Università degli Studi di Pisa

Facoltà di Ingegneria
Corso di Laurea Specialistica in Ingegneria Informatica
per la Gestione d'Azienda

Tesi di laurea specialistica

Reengineering di un modulo di un sistema ERP: progettazione e sviluppo del prototipo

Relatori:
Prof. Cinzia Bernardeschi
Prof. Francesco Marcelloni
Ing. Michele Barbagli

Candidato:
Matteo Piacentini

Anno Accademico 2009/2010

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Il presente lavoro di tesi illustra la progettazione e lo sviluppo di un prototipo di un'applicazione *web* che permette di replicare un applicativo preesistente presso un'azienda che commercializza oli lubrificanti per motori, utilizzando diverse tecnologie rispetto alle attuali, per migliorarne alcuni aspetti e per permettere di spostare parte della gestione dell'azienda sul *web*.

Prima di iniziare lo sviluppo del prototipo è stato necessario effettuare una valutazione delle tecnologie presenti sul mercato per realizzarlo nel modo più efficace e più efficiente.

Nella prima parte della tesi verrà quindi illustrata una valutazione di alcune delle possibili alternative presenti sul mercato, soffermandosi soprattutto sulle tecnologie che possono permettere di velocizzare lo sviluppo dell'applicazione stessa e di fornire un adeguato supporto.

In particolare nel capitolo 1 verrà effettuata una panoramica delle tecnologie attualmente presenti nell'ambito dello sviluppo di applicazioni *web*, accennando alle piattaforme, agli *application server* e ai *framework* utilizzati per lo sviluppo.

Nel secondo capitolo ci si concentrerà sulla valutazione dei principali *framework* di sviluppo *Java*, descrivendo le caratteristiche di cinque di loro ed effettuando dei confronti tra di essi. I *framework* saranno presentati in maniera molto generale, limitandosi a descriverne il funzionamento di base in maniera da illustrare con più facilità le differenze tra di essi e i relativi vantaggi o svantaggi. Per uno studio più approfondito dei vari *framework*, che esula dagli scopi di questa tesi, si rimanda a testi più specifici, quali ad esempio quelli indicati in bibliografia.

Nella seconda parte della tesi si analizzerà l'attività di sviluppo del prototipo, soffermandosi inizialmente sulle caratteristiche generali dell'applicazione, per poi passare ad analizzare il modulo di gestione degli ordini del sistema.

Nel terzo capitolo si passerà quindi all'illustrazione dell'applicativo vero e proprio, indicando l'ambito in cui si andrà a porre il progetto e le motivazioni che hanno portato alla sua realizzazione, gli obiettivi da raggiungere, i criteri nella scelta delle tecnologie utilizzate, per poi passare ad illustrare l'architettura del sistema ed il suo funzionamento.

Nella descrizione del prototipo realizzato si cercherà di illustrare le scelte legate principalmente alla modularizzazione e di conseguenza alla manutenibilità del sistema stesso; trattandosi infatti della replicazione di un sistema preesistente non si dovrà necessariamente modificare le logiche di business caratterizzanti l'applicativo, ma piuttosto fare in modo che il sistema sia facilmente modificabile ed integrabile con nuove funzionalità.

Nel quarto capitolo si approfondirà il modulo di gestione degli ordini, che è la funzionalità principale richiesta dal cliente, soffermandosi sulle caratteristiche funzionali e sulle migliorie apportate rispetto al sistema originale, analizzando quindi nel dettaglio le operazioni di inserimento, visualizzazione, modifica ed eliminazione degli ordini di clienti o fornitori dell'azienda.

Nell'ultimo capitolo infine verranno indicati i possibili sviluppi futuri dell'applicativo.

Il lavoro è stato svolto presso la Shinteck s.r.l. di Pontedera.

Capitolo 1

Stato dell'arte

1.1 Applicazioni web

Al giorno d'oggi un'azienda che non utilizza almeno in parte *internet* rischia di essere fuori dal mercato, in qualsiasi settore operi. In molti casi, in base al settore di appartenenza, un'azienda può limitarsi ad utilizzare la rete esclusivamente come vetrina virtuale per i propri prodotti o servizi, senza di fatto utilizzarla per la produzione, la gestione o comunque per tutte le altre attività che non siano strettamente riconducibili ad un'attività di *marketing*.

Sono però sempre in aumento le aziende che sentono il bisogno di utilizzare la rete per controllare direttamente l'evolversi della produzione, per gestire le informazioni aziendali in maniera dinamica, ma anche per fornire un servizio più veloce e preciso al cliente.

Questi sono alcuni degli aspetti per i quali si è visto un aumento dell'utilizzo delle applicazioni *web* per la gestione di qualsiasi tipo di azienda, non solo di medio-grande dimensione, ma anche e soprattutto per le piccole-medie imprese.

Per non rischiare di creare incomprensione sull'utilizzo del termine è importante precisare che le applicazioni *web* sono applicazioni accessibili tramite qualsiasi tipo di *network*, quindi attraverso non solamente la rete *internet*, ma anche una normalissima rete aziendale *intranet*.

La forza principale delle applicazioni *web* risiede infatti nella possibilità di accedere alle funzioni aziendali tramite l'utilizzo di un comunissimo *browser web*, consentendo un notevole risparmio di tempo e denaro, oltre ad una maggior velocità nella navigazione all'interno dell'applicazione.

1.2 Architettura a livelli

La progettazione di un'applicazione *web* ha come prerequisito la selezione di un'architettura per il suo sviluppo. Il paradigma più utilizzato per descriverla è quello dell'architettura "a livelli" o *Layered Application Architecture*. Tale paradigma prevede che un sistema *software* sia composto da tre livelli distinti, ognuno con un proprio ruolo:

- *Presentation Layer*: è il livello di presentazione, che ha il compito di interagire direttamente con l'utente del sistema, quindi appunto presentargli le varie schermate con i componenti per inserire i dati o selezionarne altri, tramite l'utilizzo di un'apposita interfaccia grafica (*Graphic User Interface*);
- *Application Processing Laye*r: è il livello di *business logic*, cioè il livello in cui vengono effettuate le operazioni tipiche dell'applicazione *web*, quindi le varie elaborazioni dei dati;
- *Data Management Layer*: è il livello di gestione dei dati, in cui viene gestito l'accesso ai dati e la loro persistenza.

Dal punto di vista *hardware*, questi tre livelli distinti possono essere mappati su altrettanti livelli ed in particolare su altrettante macchine distinte; tipicamente il *Presentation Layer* è rappresentato da un *web browser*, l'*Application Processing Layer* da un *web container* ed il *Data Management Layer* da uno strumento di *Data Management*.

1.3 Piattaforme di sviluppo

Per lo sviluppo delle applicazioni *web* sono al momento utilizzate principalmente due piattaforme, entrambe molto potenti: la piattaforma *Java EE* della *Sun* e la *Microsoft .NET*.

1.3.1 Piattaforma *Java EE*

La caratteristica principale della piattaforma *Java* è sicuramente la portabilità: un programma scritto in *Java* può essere creato in ambiente *Unix* ed eseguito successivamente in ambiente *Windows* senza la necessità di modificare il codice sorgente.

Il codice è infatti scritto seguendo le regole grammaticali e sintattiche del linguaggio *Java* e si basa sulle *API Java* che sono librerie utilizzabili dallo sviluppatore come supporto per la realizzazione dei compiti più disparati; successivamente tale codice viene compilato, permettendo di ottenere un codice indipendente dalla macchina su cui è stato creato denominato *bytecode*. Tale codice viene utilizzato dalla *Java Virtual Machine* del sistema operativo utilizzato per la traduzione delle istruzioni in linguaggio macchina. E' quindi sufficiente disporre di una *Java Virtual Machine* per eseguire il *bytecode* su qualsiasi operativo si desideri in quanto è proprio tale sistema a provvedere alla fase finale della traduzione del codice.

La piattaforma *Java Enterprise Edition* è la versione *enterprise* della piattaforma *Java*, ed è quindi un insieme di specifiche che estende le funzioni della piattaforma *standard* in maniera da consentire lo sviluppo di applicazioni *misson critical* o *enterprise*.

1.3.2 Piattaforma *.NET*

La piattaforma *.NET* ha invece una caratteristica peculiare totalmente diversa in quanto si basa sulla possibilità di utilizzare il linguaggio di programmazione preferito riducendo però la portabilità alle sole piattaforme *Windows*.

Il codice scritto con il linguaggio scelto deve essere compilato con il relativo compilatore che sia però compatibile con le specifiche *.NET*. Tale compilazione produce anche in questo caso un codice intermedio chiamato *common intermediate language*. Questo codice intermedio permette di nascondere il linguaggio utilizzato, consentendo anche relazioni tra oggetti realizzati in linguaggi differenti, in quanto tutti i linguaggi vengono compilati seguendo delle specifiche precise che ne garantiscono la compatibilità.

Il codice intermedio viene eseguito tramite il *common language runtime* di *Windows* che si occupa di recuperare i vari oggetti di cui ha bisogno.

1.3.3 *Sun Java EE* vs *Microsoft .NET*

La differenza tra le due piattaforme è quindi sostanzialmente nella scelta di ottenere l'indipendenza dalla macchina o l'indipendenza dal linguaggio. Con le opportune modifiche sarebbe comunque possibile ottenere queste caratteristiche su entrambe: nel caso di *Java* sono presenti opportuni progetti che presentano l'obiettivo di sviluppare l'indipendenza dal linguaggio quali *JPython* o *PERCobol*, che mettono a disposizione compilatori che permettono di ottenere il *bytecode* da codice sorgente scritto

in vari linguaggi; nel caso di *.NET* può essere possibile ottenere la portabilità su un altro ambiente implementando un *common language runtime* per l'ambiente in questione[1].

1.4 *Application Server*

Uno strumento fondamentale per la realizzazione di applicazioni *web* è l'*application server*.

Un *application server* è un *software* che fornisce l'infrastruttura e le funzionalità di supporto, sviluppo ed esecuzione di applicazioni e componenti *server* in un contesto distribuito. Fornisce infatti una serie di servizi quali ad esempio la gestione delle autenticazioni ed autorizzazioni degli utenti, dell'accesso a *database* o delle transazioni.

1.4.1 Mercato

Esistono specifici *application server* per le varie tecnologie a disposizione degli sviluppatori; tra i più importanti per lo sviluppo in *Java* citiamo prodotti *open source* quali *Glassfish*, *JBoss*, *Geronimo*, e prodotti proprietari quali *WebSphere* e *WebLogic*. Per l'ambiente *.NET* viene utilizzato il *Microsoft Internet Information Services*.

Molto spesso viene citato tra gli *application server* anche *Apache Tomcat*. In realtà *Tomcat* è un *web server* cioè un *server* che fornisce le pagine *web* in risposta a specifiche richieste del *browser*; è sicuramente il *web server* più diffuso per lo sviluppo in *Java* tanto da poter essere quasi considerato uno *standard*, ma per quanto sia un prodotto molto valido, non può essere considerato un *application server* in quanto fornisce solo in parte i servizi di un vero *application server*, non fornendo di fatto il supporto diretto ad esempio per le transazioni, fondamentale per l'utilizzo di un'applicazione in un contesto distribuito.

1.4.2 Vantaggi

L'utilizzo di un *application server* nello sviluppo di applicazioni *web* porta numerosi vantaggi:

[1] Attualmente è presente un *framework* chiamato *Mono* che prevede di far funzionare un applicativo compilato con il *.Net framework* sotto altri sistemi (ad esempio *Linux*).

- semplificazione delle attività di sviluppo, in quanto viene creato un ambiente nel quale si possono utilizzare gli strumenti di sviluppo più diffusi sul mercato, consentendo di produrre e distribuire rapidamente applicazioni transazionali altamente scalabili;

- supporto di vari linguaggi, strumenti e piattaforme;

- robustezza, fondamentale in applicazioni aziendali, garantita da un'architettura basata sui componenti e sul bilanciamento dinamico dei carichi che assicurano un'alta disponibilità dei sistemi; è possibile infatti riconfigurare, aggiungere o rimuovere i componenti del *server* e la logica applicativa senza interruzioni nell'erogazione dei servizi;

- sicurezza, in quanto un *application server* gestisce l'accesso degli utenti fornendo la possibilità di utilizzare, per le comunicazioni tra *client* e *server*, algoritmi *standard* come quelli offerti dal protocollo *SSL*;

- gestione delle transazioni facilitata, in quanto un *application server* assicura l'integrità transazionale e la gestione affidabile dei *back-end* multipli per le risorse e i dati occupandosi personalmente di tali funzioni in quanto gestisce le interazioni con i *database* e le funzioni di *commit*, *rollback* e *recovery*;

- alte prestazioni, garantite da servizi quali il *multithreading*, il bilanciamento dinamico dei carichi di lavoro (*load balancing*), il *caching* e il *pooling* degli oggetti e delle connessioni ai *database*;

- scalabilità, in quanto un *application server* supporta il partizionamento delle applicazioni e la distribuzione in rete dei componenti fornendo anche il supporto per la gestione di un gran numero di utenti concorrenti;

- estensibilità delle funzionalità delle applicazioni, grazie all'architettura modulare degli *application server* e il supporto per i *server* e per i moduli applicativi che possono essere caricati dinamicamente.

Nello sviluppo dell'applicazione si è scelto di basarsi su *Glassfish v.3* in quanto fornisce praticamente tutte le funzionalità, e quindi tutti i vantaggi sopra esposti, non sempre ottenibili con altri *application server*.

1.5 *Pattern MVC*

Un *pattern* di fatto è un approccio *standard* utilizzato per risolvere problemi comuni; il più utilizzato nello sviluppo di applicazioni *web* è il *pattern MVC* (*Model-View-Controller*), nato negli anni '80 nell'ambiente *SmallTalk* in cui era utilizzato per la realizzazione di *Graphic User Interface* di applicazioni *desktop*.

