) Crea un'istanza della classe
    c) Inizializza la Servlet invocando il metodo *init*. Il metodo può essere sovrascritto passando come parametro un oggetto della classe *ServletConfig* al fine di permettere alla Servlet di accedere a dati di configurazione come password di file, cookies, parametri di accesso al database o dati provenienti da richieste precedenti. Una Servlet che, per qualche motivo, non è in grado di terminare l'inizializzazione lancia l'eccezione *UnavailableException*.
2. Il Web-container invoca il metodo *service* passando gli oggetti di richiesta *HttpServletRequest* e di risposta *HttpServletResponse*. Questo metodo identifica il tipo di richiesta del protocollo HTTP (GET, POST, PUT, DELETE) e invoca il corrispondente metodo Servlet *doGet(), doPost(), doPut(), doDelete()*. Altri metodi esistenti sono *doOptions()* e *doTrace()*.

3. Se il container deve eliminare l'istanza della Servlet, sia per una scelta dell'amministratore dell'applicazione sia perché da tempo inutilizzata, chiama il metodo *destroy,* il quale effettua tutte le operazioni di chiusura tra cui la dismissione delle connessioni ai database, conclusione di eventuali processi ed in generale effettua tutte le operazioni per *dismettere* l'istanza.

Una volta che una Servlet è stata inizializzata, le successive chiamate alla risorsa faranno riferimento all'unica istanza caricata in memoria e verranno creati dei thread Java, nei quali verrà invocato il metodo *service*. Essendo la creazione di un thread meno costosa in termini di risorse e veloce rispetto alla creazione di un nuovo processo, ecco che si intuisce la miglioria ottenuta con la tecnologia Servlet rispetto ai *CGI (Common Gateway Interface)*[1]*,* tecnologia in disuso.

Altri metodi importanti sono:

- *getParameter("Param")* che raccoglie il valore di un parametro dalla richiesta HTTP e restituisce un elemento di tipo String;

```
public class Servlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    String parametro =  request.getParameter("ParamName");
    }
}
```

- *getServletContext()* che ritorna il contesto (della classe ServletContext) in cui opera la Servlet per comunicare con il Web Container.
- *setAttribute(String name, Object obj)* che permette di associare alla variabile *name* un valore *obj,* che può essere utilizzato in seguito, nell'ambito della gestione della medesima richiesta.
- *getAttribute(String name)* questo metodo permette di accedere al valore associato a *name* precedentemente memorizzato.
- *getSession()* ritorna l'istanza di un oggetto che rappresenta una sessione di lavoro con l'utente.

Un possibile passaggio di parametri da utente a Servlet può essere attraverso il metodo HTTP GET, con *URL append*; ad esempio: in una pagina HTML (HyperText Markup Language) è possibile trovare

```
<a href="Path/MyServlet?ParamName=AAA">submit</a>
```

che permette a *MyServlet* (all'interno del metodo *doGet*) di inizializzare la variabile *parametro* uguale alla stringa *"AAA".*

[1] Prima forma di programmazione Server Side che consiste nel far eseguire dei veri e propri programmi dal web server. Il controllo della gestione della risposta passa dal web server al processo che fa eseguire il codice del programma

Un'importante interfaccia delle Servlet è *HttpSession* che permette di creare uno strato *stateful* tra il client e il Server. La Servlet può chiedere al Web-Container la creazione di una sessione di lavoro con il Client; il container si fa carico di associare le successive richieste del Client alla sessione, attraverso, per esempio, l'utilizzo di Cookie. La sessione ha un tempo di vita limitato e configurabile in fase di deploy dell'applicazione, scaduto il quale la sessione viene eliminata. L'oggetto *HttpSession* viene richiamato all'interno della Servlet con il metodo *getSession()* dell'interfaccia *HttpSessionRequest*.

1.4 JSP (JavaServer Pages)

Questa tecnologia permette di creare pagine di testo sia statiche sia dinamiche. Per dinamiche s'intende che il contenuto delle pagine può essere creato includendo risultati di elaborazione, dati prelevati dal database, oppure parametri inseriti dall'utente. JSP è un buon connubio tra le funzionalità dinamiche di una Servlet e la staticità di una pagina formattata ad esempio in HTML. Infatti, l'inserimento di contenuti statici e costrutti di formattazione all'interno di una Servlet è difficoltoso, per questo motivo è preferibile utilizzare la tecnologia JSP.

Vediamo in breve il contesto esecutivo. Il client richiede l'accesso al file con estensione *.jsp,* il quale viene processato dal Web-Container che lo trasforma in un file Java, lo compila, lo carica e lo esegue come una Servlet; ovviamente, la fase di traduzione, compilazione e caricamento viene effettuata solo la prima volta che viene individuata una richiesta alla risorsa oppure se la JSP è stata modificata. Questa tecnologia, inoltre, mette a disposizione una serie di tag e costrutti specifici.

Analizziamo ora i principali tag di una pagina JSP.

- *<% statement %>* all'interno di questo tag viene scritto codice, per lo più Java, che verrà interprato dalla Java Virtual Machine;

- *<%= espressione %>* all'interno del quale viene calcolata l'espressione indicata e ne viene restituito il valore;

- *<%! dichiarazione %>* all'interno di questo tag sono presenti frammenti di codice che vengono riportati al di fuori del metodo principale di gestione della richiesta, ovvero a livello di classe;

- *<%-- commento --%>* quello che è racchiuso da questo tag rappresenta un commento al codice;

- *<!-- commento -->* questo rappresenta un commento HTML. Nella pagina JSP, i commenti HTML vengono comunque processati e fanno parte del flusso di risposta. Sarà quindi possibile scrive un commento dinamico. Questa possibilità è

sfruttata nelle applicazioni per scrivere codice JavaScript, che viene eseguito dal Client, in modo dinamico a seconda dello stato dell'applicazione Web;

- *<jsp:useBean ...>* serve ad identificare (o eventualmente istanziare, in caso non esistesse) un oggetto per utilizzarlo nella pagina;

- *<jsp:setProperty ...>* permette di assegnare un valore ad una proprietà di un oggetto;

- *<jsp:getProperty ...>* serve ad ottenere il valore della proprietà di un oggetto;

- *<jsp:include ...>* esegue un'altra pagina JSP e ne include l'output;

- *<jsp:forward...>* inoltra il controllo di richiesta e risposta ad un'altra pagina JSP.

Sono inoltre presenti tre differenti tipi di direttive per la traduzione e la compilazione di una pagina JSP, in particolare

- *page* che serve a specificare alcune caratteristiche della pagina JSP;
- *taglib* che serve ad importare una libreria di tag esterna (trattati in §2.8);
- *include* che serve ad importare, senza l'esecuzione preventiva, una porzione di codice di un file.

Queste direttive sono dichiarabili attraverso il tag *<%@ direttiva %>*.

Un'importante caratteristica degli oggetti di una pagina JSP si chiama *Scope*. Lo Scope di un oggetto rappresenta la visibilità che questo ha all'interno dell'applicazione web. La seguente tabella riassume i possibili valori per lo Scope.

Scope	Visibilità	Ispezione
page	all'interno della stessa pagina	-
request	Durante tutta l'elaborazione della richiesta corrente	HttpServletRequest.getAttribute()
session	per tutto l'arco di una stessa sessione	HttpSession.getAttribute()
application	per tutta la vita dell'applicazione web	ServletContext.getAttribute()

Tabella dei diversi valori assunti dallo scope

Infine, di seguito è rappresentato un esempio di utilizzo dei tag all'interno di una pagina JSP, che contiene anche i costrutti di formattazione HTML.

```
<%@ page contentType="text/html ; charset=iso-8859-1" language="java" %>
<%@ page import="java.sql.*,java.util.*,beans.*" %>
<%@ tagliburi="es.tld" prefix="es" %>
<jsp:useBean id="utente" scope="session" type="User" />
```

```
<html>
<head>
<title>Examples</title>
<head>
<body>
Test Page
</body>
</html>
```

In questo esempio è definito il tipo di documento, detto "*contentType"*, e sono importate alcune classi java che possono essere utilizzate all'interno della pagina, con il la parola chiave "*import*". E' poi importata una libreria di tag personalizzata, "*es.tld"*, e si utilizza un oggetto di tipo *JavaBean* con scope *Session*.

1.5 JDBC (Java DataBase Connectivity)

JDBC non è altro che l'insieme delle API che definiscono le specifiche per l'accesso a database relazionali. Fa parte sia dello standard J2SE sia di JEE. L'insieme delle classi che offrono una connettività tra Java e un DBMS, implementando le interfacce dei package *java.sql* e *javax.sql*, prende il nome di driver JDBC; le principali interfacce sono: *Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, DatabaseMetaData, ResultSetMetaData*. Vi sono fondamentalmente quattro diversi tipi di Driver JDBC:

- *JDBC-ODBC bridge*: come si può intuire questi driver implementano le specifiche traducendo le operazioni JDBC in operazioni specifiche ODBC[2].
- Driver scritti parzialmente in Java ed in parte in linguaggio nativo, utilizzano componenti del produttore dello specifico DBMS.
- Driver completamente scritti in Java che comunicano con il DBMS attraverso un gateway, il quale converte le richieste/risposte in richieste/risposte specifiche per il DBMS.
- Driver completamente scritti in Java e che comunicano con il DBMS in modo diretto.

Le classi implementative sono normalmente contenute in pacchetti *.jar* che lo sviluppatore dovrà caricare nella propria applicazione. Successivamente, per poter utilizzare i driver, si dovrà aprire una connessione con il database, creare uno *Statement* per interrogare la base dati e gestire i risultati ottenuti. Di seguito è riportato una porzione di codice che potrebbe popolare una pagina JSP, in cui viene effettuata l'interrogazione di una base dati connessa tramite driver JDBC-ODBC.

[2] (Open Database Connectivity) sono API sviluppate da SQL Group nel 1992 con l'obbiettivo di rendere il protocollo di accesso ai database indipendente dal DBMS utilizzato. Il maggior supporto di queste specifiche è stato dato da Microsoft, per questo motivo i driver ODBC vengono utilizzati quasi sempre in ambiente Windows.

```
<%@ page import="java.sql.*" , java.util.*" %>
<%
// Viene caricato il driver
String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
Class.forName(driver);
String url = "jdbc:odbc:myDataSource";

// Viene creata una connessione con username e password
Connection connection =
DriverManager.getConnection(url, "myUser", "myPassword");

// Viene creato un oggetto Statement
Statement std = connection.createStatement();

// Viene eseguita un'interrogazione e raccolti i risultati
// in un oggetto ResultSet
String query = "SELECT NomeColonna1, NomeColonna2 FROM myTable";
ResultSet res = std.executeQuery(query);
%>

<table>

<%
// Vengono stampati i risultati inseriti in una tabella
while (res.next()) {
%>

<tr>
 <td><%= res.getString("NomeColonna1"); %></td>
 <td><%= res.getString("NomeColonna2"); %></td>
</tr>

<%}
// Chiusura di tutti i flussi
res.close();std.close();connection.close();
%>
```

L'oggetto di tipo *Connection* consente di ottenere una connessione con un database. La classe *DriverManager* è incaricata di scegliere l'opportuno driver registrato, attraversi l'URL e le credenziali di accesso al database passati come parametri. L'oggetto *Statement*, invece, permette di effettuare interrogazioni alla base dati in linguaggio SQL, attraverso la connessione precedentemente creata. Qualora lo stesso *Statement* SQL sia utilizzato più volte, è consigliato servirsi, in alternativa, dell'oggetto *PreparedStatement* che aumenta l'efficienza dell'esecuzione della query e l'allocazione delle risorse. Infine, il *ResultSet*, è l'oggetto che raccoglie i dati richiesti all'interno dello *Statement* e li inserisce in una struttura dati opportuna. Il metodo *getString("NomeColonna")* restituisce il valore correntemente puntato dal cursore *next()* della colonna specificata come parametro. Se il nome della colonna non è presente nel ResultSet, non viene lanciata alcuna eccezione. Esistono diversi metodi *getType("NomeColonna")*, dove *Type* indica il tipo di dato che si vuole estrarre dal *ResultSet* e che deve essere concorde al tipo di dato di *"NomeColonna"*.

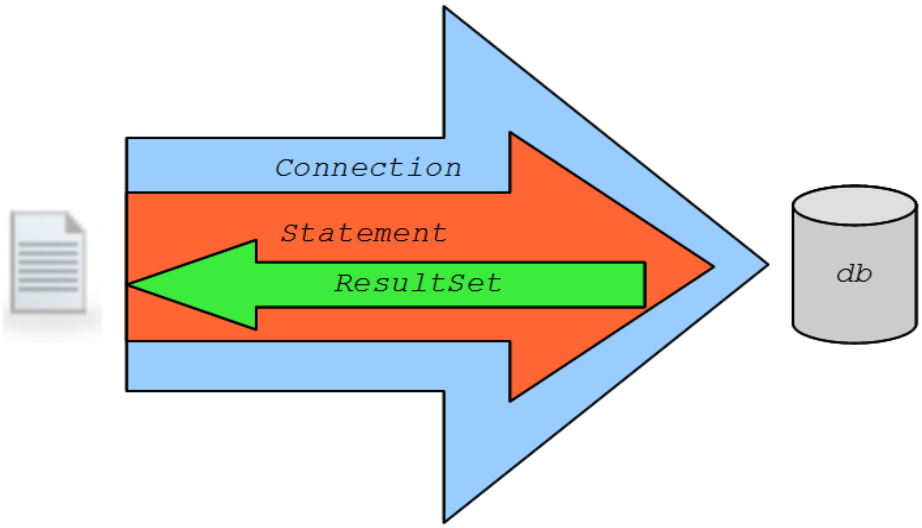

Figura 1.3 Schematizzazione degli oggetti di una connessione JDBC

1.6 JNDI (Java Naming and Directory Interfaces)

JNDI sono API che offrono servizi per identificare degli oggetti, o in generale, delle risorse con un nome, detto *nome JNDI*. Le risorse possono anche appartenere a Java Virtual Machine diverse. Le interfacce offrono nel loro insieme servizi di *naming* usando il paradigma Client/Server: il server offre i servizi JNDI e il client ne usufruisce per recuperare le risorse di cui ha bisogno. Il servizio è in esecuzione sull'Application Server JEE, il quale deve implementare le interfacce SPI (Service Provider Interface).

Normalmente, in un contesto enterprise, l'accesso ai database da parte dell'applicazione non avviene attraverso le modalità esposte in §1.5. La connessione non viene richiesta tramite *DriverManager,* ma bensì tramite un'istanza della classe *javax.sql.DataSource*, ottenuta attraverso JNDI. Questo permette all'Application Server di frapporsi tra l'applicazione e la connessione fisica al DBMS, aggiungendo nuovi servizi e funzionalità. Tra i più importanti vi è l'aggiunta di un *pool connection* che ottimizza e automatizza la connessione. Attraverso questo strumento, quando un'applicazione richiede una connessione a un database, l'Application Server verifica se vi sono connessioni aperte non utilizzate e se ciò si verifica, ritorna all'applicazione una di queste. Il vantaggio principale sta nel fatto di non richiedere, ogni volta, la negoziazione delle credenziali di autorizzazione e l'allocazione di risorse a livello di driver JDBC (e soprattutto a livello DBMS). Quando l'applicazione non ha più bisogno di utilizzare la connessione, quest'ultima viene rilasciata e resa disponibile ad altre applicazioni. Per questo motivo l'Application Server deve intercettare le invocazioni al metodo *Connection.close()* che indicano che la connessione non è più utilizzata. L'invocazione di questo metodo da parte dell'applicazione non chiude fisicamente la connessione al DBMS, ma la rilascia all'Application Server, che potrà darla

ad un altro Client. Questa modalità è stata utilizzata anche durante lo sviluppo del progetto, presentato al capitolo 3.

1.7 JavaBean ed Enterprise JavaBean (EJB)

Queste componenti sono elementi importantissimi per un'applicazione complessa che utilizzi tecnologie Java, non solo per quelle orientate al web. Un JavaBean ha la funzione principale di contenere e manipolare le informazioni ed è rappresentata da una classe Java con determinati requisiti:

- lo stato dell'istanza deve avere proprietà accessibili solo all'interno della classe stessa (visibilità privata);
- devono esservi metodi pubblici specifici per poter ottenere (*getProperty()*) e manipolare (*setProperty(value)*) le informazioni;
- il suo costruttore non deve avere argomenti;
- può comunicare i propri cambiamenti di stato con altre componenti mediante la generazione di eventi;
- l'istanza deve essere serializzabile (implementando l'interfaccia *Serializable*).

Di seguito, è rappresentato un semplice esempio d'implementazione di un JavaBean per la raccolta delle informazioni di un utente in navigazione all'interno di una generica applicazione. La proprietà *pagineViste* rappresenta il numero di pagine viste dall'utente durante la navigazione; la proprietà *user*, invece, identifica la login dell'utente.

```
public class InfoUtente implements java.io.Serializable {
 private String user;
 private int pagineViste;

// Costruttore senza argomenti.
public void InfoUtente() {
}

    // Metodi pubblico per l'accesso alle proprietà
    public String getUser() {
    return this.user;
    }
    public void setUser(String user) {
    this.user = user;
    }
    public int getPagineViste() {
    return this.pagineViste;
    }
    public void setPagineViste(int pagine) {
    this.pagineViste = pagine;
    }
}
```

Queste componenti sono state pensate principalmente per essere manipolate attraverso applicazioni grafiche per cui sono largamente utilizzate. Attualmente hanno ottenuto un notevole successo anche in applicazioni web e nei progetti di tipo enterprise.

Gli *Enterprise JavaBean* non sono altro che l'equivalente della tecnologia JavaBean in ambito enterprise ed implementano la logica di business dell'applicazione. L'introduzione di questo tipo di tecnologia nel lato server fu introdotta per risolvere i seguenti problemi:

- Persistenza dei Dati
- Gestione delle transazioni
- Controllo della concorrenza
- Standard di sicurezza
- Integrazione di componenti software, anche di natura diversa
- Invocazione di procedure remote[3]
- Fornire servizi sul web

Vi sono tre tipi di EJB:

- **EJB di sessione o *Session EJB***: la vita di questo tipo di oggetto è legata alla richiesta di utilizzo da parte del client, poiché dopo l'utilizzo viene elaborata la sua eliminazione. Per questo motivo, un Session EJB non è un elemento condiviso con altri Client ed ha, generalmente, un ciclo di vita limitato. Il suo scopo principale è quello di fornire un'interfaccia semplice per tutti i client che richiedono di accedere a determinate informazioni; una strutturazione di questo tipo impedisce che modifiche alla struttura dati di controllo comportino correzioni alle applicazioni client. In figura 1.4 è rappresentata la posizione di un EJB di sessione tra client e il gestore delle informazioni, mascherando il meccanismo per il recupero e la manipolazione dei dati.

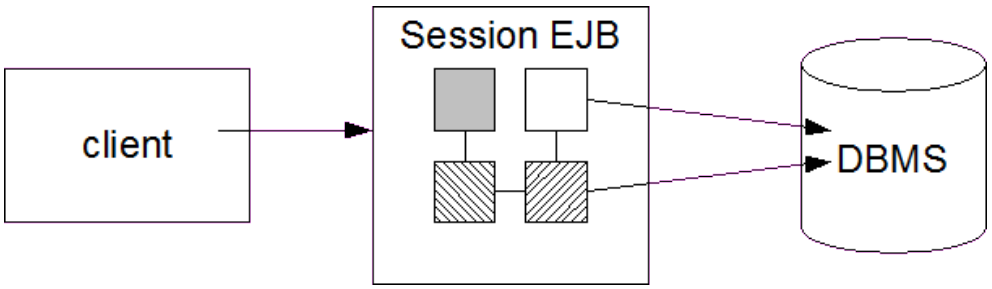

Figura 1.4 Schema del ruolo di un Session EJB

Questi elementi possono essere divisi in ulteriori due categorie:

[3] Forniscono interfacce remote accessibili direttamente tramite RMI: una classe residente su una JVM può servirsi di un EJB residente su un'altra JVM invocandone i metodi desiderati, fornendo dei parametri e ricevere dei risultati.

1. Session Statefull: portano con sé uno stato, il quale può cambiare tramite l'invocazione di un client. Lo stato può essere rappresentato da più proprietà dell'EJB stesso, il quale effettua un legame logico con il client.
2. Session Stateless: contrariamente al caso precedente, questo elemento non possiede uno stato e le sue istanze sono perciò tutte indistinguibili.

- **EJB di Entità o *Entity EJB***: forniscono la caratteristica di persistenza dei dati. Questi oggetti inglobano i dati sul lato server, come ad esempio righe di una tabella di un database o un File-System; possono essere condivisi da client diversi e sono in grado di sopportare periodi di disservizio dell'application server. Nel loro utilizzo più frequente, inglobano i dati di un database tramite oggetti Java, grazie ai quali è possibile manipolarli più agevolmente all'interno delle proprie applicazioni e senza l'utilizzo diretto di JDBC. Di norma, il client per accedere a questi Bean utilizza EJB di sessione stateless (presentati nel punto precedente), aumentando ulteriormente la stratificazione logica di queste componenti. Sia gli EJB di sessione sia di entità hanno un funzionamento sincrono, perciò il client che li invoca deve attendere la terminazione della loro elaborazione potendo, poi, continuare la propria esecuzione.

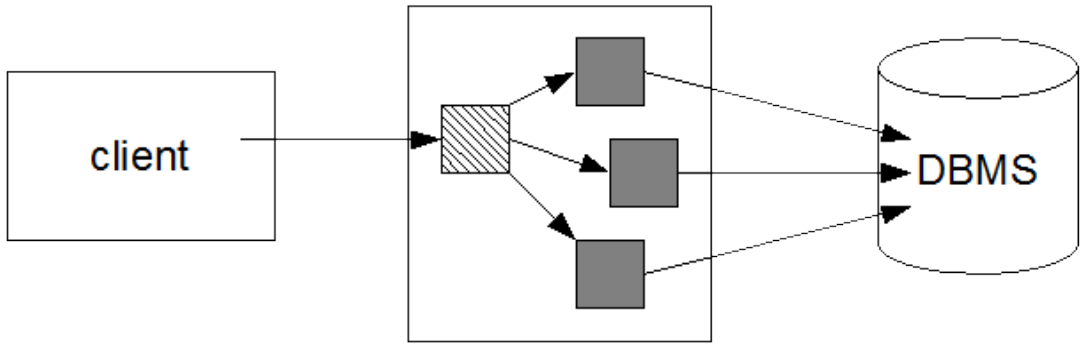

Figura 1.5 Nello schema sono rappresentati in grigio gli entity EJB e in tratteggio i session stateless EJB

Fino alla versione 2 dello standard EJB, la gestione della persistenza veniva suddivisa in due tipologie:

1. persistenza gestita dall'EJB-Container: i dati relativi ad un oggetto vengono memorizzati nella base dati dall'EJB-Container che si occupa anche della loro estrazione ottimizzata;
2. persistenza gestita da un bean: il meccanismo è delegato allo sviluppatore, che implementa nel bean il sistema di memorizzazione e recupero dei dati.

Dalla versione 3 delle specifiche EJB sono state introdotte nuove API per la gestione della persistenza, chiamate *JavaPersistence API (JPA)* che, mediante il servizio *object-relational mapping (ORM)*[4], denotano il processo di mapping dei

[4] Tecnica di programmazione che favorisce l'integrazione di sistemi software aderenti al paradigma object-oriented con sistemi RDBMS, garantendo elevata portabilità, riduzione del codice ed elevata stratificazione del sistema software.

dati, fra gli oggetti e le tabelle di un database. Un esempio di persistence provider è rappresentato dal framework *Hibernate*.

- **EJB pilotati da messaggi o *Message Driven EJB (MDB)***: sono gli unici EJB ad avere comportamento asincrono, grazie all'utilizzo delle specifiche JMS (Java Message Service). Il client che intende invocare queste componenti, deve essere in grado di pubblicare o accodare messaggi su una coda, i quali vengono raccolti dall'application server e forniti ad un MDB. Una volta ricevuto il messaggio, l'MDB può invocare in modo sincrono altri tipi di EJB, attendere i risultati di ritorno ed accodare un messaggio di fine elaborazione al client invocante; tutto questo con la possibilità da parte del client di proseguire la sua esecuzione.

Session EJB e Message Driven EJB possono gestire le transazioni in due differenti modi. Nel primo, la gestione è delegata all'EJB-Container che si prende carico di iniziare una nuova transazione e concluderla con un commit o un rollback, tutto questo seguendo la configurazione del file *ejb-jar.xml*. Nel secondo modo, le transazioni sono gestite all'interno del bean, e viene implementata direttamente dallo sviluppatore tramite l'uso dell'interfaccia *javax.ejb.EJBContext* e i metodi *begin()*, *commit()* e *rollback()*, che permettono all'EJB di comunicare con l'Application Server.

1.8 Struttura delle directory

Un'applicazione web può essere distribuita all'interno di un pacchetto con estensione .WAR.

Un archivio WAR contiene, nella directory principale, tutti i file statici come pagine html, fogli di stile, immagini e file JSP non precompilati, eventualmente raggruppati in opportune sottodirectory; inoltre, deve essere presente una directory WEB-INF contenente le componenti seguenti:

- *web.xml* : Questo file è detto deployment descriptor dell'applicazione web.
- *classes/* : è la directory che contiene le classi Java compilate come Servlet, JavaBeans e JSP precompilate.
- *lib/* : è la directory che contiene tutte le librerie necessaria al funzionamento del modulo.

Questo pacchetto può essere generato tramite svariati strumenti. All'interno del progetto presentato al capitolo 3, si è utilizzata l'utility Apache Ant, un'applicazione java che permette di automatizzare le operazioni di compilazione, organizzazione e deployment, secondo i dettagli specificati nel file di configurazione *build.xml*.