Tale metodologia di sviluppo consente una netta separazione tra *Model*, *View* e *Controller*:

- Il *Model* rappresenta la parte dell'applicazione più strettamente a contatto con i dati; dovrebbe pertanto contenere la rappresentazione dei dati utilizzati e l'infrastruttura necessaria all'interfacciamento con basi di dati; il *Model* ha anche il compito di notificare le modifiche dello stato dell'applicazione al fine di presentare ai componenti della *View* una situazione sempre aggiornata.

- La *View* si riferisce alla parte dell'applicazione che gestisce le visualizzazioni delle schermate da presentare all'utente. In tale sezione (come nel caso del *Model*) deve essere ridotta al minimo la logica di *business*, dovendosi occupare solamente della creazione dell'interfaccia grafica da presentare all'utente.

- Il *Controller* è invece la parte dedicata alla logica di *business*. In questa sezione, in base ai dati provenienti da *View* e *Model*, si decide quali azioni effettuare, quindi quale modifiche al *Model* andranno eseguite e quali schermate andranno presentate all'utente.

La figura 1.1 illustra l'interazione tra i vari attori del pattern ed i relativi ruoli principali.

Utilizzare il *pattern MVC* vuol dire costruire la propria applicazione in maniera modulare, definendo esattamente i compiti di ogni componente. Questo comporta notevoli vantaggi:

- un'architettura flessibile dell'applicazione, in quanto è possibile cambiare in tempi diversi parte della tecnologia utilizzata, ad esempio, per costruire l'interfaccia grafica;

- una maggior manutenibilità, in quanto è possibile modificare solo piccole porzioni dell'applicazione senza dover necessariamente intervenire sul resto;

- una riduzione delle linee di codice dovuta soprattutto alla possibilità di riusare vari componenti in punti diversi dell'applicazione;

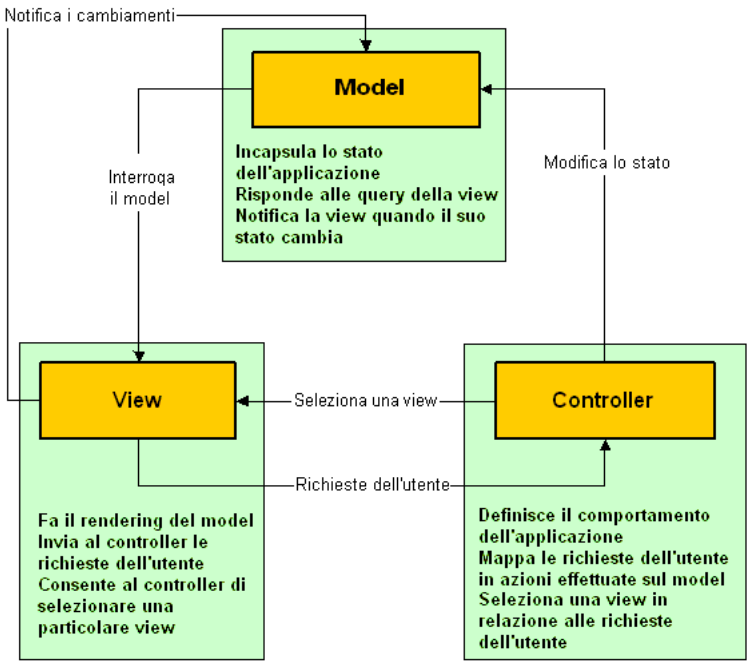


Figura 1.1: Pattern MVC

- una semplificazione dello sviluppo in team, in quanto risulta più agevole suddividere i compiti tra i vari sviluppatori;
- l'utilizzo di uno *standard* di progettazione che facilita la comprensione e le successive implementazioni di un progetto.

Ovviamente l'utilizzo di questo *pattern* comporta una complessità maggiore, che però potrebbe essere un problema solamente in caso di applicazioni piccole, mentre per le applicazioni medio/grandi rappresenta la base imprescindibile da cui partire.

1.6 *Framework MVC*

1.6.1 Introduzione

Un *framework*, nella produzione del *software*, è una struttura di supporto su cui un *software* può essere organizzato e progettato.

Con l'aumento della produzione di *software* e della complessità delle interfacce grafiche sono nati sempre più frequentemente *framework* con lo scopo di facilitare il più possibile la vita allo sviluppatore, da un lato per evitare la creazione *ex novo* di interfacce spesso molto simili da un'applicazione all'altra, ma anche e soprattutto

per assisterlo nella progettazione dell'applicazione, in maniera da far sì che essa venga scritta in maniera lineare e facilmente riadattabile ai vari contesti in base a specifici requisiti di utilizzo.

Un *framework MVC* è un *framework* che fornisce una guida allo sviluppatore nella realizzazione di applicazioni basate sul *pattern MVC* ed in più offre il supporto per gestire in maniera semplice e veloce le funzionalità tipiche di ogni applicazioni *web*. In questo modo è possibile concentrarsi sugli aspetti principali del *business*, permettendo di aumentare notevolmente la produttività. I vantaggi offerti dall'utilizzo di un *framework* sono infatti:

- disaccoppiamento della presentazione e della logica in componenti separati;
- separazione dei ruoli degli sviluppatori: gli sviluppatori hanno a disposizione differenti tipi di interfacce a seconda che siano sviluppatori di componenti di presentazione (e quindi agiscono sulle pagine dell'applicazione) o di logica (e quindi agiscono sulle classi);
- utilizzo di un unico punto di controllo centralizzato;
- facilità di *testing* dei componenti dell'applicazione;
- disponibilità di una struttura predefinita, per utilizzare il tempo a disposizione per sviluppare la logica di *business* piuttosto che il codice strutturale dell'applicazione;
- disponibilità di un gran numero di caratteristiche che lo sviluppatore medio potrebbe non essere in grado di realizzare per mancanza di esperienza o di tempo a disposizione;
- utilizzo di componenti *standard* utilizzati e quindi testati dalla comunità degli sviluppatori;
- stabilità, in quanto vista l'estesa comunità di sviluppatori è certamente più sicuro il codice dei *framework* piuttosto che il codice personalizzato;
- supporto della comunità degli sviluppatori, con un numero sempre maggiore di *tutorial* e gruppi di discussione presenti nella rete;
- riduzione del tempo e del costo di apprendimento, in quanto gli sviluppatori con esperienze nello sviluppo di un *framework* tendono a produrre in maniera più rapida;

- semplificazione dell'internazionalizzazione, tramite un supporto efficiente fornito dalla maggior parte dei *framework*;
- semplificazione della validazione degli *input,* in quanto la maggior parte dei *framework* dispongono di un apposito supporto che permette di semplificare tale operazione spesso ripetitiva.

Un *framework* è solitamente definito da una serie di classi astratte che saranno implementate dall'applicazione da realizzare; in questo modo il programmatore ha a disposizione una ben definita struttura[2].

In molti casi un *framework* è semplicemente un'aggiunta alle librerie a *run-time* del linguaggio utilizzato. Infatti per la maggior parte dei linguaggi di programmazione troviamo esempi di *framework*:

- il *C++* ha vari *framework* che rappresentano un'estensione della libreria *standard C++*: *Standard Template Library*, *Active Template Library*, *Microsoft Foundation Classes*, *Qt*, *wxWidgets*;
- il *C* in aggiunta alla libreria *standard libc* (per l'ambiente *Unix*) o *CRT* (in ambiente *Microsoft*), presenta vari *framework* quali ad esempio il *GIMP Toolkit*;
- il *C#* non ha una propria libreria di *run-time*, ma si può appoggiare sul *Framework .NET* o su *VisualBasic .NET*;
- il *Delphi* di *Borland* si basa sulla *Visual Component Library*, ma può essere utilizzato anche il *Framework .NET*;
- il *PHP* può utilizzare *framework* quali *Zend Framework*, *Seagull* o *Jamp*;
- il *Perl* può utilizzare *Catalyst*;
- *RPG* presenta *framework* come *Jeniux framework*, *WebFacing*, *GUI/400*;
- il *Python* può utilizzare *Twisted*;
- il *Java*, che sarà analizzato più in dettaglio nel seguito, presenta *framework* come *Struts*, *JavaServer Faces*, *Spring*, *Wicket*, *Vaadin*, *Cocoon* e molti altri.

[2] *Framework* significa proprio "struttura" o "intelaiatura"

1.6.2 *Framework MVC Java*

L'incremento dello sviluppo di applicazioni *web* basate sulla piattaforma *Java* ha portato un conseguente incremento dei *framework* a disposizione degli sviluppatori.

Se pochi anni fa eravamo soltanto agli albori dello sviluppo tramite l'ausilio di *framework* ora ci si è accorti che il loro utilizzo è indispensabile per garantire una progettazione e uno sviluppo veloci, robusti e manutenibili. Di conseguenza la comunità degli sviluppatori *Java* ha cominciato a rendere disponibili moltissimi *framework*, alcuni innovativi, altri con caratteristiche molto (forse troppo) simili a quelli già esistenti.

La presenza di un gran numero di *framework* può essere considerato sia un vantaggio che uno svantaggio. Sicuramente avere a disposizione molte opzioni può essere un vantaggio perché può garantire con sufficiente sicurezza che ci sia almeno un *framework* che può aiutare lo sviluppatore nella creazione della propria applicazione *web*. Nello stesso tempo però sorge il problema di capire quale può essere effettivamente utile per i propri scopi, in quanto è assai improbabile, per non dire impossibile, che uno sviluppatore si documenti su tutti i *framework* messi a disposizione dalla comunità *Java* per scegliere quale può essere il migliore, perché correrebbe il rischio di impiegare più tempo a documentarsi sulle tecnologie disponibili che non a realizzare il proprio lavoro.

Nonostante le caratteristiche simili che possono avere i *framework*, si possono distinguere due modelli principali di sviluppo:

- modello basato sulla struttura *JSP/servlet* (conosciuto anche come *Model* 2);
- modello a componenti.

1.6.2.1 *Modello JSP/servlet*

Il modello *JSP/servlet* è basato appunto sull'utilizzo di pagine *JSP* e di *servlet.*

Una pagina *JSP* (*JavaServer Pages*) è una tecnologia *Java* che permette di inserire in una pagina *web* contenuti dinamici. Si basa su un insieme di *tag* speciali con cui possono essere invocate funzioni predefinite o codice *Java.* Al momento della prima invocazione le pagine *JSP* vengono tradotte automaticamente in *servlet* da un compilatore *JSP*. Per questo motivo il *web server* deve contenere, oltre al motore *JSP*, anche un *servlet container.*

Una *servlet* è un oggetto *Java* che permette di realizzare le funzioni applicative dell'applicazione. Una *servlet* è spesso utilizzata come *controller* nell'applicazione

del *pattern MVC*, in quanto, una volta invocata, può decidere quale altra pagina visualizzare o quale operazione effettuare.

Secondo il modello *JSP/servlet*, chiamato anche modello richiesta/risposta, il *framework* lavora in termini di intere richieste e di intere pagine. Il principio di funzionamento in linea di massima è il seguente:

1. un utente effettua un'interazione con una pagina *JSP*, magari inserendo ed inviando dei dati tramite un *form*;

2. tali dati vengono inviati ad una *servlet* (o ad un oggetto con funzioni analoghe) che li utilizza per effettuare determinate operazioni;

3. la *servlet* indirizza l'applicazione verso la nuova pagina da visualizzare.

Quindi un *framework* basato su tale modello è un *framework* in cui l'ottica dello sviluppatore è concentrata sul flusso di esecuzione dell'applicazione, in quanto è possibile identificare una serie di passi definiti che un utente generico può effettuare navigando all'interno dell'applicazione in maniera semplice; si ha infatti un continuo passaggio da una pagina *JSP* ad una *servlet*, poi ancora ad un'altra pagina *JSP* e poi ancora ad una *servlet*, etc....

In tale modello lo sviluppatore si trova spesso a lavorare con gli oggetti contenuti nelle richieste e nelle risposte *Http*[3].

1.6.2.2 Modello a componenti

Il modello a componenti è basato sul fatto che l'applicazione viene creata tramite l'aggiunta appunto di componenti predefiniti (*widget*), con specifiche funzioni, che possono essere riusati molto facilmente all'interno dell'applicazione e con i quali l'utente interagisce quasi direttamente.

In questo caso infatti non necessariamente tra un'azione e l'altra dell'utente vengono gestite intere pagine *web*, ma, nella maggior parte dei casi, l'interazione avviene solamente con un componente, permettendo quindi un'interazione più veloce e funzionale. In questi casi si fa spesso ricorso alla tecnologia *AJAX*, che permette appunto di ridurre il carico delle richieste consentendo in alcuni casi invii parziali delle informazioni. Lo sviluppo di applicazioni *HTML* con *AJAX* infatti si basa su uno scambio di dati in *background* fra *web browser* e *server*, che consente l'aggiornamento dinamico di una pagina *web* senza esplicito ricaricamento da parte

[3] Anche se sempre più spesso i *framework* tendono a nascondere la gestione diretta delle richieste allo sviluppatore, permettendogli di lavorare ad un livello di astrazione più alto.

dell'utente. *AJAX* è asincrono nel senso che i dati *extra* sono richiesti al *server* e caricati in *background* senza interferire con il comportamento della pagina esistente. Normalmente le funzioni richiamate sono scritte con il linguaggio *JavaScript*.

Rispetto al precedente modello però si può riscontrare qualche difficoltà in più nel seguire l'andamento del flusso di esecuzione dell'applicazione, proprio perché si tende ad avere un livello di astrazione più alto del comportamento dell'applicazione. In molti casi infatti il programmatore interagisce tramite componenti che effettuano alcune operazioni trasparenti al programmatore.

Capitolo 2

Scelta delle tecnologie *Java*

La garanzia di una maggior indipendenza dal linguaggio è la caratteristica che ha fatto propendere nella scelta per lo sviluppo dell'applicazione oggetto di questa tesi verso l'utilizzo della tecnologia *Java*.