Il file *web.xml* ha il compito di descrivere tutti gli aspetti fondamentali dell'applicazione web, una configurazione che il web container necessita di conoscere per il corretto funzionamento. Di seguito è riportato un esempio della struttura del file web.xml e dei parametri più importanti contenuti all'interno.

```
<?xml version="1.0" encoding="UTF-8"?>
    <web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    …
    </web-app>
```

All'interno dei tag <*web-app*> è contenuta l'intera configurazione tra cui nome e descrizione dell'applicazione. Sono poi contenuti i riferimenti alle classi Servlet che compongono il modulo web ed eventuali parametri del contesto Servlet;

```
…
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>it.myPackage.web.sample.SampleServlet</servlet-class>
    <init-param>
        <param-name>parametri</param-name>
        <param-value>/WEB-INF/param.xml</param-value>
    </init-param>
</servlet>
…
```

La configurazione del mapping della Servlet è necessaria per comunicare quali URL devono essere gestiti dalla Servlet stessa;

```
…
<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
…
```

E' possibile indicare una pagina iniziale, qualora l'utente non richiedesse una specifica pagina;

```
…
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
…
```

È definita una pagina di errore, in caso l'utente richieda una risorsa inesistente;

```
…
<error-page>
    <error-code>404</error-code>
    <location>/errore404.jsp</location>
</error-page>
…
```

Sono contenute le dichiarazioni delle librerie di tag utilizzate all'interno dall'applicazione;

```
…
<taglib>
<taglib-uri>c.tld</taglib-uri>
<taglib-location>/WEB-INF/c.tld</taglib-location>
</taglib>
…
```

Definizione di aree protette del modulo web che per l'accesso necessitano di autenticazione;

```
…
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Admin</web-resource-name>
        <description>Area di amministrazione</description>
        <url-pattern>*.pdf</url-pattern>
        <url-pattern>*.do</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>user</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/admin.jsp</form-login-page>
        <form-error-page>/error.jsp</form-error-page>
    </form-login-config>
</login-config>
<security-role>
    <role-name>user</role-name>
</security-role>
…
```

Infine, eventuali riferimenti a EJB (§ 1.7).

```
…
<ejb-ref>
    <ejb-ref-name>EJB/MyEJB</ejb-ref-name>
    <ejb-ref-type>Session</ej-ref-type>
    <home>it.myPackage.web.ejb.MyEJB</home>
    <remote>it.myPackage.web.ejb.MyEJB</remote>
</ejb-ref>
…
```

2 Apache Struts

In questo capitolo verrà fornita un'introduzione ai pattern architetturale Model 1 e Model 2 (Model-View-Controller) in ambito delle applicazioni web. Saranno sottolineati i motivi che spingono i progettisti all'utilizzo di tali pattern architetturali e le tecnologie Java che li supportano. Sarà trattato in dettaglio il funzionamento della versione 1 del framework Struts, utilizzata nello sviluppo del progetto "Startup". Infine, sarà presentata una panoramica sulle differenze tra le due versioni di Struts, e le principali alternative a questo framework.

2.1 Pattern

Un'importante fase nella progettazione di applicazioni software è quella della scelta di un'opportuna architettura, che definisce le linee guida allo sviluppo del progetto. Definire bene tale processo è utile sia per standardizzare il modello di sviluppo del progetto corrente, sia per le applicazioni future. L'impiego di un adeguato pattern porta numerosi vantaggi tra cui:

1. Incrementa il riutilizzo del codice.
2. Facilita il lavoro in team e la pianificazione del progetto, dividendo quest'ultimo in componenti indipendenti delegabili a gruppi di lavoro differenti.
3. Aiuta la manutenzione del codice, grazie anche all'abbassamento della curva di apprendimento (per i progetti più complessi).
4. Aumenta la flessibilità delle applicazioni e incrementa della scalabilità.

Per applicazioni semplici, il codice che gestisce i dati da memorizzare o da visualizzare può essere incluso direttamente nelle pagine JSP. Similmente è possibile delegare alle classi Servlet anche la parte di presentazione dei dati e la gestione delle richieste dell'utente. Questo modello è definito dal paradigma *Model 1*, schematizzato in figura 2.1; è importante ribadire che tale modello può essere utilizzato solo nel caso

in cui l'applicazione possieda scarsa complessità di interazione tra client e server, funzioni limitate e poche pagine.

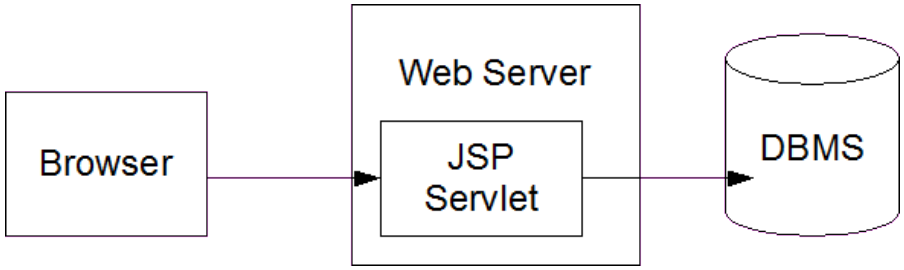

Figura 2.1 Paradigma Model 1

Molti linguaggi HTML-based, come *PHP*[1] e *ASP*[2], utilizzano unicamente questo paradigma sfruttando la sua semplicità, in quanto, le tecnologie utilizzate richiedono minor conoscenze tecniche e minor tempo per il rilascio delle versioni. La scelta impropria di questo paradigma all'interno dei propri progetti può portare, però, ad alcuni inconvenienti:

- Eccessivo codice di controllo all'interno delle pagine JSP
- Eccessivo codice di presentazione all'interno delle Servlet
- Scarsa efficienza complessiva

2.2 Il paradigma Model 2: il pattern architetturale MVC

Il modello più utilizzato in ambito delle applicazioni orientate al web (ma non solo) è il pattern *Model-View-Controller (MVC)*. Fu introdotto per la prima volta nella progettazione del linguaggio *SmallTalk*, nel 1992. Questo pattern permette una maggiore strutturazione del codice, con un aumento della manutenibilità software e una suddivisione dell'applicazione in sottosistemi. Per comprendere la filosofia del modello, in figura 2.2, è riportata la schematizzazione delle tre componenti in cui si divide.

[1] (PHP: Hypertext Preprocessor) Linguaggio di scripting interpretato con licenza GPL, viene utilizzato principalmente per creare applicazioni lato server e piccoli script stand-alone.

[2] (Active Server Pages) altro linguaggio di scripting di proprietà di Microsoft Corp., progetto ufficialmente abbandonato in favore di ASP .NET.

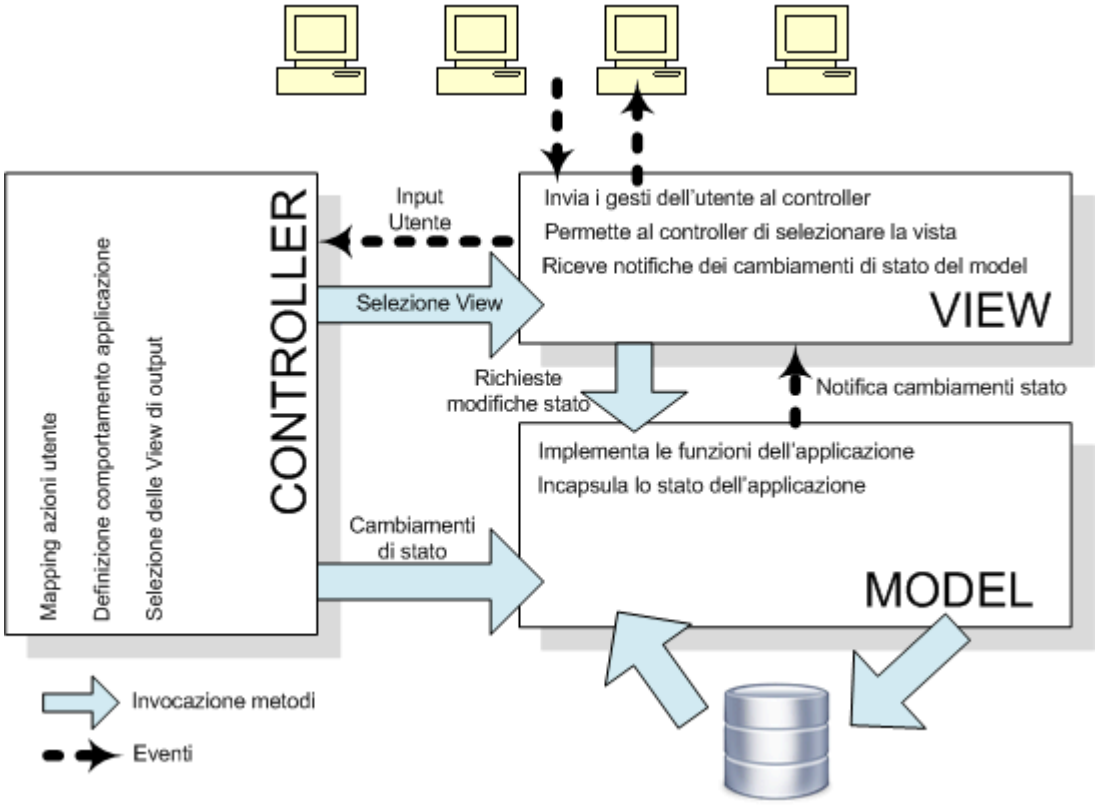

Figura 2.2 Implementazione comune del pattern MVC

Le componenti sono:

- la *vista (view)* dove viene gestita la presentazione dei dati. E' rappresentata da tutto quel codice di presentazione che permette all'utente[3] di operare le richieste. All'interno di questo livello lavorano sia programmatori che grafici che curano la parte estetica della presentazione;
- il *modello (model)* che rappresenta e gestisce i dati, tipicamente persistenti su database;
- il *controllore (controller)* che veicola i flussi di interazione tra vista e modello, organizzando il comportamento dell'applicazione e tracciando la navigazione con meccanismi che creano uno stato associato all'utente. Nello specifico, intercetta le richieste HTTP del client e traduce ogni singola richiesta in una specifica operazione per il model; in seguito può eseguire lui stesso l'operazione oppure delegare il compito ad un'altra componente. In ultima, seleziona la corretta vista da mostrare al client ed inserisce, eventualmente, i risultati ottenuti.

All'interno di un'architettura 2-tier, dove vi è interazione diretta con il database, è possibile implementare il pattern MVC dentro il Web-Tier, utilizzando Servlet per il

[3] Ad esempio, tramite un Browser

controllo, JSP per la vista e, per il model, classi Java o JavaBeans che comunicano direttamente con il DBMS tramite JDBC. Questo è il modello che è stato utilizzato per la progettazione della web application presentata nel capitolo 3.

Nel contesto JEE, invece, il model è realizzato mediante l'uso della tecnologia EJB ed è gestito da un EJB Server; la vista è realizzata con JSP e tecnologie di trasformazione XML–HTML, ed il controller con EJB e Servlet.

2.3 Introduzione al framework

Nel precedente paragrafo, si è parlato del pattern MVC al fine di introdurre un framework di sviluppo famoso nel panorama delle applicazioni web, che ha saputo implementare in modo efficiente il modello architetturale. In primis, è bene definire il concetto di framework: è una struttura di supporto per la produzione di software che offre soluzioni di modularità e riusabilità. In dettaglio, un framework è rappresentato da una serie di librerie costituite da classi e interfacce, in uno o più linguaggi di programmazione, che cooperano alla risoluzione di uno specifico problema ed aumentano la velocità di sviluppo dei progetti. Un buon framework è anche in grado di fornire una valida soluzione per diversi tipi di applicazioni, ma non per questo può essere considerata una scelta sempre valida:

- una scelta inappropriata del framework, per valutazioni errate del problema in esame, può compromettere la buona riuscita del progetto;
- forzare l'utilizzo di un framework inadatto al problema, può portare ad un prodotto estremamente inefficiente.

Per questo motivo in fase di progettazione, qualora non esistesse un framework che si adatti al problema in esame, è bene ricorrere ad una implementazione personalizzata.

La prima versione del framework Struts, ad opera di *C.R.McClanahan*[4] nel 2000, è stata (ed è tuttora) largamente utilizzata. Nel 2008 è stata presentata la seconda versione, che però si differenzia in modo profondo dalle versioni precedenti, tanto da essere considerata un nuovo framework.

Struts 2.0 è stato originariamente sviluppato dal team WebWork da cui prende inizialmente il nome Webwork 2. Struts riguarda il Web-Tier, il quale permette all'applicazione di comunicare ed interoperare con i client attraverso il web.

[4] Programmatore del gruppo Apache che ha collaborato anche alla realizzazione delle specifiche Servlet e JSP

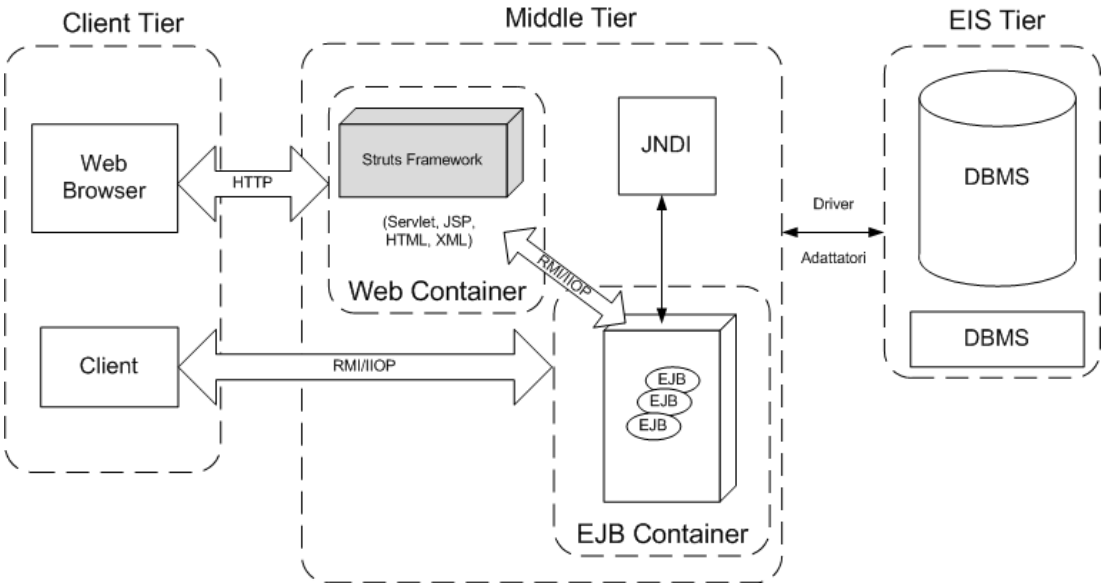

Figura 2.3 Posizione del framework Struts all'interno dell'architettura 3-Tier

E' basato sulle tecnologie Servlet, JSP e XML e le principali componenti sono:

- *ActionServlet*
- *RequestProcessor*
- *struts-config.xml*
- *Action*
- *ActionForm*
- *ActionMessage*

Di seguito sarà analizzato il funzionamento della versione 1 del framework e successivamente si sottolineeranno le principali differenze con la versione successiva.

2.4 ActionServlet e RequestProcessor

ActionServlet è la Servlet che funziona da Controller del pattern MVC, e gestisce tutte le richieste dell'applicazione. Estende la classe *javax.servlet.http.HttpServlet* e quindi implementa tutti i metodi presentati in §1.1, inclusi *doGet()* e *doPost()*. Per poterla utilizzare deve essere definita all'interno del file *web.xml* (introdotto in §1.8) della propria applicazione:

```
…
<servlet>
<servlet-name>action<servlet-name>
<servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
<init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
</init-param>
</servlet>
…
```

I parametri d'inizializzazione possono essere: *config* (presente nell'esempio precedente) che specifica il path assoluto del file XML di configurazione dell'ActionServlet e *validating*, che indica l'attivazione di un parser XML per la correttezza sintattica del file di configurazione. Vanno poi specificate il tipo di richieste che Struts deve gestire e che tipicamente hanno estensione **.do*.

```
…
<servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
…
```

Inoltre, è possibile inizializzare la Servlet (invocazione preventiva del metodo *init*) in fase di inizializzazione del modulo web con la configurazione

```
<servlet>
…
<load-on-startup>1</load-on-startup>
…
<servlet>
```

Al fine di snellire il file di configurazione struts-config.xml, dalla versione 1.1 è stata introdotta la possibilità di suddividere la propria applicazione in sotto-applicazioni più piccole, ognuna con il proprio file di configurazione. L'ActionServlet delega alle istanze della classe *RequestProcessor* il controllo delle richieste/risposte delle singole sottoapplicazioni.

Di seguito sono riportati gli step di funzionamento generati da una richiesta dell'utente:

1. L'utente richiede la risorsa *autenticazione.do* attraverso il metodo HTTP GET.
2. Il Web Server invoca il metodo *doGet()* della Servlet ActionServlet.
3. Il Web Server invoca la giusta istanza del RequestProcessor al quale appartiene la risorsa, attraverso il parametro *config* del *web.xml.*
4. L'istanza del RequestProcessor consulta la configurazione *struts-config.xml* (la struttura dati associata) e ricava l'*azione*[5] corrispondente alla risorsa richiesta ed esegue il metodo *execute()*.

   ```
   <action path="/autenticazione" type="AuthenticationAction">
       <forward name="success" path="/WEB-INF/jsp/welcome.jsp" />
   </action>
   ```

   Sopra è presentato il codice dello struts-config.xml con il quale si indica che, alla richiesta della risorsa *autenticazione,* dovrà essere invocata un'istanza della classe *AuthenticationAction*. Una volta terminata l'esecuzione del metodo *execute*, se

[5] Rappresentata da un'istanza di una AuthenticationAction che estende la classe org.apache.struts.action.Action del framework, e che implementa il metodo execute()

il messaggio di ritorno dell'azione sarà "*success*", il controller delega la generazione del codice di risposta alla pagina *welcome.jsp*.

2.5 struts-config.xml

Come detto nel paragrafo precedente, il file di configurazione *struts-config.xml*, determina l'azione da intraprendere durante la richiesta di una risorsa. E' possibile però definire ulteriori elementi di configurazione:

- *form-beans*: serve a specificare le classi del tipo *ActionForm* che si occupano dell'incapsulamento dei parametri provenienti da una richiesta, attraverso un *form* HTML.

```
<form-beans>
    <form-bean name="form-autenticazione" type="AuthenticationForm" />
</form-beans>
```

- *global-forward*: si occupa di definire la corrispondenza tra un nome fittizio e un'azione. In questo modo, anche una richiesta di tipo GET o POST non conterrà il nome effettivo della classe e i riferimenti globali a tale risorsa potranno essere gestiti tutti qui.

```
<global-forward>
    <forward name="login" path="AuthenticationAction">
</global-forward>
```

- *action-mappings*: è la parte più importante del file di configurazione. Definisce la corrispondenza tra la richiesta HTTP e l'*Action* corrispondente. All'interno sono definiti degli elementi *action* con attributi diversi.

Attributo	Descrizione
path	Indica l'URI dell'azione
type	Identifica il nome completo della classe Action da utilizzare, compreso il package che la contiene
name	Nome del form bean che verrà utilizzato da questa Action e che deve essere precedentemente definito dall'elemento *form-beans*
input	Nome dell'Action o della pagina che contiene i campi del form bean. Nell'uri specificato, sarà reinderizzato l'utente in caso di errori nella compilazione del form
parameter	Valori di eventuali parametri da passare all'istanza Action
scope	Indica lo scope dell'istanza ActionForm, che può essere limitato alla richiesta oppure avere persistenza in sessione
validate	Si pone a TRUE tale parametro, se si vuole sottoporre a validazione i parametri inseriti dall'utente. La validazione avviene attraverso l'invocazione del metodo validate della classe ActionForm

Attributi dell'elemento action di action-mappings

- *controller*: serve a configurare l'istanza del RequestProcessor che gestisce l'applicazione.

Attributo	Descrizione
contentType	Specifica il tipo di output. Se non specificato è "text/html"
locale	Ha valore true se si vuole in sessione l'oggetto java.util.Locale
nocache	Se ha valore true indica, all'interno dell'header di risposta, che la pagina non deve essere inserite nella cache del browser.
processorClass	La classe che elabora la richiesta dell'utente

Attributi dell'elemento controller

- *message-resources*: utile per configurare gli oggetti che gestiscono i messaggi di testo dipendenti dalla lingua (e quindi dalla classe java Locale, §2.7)

Attributo	Descrizione
factory	Indica la classe Java che si occupa della gestione dei messaggi, di default è org.apache.struts.util.PropertyMessageResourcesFactory
parameter	Normalmente contiene il path del file che contiene le coppie chiave-messaggio, ma in generale è un valore passato come parametro alla classe di gestione.
null	Se posta TRUE, se durante il caricamento di una pagina manca il valore di una chiave associata al messaggio viene segnalato.

Attributi dell'elemento message-resources

Altri elementi sono *global-exception* che configura il comportamento delle eccezioni Java e *plug-in* che da' la possibilità di inizializzare una classe in fase di avvio dell'applicazione, per fornire funzionalità aggiuntive di terze parti. Tutti gli elementi devono essere inseriti all'interno dell'elemento *struts-config*

```
<struts-config>
…
</struts-config>
```

2.6 Action, ActionFordward, ActionMessage, ActionForm

Le classi *Action*, dette anche *azioni*, hanno il compito di definire il comportamento del controller e sono invocate dal RequestProccessor, tramite il file di configurazione struts-config.xml. Una classe azione, per essere definita tale, deve estendere la classe *org.apache.struts.action.Action*. Al suo interno, sono catturate le informazioni provenienti dalla richiesta utente, inviate al Model e selezionate le opportune View, dove veicolare i risultati ottenuti. L'azione deve poi gestire i parametri Session e Request (validità e scadenza) ed eventuali eccezioni. Di seguito è rappresentato un esempio di azione che gestisce i dati di una pagina di Login.

```
package examples;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionForward;

public class LoginAction extends Action {

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
                                 throws Exception
    {    String username = ((LoginForm) form).getUsername();
         String password = ((LoginForm) form).getPassword();

         if ( !username.equals("admin") || !password.equals("12345"))
         // il controllo va' alla risorsa
         // specificata nello struts-config.xml con il messaggio "failure"
            return mapping.findForward("failure");

         // store authentication info on the session
         HttpSession session = request.getSession();
         session.setAttribute("authentication", username);

         // ritorna il controllo alla risorsa
          // specificata nello struts-config.xml con il messaggio "success"
         return mapping.findForward("success");

}
```

Nella classe che estende Action deve essere implementato il metodo *execute* che è il metodo invocato dal RequestProcessor al momento della richiesta dell'utente. I parametri *HTTPServletRequest* e *HTTPServletResponse* sono gli oggetti Java rispettivamente per la richiesta e la risposta, come già visto in §1.3.

L'oggetto *mapping* della classe *ActionMapping* è un *JavaBean* che contiene tutte le impostazioni necessarie a soddisfare la richiesta, raccolte dal file di configurazione struts-config.xml, nell'elemento *action-mapping*. La conversione da XML a JavaBe-

an è eseguita in fase d'inizializzazione del RequestProcessor tramite la libreria *commons-digester* che si occupa del parsing XML. Una volta localizzati gli elementi *action*, questi vengono memorizzati in un oggetto della classe *ActionMapping*. Nell'esempio precedente, quest'oggetto viene utilizzato per ricavare l'oggetto ActionForward (tramite il metodo *findForward*), contente il path della pagina di risposta e corrispondente al messaggio *"success"* (in caso di autenticazione avvenuta) o al messaggio *"failure"* (autenticazione fallita). L'istanza di *ActionForward* contiene la stringa che verrà confrontata con l'elemento *forward* o *include* del file di configurazione struts-config.xml, determinando così la navigazione al passo successivo. Nell'implementazione di una classe Action bisogna tener conto che la Servlet di controllo crea un'unica istanza di tale servizio per soddisfare tutte le richieste, per questo motivo è necessario creare una classe thread-safe. Per fare ciò, nell'azione, è sufficiente utilizzare solo variabili locali e gestire la sincronizzazione dell'accesso alle risorse. In questo esempio viene anche utilizzata la classe *HTTPSession* per salvare in sessione il dato riferito alla username.

Nell'istanza *form,* della classe custom *LoginForm,* sono contenuti i dati di una richiesta di tipo HTTP POST o GET. Un possibile codice HTML contenuto in una pagina HTML o JSP potrebbe essere il seguente:

```
<form action="/LoginAction" method="post">
    <input type="text" name="username" />
    <input type="text" name="password" />
    <input type="submit" name="submit" />
</form>
```

L'istanza ActionForm, che intende raccogliere i dati della precedente form, deve essere un JavaBean contenente i metodi *get* e *set,* corrispondenti ai nomi dei parametri di input. Nell'esempio seguente, è definita una possibile classe LoginForm che fa riferimento al form HTML dell'esempio precendente.

```
package examples;
import org.apache.struts.action.ActionForm;

public class LoginForm extends ActionForm {

    private String username;
    private String password;

    public void setUsername(String username){
    this.username = username;

    }
    public String getUsername(){
    return username;
    }

    public void setPassword(String password){
    this.password = password;
    }
}
```

La definzione dell'ActionForm all'interno di *struts-config.xml* risulta:

```
…
<form-bean name="login" type="examples.LoginForm" />
…
```

Dopo aver visto alcune delle principali componenti della versione 1 di Struts, con riferimento alla figura 2.4 possiamo definire il flusso esecutivo generato da una richiesta dell'utente:

1. Il client invia una richiesta HTTP.
2. La richiesta è ricevuta dall'ActionServlet (1) che delega il controllo al RequestProcessor (2) della corretta sotto-applicazione.
3. Viene popolato l'ActionForm associato alla richiesta con i dati del form e l'ActionMapping con gli oggetti relativi alla Action associata alla richiesta (4). Tutti i dati di configurazione sono stati letti in fase di startup dell'applicazione (0) dal file struts-config.xml.
4. L'Action si interfaccia con il model che implementa la logica applicativa (6). Al termine dell'elaborazione restituisce al RequestProcessor un ActionForward (7) contenente l'informazione del path della vista da fornire all'utente.
5. Il RequestProcessor restituisce il controllo all'ActionServlet (8) che esegue il forward alla vista (9) specificata dall'ActionForward.