Nel seguito quindi verranno presi in considerazione solamente gli aspetti relativi allo sviluppo in *Java*.

2.1 Comparazione *Framework MVC Java*

Per scegliere quale *framework* utilizzare per la propria applicazione è necessario avere ben chiari i requisiti che dovrà rispettare ed il *team* di sviluppo che si ha a disposizione per realizzarla.

In questa sezione saranno analizzate brevemente le caratteristiche di cinque tra i *framework MVC Java* più importanti e poi verrà effettuato un confronto per avere una base su cui partire per stabilire quale *framework* può essere migliore per l'applicazione che si intende sviluppare.

Verranno quindi presentati *Struts 2* e *Spring MVC*, che sono *framework* che seguono il modello *JSP/servlet*, *Wicket* e *Vaadin*, che sono *framework* a componenti puri, e *JavaServer Faces 2* che è considerato un *framework* a componenti, ma che può essere anche visto come ponte di collegamento tra le due tipologie di *framework*.

E' importante sottolineare che, anche dopo un'attenta comparazione tra i *framework*, non è possibile stabilire quale sia migliore degli altri; ognuno ha le sue caratteristiche, ma i cinque *framework* presentati sono tutti molto robusti e aiutano lo sviluppatore a realizzare tutte le principali funzioni caratteristiche di un'applicazione *web*.

Questa analisi vuole semplicemente porre le basi di una scelta, comunque personale in base alle esigenze di sviluppo.

2.1.1 *Struts* 2

2.1.1.1 Introduzione

Apache Struts [2] è nato nel 2000 per mano di Craig McClanahan ed è stato ufficialmente rilasciato come versione 1.0 nel luglio 2001. *Struts* è stato tra i primi e sicuramente tra i più importanti *Java framework open source* per lo sviluppo di applicazioni *web*, tanto da diventare uno *standard de facto* per molti anni.

La prima versione di *Struts* è tuttora il *framework* più diffuso proprio in quanto la sua introduzione è stato un notevole cambiamento positivo nella metodologia di lavoro di molti programmatori e soprattutto perché per anni non ci sono state grandi alternative di rilievo.

Verso la metà degli anni 2000 sono stati però sviluppati vari *framework* con caratteristiche più innovative rispetto a quelle di *Struts 1*, quindi nella comunità *Struts* si è sentita la necessità di cambiare gradualmente la metodologia di sviluppo cercando di far lavorare gli sviluppatori ad un livello di astrazione più alto rispetto a quello della prima versione. Proprio per questo nel 2007 viene rilasciata una nuova *release* di *Struts*, il cui obiettivo è infatti quello di semplificare ulteriormente lo sviluppo di applicazioni *web* offrendo numerose caratteristiche che permettano di lasciare al *framework* alcuni aspetti che non riguardano in maniera diretta lo sviluppo dell'applicazione *web* da realizzare.

La tabella 2.2 rappresenta le differenze tra le due versioni di *Struts* e può essere molto utile per chi già conosce *Struts 1* per avere un'idea delle maggiori novità introdotte.

2.1.1.2 Struttura e funzionamento

I componenti principali di un'applicazione basata su *Struts* 2 sono:

- *web.xml*: è il *file* di configurazione caratterizzante un'applicazione *web*; in questo *file* va inserita la dichiarazione del "*FilterDispatcher*", ovvero del *controller* base di *Struts* che si occupa di gestire e smistare tutte le richieste;
- *struts.xml*: è il *file* di configurazione principale utilizzato da *Struts* 2 per l'inizializzazione ed il funzionamento delle varie risorse. In tale *file* vengono

	Struts 1	***Struts 2***
Front Controller	Compito svolto da un'unica *ActionServlet*	Compito svolto da un unico *FilterDispatcher*
Classi *Action*	Estendono una classe base astratta che è dipendente dal *framework*	Possono sia estendere una classe, sia implementare un'interfaccia in base ai servizi richiesti. Le *Action* non sono dipendenti dal *container* in quanto sono semplici *POJO*
Dipendenza dalle *ServletAPI*	*Action* fortemente dipendenti dalle *servlet API* in quanto gli oggetti *HttpServletRequest* ed *HttpServletResponse* sono passati al metodo *execute()* delle *Action*	E' possibile comunque accedere alla richiesta e alla risposta, ma non è indispensabile accedere agli oggetti *HttpServletRequest* e *HttpServletResponse*
Validazione	Oltre al metodo *validate()* dell'*Action Form* è possibile estendere il comportamento del *Commons Validator*	Oltre al metodo *validate()* è possibile utilizzare l'*XWork validation framework* che supporta anche la validazione a catena
Threading Model	Le *Action* devono essere *Thread-safe* o *synchronized*; c'è infatti una sola istanza di una classe *Action* che cattura le richieste per quella *Action*. Questo comporta delle restrizioni e una notevole attenzione nello sviluppo	Le classi *Action* sono istanziate per ogni richiesta, quindi non ci sono problemi di creare codice *Thread-safety*
Testabilità	Qualche difficoltà per testare le *Action* in maniera indipendente	Le *Action* possono essere testate istanziandole, settando delle proprietà ed invocando dei metodi
Recupero dell'*input*	Usa gli oggetti *Action Form* per catturare gli *input* e tali oggetti estendono classi base dipendenti dal f*ramework*. Inoltre i *JavaBeans* non possono essere usati come *Action Form* così è necessario scrivere molto codice ridondante	Usa le proprietà delle *Action* per catturare gli *input* quindi evita il secondo livello di classi in ingresso. In più le proprietà delle *Action* possono essere accedute via *web* tramite le *taglibs*
Expression Language	Si integra con *JSTL* quindi utilizza *JSTL-EL*	Può utilizzare *JSTL*, ma può anche utilizzare un *EL* più potente e flessibile chiamato *Object Graph Notation Language*
Collegamento dei valori alle viste	Usa il meccanismo *standard* delle *JSP* per collegare gli oggetti provenienti dal *model* nel contesto della pagina per accedervi	Usa una strategia "*Value Stack*" che permette di riusare delle *view* che potrebbero avere nomi di proprietà uguali, ma tipi differenti
Conversione di tipi	Usa *Common-BeanUtils* per le conversioni dei tipi. Queste conversioni sono configurabili per classi e non per istanze	Usa *OGNL* per le conversioni dei tipi.
Controllo dell'esecuzione delle *Action*	Supporta differenti *RequestProcessor* (*lifecycle*) per ogni modulo, ma tutte le *Action* in un modulo devono condividere lo stesso *lifecycle*	Supporta la creazione di differenti *lifecycle*. I *Custom Stack* possono essere creati e usati con differenti *Action* a seconda del bisogno

Tabella 2.2: Differenze tra *Struts* 1 e *Struts* 2

dichiarate le varie *action* utilizzate ed il *mapping* di tale *action*. Questo *file* quindi è una sorta di "guida" per il *framework*, in quanto è qui che vengono identificati i possibili flussi di esecuzione dell'applicazione;

- *action*: in queste classi *Java*, gestite come *Plain Old Java Objects*, vengono definite le varie operazioni di *business logic* caratteristiche dell'applicazione; il risultato del metodo principale "*execute()*" è una stringa che, in base alla configurazione indicata in *struts.xml* (tramite definizione di specifici oggetti *Result*) permette al *framework* di presentare la *view* opportuna;
- *interceptors*: tali oggetti sono in sostanza dei filtri che possono essere eseguiti prima dell'esecuzione delle *Action* per effettuare una parte delle elaborazioni necessarie (spesso ripetitive ed indipendenti dalla *business logic*);
- pagine *JSP*: sono le pagine che rappresentano le varie viste dell'applicazione;
- altri *file* di configurazione: comprendono i vari altri *file* di configurazione o di *utility* che possono essere inseriti in un'applicazione *Struts* 2.

Il funzionamento di massima di un'applicazione basata su *Struts* 2 è quello indicato in figura 2.1.

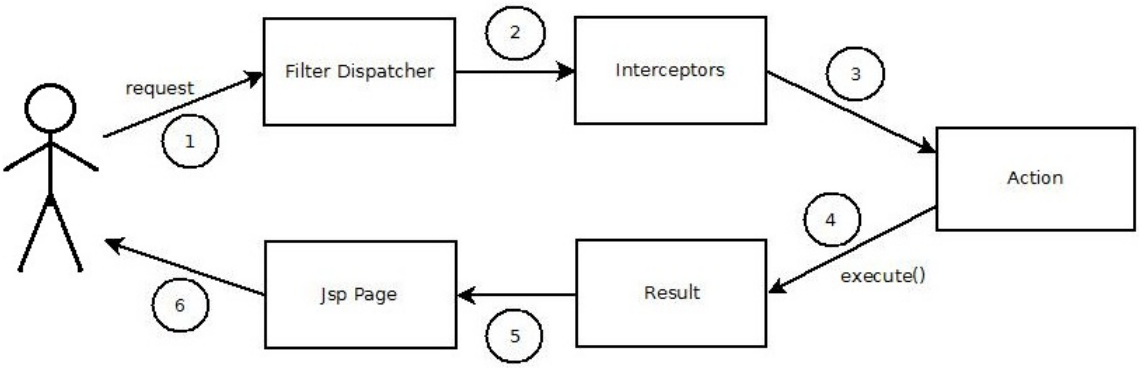


Figura 2.1: Ciclo di vita di una richiesta in un'applicazione basata su *Struts* 2

Quindi nel dettaglio:

1. L'utente effettua una richiesta al *server* per qualche risorsa tramite l'accesso ad una pagina dell'applicazione
2. Alla richiesta sono applicati opportuni oggetti *Interceptors* che effettuano alcune operazioni ripetitive (come ad esempio la validazione dei campi di un *form*)
3. Il *FilterDispatcher* analizza la richiesta e determina la *Action* appropriata basandosi su quanto indicato nel *file struts.xml*

4. Se all'interno della richiesta sono presenti dei dati inviati ad esempio tramite un *form*, i valori dei campi di tale *form* vengono associati direttamente con le proprietà della *Action* appropriata tramite l'utilizzo degli *Struts tag* e temporaneamente memorizzati in esse (le *Action* sono quindi dei *JavaBean* con specifiche proprietà che devono possedere dei metodi "*setter*" e "*getter*" per interagire con le pagine associate ad esse). Dopo tale memorizzazione viene mandato in esecuzione il metodo "*execute()*" della *Action* che effettua le necessarie operazioni di *business logic* e restituisce una stringa contenente il nome di un oggetto *Result*

5. L'oggetto *Result* prepara l'*output* (e quindi la pagina da restituire) secondo quanto indicato nel *file struts.xml*

6. La pagina viene visualizzata dall'utente

Maggiori dettagli sul funzionamento di *Apache Struts* 2 possono essere letti in [3].

2.1.2 *Spring MVC*

2.1.2.1 Introduzione

Spring MVC è un modulo del *framework Spring*, che rappresenta una soluzione completa per lo sviluppo di qualsiasi applicazione *Java*. *Spring* [4] è nato nel 2002 da un'idea di Rod Johnson, ma solo dopo alcuni anni è stato aggiunto il modulo relativo allo sviluppo di applicazioni *web* basate sul *pattern MVC*; tale aggiunta è stata effettuata in risposta alle funzionalità non completamente soddisfacenti del *framework MVC* per eccellenza fino a quel momento: *Struts* 1. Tale *framework* infatti non disponeva ancora delle funzionalità dell'*Aspect Oriented Programming* e dell'*Inversion Of Control*, che hanno fatto la fortuna di *Spring* (che le ha rese popolari) e sono state successivamente sviluppate dalla maggior parte dei *framework MVC*, tra cui anche il già citato *Struts* 2.

2.1.2.2 Struttura e funzionamento

I componenti principali di un'applicazione basata su *Spring MVC* sono:

- *web.xml*: nel descrittore di *deployment* deve essere inserita la dichiarazione della "*DispatcherServlet*", ovvero del *controller* base di *Spring MVC* che si occupa di gestire le richieste smistandole ai vari *controller*;

- *dispatcher-servlet.xml*[1]: questo *file* di configurazione contiene la dichiarazione dei *bean* da istanziare e dei relativi *mapping*; è quindi la guida del *framework* all'utilizzo dei componenti inseriti nell'applicazione;
- pagine *JSP*: sono le pagine che rappresentano le varie viste dell'applicazione;
- *command object*: sono *JavaBean* che rappresentano gli oggetti dell'applicazione e sono popolati dai campi dei *form HTML* a cui sono legati;
- *controller*: classi *Java* che in base ai dati ricevuti dalla *view* e dal *model* restituiscono le viste opportune tramite oggetti di tipo "*ModelAndView*", interpretati dal *framework*;
- *validator*: classi *Java* che gestiscono la validazione dei dati che dalle *view* devono arrivare ai *controllers*.

Il funzionamento di massima di un'applicazione basata su *Spring MVC* è illustrato in figura 2.2.

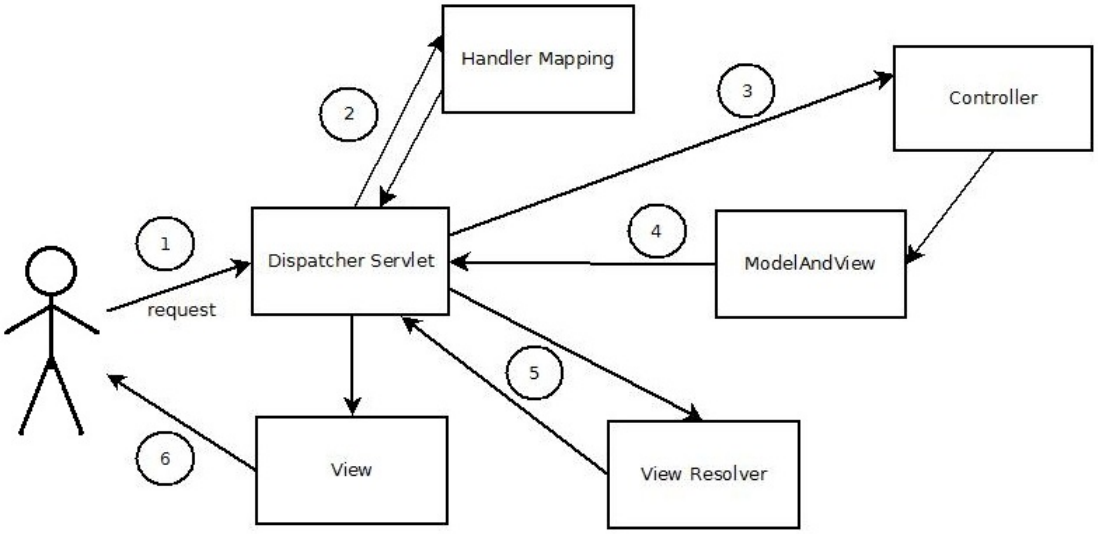


Figura 2.2: Ciclo di vita di una richiesta in un'applicazione basata su *Spring MVC*

Nel dettaglio:

1. L'utente effettua una richiesta al *server* per qualche risorsa

[1]Il *file* di configurazione di *Spring MVC* può in realtà avere anche un nome diverso da quello indicato, in quanto ha il nome che viene dichiarato nel *file web.xml*. Ad esempio se il nome della *DispatcherServlet* viene dichiarato come "*spring*" invece di "*dispatcher*", il *file* di configurazione sarà *spring-servlet.xml*

2. La *DispatcherServlet* consulta l'*HandlerMapping*[2] per trovare il *controller* opportuno

3. La *DispatcherServlet* invoca il *Controller* associato alla richiesta

4. Il *Controller* processa la richiesta e ritorna un oggetto *ModelAndView* alla *DispatcherServlet*

5. La *DispatcherServlet* manda il nome della *view* al *ViewResolver*[3] per trovare la pagina da visualizzare

6. La *DispatcherServlet* passa l'oggetto del *model* alla *View* per restituire il risultato

Per un ulteriore approfondimento sul frame *Spring MVC* si rimanda a [5].

2.1.3 *JavaServer Faces 2*

2.1.3.1 Introduzione

JavaServer Faces [6] è un *framework Java*, le cui specifiche sono fornite direttamente dalla *Sun*, nato nel 2004 per facilitare la costruzione di interfacce utente per applicazioni *web*.

JSF mette infatti a disposizione una serie di componenti predefiniti che lo sviluppatore deve semplicemente richiamare nelle proprie pagine *web*. L'utente, interagendo con tali componenti di interfaccia scatena determinati eventi che vengono gestiti dal *framework* tramite opportuni *listener*.

JavaServer Faces può essere quindi visto come un ibrido tra i due modelli di *framework* analizzati, in quanto è di fatto un *framework MVC* del modello a componenti, ma conserva ancora numerosi aspetti tipici dei modelli di *framework* classici quale ad esempio l'utilizzo di un *file* di configurazione per gestire direttamente il flusso di esecuzione dell'applicazione.

Nel 2009 è uscita la versione 2 del *framework* (v. [7] e [8]) che ha introdotto numerosi miglioramenti tra i quali la semplificazione del file principale di configurazione tramite l'introduzione del meccanismo delle annotazioni e della navigazione implicita, il supporto diretto per la tecnologia *AJAX* e l'introduzione della tecnologia *Facelets* come *default* per la parte di presentazione al posto delle pagine *JSP*

[2] Componente che si occupa di gestire il *mapping* delle varie richieste

[3] Componente che in base a quanto dichiarato nel *file* di configurazione associa il nome di una stringa ad una pagina specifica

che vengono considerate deprecate. Queste ed altre differenze sono descritte nella tabella 2.4, che permette ad uno sviluppatore *JSF* 1 di capire le principali novità introdotte dalla nuova versione.

	JSF 1	***JSF 2***
Presentation Technology	Utilizzo di *JSP* e *Facelets*	*Facelets* inserito come *standard*
Estensione dei componenti	Molte operazioni per definire un nuovo componente	Funzionalità *composite components* per definire un nuovo componente modificando soltanto un unico *file xml*
AJAX	Supporto più limitato	Possibilità di utilizzo del *tag* <*f:ajax*> e di nuove *API*
Salvataggio dello stato	E' possibile salvare soltanto l'intero stato del *component tree*	E' possibile effettuare il salvataggio parziale dello stato agendo quindi solo sulle modifiche dell'albero dei componenti
Tipologie di eventi	Gestione di *FacesEvent* e *PhaseEvents*	Aggiunta dei *SystemEvents* divisi a sua volta in *global system events* e in *component system events*
Navigazione	Gestione della navigazione esclusivamente tramite *faces-config.xml*	Inserimento della navigazione implicita (senza il passaggio dal *faces-config.xml*) e di navigazione condizionale
Gestione richieste	Possibilità di gestione di richieste di tipo *POST*	Possibilità di gestione di richieste di tipo *POST* e di tipo *GET*
Scope	Presenza di *Application Scope*, *Session Scope* e *Request Scope*	Aggiunta alle precedenti di *View Scope*, *Flash Scope* e *Custom Scope*
Annotazioni	Configurazione necessariamente inserita nel *faces-config.xml*	Possibilità di utilizzo delle annotazioni per eliminare codice dal file di configurazione
Validazione	Utilizzo di <*f:validateBean*>	Introduzione di <*f:validateRequired*> e <*f:validateRegexp*>
Tree Visiting	Presenti solo alcune *API* per la navigazione dell'albero dei componenti	Aggiunta di ulteriori *API* per la navigazione dell'albero dei componenti quali ad esempio *UIComponent.visitTree()*

Tabella 2.4: Differenze tra *JSF* 1 e *JSF* 2

2.1.3.2 Struttura e funzionamento

I componenti principali di un'applicazione basata su *JSF* 2 sono:

- *web.xml*: nel descrittore di *deployment* dell'applicazione *web* va inserita la dichiarazione della "*FacesServlet*", ovvero del controllore principale dell'applicazione che si occupa di gestire tutte le richieste;

- *faces-config.xml*: è il file di configurazione di un'applicazione creata con *JSF* e contiene la dichiarazione dei vari oggetti utilizzati dall'applicazione (salvo quelli dichiarati tramite il meccanismo delle annotazioni) e le regole di navigazione tra le varie pagine;
- *managed bean*: sono *bean* riconosciuti ed utilizzati dal *framework JSF*; vanno dichiarati nel *faces-config.xml* oppure marcati con l'annotazione specifica;
- altre classi non utilizzate come *managed bean;*
- altri *file* di configurazione quali ad esempio quelli per i messaggi utilizzati dall'applicazione.

Il ciclo di vita di una richiesta in un'applicazione basata su *JSF* 2 è illustrato in figura 2.3.

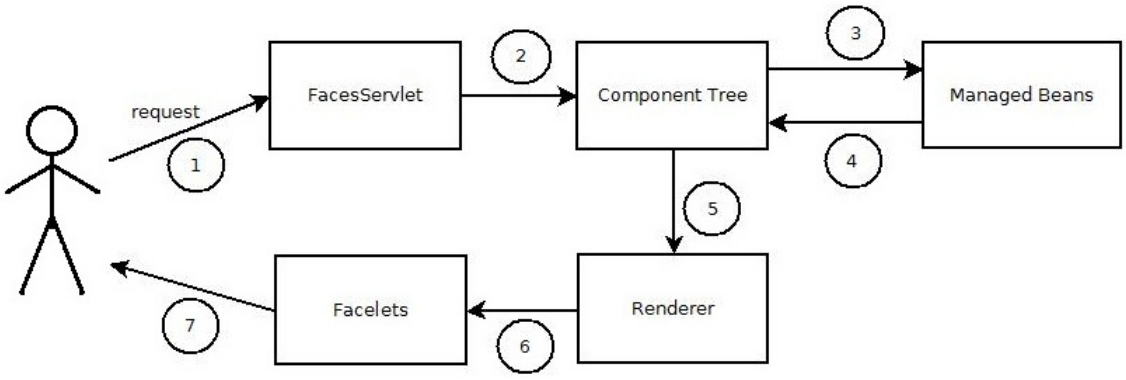


Figura 2.3: Ciclo di vita di una richiesta in un'applicazione basata su *JSF* 2

Nel dettaglio quindi:

1. L'utente effettua una richiesta al *server* per una qualche risorsa
2. La *FacesServlet* si prende carico della richiesta e recupera l'albero dei componenti (creandolo se si tratta della prima richiesta), memorizzandovi i dati associati alla richiesta ed effettuando le opportune operazioni di validazione
3. I valori dei componenti della pagina sono associati alle proprietà dei *Managed bean*[4] coinvolti
4. Vengono eseguiti gli *action method* dei *bean* che eseguono la logica desiderata e consentono così l'aggiornamento dell'albero dei componenti

[4]detti anche *backing bean*

5. Viene effettuata un'iterazione sull'albero dei componenti e per ognuno di essi viene individuato il "*Renderer*" opportuno, ovvero la classe incaricata di tradurre il componente in una rappresentazione comprensibile lato *client*

6. Ogni *Renderer* provvede a tradurre il componente aggiornando la pagina *web*

7. La pagina *web* viene visualizzata dall'utente tramite il *browser*

2.1.4 *Wicket*

2.1.4.1 Introduzione

Wicket è nato nel 2004 per mano di Jonathan Locke, ma la versione 1.0 è stata ufficialmente rilasciata nel 2005, per poi diventare un vero e proprio progetto *Apache* nel 2007 [9].

Apache Wicket è un *framework* che rispecchia il modello a componenti, infatti gli elementi di *markup* utilizzati per caratterizzare le azioni gestite dal *framework* sono associati a veri e propri componenti *Java* che devono essere letteralmente aggiunti alla pagina da visualizzare, come sarà più chiaro nel seguito.

Infatti un'applicazione *Wicket* è costituita da un albero di componenti a cui sono associati opportuni *listener* che gestiscono gli eventi scatenati dalla pressione di un *link* o dall'invio dei dati di un *form*.

Wicket permette di semplificare al massimo programmazione *web* vera e propria, infatti nelle pagine *web* viene gestito esclusivamente il *layout* della pagina, mentre la *business logic* viene gestita interamente tramite linguaggio *Java*. Questo permette di evitare l'utilizzo di pagine *JSP* in quanto non è necessario inserire alcuna logica nelle pagine *web*, ma solamente *markup*, quindi per le viste vengono utilizzate soltanto pagine *HTML*.

Con *Wicket* è possibile sviluppare le pagine *web* in maniera totalmente indipendente rispetto al resto dell'applicazione. E' sufficiente infatti inserire un semplice attributo ad alcuni componenti delle pagine *web* per associare il loro contenuto alla logica di *business* dell'applicazione.

2.1.4.2 Struttura e funzionamento

I componenti principali di un'applicazione basata su *Wicket* sono:

- *web.xml*: è il descrittore di *deployment* di ogni applicazione *web*; in questo *file* va inserita la dichiarazione del "*WicketFilter*", ovvero del filtro che ge-

stirà determinate richieste indirizzandole alla classe base di *Wicket* che avrà la funzione di *controller* centrale;

- "*Controller*" di *Wicket* che è una classe *Java*, chiamata per prima, che estende la classe "*org.apache.wicket.protocol.http.WebApplication*" e di fatto rappresenta l'applicazione stessa;

- pagine *web* in formato *HTML* associate ognuna ad una classe *Java* con lo stesso nome; ogni *tag* associato ad un componente *Wicket* lato *Java* deve avere l'attributo "*wicket:id*";

- classi *Java* che estendono la classe "*org.apache.wicket.markup.html.WebPage*" e che definiscono la logica di *business* che interagisce con le pagine *web* tramite l'inserimento di tutti i componenti necessari nelle pagine stesse.

Il *Controller* di *Wicke*t non prevede il controllo del flusso tra varie pagine, ma piuttosto tra varie classi *Java*, nelle quali è inserito il codice per la gestione dei componenti della pagina associata a tale classe.

Il comportamento alla base di *Wicket* è quindi concettualmente molto semplice, soprattutto per uno sviluppatore *Java*: bastano infatti poche linee di codice per definire la configurazione di alcune pagine *web*. Infatti la dichiarazione del *WicketFilter* avviene nel *file web.xml*, ma non esiste un *file* di configurazione analogo ad esempio a vari *file XML* di configurazione di *Struts* 2 e di *Spring MVC*.

Tutte le operazioni principali sono quindi gestite esclusivamente tramite codice *Java*; la validazione dei campi di un *form*, ad esempio, viene gestita esclusivamente tramite codice *Java* nella classe associata alla pagina *web* in cui è presente il *form*. Non è richiesta alcuna scrittura di *file XML* o alcun attributo "*validate*" nella pagina *web*.

Il funzionamento di massima di un'applicazione basata su *Wicket* è quello indicato in figura 2.4.

Analizzando il ciclo di vita di una richiesta nel dettaglio si ha:

1. L'utente effettua una richiesta al *server* per qualche risorsa

2. Il *WicketFilter* analizza la richiesta delegandola al *controller* di *Wicket*

3. Viene eseguito il metodo "*getHomePage()*" del *controller* che restituisce la classe indicata come *home page*

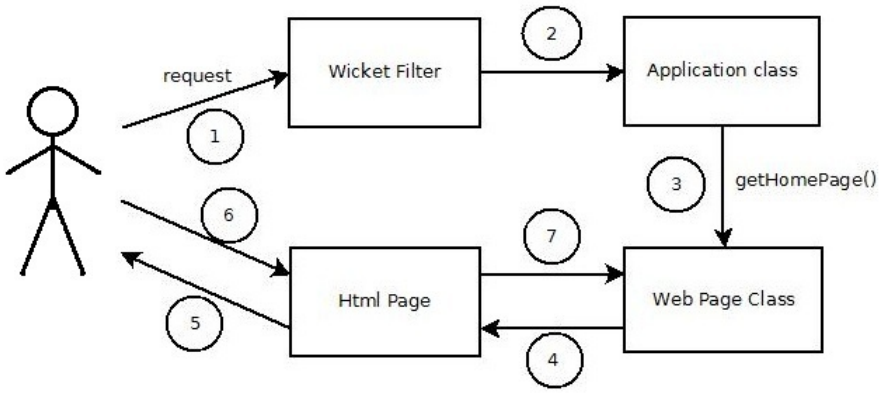


Figura 2.4: Ciclo di vita di una richiesta in un'applicazione basata su *Wicket*

4. La classe selezionata esegue il codice per creare i componenti della pagina *web* associati agli identificatori "*wicket:id*" presenti nella pagina *HTML* legata alla classe stessa

5. Viene visualizzata la pagina *HTML* all'utente

6. L'utente interagisce con la pagina *web* scatenando determinati eventi

7. Tali eventi vengono gestiti dalla classe legata alla pagina che effettua le opportune elaborazioni

Maggiori approfondimenti su *Apache Wicket* possono essere letti in [10].