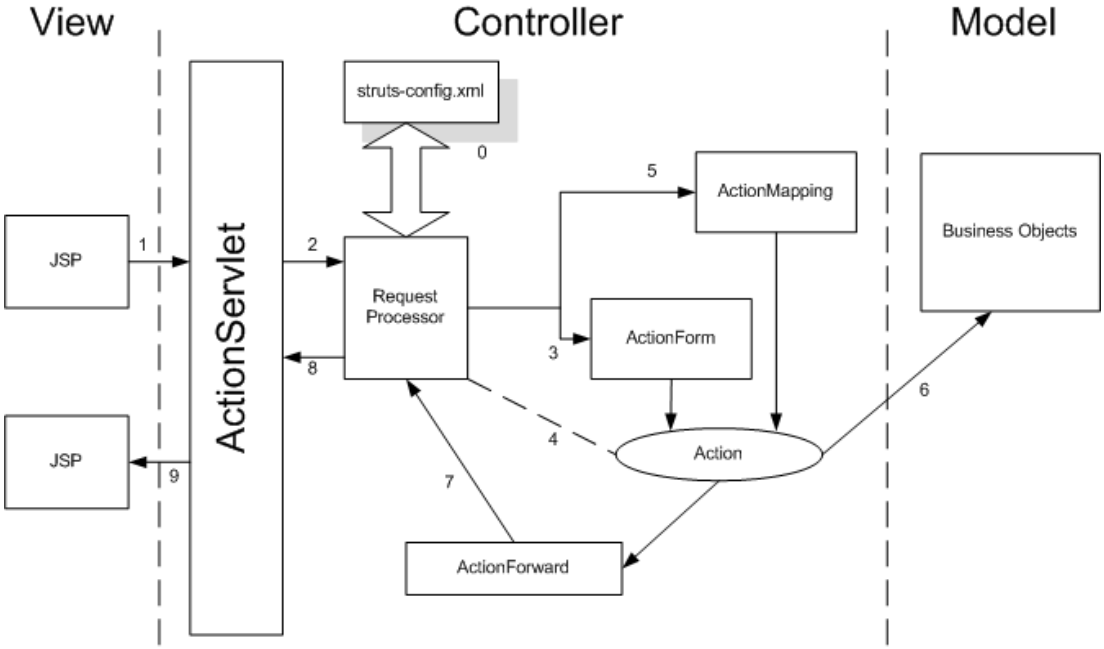

Figura 2.4 Schema del flusso esecutivo a seguito di una richiesta da parte dell'utente

2.7 Localizzazione

Con *localizzazione* s'intende la capacità di un'applicazione di adattare il proprio output con viste aventi caratteristiche nazionali, corrispondenti all'utente che sta effettuando le richieste. Le caratteristiche principali che possono essere interessate sono testi in lingua, date e valute. Poiché molte delle applicazioni web necessitano di una visibilità internazionale, Struts ha introdotto un meccanismo di *internazionalizzazione* che facilita il raggiungimento di tale scopo. La classe interessata è *org.apache.struts.util.MessageResources* che, tramite il metodo *getMessage(Locale locale, String key)*, permette di recuperare una stringa, identificata da un valore chiave e da uno specifico *Locale*. La classe *Locale* fa parte della libreria Java *java.util* e gestisce le impostazioni locali dell'utente, mentre, *MessageResources* estende i servizi della classe *java.util.ResourceBundle* che servono a mappare una chiave con uno specifico messaggio. E' possibile, inoltre, ricavare il messaggio inserendo dei valori passati come argomenti nel metodo *getMessage(Locale locale, String key, Object[] args)*, di modo da creare (tramite la classe *java.text.MessageFormat*) messaggi completi di valori, noti solo al termine dell'esecuzione.

I valori delle chiavi sono contenuti all'interno del file *Resources.properties*, il file di proprietà di Struts. Questo va indicato nel *web.xml*, oppure nello struts-config.xml come rappresentato nell'esempio seguente.

```
<struts-config>
…
<message-resources parameter="examples.ApplicationResources" null="false">
…
</struts-config>
```

Di seguito è rappresentato un possibile file di proprietà *ApplicationResources.properties* che contiene i messaggi nella lingua di default dell'applicazione, in questo caso in italiano.

```
label.header.user = Benvenuto, {0}
label.header.title = Applicazione Web di Esempio
```

L'equivalente file di proprietà *ApplicationResources_en.properties* per i messaggi in lingua inglese sarà:

```
label.heade.user = Welcome, {0}
label.header.title= Web Application Sample
```

Si noti che la sintassi *{0}* riferisce ad un argomento che verrà passato durante l'esecuzione e che completerà il messaggio da visualizzare.

In caso una JSP o un'Action facessero riferimento ad una label mancante delle lingue secondarie, verrà visualizzato il valore della label del file di default, in questo caso *ApplicationResources.properties*; se anche in questo file la label fosse non specificata, all'utente verrà mostrata la stringa "*null"* o "*???key???*" in base alle impostazioni

del parametro *null*, in struts-config.xml. Infine, per riferirsi ad una messaggio contenuto in questi file di properties, in una pagina JSP si utilizza la sintassi:

```
<bean:message key="label.header.title"/>
<bean:message key="label.header.user" arg0=”Anonymous” />
```

Mentre in una Servlet si utilizza il codice:

```
MessageResources messageResources = getResources(request);
messageResources.getMessage(request.getLocale(),"label.header.title");
//con passaggio del parametro
messageResources.getMessage(request.getLocale(),"label.header.user","
Anonymous”);
```

2.8 Tag Library

Tra le funzionalità messe a disposizione da Struts vi sono anche alcune librerie, le *struts-taglib*, le quali contegono file in formato xml con estensione *.tld,* che definiscono i tag utilizzabili all'interno delle pagine JSP. L'utilizzo di queste librerie permettono un più rapido sviluppo della View, rendendola più comprensibile e pulita. Le principali librerie di tag sono:

- *struts-bean.tld*
- *struts-form.tld*
- *struts-html.tld*
- *struts-logic.tld*
- *struts-tiles.tld*
- *struts-template.tld*

Ogni qual volta si voglia utilizzare delle librerie di tag nelle proprie applicazioni web, vanno inseriti i riferimenti nel file di configurazione dell'applicazione, il web.xml. Il codice da inserire sarà:

```
…
<taglib>
 <taglib-uri>struts-bean</taglib-uri>
 <taglib-location>/WEB-INF/taglib/struts-bean.tld</taglib-location>
</taglib>
…
```

Inoltre, la libreria deve essere importata all'interno della pagina JSP in questo modo:

```
<%@ taglib uri="/WEB-INF/taglib/struts-html.tld " prefix="html" %>
<%@ taglib uri="/WEB-INF/taglib/struts-bean.tld " prefix="bean" %>
<%@ taglib uri="/WEB-INF/taglib/struts-logic.tld" prefix="logic" %>
```

E' possibile utilizzare librerie personalizzate, definendone di proprie con estensione *.tld*.

Il primo esempio di tag riguarda la famiglia bean, *bean:message*.

```
<bean:message key="label.header.user">
```

Questo tag permette di inserire un testo gestito tramite localizzazione, che potrà essere completato dinamicamente con uno o più parametri.

Un altro tag è *bean:define*, con il quale si indicano le istanze di un bean da utilizzare nella propria pagina JSP

```
<bean:define id="userlogin"
             name="user.key"
             property="user"
             type="examples.beans.User"
             scope="request" />
```

Nell'esempio precedente si rende disponibile un oggetto identificato da *user* della classe *examples.beans.User* e ricavato dalla proprietà *userlogin* contenuta nell'oggetto associato all'attributo *user.key*. L'oggetto avrà validità (scope) solo per la richiesta corrente.

Un ultimo esempio di tag bean è *bean:write.* Permette di trascrivere direttamente il contenuto di una proprietà di un bean.

```
<bean:write name="userlogin" property="user">
```

Un'altra famiglia di tag è quella con radice "*html*" che è stata introdotta per poter meglio integrare gli oggetti html utilizzati dal framework, in particolare la scrittura di form associati alle relative ActionForm e Action. Si noti che questa libreria sostituisce i principali tag HTML standard. In particolare, il tag *html:link* permette di accedere alla configurazione struts-config.xml per inserire il valore effettivo dell'URL; gli attributi che permettono di fare ciò sono: *forward* (si riferisce all'elemento *global-forward*), *action* (si riferisce all'elemento action) e *href* (si indica un normale url di una risorsa, anche esterna).

Tag Struts	Equivalente sintassi HTML
<html:html>	<html>
<html:link>	<a href="">
<html:form>	<form>
<html:option>	<option>
<html:text>	<input type="text">
<html:textarea>	<textarea>
<html:select>	<input type="select">
<html:submit>	<input type="submit">
<html:button>	<input type="button"
<html:checkbox>	<input type="checkbox">

Sintassi dei tag html Struts e differenze con lo standard HTML

La libreria di tag "*logic"* fornisce un'estensione per la gestione logica della presentazione. Di seguito è rappresentato un esempio del tag *logic:present* che verifica il con-

tenuto dell'oggetto associato ed eventualmente esegue delle operazioni, una sorta di esecuzione condizionata.

```
<logic:present name="userlogin" property="user">
Benvenuto,
<bean:message name="userlogin" property="user">
</logic:present>
```

Nell'esempio precedente, il messaggio di benvenuto verrà visualizzato solo se l'oggetto "userlogin" è valorizzato.

Un altro tag importante è *logic:iterate*: come si può intuire questo elemento permette di effettuare iterazioni del codice contenuto al suo interno, per ogni dato presente nell'oggetto a cui fa riferimento. Ne è un esempio il codice seguente:

```
<p>Lista libri</p>
<logic:iterate id="book" type="examples.Book" name="books">
- <bean:write name="book" property="name" /><br />
</logic:iterate>
```

Vi sono altri tag di tipo logic che si comportano in modo simile ai precedenti, implementando alcune delle più frequenti istruzioni condizionate:

- *logic:notPresent*
- *logic:Match, logic:notMatch*
- *logic:Empty, logi:notEmpty*

Questi si comportano similmente al primo esempio visto, mentre i tag

- *logic:greaterThan, logic:greaterEqual*
- *logic:lessThan, logic:lessEqual*

si utilizzano per valutare oggetti contenenti numeri o stringhe.

La libreria con prefisso "*tiles*", offre la possibilità di definire un modello per le proprie pagine JSP, al fine di poter riutilizzare le parti di codice che si ripetono in più punti della View. La struttura della pagina, ovvero l'*astrazione* di questa, è definita in un file XML, che funziona similmente ad un template. Per poter utilizzare questa libreria è necessario, oltre l'importazione nelle pagine JSP, inserire un elemento *plug-in* nello *struts-config* del modulo web:

```
<plug-in className="org.apache.struts.tiles.TilesPlugin">
<set-property property="definitions-config" value="tiles-config" />
</plug-in>
```

Il file XML permette di definire le componenti principali, costituenti una pagina in una struttura di base e lasciando la specializzazione di ogni singola componente ad un ulteriore elemento del file di configurazione dei tiles, detta *specializzazione*. Un esempio classico di strutturazione di una pagina è mostrato nell'esempio seguente, che è parte del file xml di configurazione:

```
<tiles-definitions>
<definition name=”layout.page” path=”layout.jsp”>
    <put name="title" type="string">Layout</put>
    <put name="header" type="page">/WEB-INF/jsp/tiles/header.jsp</put>
    <put name="footer" type="page">/WEB-INF/jsp/tiles/footer.jsp</put>
    <put name="menu" type="page">/WEB-INF/jsp/tiles/menu.jsp</put>
    <put name="content" type="page">/WEB-INF/jsp/tiles/content.jsp</put>
</definition>
…
</tiles-definitions>
```

Una possibile specializzazione di una pagina che estende l'esempio precedente potrebbe essere:

```
<definition name=”bookSearch.page” extends=”layout.page”>
<put name=”title” type=”string”>Ricerca Libro</put>
<put name=”content” type=”page”>/WEB-INF/jsp/tiles/book-search.jsp</put>
</definition>
```

Si noti che tale definizione della pagina eredita le componenti header.jsp, footer.jsp e menu.jsp e sovrascrive i contenuti che riguardano il titolo (definendone uno nuovo) e il contenuto principale (book-search.jsp). Infine è possibile utilizzare i tiles nel path di un'action nel file struts-config.xml:

```
…
<action path=”book-search” type=”examples.SearchBookAction”>
<forward name=”success” path=”bookSearch.page” />
</action>
…
```

In questo modo, in caso di successo, l'azione passerà il controllo alla pagina rappresentata dal tiles *bookSearch.page*, che conterrà la definizione di specializzazione, ma anche gli elementi di base non sovrascritti (*layout.page*).

2.9 Alternative

Le alternative al framework Struts 1 sono diverse, tra cui vi è Struts 2 che presenta notevoli differenze rispetto la prima versione e che rende difficile l'upgrade. Struts 2 oltre a migliorare, sia in efficienza sia in efficacia le funzionalità della prima versione, possiede un meccanismo di controllo manuale dell'esecuzione delle Action; la flessibilità del linguaggio espressivo di JSP è migliorata grazie all'uso della tecnologia OGNL[6]. Nella tabella seguente sono riassunte le principali differenze tra i due framework.

[6] (Object-Graph Navigation Language) Liguaggio opensource utilizzato soprattutto nelle tag library dello standard JEE che consente una facile manipolazione delle proprietà di oggetti Java.

Struts 1.x	Struts 2.x
Il controllo è delegato ad una Servlet detta ActionServlet	Il controllo è delegato ad un filtro[7] detto FilterDispatcher
Le azioni sono ottenute mediante estensione della classe Action	Le azioni sono ottenute implementando l'interfaccia Action
Le Action sono dei singleton. Devono essere implementate in maniera thread-safe poiché esiste una sola istanza che si occupa di gestire tutte le richieste.	Le Action sono istanziate una per ogni richiesta, perciò non vi sono problematiche legate alla gestione dei thread.
Le proprietà di interesse vengono manipolate tramite un oggetto della classe ActionForm, nella quale è necessario definire i metodi get e set.	Le proprietà di interesse vengono manipolate direttamente nelle classi Action, nelle quali è necessario definire i metodi get e set.
Il metodo execute di un action restituisce un oggetto di tipo ActionForward	Il metodo execute di una action restituisce un oggetto String
Requisiti necessari: Servlet API 2.3, Java 4 o superiore, JSP API 2.0	Requisiti: Servlet API 2.4, JSP API 2.0, Java 5 o superiore

Principali differenze tra Struts 1 e Struts 2

La tecnologia JavaServer Faces e il framework Spring rappresentano le più famose alternative a Struts, per lo sviluppo di applicazioni web. La prima, è rappresentata da un insieme di specifiche create da Sun Microsystems e dall'ideatore di Struts. JSF è basata anch'essa sul paradigma Model-View-Controller ed è stata inclusa in forma nativa nelle specifiche JEE, dalla versione 5. Facendo parte dello standard, questa tecnologia è ampiamente supportata, godendo di una documentazione ufficiale dettagliata. Diverse società hanno proposto la loro implementazione delle specifiche, tra cui Apache, con il nome MyFaces, Oracle (ADF Faces[8]) e Sun. Oltre a quelle già citate, le principali caratteristiche sono:

- integrazione con la tecnologia Unifield Expression Language (EL): l'accesso ai bean agilmente manipolati da questo potente metodo espressivo;
- un sistema integrato per la creazione di template, simile alle Tiles di Struts;
- controllo centralizzato del caricamento delle viste;
- un sistema per la gestione dell'internazionalizzazione dell'applicazione;
- supporto server-side del paradigma di *programmazione ad eventi*;

[7] Implementa l'interfaccia Filter. Similmente ad una Servlet possiede i metodi init, a cui viene passata la configurazione di inizializzazione, e il metodo doFilter che fornisce il servizio chiamato dal container ogni qual volta una richiesta tenta di accedere ad una risorsa. Il metodo doFilter accetta anche una parametro della classe FilterChain che permette al filtro di passare il contesto di richiesta/risposta all'elemento successivo della catena. Analogamente ad una Servlet è presente un metodo destroy chiamato dal container al per dismettere il servizio.

[8] Application Develpment Framework Faces

- fornisce librerie di tag in sintassi XML, anche personalizzabili;

Il framework Spring, invece, si è reso celebre per la sua pragmatica adattabilità a risolvere una larga gamma di problematiche comuni. È basato sulla tecnica dell'*inversion of Control*[9] sul paradigma di *programmazione orientata agli aspetti*[10]. Spring fornisce anch'esso un'implementazione del pattern MVC, maggiormente flessibile, tale da poter facilitare l'integrazione con altri framework.

Questo framework si è rivelato, inoltre, particolarmente adatto ad applicazioni di tipo enterprise, poiché è riuscito ad integrare modelli di persistenza sia standard (EJB) che non (*framework Hibernate*).

[9] Pattern di programmazione che prevede il disaccoppiamento di tutte le componenti di un sistema, le cui dipendenze sono gestite esternamente da una terza entità, un ambiente che gestisce le invocazioni e l'istanziazione. La tecnica di Dependency Injection implementa questa politica e permette slegare il codice applicativo con la logica d'inizializzazione.

[10] Può estedendere la programmazione ad oggetti, fornendo elementi e funzioni comuni a più oggetti, diminuendo la ridondanza di codice e la aumentando la flessibilità dell'applicazione.

3 Progetto “Startup”

In questo capitolo saranno presentati gli aspetti progettuali e implementativi riguardanti lo sviluppo di un’applicazione web, all’interno del progetto “Startup”. L’attività ha messo in opera le tecnologie e gli strumenti discussi nei precedenti capitoli ed è stata suddivisa in quattro fasi: raccolta e analisi dei requisiti, progettazione, implementazione e deployment. All’interno di questo capitolo sono stati raccolti i risultati più importanti che riguardano l’applicazione sviluppata.

3.1 Requisiti

Il progetto in esame riguarda lo sviluppo di un’applicazione accessibile dagli utenti tramite browser web, che gestisca il caricamento delle consuntivazioni ore dei dipendenti, in una base dati preesistente. Nel dettaglio, l’applicazione deve consentire ai dipendenti dell’azienda:

1. la dichiarazione delle giornate e delle ore lavorative effettuate, per un determinato mese;
2. consultazione delle comunicazioni di consuntivazione precedentemente inviate.

I mesi di riferimento per la dichiarazione potranno fare riferimento a tutto l’arco dell’anno solare precedente alla data di comunicazione, fino al mese corrente. Stesse considerazioni valgono per la consultazione delle comunicazioni.
L’applicazione dovrà tener conto della multi-nazionalità dei dipendenti dell’azienda, perciò si dovrà adottare una modalità efficiente di gestione dell’internazionalizzazione. La base di dati di riferimento è gestita tramite il DBMS PostgreSQL. Le informazioni d’interesse sono riportate in figura 3.1.

Figura 3.1 Schema della base di dati

Ogni dipendente della base dati è identificato dal suo numero di matricola; le comunicazioni ore, chiamate anche "fogli ora", sono identificati dalla matricola del dipendente che le ha effettuate, dal mese e dall'anno di riferimento. Come ultimo requisito tecnico, l'applicazione dovrà essere integrata al WebServer Apache Tomcat versione 5.5 e potrà essere supportata dalla versione Java 1.6.

3.2 Progettazione

Dopo l'analisi dei requisiti si è deciso di utilizzare il pattern MVC, presentato nel paragrafo 2.2. A supporto dello sviluppo si è utilizzato il framework Struts al fine di rendere ottimale la gestione dell'internazionalizzazione, garantendo brevi tempi per

il rilascio di una versione funzionante e assicurando una certa flessibilità per eventuali estensioni future. In questo caso, la componente "View" dell'applicazione sarà esclusivamente composta da pagine JSP, mentre il controllo del flusso di navigazione sarà affidato all'ActionServlet del framework utilizzato. Infine, trovandoci in una situazione di bassa complessità, non sarà necessario affiancare alcun EJB Container o altri sistemi di gestione della persistenza, lasciando il ruolo di "model" al DBMS.

In figura 3.2 è riportato lo schema di navigazione che l'utente potrà seguire all'interno dell'applicazione:

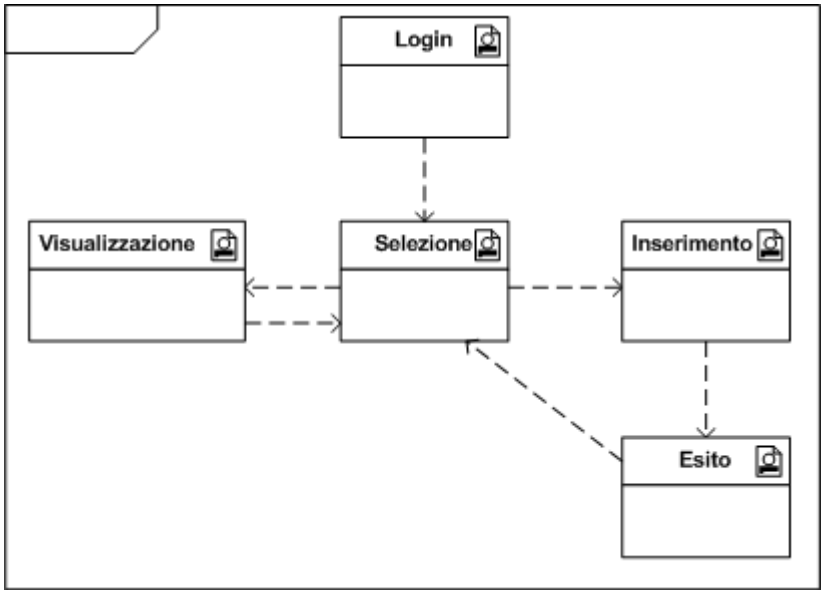

Figura 3.2 Schema di navigazione all'interno dell'applicazione

La pagina iniziale contiene il form per l'autenticazione. Qui vengono richiesti username e password per l'accesso alle funzionalità del programma. Dopo l'autenticazione, l'utente può selezionare da un menu un determinato mese e anno (con i vincoli imposti dai requisiti) e proseguire verso due direzioni:

- Se ha selezionato un mese per il quale ha già inviato la comunicazione, viene visualizzato il dettaglio (e la possibilità di tornare al menu precedente).
- Se ha selezionato un mese per il quale non ha inviato una comunicazione, sarà indirizzato verso la pagina che conterrà il form d'inserimento e invio delle ore lavorative svolte, che a sua volta ridirezionerà verso una pagina di esito d'inserimento, con la possibilità di ritorno al menu principale.

Saranno ora trattati in dettaglio gli aspetti progettuali legati allo specifico pattern e al framework di sviluppo utilizzati, facendo riferimento allo schema precedente.

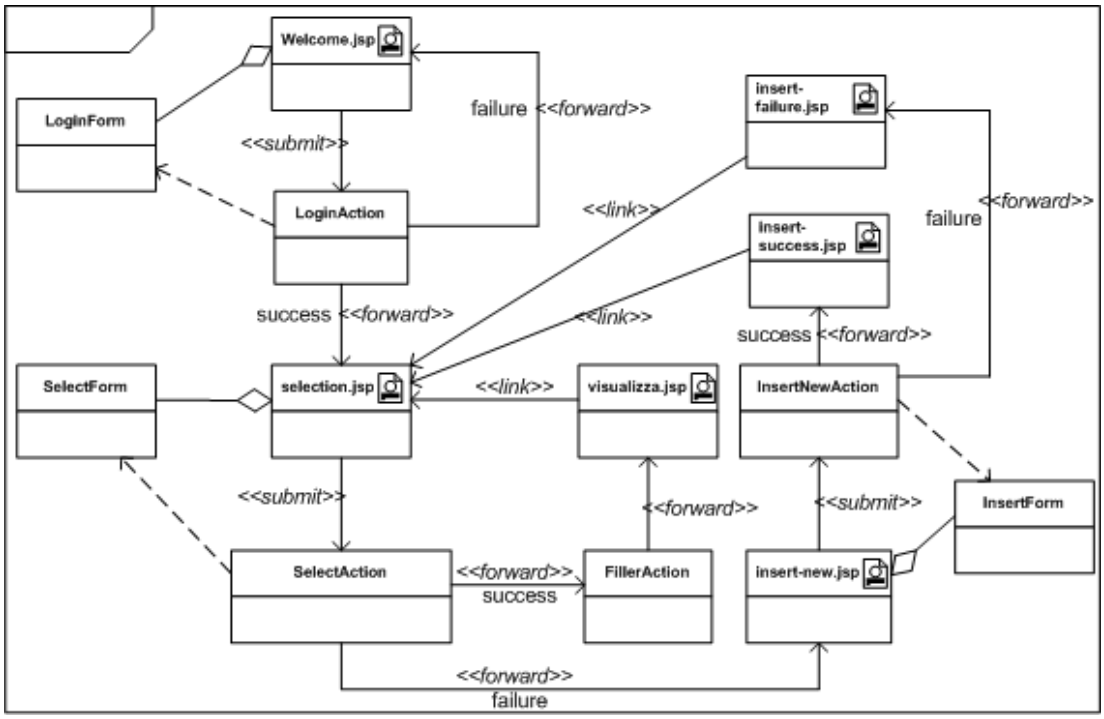

Figura 3.3 Schema delle azioni, dei bean e delle pagine coinvolte

La classe *LoginAction* si occuperà di controllare, mediante un'interrogazione al database, le credenziali immesse dall'utente, che potranno essere recuperate dal bean *LoginForm*. In condizione di successo, l'utente verrà ridirezionato alla pagina *selection.jsp* contenente il menu di selezione, in caso contrario tornerà alla pagina di login con un messaggio di errore. L'azione *SelectAction* esaminerà le informazioni selezionate attraverso il bean *SelectForm*. Se già presente al database un foglio ore corrispontendente alla selezione, un'altra azione, *FillerAction*, si farà carico di estrarre il dettaglio della comunicazione dalla base dati e mostrarlo attraverso la pagina *visualizza.jsp*. Qualora non fosse presente alcun foglio ore per il mese e l'anno selezionati, l'utente andrà inserire il nuovo dettaglio nella pagina *insert-new.jsp*. Infine, l'azione InsertNewAction raccoglie il dettaglio compilato dall'utente dal bean InsertForm ed esegue l'inserimento nel database. Dopo l'inserimento verrà visualizzata la pagina di esito insert-success.jsp se tutto è andato a buon fine, insert-failure.jsp in caso contrario.