2.1.5 *Vaadin*

2.1.5.1 Introduzione

Vaadin è un *framework open source* per la creazione di applicazioni *web* derivante da alcuni progetti iniziali da cui è nato nel 2007 l'*IT Mill toolkit*, diventato ufficialmente *Vaadin* nel 2009.

Anche *Vaadin* è un *framework* basato sul modello a componenti, ma la caratteristica più importante è che con *Vaadin* lo sviluppatore si occupa solo di gestire codice *Java*, e teoricamente può non conoscere minimamente il linguaggio *HTML* o *Javascript*. Le pagine vengono infatti create dinamicamente dal *framework* tramite opportuni *widget* inseriti appunto in classi *Java*.

2.1.5.2 Struttura e funzionamento

Vaadin consiste in un *server-side framework* (*Vaadin* appunto) e un *client-side engine* (*Google Web Toolkit*) che provvede a creare dinamicamente le pagine *web* da presentare all'utente a partire dal codice *Java* dell'applicazione. I componenti *default* di *Vaadin* possono essere estesi utilizzando proprio i *widget* e i temi di *GWT*.

In pratica *GWT* si occupa della parte di *presentation*, mentre *Vaadin* della parte di *application logic*.

Il *framework* riceve le richieste tramite un "*Terminal Adapter*" ed interpreta gli eventi a cui sono associati i componenti di *User Interface*. L'applicazione effettua poi i cambiamenti necessari ai componenti e li restituisce al *web browser* creando la nuova pagina *web* (tramite tecnologia *AJAX*). Un'applicazione basata su *Vaadin* non ha alcuna pagina *JSP* di *default*, ma ha invece una classe *Java* di *default* che estende "*com.vaadin.Application*".

Come già visto nel caso di *Wicket* non sono presenti *file XML* di configurazione aggiuntivi al descrittore di *deployment* di ogni applicazione *web*.

La classe di default ha un metodo "*init()*" che crea la pagina di benvenuto tramite codice *Java*, creando letteralmente la finestra, e tramite il quale lo sviluppatore può interagire aggiungendo i componenti di *User Interface* forniti da *Vaadin* che ritiene più opportuni.

I componenti principali di un'applicazione *Vaadin* sono quindi:

- *web.xml*: descrittore di *deployment* in cui viene dichiarata l'"*ApplicationServlet*" di *Vaadin* che gestirà le richieste smistandole verso la classe principale dell'applicazione;
- "*Controller*" di *Vaadin*, che è rappresentato dalla classe principale che estende "*com.vaadin.Application*" e che crea la finestra principale dell'applicazione;
- ulteriori classi *Java* tramite le quali creare le altre viste dell'applicazione da aggiungere alla finestra principale.

Il ciclo di vita di una richiesta gestita da un'applicazione basata su *Vaadin* è indicato in figura 2.5.

Nel dettaglio:

1. L'utente effettua una richiesta al *server* per qualche risorsa
2. L'*ApplicationServlet* analizza la richiesta delegandola al *controller* di *Vaadin*

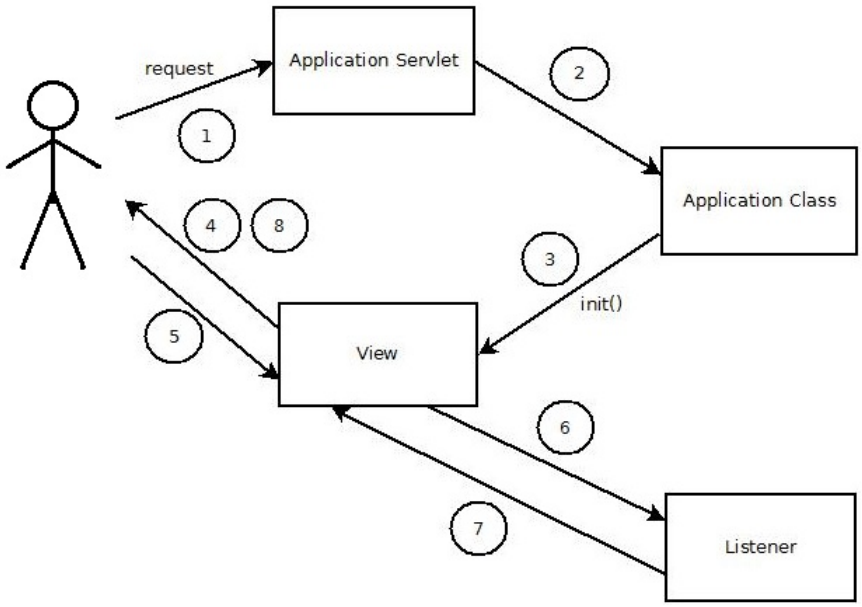


Figura 2.5: Ciclo di vita di una richiesta in un'applicazione basata su *Vaadin*

3. Viene eseguito il metodo "*init()*" che crea la finestra principale con tutti i componenti necessari

4. Il *client engine* di *GWT* provvede a creare la relativa finestra nel *browser* per l'interazione con l'utente

5. L'utente interagisce con la finestra creata generando determinati eventi

6. Tali eventi vengono catturati dagli opportuni *listener*

7. I *listener* provvedono a effettuare le opportune modifiche alla pagina

8. La pagina viene visualizzata all'utente

Una guida importante per capire a fondo il funzionamento di *Vaadin* è [12], scaricabile direttamente dal sito di ufficiale [11].

2.1.6 Scelta del *framework*

I *framework* analizzati rappresentano alcune delle più importanti alternative presenti sul mercato. Alcuni di questi *framework* garantiscono l'affidabilità ed il supporto grazie al loro grande utilizzo nella comunità degli sviluppatori *Java*. Altri, più recenti ma non meno potenti, forniscono delle caratteristiche nuove che colpiscono per le loro potenzialità.

Nel seguito il confronto tra i vari *framework* verrà fatto inizialmente in base alla tipologia. Per dare un'idea di base della differenza strutturale imposta dai *framework* verrà effettuato un confronto prendendo come esempio una semplicissima applicazione composta da due pagine *web*, la prima contenente un *form* di *login* (figura 2.6), la seconda contenente un messaggio di benvenuto nel caso i dati inseriti nel *form* siano quelli corretti (figura 2.7).

Figura 2.6: Schermata di *login* dell'applicazione di esempio

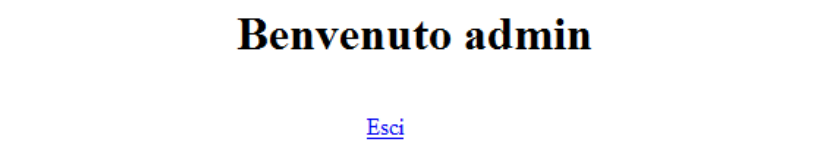


Figura 2.7: Schermata di accesso avvenuto dell'applicazione di esempio

2.1.6.1 *Struts* 2 vs *Spring MVC*

Un primo confronto può essere fatto analizzando i due *framework* che seguono il modello *JSP/Servlet*. Entrambi i *framework* sono molto maturi ed utilizzati moltissimo nella comunità *Java*.

I sostenitori di *Spring* indicano nel relativo *modulo MVC* la scelta migliore tra i due per lo sviluppo di applicazioni *web*, basandosi sul fatto che *Spring MVC* utilizzi una metodologia di programmazione più moderna e più ad alto livello rispetto a *Struts*. In realtà questo è vero soltanto in caso di confronto con la prima versione di *Struts*, in quanto successivamente le cose sono cambiate notevolmente anche per tale prodotto.

Analizzando più approfonditamente i due *framework* si può intuire quanto le differenze, indicate di seguito, siano molto minori di quanto ci si possa effettivamente aspettare.

Action vs Controller + Command Object

La differenza forse più sostanziale che si può notare tra la struttura dei due *framework* è il ruolo delle *Action* che svolgono in *Struts* 2 i compiti svolti dai *Command Object* e dai *Controller* di *Spring MVC*. Infatti come accennato precedentemente nella sezione relativa a *Struts* le *Action* comprendono sia la dichiarazione delle proprietà associate ai campi di un *form* con i relativi metodi *getter* e *setter* (analogamente a quanto avviene nei *Command Object* di *Spring MVC*), sia il metodo che esegue la logica di *business* e restituisce un risultato analizzato dal *framework* che provvederà a restituire la vista opportuna (come avviene per i *Controller* di *Spring MVC*).

Il codice relativo alla *Action* di *Struts* che gestisce la procedura di *login* dell'applicazione di esempio avrà il contenuto seguente:

```
public class Login extends ActionSupport {

   private String username;
   private String password;

   public String getPassword() {
      return password;
   }

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

   public String getUsername() {
      return username;
   }

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

   @Override
   public String execute() throws Exception {
      if (getUsername().equals("admin") && getPassword().equals("admin")) {
         addActionMessage("Benvenuto " + getUsername());
         return SUCCESS;
      } else {
         addActionError("Credenziali errate");
         return ERROR;
      }
   }
}
```

Come si può vedere dal codice, in una *Action* di *Struts* sono presenti sia delle proprietà, che tramite i metodi "*setProperty()*" e "*getProperty()*" saranno collegate direttamente ai campi del *form*, sia il metodo "*execute()*", che contiene la logica di *business* da eseguire.

Con *Spring MVC* è necessario utilizzare invece due classi per ogni *form* gestito, la prima contenente le proprietà ed i relativi metodi di accesso relative ai campi del *form*:

```
public class Utente {

   private String username;

   private String password;


   public String getPassword() {
      return password;

   }


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

   }


   public String getUsername() {
      return username;

   }


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

La seconda contenente esclusivamente la logica di *business*:

```
public class LoginController extends SimpleFormController {


   public LoginController() {
      setCommandClass(Utente.class);
      setCommandName("utente");

   }


   @Override
   protected ModelAndView onSubmit(Object command) throws Exception {
      Utente utente = (Utente) command;
      ModelAndView mv = null;
```

```
        if (utente.getUsername().equals("admin")
              && utente.getPassword().equals("admin")) {
           mv = new ModelAndView(getSuccessView());
           mv.addObject("messaggio", "Benvenuto " + utente.getUsername());
        else {
           mv = new ModelAndView(getFormView());
           mv.addObject("utente", utente);
           mv.addObject("messaggio", "credenziali.errate");
        }
        return mv;
     }
  }
```

Questa differenza strutturale comporta dei vantaggi e degli svantaggi. Sicuramente raccogliere tali funzioni in unico file permette di tenere la struttura dell'applicazione più compatta, in quanto la divisione che avviene in *Spring MVC* comporta necessariamente la proliferazione di molte classi. Nello stesso tempo però dal punto di vista della progettazione può essere sicuramente una soluzione migliore separare le due funzioni in classi distinte in maniera da avere una maggior manutenibilità ed una maggior divisione dei compiti.

É infatti possibile gestire in maniera separata la struttura degli oggetti che mantengono (almeno temporaneamente) i dati, dagli oggetti che invece utilizzano tali dati, ma fanno anche elaborazioni che possono coinvolgere più oggetti, come risulta più chiaro dalla figura 2.8 che illustra proprio le differenze strutturali tra l'applicazione di esempio creata con *Struts* 2 e quella creata con *Spring MVC*.

Questa differenza può essere utile nella scelta del *framework* in base al numero di componenti del *team* di sviluppo: la struttura più raccolta di *Struts* 2 consente di aver maggiormente sotto controllo i componenti dell'applicazione ed è probabilmente una soluzione migliore per *team* piccoli; la struttura più modulare di *Spring MVC* consente di dividere maggiormente i compiti in quanto nei *Controller* è gestita solo la logica di *business*, mentre nei *Command Object* è gestito il *mapping* dei campi del *form*.

UI Tag

Struts 2 fornisce un numero di *UI tag* maggiore con i quali è possibile fare quasi di tutto. La stessa cosa non si può dire per *Spring MVC*, infatti molto spesso gli sviluppatori tendono a ricorrere all'utilizzo di *JSTL* per colmare questa lacuna.

Questo comporta una maggior pulizia del codice di *Struts* 2, come risulta evidente anche nella semplice applicazione presentata, poiché per quanto riguarda *Struts* 2 è necessario utilizzare una sola tipologia di *tag* speciali (quelli con *namespace* 's'):

.

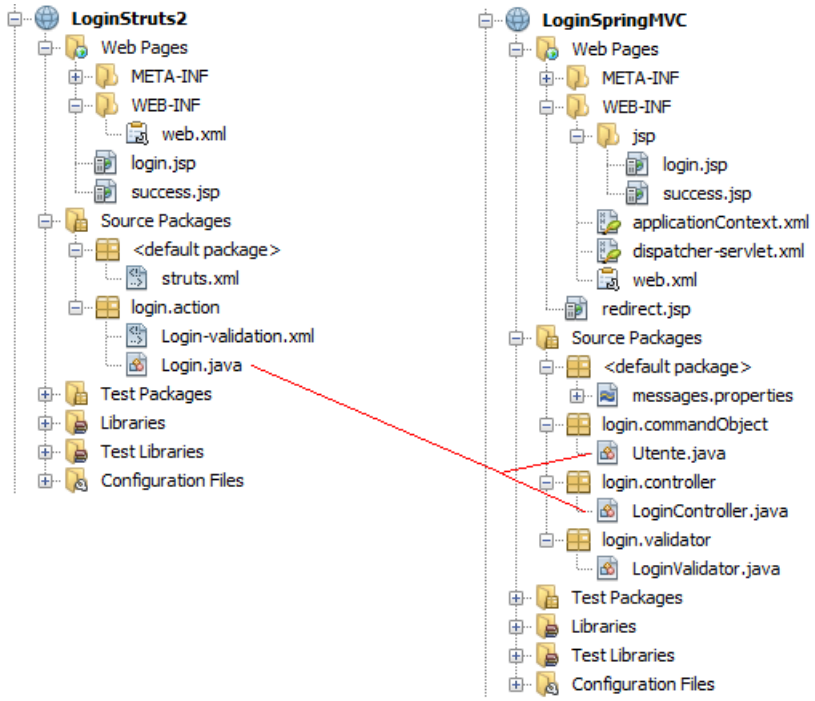


Figura 2.8: Differenza strutturale *Struts* 2 - *Spring MVC*