Per aumentare le prestazioni delle operazioni di connessione al DBMS, saranno utilizzate alcune funzionalità messe a disposizione dal Webserver Tomcat e dalle tecnologie JNDI (§1.6) e JDBC (§1.5), permettendo di instaurare un pool di connessioni. Questa modalità consente di ottenere una connessione alla base dati tramite un DataSource messo a disposizione dal Webserver, anziché utilizzando il DriverManager (§1.5). L'istanza del datasource viene ottenuta attraverso il servizio *"lookup"* di JNDI.

3.3 Implementazione

Per prima cosa saranno analizzate le interrogazioni e gli aggiornamenti che interessano la base dati.

- Controllo delle credenziali di accesso:

```
SELECT nome,
cognome
FROM dipendente
WHERE login = '000000'
AND password = 'abcde'
```

- Interrogazione per verificare la presenza di una comunicazione per una determinata selezione:

```
SELECT 1 as present
FROM dipendente NATURAL JOIN foglio_ore
WHERE login = '000000'
AND mese = 1
AND anno = 2010
```

- Recupero della matricola del dipendente dalla username

```
SELECT matricola
FROM dipendente
WHERE login = '000000'
```

- Recupero del dettaglio della comunicazione, dati mese, anno e matricola

```
SELECT
ore_giorno1, ore_giorno2, ore_giorno3, ore_giorno4, ore_giorno5,
ore_giorno6, ore_giorno7, ore_giorno8, ore_giorno9, ore_giorno10,
ore_giorno11, ore_giorno12, ore_giorno13, ore_giorno14, ore_giorno15,
ore_giorno16, ore_giorno17, ore_giorno18, ore_giorno19, ore_giorno20,
ore_giorno21, ore_giorno22, ore_giorno23, ore_giorno24, ore_giorno25,
ore_giorno26, ore_giorno27, ore_giorno28, ore_giorno29, ore_giorno30,
ore_giorno31,
totale_ore
FROM foglio_ore
WHERE matricola = '000000'
AND anno = 2010
AND mese = 1
```

- Inserimento di una nuova comunicazione

```
INSERT INTO foglio_ore (ore_giorno1, ore_giorno2, ore_giorno3, ore_giorno4,
ore_giorno5, ore_giorno6, ore_giorno7, ore_giorno8, ore_giorno9,
ore_giorno10, ore_giorno11, ore_giorno12, ore_giorno13, ore_giorno14,
ore_giorno15, ore_giorno16, ore_giorno17, ore_giorno18, ore_giorno19,
ore_giorno20, ore_giorno21, ore_giorno22, ore_giorno23, ore_giorno24,
ore_giorno25, ore_giorno26, ore_giorno27, ore_giorno28, ore_giorno29,
ore_giorno30, ore_giorno31,totale_ore, data_compilazione
) values(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
CURRENT_DATE)
```

Per poter utilizzare il pool di connessioni è necessario creare un file di configurazione xml, chiamato *context.xml*, che il Webserver Tomcat leggerà automaticamente in fase di inizializzazione dell'applicazione. Qui compaiono i parametri d'inizializzazione delle connessioni tramite *DataSource*, i driver JDBC da utilizzare e altre configurazioni delle risorse dell'applicativo. Il file deve essere contenuto nella cartella *META-INF* del progetto, mentre i driver JDBC devono essere inseriti nella cartella delle librerie del Webserver (anziché nel path delle librerie di progetto). Di seguito è riportato il suo contenuto.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/foglio_ore">
<Resource name="jdbc/postgres" auth="Container"
        maxActive="100" maxIdle="30" maxWait="10000"
        username="webuser" password="webuser"
        driverClassName="org.postgresql.Driver"
        url="jdbc:postgresql://localhost:5432/postgres"
        logAbandoned="true" removeAbandoned="true"
        removeAbandonedTimeout="60"
        type="javax.sql.DataSource" />
        </Context>
```

La classe che si occupa di implementare il pool di connessione si chiama *ConnectionPool*. In questa classe è gestito, in modo thread-safe, l'utilizzo del pool di connessioni e contiene i metodi per ottenere una connessione dal pool e per rilasciare la risorsa.

```
package it.aive.web.Struts;

import java.sql.Connection;
import java.sql.SQLException;
import javax.naming.InitialContext;
import javax.sql.DataSource;

public class ConnectionPool {
    private static DataSource dataSource = null;
    private ConnectionPool(){
        try { InitialContext ic = new InitialContext();
             dataSource =
            (DataSource)ic.lookup("java:/comp/env/jdbc/postgres");
         }catch (Exception ex)
         {ex.printStackTrace();}
    }
    private static class ConnectionPoolHolder {
        private final static ConnectionPool pool = new ConnectionPool();
    }
    public static ConnectionPool getInstance(){
        return ConnectionPoolHolder.pool;
    }
    public Connection getConnection()
    {  try { return dataSource.getConnection();}
       catch (SQLException ex)
        { ex.printStackTrace();
          return null;
        }
    }
    public void freeConnection(Connection c)
    { try{if(c != null) {
```

```
            c.close();
          }
        }
        catch (SQLException ex)
        {ex.printStackTrace();
        }
    }
}
```

Il deployment descriptor, il file *web.xml* (§1.8), conterrà solo le informazioni principali di default, lasciando il dettaglio della configurazione al file *struts-config.xml*. Qui va definita la Servlet di controllo *ActionServlet*.

```
<?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>Foglio Ore - Aive S.p.A.</display-name>

  <!-- Standard Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
 </servlet>

  <!-- Standard Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- The Usual Welcome File List -->
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Vediamo ora il cuore dell'applicazione, il file di configurazione *struts-config.xml* (§2.5). In questo file è facilmente configurabile tutta la logica di controllo dell'applicazione. Di seguito è riportata la sua implementazione per il problema in esame.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
          "http://struts.apache.org/dtds/struts-config_1_3.dtd">

<struts-config>
```

```
<!-- =================Form Bean Definitions================== -->

<form-beans>
    <form-bean name="LoginForm" type="it.aive.web.Struts.LoginForm" />
    <form-bean name="SelectForm" type="it.aive.web.Struts.SelectForm" />
    <form-bean name="InsertNewForm"
    type="it.aive.web.Struts.InsertNewForm"/>
</form-beans>

<!-- ========== Global Exception Definitions ================= -->

<global-exceptions></global-exceptions>

<!-- ========== Global Forward Definitions =================== -->

<global-forwards>
<!-- Ogni volta che si tenterà di accedere alla risorsa "welcome" verrà
intrapresa l'azione Welcome.do (reinderizzamento alla pagina
/pages/Welcome.jsp) -->
<forward name="welcome" path="/Welcome.do"/>
</global-forwards>

<!-- ========== Action Mapping Definitions ================== -->
<action-mappings>

<action path="/Welcome" forward="/pages/Welcome.jsp"/>

<action
    path="/login"
    type="it.aive.web.Struts.LoginAction"
    name="LoginForm"
    scope="session"
    validate="true"
    input="/pages/Welcome.jsp">
    <forward name="success" path="/pages/success.jsp"/>
    <forward name="failure" path="/pages/Welcome.jsp"/>
</action>

<action
    path="/select"
    type="it.aive.web.Struts.SelectAction"
    name="SelectForm"
    scope="session"
    validate="false"
    input="/pages/selection.jsp">
    <forward name="success" path="/filler.do"/>
    <forward name="failure" path="/pages/insert-new.jsp"/>
</action>

<action
    path="/filler"
    type="it.aive.web.Struts.FillerAction"
    name="SelectForm"
    scope="request"
    validate="true">
    <forward name="success" path="/pages/visualizza-ore.jsp"/>
</action>

<action
    path="/insertnew"
    type="it.aive.web.Struts.InsertNewAction"
```

```
    name="InsertNewForm"
    scope="request"
    validate="true"
    input="/pages/insert-new.jsp">
    <forward name="success" path="/pages/insert-success.jsp"/>
    <forward name="failure" path="/pages/insert-failure.jsp"/>
</action>

<action path="/success-insert" forward="/pages/selection.jsp"/>
<action path="/failure-insert" forward="/pages/selection.jsp"/>

</action-mappings>

<!-- ========== Message Resources Definitions =========-->

<message-resources parameter="MessageResources" null="false" />

<!-- ========== Plug Ins Configuration ================-->
<!-- ========== Validator plugin ======================-->
<!-- Validatore di default del framwork -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property
    property="pathnames"
    value="/org/apache/struts/validator/validator-rules.xml,
        /WEB-INF/validation.xml"/>
</plug-in>

</struts-config>
```

Si noti che, nel precedente codice, è stata inserita la dichiarazione di un validatore che si riferisce al file *validation.xml*. In questa componente è possibile definire, in modo facile e rapido, i parametri di validazione dei campi di un form: se la validazione da' esito negativo, sarà riportato un messaggio di errore alla pagina contente il form. Di seguito è riportato un esempio di configurazione del plugin per il form di login:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE form-validation PUBLIC
         "-//Apache Software Foundation//DTD Commons Validator Rules
          Configuration 1.3.0//EN"
     "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
    <global></global>
    <formset>

        <form name="LoginForm">
            <field
                property="username"
                depends="required">
                <arg key="LoginForm.username"/>
            </field>
            <field
                property="password"
                depends="required,mask">
                <arg key="LoginForm.password"/>
                <var>
                    <var-name>mask</var-name>
```

```
                <var-value>^[0-9a-zA-Z]*$</var-value>
            </var>
        </field>

    </form>

</form-validation>
```

In alternativa a questa modalità, è possibile effettuare una validazione dei campi all'interno del metodo sovrascritto *validate()* del bean associato al form. Nell'esempio seguente è riportato il contenuto del bean LoginForm.

```
package it.aive.web.Struts;
import org.apache.struts.action.*;
import javax.servlet.http.HttpServletRequest;

public class LoginForm extends ActionForm
{
    private String username=null;
    private String password=null;

    public void setUsername(String username){
        this.username=username;
    }

    public String getUsername(){
        return this.username;
    }

    public void setPassword(String password){
       this.password=password;
    }

    public String getPassword(){
       return this.password;
    }
    public void reset(ActionMapping mapping,HttpServletRequest request){
        username=null;
        password=null;
    }
    public ActionErrors validate(ActionMapping mapping, HttpServletRequest
    request){
        ActionErrors errors = new ActionErrors();
        if(request.getSession().getAttribute("username") == null) {
           if(username == null || username.length() > 32 ||
              username.length() == 0 || password == null ||
              password.length() > 32 || password.length() == 0) {
              errors.add("username",new ActionMessage("welcome.errors"));
           }
        }
           return errors;
    }
}
```

Il metodo *validate(),* restituisce un oggetto di tipo *ActionError* che potrà essere catturato dalla pagina chiamante tramite un apposito tag delle librerie Struts, che si incari-

ca di stampare tutti i messaggi di errore contenuti nell'oggetto restituito. La pagina iniziale Welcome.jsp risulterà:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
<html:html>
<head>
<title><bean:message key="welcome.title"/></title>
<html:base/>
</head>
<body bgcolor="white">

<logic:notPresent name="org.apache.struts.action.MESSAGE"
scope="application">
  <font color="red">
    ERROR:  Application resources not loaded -- check servlet container
    logs for error messages.
  </font>
</logic:notPresent>

<h3><bean:message key="welcome.heading"/></h3>

<logic:equal name="login" value="failure">
  <span style="color:red"><bean:message key="welcome.loginfail.label" />
  </span>
</logic:equal>

<html:errors />

<logic:notPresent name="username">
<html:form action="/login" method="post">
Username: <html:text property="username" size="30" maxlength="32"/><br />
Password: <html:password property="password" size="30" maxlength="32"/><br
/>
<html:submit><bean:message key="welcome.button.login" /> </html:submit>
</html:form>
</logic:notPresent>

<logic:present name="username" parameter="username">
<p><b><bean:message key="welcome.loginsucc.label"
arg0="/foglio_ore/login.do" />
</b></p>
</logic:present>

</body>
</html:html>
```

Tutte le etichette di testo sono contenute nei file *MessageResources.propertie*s (in lingua inglese) e *MessageResources_it.properties* (in lingua italiana); il secondo, anche se non esplicitamente indicato nel file di configurazione *struts-config.xml,* sarà automaticamente caricato nel caso l'utente abbia localizzazione italiana. Nell'esempio seguente è riportato una parte del file di risorse italiano.

```
# -- Welcome page
welcome.title=Azienda S.p.A - Compilazione Foglio Ore
welcome.button.login=Entra
```

```
welcome.heading=Consuntivo Ore
welcome.loginfail.label=Username o password non corretti. Riprovare o
contattare l'<a href="mailto:test@test.com">amministratore del sistema</a>
welcome.loginsucc.label=Hai già effettuato il login,
    <a href="{0}">clicca qui</a> per rientrare.
welcome.errors=Il formato dei campi Username/password non sono corretti.

# -- Select Month/Year Page --
select.welcome.label=Benvenuto, {0}
select.username.label=La tua username e' {0}
select.selectmessage.label=Seleziona una preferenza:
select.month.label=Mese
select.year.label=Anno
```

Infine, viene presentata una classe azione, LoginAction, nella quale viene sovrascritto il metodo *execute* della superclasse Action. Per l'esecuzione delle interrogazioni e gli aggiornamenti della base dati è stata utilizzata la classe *PrepareStatement*, uno statement SQL parametrizzato che permette al DMBS di ottimizzare le interrogazioni successive.

```
package it.aive.web.Struts;

[…]

public class LoginAction extends Action
{
    public ActionForward execute( ActionMapping mapping,
                                  ActionForm form,
                                  HttpServletRequest request,
                                  HttpServletResponse response) throws
                                                          Exception{


    LoginForm loginform = (LoginForm) form; //Bean del form di richiesta
    // se in sessione è già presente username e password
    // non faccio la ricerca nel database
    HttpSession session = request.getSession();
    String session_username=(String) session.getAttribute("username");
    if(session_username != null ) {
        return mapping.findForward("success");
    } else {
    // verifico username e password nel db
    // Connection Pool
    ConnectionPool pool = ConnectionPool.getInstance();

    //Ottengo una connessione dal pool
    Connection conn = pool.getConnection();


    PreparedStatement ps = null;
    ResultSet rs = null;

    String query = "SELECT cognome, nome FROM dipendente WHERE login = ? " +
                   "AND password = ?"; // i punti domanda verranno
                   sostituiti con i valori effettivi con gli appositi metodi
                   setType(int numeroCampo, type valore)
    //creazione del dello statement precompilato
    ps=conn.prepareStatement(query);
```

```
    ps.setString(1, loginform.getUsername());
    ps.setString(2, loginform.getPassword());
    rs = ps.executeQuery();

    if(rs.next()){ //se il ResultSet non è vuoto
        String nome = rs.getString("nome") + " " +  rs.getString("cognome");

        // Pongo in sessione alcune oggetti che saranno utilizzate
        // successivamente
        session.setAttribute("nome", nome);
        session.setAttribute("username",loginform.getUsername());
        session.setAttribute("password",loginform.getPassword());
        request.setAttribute("login", "success");

       //Rilascio della connessione
        pool.freeConnection(conn);
        //Ritorno il controllo al Request Processor con il messaggio SUCCESS
       return mapping.findForward("success");
    } else {
    request.setAttribute("login", "failure");
    // Svuoto i campi compilati
    loginform.reset(mapping, request);
     pool.freeConnection(conn);
   //Ritorno il controllo al Request Processor con il messaggio FAILURE
     return mapping.findForward("failure");
    }
  }//else -- username/password not in session
}//execute method
}//LoginAction class
```

3.4 Deployment

Questa operazione è fondamentale per rendere disponibile e utilizzabile l'applicazione. Per fare ciò sono presenti alcuni strumenti che automatizzano le operazioni di compilazione, creazione delle directory e del pacchetto compresso dell'applicazione (WAR, §1.8). Per questo progetto si è deciso di utilizzare l'utility Apache Ant, la quale si trova integrata anche all'interno dell'ambiente di sviluppo Eclipse, l'IDE (Integrated development enviroment) utilizzato per l'implementazione dell'intero progetto. Come già detto, Ant è un tool per l'automatizzazione del processo di build dell'applicazione; differentemente all'utility MAKE in ambiente Unix, Ant è principalmente orientato allo sviluppo in ambiente Java. La principale caratteristica di questa applicazione è la portabilità; lo sviluppatore, infatti, può utilizzare questo strumento in ogni ambiente di sviluppo e con una particolare flessibilità sulla sintassi dei path utilizzati. Attraverso un semplice script in formato XML è possibile dare tutte le indicazioni necessarie per effettuare le operazioni di build. La cartella di lavoro del progetto si presenta come esposto di seguito:

```
+-- foglio_ore [DIR]
       |-- src [DIR]
            |-- it [DIR]
                |-- azienda [DIR]
```

```
                    |-- web [DIR]
                        |--Struts [DIR]
                              |-- file sorgenti java del progetto
                              |-- MessageResources_it.properties
                              |-- MessageResources.properties
      |-- Web-Content [DIR]
              |-- index.jsp [questa pagina contiene
                            solo un ridirezionamento]
              |-- META-INF [DIR]
                        |-- context.xml
              |-- pages [DIR]
                        |-- [pagine jsp]
                        |-- [fogli di stile]
              |-- WEB-INF [DIR]
                        |-- build.xml [script Ant]
                        |-- struts-config.xml
                        |-- validation.xml
                        |-- web.xml
                        |-- lib [DIR]
                             |-- [librerie di progetto]
```

A partire da questa situazione, tramite lo script Ant *build.xml*, è necessario generare un pacchetto war che contenga la struttura di directory seguente.

```
+--foglio_ore [DIR]
        |-- META-INF [DIR]
               |-- context.xml
        |-- pages [DIR]
        |-- WEB-INF
               |-- index.jsp
               |-- struts-config.xml
               |-- validate.xml
               |-- web.xml
               |-- build.xml
               |-- lib
               |-- classes [DIR]
                      |-- it [DIR]
                          |-- azienda [DIR]
                               |-- web [DIR]
                                   |--Struts [DIR]
                                       |-- file binari
                                       |-- MessageResources_it.properties
                                       |-- MessageResources.properties
```

Per prima cosa è necessario settare il path assoluto con il quale Ant creerà i percorsi relativi. Questo può essere fatto inserendo una variabile d'ambiente nell'IDE di sviluppo.

Per iniziare, definiamo l'elemento principale dello script che specifica il nome del progetto, path di partenza per il calcolo dei riferimenti e il target di default. Ogni altro elemento sarà contenuto in esso.

```
<project name="foglio_ore" basedir="../" default="all">
[…]
</project>
```

Il target è una lista di operazioni da eseguire per ottenere una specifica funzionalità; ogni qual volta si vorrà utilizzare un target sarà necessario eseguire Ant e specificare come argomento il target d'interesse.

Le *property* sono variabili che possono essere utilizzare all'interno dello script con la sintassi *${propertyName}*.

```
<!-- Definizione del path della libreria di sistema -->
<property name="servlet.jar" value="C:\Utility\Tomcat 5.5\common\lib\"/>

<!--Definizione del path dove sarà creato il pacchetto war-->
<property name="distpath.project" value="../../build/dist"/>

<!-- Definizione del nome del progetto, nome del deploy  e versione -->
<property name="project.title" value="Foglio Ore - Azienda S.p.A."/>
<property name="project.distname" value="foglio_ore"/>
<property name="project.version" value="1.1"/>

<!-- definizione delle librerie -->
<path id="compile.classpath">
        <pathelement path ="lib/commons-beanutils-1.7.0.jar"/>
        <pathelement path ="lib/commons-chain-1.1.jar"/>
        <pathelement path ="lib/commons-digester-1.8.jar"/>
        <pathelement path ="lib/commons-logging-1.0.4.jar"/>
        <pathelement path ="lib/commons-validator-1.3.1.jar"/>
        <pathelement path ="lib/oro-2.0.8.jar"/>
        <pathelement path ="lib/struts-core-1.3.8.jar"/>
        <pathelement path ="lib/struts-taglib-1.3.8.jar"/>
        <pathelement path ="lib/struts-tiles-1.3.8.jar"/>
        <pathelement path ="${servlet.jar}/servlet-api.jar"/>
        <pathelement path ="classes"/>
        <pathelement path ="${classpath}"/>
</path>
```

Il target *prepare* è un target di inizializzazione e crea il timestamp dei file di output.

```
<target name="prepare">
   <tstamp/>
 </target>
```

Il target *resources* copia tutte le risorse che non richiedono compilazione, nella cartella classes.

```
<target name="resources">
 <copy todir="classes/" includeEmptyDirs="no">
   <fileset dir="../../src/">
     <patternset>
       <include name="**/*.conf"/>
       <include name="**/*.properties"/>
       <include name="**/*.xml"/>
     </patternset>
  </fileset>
 </copy>
</target>
```

Nel target seguente vengono compilati tutti i sorgenti del progetto, indicando la dipendenza dei target tramite la sintassi *depends="targetName1,targetName2"*.

L'elemento *javac* rappresenta un costrutto specifico per la compilazione attraverso il compilatore JAVAC del pacchetto JDK di Java.

```
    <target name="compile" depends="prepare,resources">
        <javac srcdir="../../src" destdir="classes">
            <classpath refid="compile.classpath"/>
        </javac>
    </target>
```

Il target seguente definisce il modo in cui devono essere eliminate le risorse generate dal processo di build.

```
<target name="clean"
      description="Prepare for clean build">
      <delete dir="classes"/>
      <delete dir="../../build/"/>
      <mkdir  dir="classes"/>
</target>
```

Il target project effettua i precedenti tre target *clean,prepare* e *compile* per l'operazione completa di build del progetto.

```
<target name="project" depends="clean,prepare,compile"/>
```

Il target seguente crea solamente il pacchetto WAR dell'applicazione, senza l'operazione di build.

```
<!-- Create binary distribution -->
<target name="dist"
        description="Create binary distribution">

      <mkdir
        dir="${distpath.project}"/>
      <jar
        jarfile="${distpath.project}/${project.distname}.jar"
        basedir="./classes"/>
      <copy
        file="${distpath.project}/${project.distname}.jar"
        todir="${distpath.project}"/>

      <war
        basedir="../"
        warfile="../../${project.distname}.war"
        webxml="web.xml">
        <exclude name="**/${distpath.project}/**"/>
       </war>
      <move file="../../${project.distname}.war"
tofile="${distpath.project}/${project.distname}.war" />

</target>
```

Infine, il target *all* che esegue l'operazione di build e crea la distribuzione.

```
<target name="all" depends="project,dist"/>
```

Per la messa in opera dell'applicazione nel Webserver sarà sufficiente copiare il file .war della distribuzione nella cartella di root del web server ed eseguire una re-inizializzazione dell'istanza del server.

Conclusioni

Posso sicuramente ritenermi soddisfatto di come si è svolta l'intera attività di tirocinio e dei risultati ottenuti. Innanzitutto, ho potuto affrontare problemi pratici che riguardano la produzione di software, sia dal lato progettuale sia da quello implementativo. Ho potuto costatare che la scelta della tecnologia Java spesso è associata a realtà aziendali piuttosto grandi e stratificate. La complessità di Java, infatti, ne scoraggia l'utilizzo agli sviluppatori di applicazioni (web e non) di media e bassa difficoltà, i quali preferiscono soluzioni più semplici come PHP, ASP e altri linguaggi di scripting.

Inoltre, ho potuto rafforzare l'importanza della standardizzazione del processo di produzione di software: l'adozione di framework di sviluppo permette di avere una struttura adattabile e flessibile e di risolvere una certa gamma di problematiche comuni.

Grazie al progetto "Startup", ho potuto misurarmi con problemi legati alla qualità, con particolare attenzione all'efficienza delle soluzioni proposte, in un ambiente Java multi-thread.

Infine, posso dire che l'attività di tirocinio è stata utile per l'inserimento nel mondo del lavoro: ho avuto, infatti, l'opportunità di proseguire l'attività lavorativa presso alcuni clienti di Aive S.p.A..

Bibliografia

[1] Bodoff S., Green D., Haase K., Jendrock E., Pawlan M., Stearns B., The J2EE™ Turorial, Sun Microsystem, 2002.

[3] Fowler M., Patterns of enterprise application architecture, Addison-Wesley, 2003

[4] Alure D., Crupi J., Malks D., Core J2EE Patterns, Sun Microsystem, 2003

[2] Goodwill J., Mastering Jakarta Struts, Wiley, 2002.

[3] Cavaness C., Programming Jakarta Struts, O'Reilly, 2002.

[4] Husted T., Domoulin C., Franciscus G., Winterfeldt D., Struts in Action, Manning, 2003.

[5] Brown D., Davis C.M., Stanlick S., Struts 2 in Action, Manning, 2008.

[6] Documentazione aziendale Aive sull'architettura JEE.

[7] The Apache Software Foundation, Struts 2, http://struts.apache.org.

[8] Sun Microsystems, JavaServer Faces Technology, http://java.sun.com.

[9] Spring Source, Spring Framework, http://www.springsource.org/.

[10] JBoss Community, Hibernate, http://www.hibernate.org.

Ringraziamenti

Vorrei ringraziare, innanzitutto, il professor Giorgio Clemente per avermi seguito durante l'attività di tirocinio e durante la realizzazione di questa relazione.

Ringrazio l'azienda Aive S.p.A. per avermi concesso tempo e spazio presso le loro strutture e tutto il personale che ha contribuito alla mia formazione.

Un immenso grazie va ai miei genitori e a mia sorella, per il loro immancabile aiuto in ogni momento del mio percorso universitario.

Ringrazio Sara che durante gli studi mi ha sostenuto sempre e con forza, in ogni mia scelta.

Ringrazio, infine, i miei parenti e i miei amici per l'affetto che mi hanno sempre dimostrato.