```
.
<body>
  <h1 align="center">Inserire i dati per l'accesso al sistema</h1>
  <s:form action="Login" validate="true">
    <table align="center">
      <tr>
        <td><s:textfield label="Username" name="username" size="10"/></td>
      </tr>
      <tr>
        <td><s:password label="Password" name="password" size="10"/></td>
      </tr>
    </table>
    <table align="center">
      <tr><td><s:actionerror/></td></tr>
    </table>
    <table align="center">
      <tr>
        <td><s:submit value="Invia dati"/></td>
        <td><s:reset /></td>
      </tr>
    </table>
  </s:form>
</body>
.
.
```

Con *Spring MVC* invece è necessario utilizzare altre librerie di *tag* personalizzate per gestire alcune operazioni. Nell'esempio che segue viene illustrato come siano

effettivamente utilizzati vari *tag* speciali con differenti *namespace* (*form*, *spring* e c[5]) semplicemente per realizzare con meno sforzo possibile ciò che con *Struts* era effettuato tramite una sola libreria:

```
.
.
<body>
   <h1 align="center">Inserire i dati per l'accesso al sistema</h1>
   <form:form method="post" commandName="utente">
      <table align="center">
         <tr>
            <td>Username:</td>
            <td><form:input path="username" size="10"/></td>
         </tr>
         <tr>
            <td>Password:</td>
            <td><form:password path="password" size="10"/></td>
         </tr>
      </table>
      <table align="center">
         <tr><td><form:errors path="*"/></td></tr>
         <c:if test="${messaggio != null}">
            <tr><td><spring:message code="${messaggio}"/></td></tr>
         </c:if>
      </table>
      <table align="center">
         <tr>
            <td><input type="submit" value="Invia dati"></td>
            <td><input type="reset" value="Reset"></td>
         </tr>
      </table>
   </form:form>
</body>
.
.
```

Integrazione con altri framework

Spring MVC da questo punto di vista segna un punto a suo vantaggio. In effetti entrambi i *framework MVC* possono essere integrati con *Spring* che al momento è considerato uno dei migliori *framework* che riesce a coprire tutti i principali aspetti dello sviluppo. Il fatto che però *Spring MVC* sia un modulo di *Spring* consente un'integrazione più veloce, in quanto la logica alla base dei due *framework* è molto più simile di quanto non lo sia tra *Struts* 2 e *Spring*.

Documentazione

Struts è stato uno dei primi *framework MVC* e sicuramente è stato tra i più utilizzati. Questo consente di avere un'abbondante documentazione anche se nella maggior parte dei casi è relativa alla prima versione.

[5]dalla libreria *JSTL standard*

Spring è in notevole crescita e di conseguenza anche il relativo modulo *MVC*, quindi anche se la documentazione al momento è minore, da questo punto di vista si può dire che non ci siano grossi problemi in nessuno dei due casi.

Utilizzo

Struts 1 ha raggiunto una notevolissima diffusione, anche perché quando fu proposto sul mercato non vi erano molti *framework* evoluti. Di conseguenza risultano ancora oggi presenti moltissime applicazioni basate su *Struts* 1. Questo è ovviamente un vantaggio per chi intende utilizzare *Struts* 2 perché, nonostante le grosse differenza tra le due versioni, molti concetti di base sono analoghi, e di conseguenza risulta più facile una migrazione da *Struts* 1 a *Struts* 2 che non da *Struts* 1 a *Spring MVC*.

Questo ha portato ad un livello di robustezza leggermente maggiore di *Struts* 2 rispetto a *Spring MVC*.

2.1.6.2 *JSF 2 vs Wicket vs Vaadin*

Per quanto riguarda i *framework* a componenti analizzati siamo sicuramente in presenza di una maggior differenziazione nella struttura di un'applicazione *web* realizzata basandosi su di essi.

La differenza più sostanziale può essere notata andando ad analizzare la tecnologia con cui viene creata la parte di presentazione:

- *JSF* 2 utilizza le *Facelets* come pagine *web* presentata all'utente, basandosi quindi sulla tecnologia *XHTML*;
- *Wicket* utilizza *HTML* puro;
- *Vaadin* non utilizza, come visto, alcuna tecnologia classica per lo sviluppo di pagine *web*, ma si basa piuttosto sulla creazione delle stesse utilizzando esclusivamente codice *Java* interpretato da un opportuno *Renderer*.

Tralasciando per un momento *JSF* 2 e *Wicket* è opportuno soffermarsi sulla novità introdotta da *Vaadin*. L'utilizzo esclusivo di codice *Java*, porta sicuramente dei vantaggi, ma anche degli svantaggi.

La creazione di pagine *web* è spesso un'operazione fastidiosa per uno sviluppatore *Java*, che deve perdere molto tempo a creare la struttura delle pagine utilizzando un linguaggio "più primitivo" come l'*HTML* o l'*XHTML*.

Da questo punto di vista *Vaadin* offre la possibilità di imparare, almeno dal punto di vista teorico, un solo linguaggio, anche se è difficile trovare uno sviluppatore di

applicazioni *web* che non abbia un po' di dimestichezza con il linguaggio *HTML*; è bene sottolineare però che a volte può essere utile avere a disposizione la possibilità di metter mano direttamente al sorgente della pagina, magari per piccoli cambiamenti personalizzati.

Purtroppo programmare le interfacce *web* utilizzando codice *Java* non è così immediato per ogni sviluppatore, di conseguenza *Vaadin* rischia di avere tempi di apprendimenti più lunghi, almeno che lo sviluppatore non abbia grande dimestichezza con linguaggi quali *Swing* o *AWT*, e questo potrebbe essere un piccolo ostacolo per alcuni sviluppatori.

Prima di pensare all'eventuale utilizzo di *Vaadin* è però fondamentale sottolineare il fatto che tale *framework* è stato creato per la realizzazione di applicazioni *web* anche complesse dal punto di vista della logica di *business* e non semplici siti *web* in cui una grafica accattivante potrebbe avere un ruolo fondamentale.

Realizzare la semplice applicazione di esempio con *Vaadin* può essere molto interessante per alcuni, un'attività particolarmente complessa per altri, come è facilmente comprensibile dal codice della classe che rappresenta la finestra contenente il *form* di *login*:

```
public class FormLogin extends VerticalLayout {


   TextField username;
   TextField password;
   Form loginForm;

   VerticalLayout appLayout;
   public FormLogin() {
      username = new TextField("username");
      password = new TextField("password");
      password.setSecret(true);
      loginForm = new Form();
      loginForm.setCaption("Inserire i dati per l'accesso al sistema");
      loginForm.addField(username, username);
      loginForm.addField(password, password);
      loginForm.getField(username).setRequired(true);
      loginForm.getField(password).setRequired(true);
      addComponent(loginForm);
      Button resetButton = new Button("Reset", new ClickListener() {
         public void buttonClick(ClickEvent event) {
            loginForm.getField(username).setValue("");
            loginForm.getField(password).setValue("");
         }
      });
      addComponent(resetButton);
      Button loginButton = new Button("Login", new ClickListener() {
         public void buttonClick(Button.ClickEvent event) {
            String user = (String) loginForm.getField(username).getValue();
```

```
            String pass = (String) loginForm.getField(password).getValue();
            if (validateUser(user, pass)) {
               loggedIn(user);
            } else {
               loginForm.setComponentError(new UserError("username e/o password errati"));
            }
         }
      });
      addComponent(loginButton);
   }

   private boolean validateUser(String username, String password) {
      if (username.equals("admin") && password.equals("admin")) {
         return true;
      }
      return false;
   }

   private void loggedIn(String username) {
      .
      .
   }
}
```

A prima vista il codice può sembrare un po' complesso per la semplice operazione da effettuare, ma è importante sottolineare che basta implementare il metodo "*loggedIn()*", in maniera che crei la nuova finestra con il messaggio di benvenuto, per avere a disposizione l'applicazione funzionante senza alcun *file* aggiuntivo, in quanto in tale codice è presente sia la creazione della pagina iniziale di *login*, sia la logica che gestisce la procedura di accesso.

Anche *Wicket* è un *framework* a componenti, ma con caratteristiche sicuramente differenti rispetto a *Vaadin*: come visto utilizza sia *Java*, per la parte di *business logic*, che *HTML*, per la parte di presentazione.

Questo comporta il vantaggio di avere una separazione netta tra i compiti delle varie parti dell'applicazione e ciò può comportare un notevole vantaggio soprattutto per quanto riguarda la divisione dei compiti all'interno di un eventuale *team* di sviluppo polifunzionale; infatti l'utilizzo di puro *HTML* per la parte di presentazione, con la sola aggiunta degli attributi che consentono di collegare i componenti lato *client* a quelli lato *server*, permette di utilizzare ad esempio un *team* esperto di grafica *web*, anche senza conoscenze di *Java*, per sviluppare la sola parte di presentazione in quanto la parte di *business logic* è sviluppabile in maniera quasi completamente indipendente e quindi può essere affidata ad un team esperto di sviluppo

Java. La pagina di accesso dell'applicazione *web* di esempio presenta ad esempio il seguente codice *HTML*:

```
.
.
<body>
   <h1 align="center">Inserire i dati per l'accesso al sistema</h1>
   <form wicket:id="loginForm">
      <table align="center">
         <tr>
            <td>Username:</td>
            <td>
               <input type="text" wicket:id="username" size="10"/>
            </td>
         </tr>
         <tr>
            <td>Password:</td>
            <td>
               <input type="password" wicket:id="password" size="10"/>
            </td>
         </tr>
      </table>
      <table align="center">
         <tr>
            <td><input type="submit" value="Invia dati"/></td>
            <td><input type="reset"/></td>
         </tr>
      </table>
   </form>
   <table align="center">
      <tr><td><span wicket:id="messaggi"/></td></tr>
   </table>
</body>
.
.
```

Come si può vedere è puro codice *HTML* e quindi molto semplice da scrivere e modificare per uno sviluppatore *web*, anche se contiene un discreto numero di linee di codice.

Come detto però per ogni pagina *web* deve essere presente una coppia di *file*, uno *HTML* ed uno *Java.* Questo comporta ulteriore codice come quello seguente, relativo appunto alla classe *Java* che contiene i componenti della pagina di *login*:

```
public class Login extends WebPage {
   private String username;
   private String password;

   public Login() {
      FeedbackPanel feedback = new FeedbackPanel("messaggi");
      add(feedback);
      Form loginForm = new Form("loginForm", new CompoundPropertyModel(this)) {
```

```
            @Override
            protected void onSubmit() {
               if (username.equals("admin") && password.equals("admin")) {
                  PageParameters pp = new PageParameters();
                  pp.add("messaggio", "Benvenuto " + username + "!");
                  setResponsePage(Success.class, pp);
               } else {
                  error("Username e/o password errati");
               }
            }
         };
         loginForm.add(new TextField("username").setRequired(true));
         loginForm.add(new PasswordTextField("password").setRequired(true));
         add(loginForm);
      }
   }
```

Si può notare come venga praticamente creata una pagina aggiungendo i componenti che interagiscono direttamente con l'utente, gestendo la stessa interazione con esso.

Wicket permette quindi una precisa separazione dei compiti, ma anche un notevole aumento delle linee di codice in quanto i componenti utilizzati vengono in pratica dichiarati due volte (tramite il *markup* nella pagina *HTML* e tramite i componenti *Java* nella relativa classe). Dalla figura 2.9 si vede il confronto tra la struttura dell'applicazione di esempio creata con *Wicket* e quella creata con *Vaadin*.

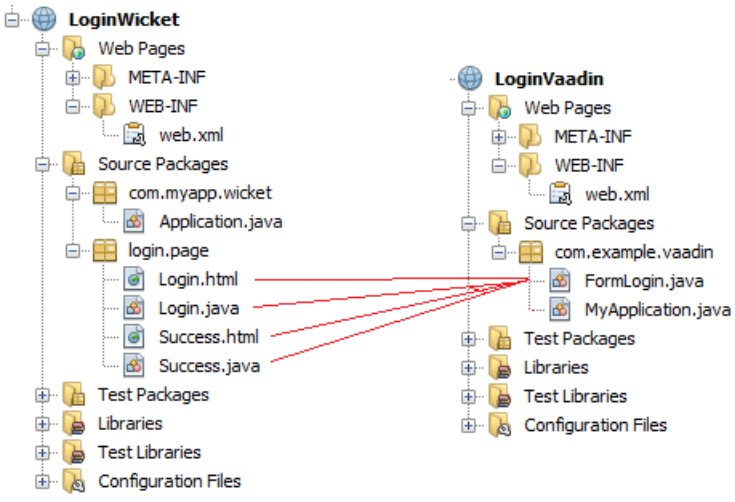


Figura 2.9: Differenza strutturale *Wicket* - *Vaadin*

JavaServer Faces, come detto nella sezione dedicata, è un *framework* a componenti che ha però alcune caratteristiche tipiche di *framework* come *Struts* o *Spring*. Storicamente (riferendosi alla prima versione) è il primo *framework* a componenti uscito tra i tre qui presentati; di conseguenza, nonostante le numerose novità

introdotte dalla seconda versione presenta meno particolarità rispetto a *Wicket* e *Vaadin*. Questo non è necessariamente uno svantaggio perché *JSF* 2 presenta caratteristiche di robustezza che gli altri due *framework* non possono al momento garantire, prima su tutte il supporto che la comunità degli sviluppatori può fornire grazie al notevole utilizzo negli anni passati, almeno della prima versione del *framework*.

Se si va analizzare l'applicazione di esempio utilizzata per l'analisi degli altri *framework* si può notare come la struttura di tale applicazione (v. figura 2.10) risulta molto simile a quella realizzata con *Struts*, nonostante si tratti di *framework* di diversa tipologia.

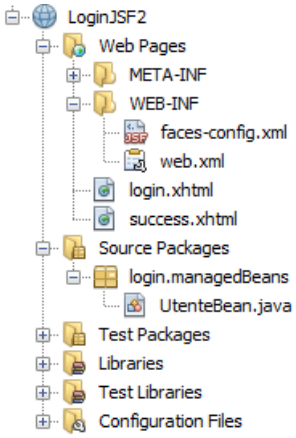


Figura 2.10: Struttura JSF 2

Rispetto alla struttura di *Struts* 2 (visibile in figura 2.8) infatti si nota soltanto l'assenza di un *file .xml* relativo alla validazione, gestita da *JSF* 2 in maniera diversa tramite il *listener* associato al componente con il contenuto da validare.

In effetti in *JSF* 2 siamo in presenza di componenti che non vengono però dichiarati esplicitamente in *Java* come avviene con *Wicket* o con *Vaadin*, ma semplicemente nella pagina *XHTML* in cui sono presenti tramite l'utilizzo delle specifiche librerie di *JSF*. Infatti la pagina di accesso dell'applicazione presenta la seguente struttura:

```
.
.
<h:body>
   <h1 align="center">Inserire i dati per l'accesso al sistema</h1>
   <h:form>
      <h:panelGrid columns="2">
         <h:outputText value="Username:"/>
         <h:inputText value="#{utente.username}"/>
         <h:outputText value="Password:"/>
         <h:inputSecret value="#{utente.password}"/>
```

```
            <h:commandButton value="Login" action="#{utente.login}"/>
         </h:panelGrid>
      </h:form>
   </h:body>
   .
   .
```

I *tag* con *namespace* 'h' (che si riferisce alla specifica libreria *HTML* di *JSF*) permettono infatti di inserire direttamente nella pagina componenti gestiti dalla *FacesServlet* di *JSF* e quindi collegati ad opportuni *listener* che si occuperanno di gestire l'interazione con essi. In maniera simile a quanto avveniva in *Struts* 2 i campi del *form* vengono associati alle proprietà di una classe *Java*, in questo caso un *ManagedBean*[6]:

```
@ManagedBean(name="utente")
public class UtenteBean {

   private String username;
   private String password;

   public UtenteBean() { }

   public String login() {
      String outcome;
      if ((username != null && username.trim().equals("admin"))
         && (password != null && password.trim().equals("admin"))) {
         outcome = "success";
      } else {
         outcome = "failure";
      }
      return outcome;
   }

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

   public String getUsername() {
      return username;
   }

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

[6]La somiglianza tra i due meccanismi è solamente dal punto di vista dello sviluppatore per quanto riguarda la scrittura del codice, in quanto, come analizzato precedentemente, il funzionamento dei due *framework* è differente, essendo utilizzati in questo caso dei veri e propri componenti.

```
    }

    public String getPassword() {
       return password;
    }
}
```

2.1.6.3 Quale scegliere

L'obiettivo di questa panoramica su alcune delle possibilità offerte nell'ambito dei *framework MVC Java* per lo sviluppo di applicazioni *web* è quello di dare alcuni suggerimenti di base utili per la scelta del supporto da utilizzare per lo sviluppo.

Sono stati scelti *framework* molto maturi e robusti, che offrono un notevole supporto da parte della comunità degli sviluppatori ed una serie di caratteristiche che permettono di realizzare tutte le funzionalità più tipiche delle applicazioni *web*.

Proprio per questo non è possibile affermare a priori quale *framework* sia migliore e quale peggiore, ma occorre analizzare l'ambito di utilizzo. Un *framework* per definizione è un supporto allo sviluppo, quindi molto dipende dalla tipologia e dalle abitudini di programmazione di uno sviluppatore. La metodologia di programmazione di alcuni potrebbe essere difficoltosa o comunque non ottimale per altri.

La scelta del *framework* dipende quindi in gran parte dal *team* di sviluppo che sia ha a disposizione e dalla tipologia di progetto che si è in procinto di iniziare.

Struts 2 e *Spring MVC* sono infatti soluzioni stabili e molto mature, ma non garantiscono le caratteristiche migliori in quanto a modularità e riusabilità, peculiari invece nei *framework* a componenti. *Spring MVC* garantisce una miglior modularità rispetto a *Struts* 2, che però potrebbe diventare un aspetto negativo nel caso lo sviluppo sia portato avanti da un *team* molto piccolo o addirittura da una sola persona.

I *framework* a componenti sono spesso caratterizzati da una semplicità maggiore ed una possibilità di separazione del codice più ampia che li rende in generale preferibili per *team* di sviluppo più ampi ed eterogenei.

In particolare *Wicket* grazie alla sua completa separazione tra presentazione e logica permette di sfruttare al massimo la possibile diversità di conoscenze all'interno del *team* consentendo di sfruttare le migliori capacità di ogni individuo. Non è invece consigliabile per sviluppatori singoli per la sua ridondanza in alcune fasi dello sviluppo.

Vaadin, trattandosi di puro *Java*, può avere un *target* di utilizzo che può essere leggermente più limitato, in quanto è più difficile da apprendere per chi non ha dimestichezza con la metodologia di programmazione a eventi pura. La disponibilità di componenti però è in continuo aumento e questo lo rende particolarmente appetibile per sviluppatori più "sperimentali".

JavaServer Faces ha il vantaggio notevole di essere lo *standard* della *Sun* e ciò è sicuramente una garanzia in più. La versione 2 ha portato alcune semplificazioni che hanno permesso di migliorare alcuni aspetti della prima versione che lo rendevano più simile a *framework* come *Struts* e *Spring*, con la complessità che li caratterizza. Tale *framework* può essere quindi un esempio di ottimo compromesso per chi vuole cercare di unire i vantaggi di entrambi i modelli cercando di eliminarne il più possibile gli svantaggi.

Capitolo 3

Sviluppo del prototipo

3.1 Ambito

Il progetto di seguito presentato è la realizzazione di una parte di un'applicazione *web* per la Maroil-Bardhal Italia, una ditta toscana che commercializza oli lubrificanti per motori.

Tutte le attività di gestione dell'azienda al momento sono realizzate presso l'azienda stessa mediante l'ausilio di un sistema gestionale basato sull'*As/400* della *IBM* (v. appendice A) realizzato tramite linguaggio *RPG*.

L'architettura del sistema attuale è rappresentata in figura 3.1.

Tale sistema si basa sul paradigma *client/server*. Il *server* è appunto rappresentato dall'*As/400* che mantiene le strutture dati ed effettua le elaborazioni, mentre presso ogni *pc* presente in azienda è installato un programma *client* visuale che si occupa di fornire l'interfaccia grafica attraverso la quale i dipendenti aziendali possono interagire con il sistema. Ogni *client* dialoga direttamente con il *server* mandando richieste ed attendendo le risposte.

Il *framework* con cui è stato sviluppato il sistema attuale è il *Visual RPG* della *IBM*, che è un programma *As/400 oriented*.

Questa scelta porta il notevole svantaggio di avere una totale assenza di portabilità in quanto si è in presenza di un'architettura nativa.

La Maroil ha deciso di investire nella realizzazione di un nuovo verticale nell'ottica di un processo generale di ammodernamento e di crescita dell'azienda.

Il processo di ammodernamento del sistema informativo aziendale è già iniziato con la realizzazione di un nuovo sito *internet* che ha aumentato la visibilità dell'azien-

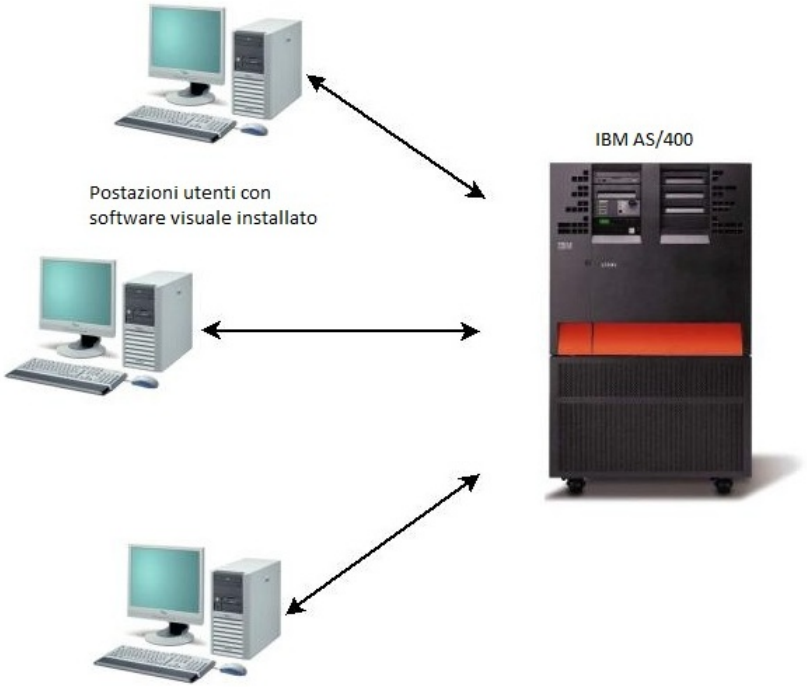


Figura 3.1: Architettura del sistema attuale

da e con la realizzazione di una *intranet*, che si basa su un *DBMS* non proprietario, che si occupa della redazione di *report* giornalieri.

Al momento l'azienda, dal punto di vista del sistema informativo aziendale, si trova però in una situazione di stallo. I sistemi informativi aziendali sono divenuti un ostacolo al *business* in quanto, allo stato dell'arte, l'azienda risulta essere totalmente dipendente dal proprio fornitore e nello stesso tempo il verticale attuale risulta di fatto essere obsoleto e difficilmente estensibile.

Inoltre finora le informazioni contenute nel sistema erano ad uso e consumo degli operatori interni all'azienda in quanto, al contrario di quanto accade oggi, non c'era né la necessità né l'intenzione di pubblicare informazioni o servizi verso l'esterno.

Allo stato attuale l'azienda si trova infatti spesso in difficoltà nel dare risposte certe, rapide e precise, proprio perché i dati sono attinti da un sistema gestionale da anni concepito per erogare informazioni ad un pubblico interno all'azienda diviso tra impiegati e agenti, tutti conoscitori dell'azienda stessa. Il gestionale attuale quindi utilizza un linguaggio comprensibile solo per chi è interno all'azienda e questo diventa un problema nel momento in cui sorge la necessità di esternalizzare alcuni servizi a personale esterno.

C'è anche un problema legato ai notevoli limiti prestazionali e non solo, in quanto

il sistema risulta essere cresciuto in seguito all'aggiunta di numerose funzionalità che però si sono rivelate mal integrate con il resto. Sono state infatti evidenziate difficoltà di analisi e di intervento ogni volta che è stato necessario effettuare una modifica al sistema.

Inoltre come detto l'azienda prevede di spostare alcune attività di gestione sul *web*, in maniera da usufruire di tutti i vantaggi derivanti da questa scelta, primi fra tutti la possibilità di effettuare una gestione completa senza la necessità di essere presenti *in loco*, con la conseguenza di velocizzare la procedura di aggiornamento delle informazioni, oltre alla possibilità di essere più vicini al cliente.

3.2 Obiettivi

L'obiettivo preposto è quindi la realizzazione di un'applicazione *web* che sostituisca il sistema attuale ampliandone prestazioni e funzionalità.

Nell'ambito di questo lavoro di tesi l'obiettivo è di creare l'applicazione stessa, prevedendo fin da subito un'alta manutenibilità per permettere l'aggiunta delle funzionalità future, ed inserire un modulo per la gestione degli ordini dei clienti dell'azienda.

La scelta di partire da tale modulo è dettata dal fatto che attualmente la gestione degli ordini è l'aspetto più urgente, soprattutto nell'ottica di una gestione via *web* degli stessi. L'intento è infatti quello di fornire un prodotto immediatamente funzionante, anche se con funzionalità inizialmente limitate rispetto a quelle offerte dal sistema preesistente, permettendo ai dipendenti dell'azienda di usufruire in tempi brevi delle funzionalità per cui è maggiormente richiesto un accesso via *web*.

Le scelte strategiche riguardo alle tecnologie da utilizzare sono state effettuate proprio in base agli obiettivi da raggiungere, infatti ci si è basati nello sviluppo dell'applicativo sul *framework JavaServer Faces* 2.

JavaServer Faces infatti, per le caratteristiche precedentemente discusse, garantisce la robustezza ed il supporto necessario per realizzare tale applicazione, riducendo al minimo il rischio di trovarsi in situazioni in cui sia difficile realizzare una specifica funzionalità a causa di un supporto o un'integrazione insufficiente del *framework*. Trattandosi di un'applicazione che prevederà numerose aggiunte future si è scelto anche di basarsi sulla seconda versione del *framework*, in quanto fornisce numerose funzionalità aggiuntive e soprattutto, trattandosi di una *release* piuttosto giovane, ma avendo una base molto solida, si prevede che la comunità dei suoi utilizzatori

cresca notevolmente, garantendo tutto il supporto che potrebbe essere necessario in futuro.

Per avere a disposizione un'interfaccia grafica più completa si è deciso di basarsi per la parte di presentazione su un altro *framework* da integrare con *JSF* 2 che fornisca una serie di componenti predefiniti da utilizzare all'interno dell'applicazione e la scelta è ricaduta su *PrimeFaces* (per cui si rimanda a [13] e [14]). Tale scelta è in parte legata alla disponibilità di componenti e alla loro realizzazione, ed in parte al fatto che avendo scelto di utilizzare *JSF* 2 è stato necessario sceglierne uno che fornisse una *release* stabile compatibile con tali specifiche.

3.3 Architettura

L'architettura prevista per la nuova applicazione è illustrata in figura 3.2.

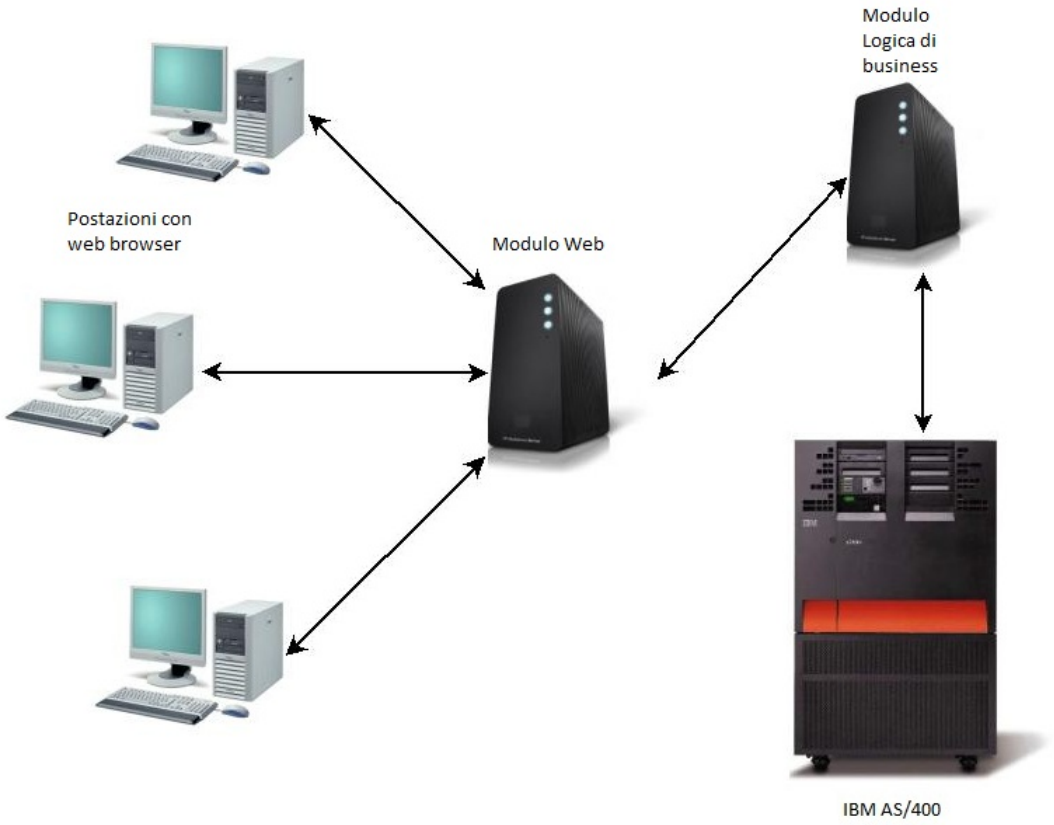


Figura 3.2: Architettura prevista per il nuovo sistema

Il *server* dell'applicazione è diviso in due moduli principali, il primo che si occupa della gestione delle schermate da presentare all'utente per l'interazione con esso, ed il

secondo che si occupa di gestire la logica di *business* con il relativo interfacciamento al *database*[1].

Gli utenti in questo caso utilizzeranno postazioni in cui sarà sufficiente un semplice *browser web* per accedere all'indirizzo dell'*application server* che si occupa di gestire l'interazione degli utenti stessi.

Per quanto riguarda l'accesso ai dati presenti sull'*As/400* è opportuno fare un'apposita riflessione.

Al momento come detto i dati sono tutti mantenuti sull'*As/400*. Per evitare di rimanere dipendenti da tale macchina, con il rischio di essere limitati negli sviluppi futuri l'azienda prevede di effettuare una migrazione verso un altro sistema di gestione dei dati, magari più performante e più innovativo.

Al momento però i dati gestiti dal sistema sono dati sensibili ed attuali, di conseguenza si è ritenuto opportuno dare la priorità al miglioramento dell'accesso ai dati piuttosto che alle modalità di mantenimento degli stessi. Allo stato attuale infatti non ci sono problemi di sicurezza dei dati, di conseguenza si prevede la migrazione effettiva delle informazioni solamente in tempi futuri.

3.4 L'applicazione

3.4.1 Creazione

Per la realizzazione del progetto ci si è basati su *Netbeans*, uno degli *IDE open source* di sviluppo più diffusi in commercio, utilizzando la versione 6.8.

Tramite *Netbeans* si è quindi provveduto a creare i due moduli che costituiscono l'applicazione:

- una nuova "*Web Application*", assegnandole un nome ed indicando inoltre l'utilizzo dell'*application server Glassfish* v3 (figura 3.3);

- un nuovo "*EJB Module*", assegnando anche ad esso il nome ed indicando l'utilizzo di *Glassfish* (figura 3.4);

3.4.2 Configurazione dell'*application server*

Per poter accedere ai dati contenuti nell'*As/400* dell'azienda in maniera da poter effettuare operazioni su di essi è necessario innanzitutto configurare l'*application*

[1] Al momento ancora interno all'As/400

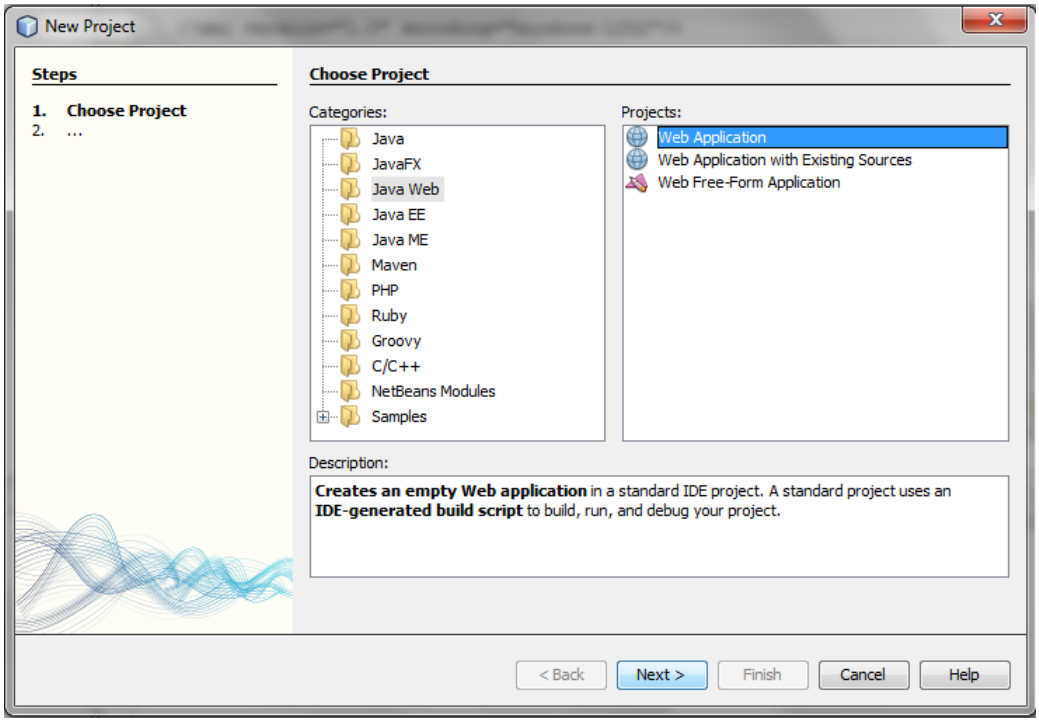


Figura 3.3: Finestra di Netbeans di creazione del modulo “Web Application”

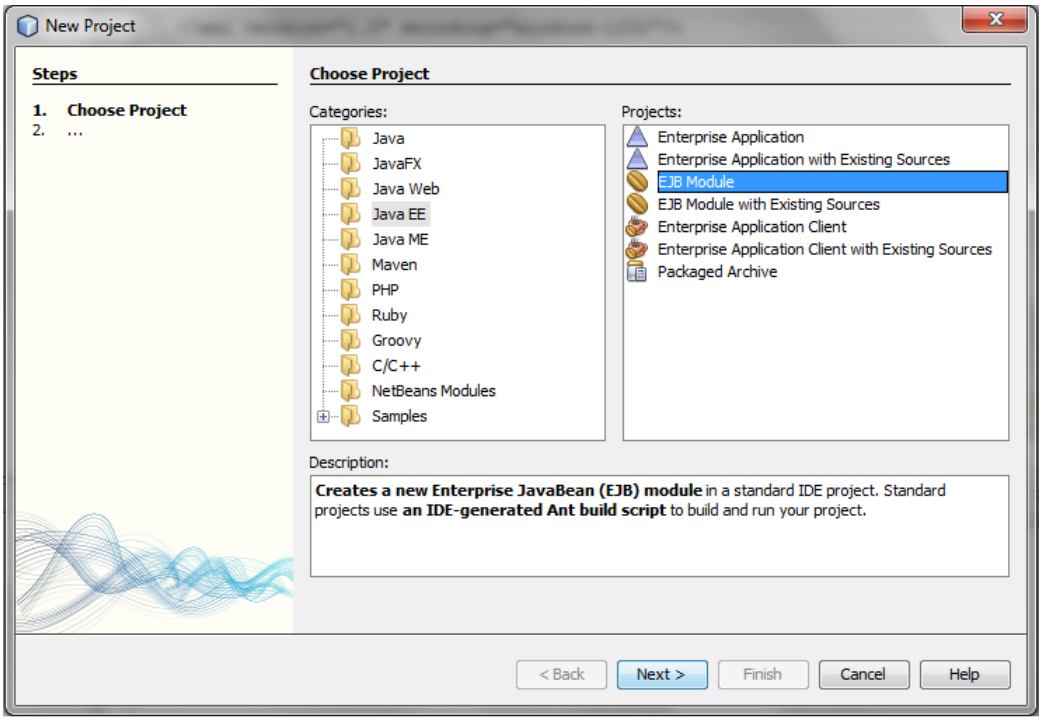


Figura 3.4: Finestra di Netbeans di creazione del modulo “EJB Module”

server utilizzato indicando le impostazioni necessarie per tale accesso. Si rimanda all'appendice B per i passi necessari ad effettuare tale configurazione.

3.4.3 Realizzazione della struttura delle pagine *web*

Tramite l'utilizzo delle *Facelets* di *JSF* 2 e del supporto dell'*IDE* utilizzato è stato possibile creare la struttura delle pagine in maniera molto semplice, garantendo nello stesso tempo il massimo della flessibilità per quanto riguarda l'espansione del sistema. E' sufficiente infatti cliccare con il tasto destro del *mouse* sull'applicazione nel *workspace* di *Netbeans* e selezionare "*New -> Facelets Template*" per avere a disposizione la possibilità di scegliere fra vari *template* predefiniti per la creazione di una nuova pagina *.xhtml*.

Vista la complessità che potrebbe raggiungere l'applicazione *web* è stato scelto un modello che preveda la suddivisione della pagina in cinque parti, anche se tale scelta può essere facilmente modificabile in futuro. Il corpo del modello della pagina caratterizzante la nostra applicazione *web* è quindi definito dal seguente codice (*file main_template.xhtml*) :

```
.
.
<div id="left">
   <div id="logo">
      <img src="path logo azienda."/>
   </div>
   <ui:insert name="left">Left</ui:insert>
</div>
<div id="header">
   <ui:insert name="header"></ui:insert>
</div>
<div id="right">
   <ui:insert name="right">Right</ui:insert>
</div>
<div id="content" class="right_content">
   <p:ajaxStatus onstart="statusDialog.show();" onsuccess="statusDialog.hide();"/>
   <p:dialog modal="true" widgetVar="statusDialog" header="Caricamento dati" draggable="true"
             fixedCenter="true" width="170px">
      <h:outputText value="#{bundle.WaitingMessage}"/>
   </p:dialog>
   <ui:insert name="content">Content</ui:insert>
</div>
<div id="bottom">
   <ui:insert name="bottom">Maroil - Bardahl</ui:insert>
</div>
.
.
```

Come si può vedere la pagina è organizzata in cinque contenitori separati, ognuno con un proprio identificatore. All'interno di essi è presente un *tag <ui:insert>* che serve per specificare l'inserimento di una porzione di codice che può essere ridefinita nelle pagine che utilizzano tale *template* come sarà più chiaro in seguito.

In tale pagina modello sono stati inseriti il percorso dell'immagine del logo aziendale e la definizione di una finestra di dialogo che appare ogni volta si ha un'operazione di caricamento tramite tecnologia *AJAX* in maniera da fare in modo che tali oggetti siano presenti in ogni pagina che utilizza tale *template*.

Questo *file*, il cui scopo è quello di contenere solo il modello della struttura della pagina, viene parzialmente esteso dal contenuto del file "*main_template_with_menu-.xhtml*":

```
.
.
<ui:composition template="main_template.xhtml">
   <ui:define name="left">
       <h:form id="form">
          <h3>Menu</h3>
          <p:menu widgetVar="tieredMenu" context="'start','tl','br'" tiered="true"
                  style="width:180px">
             .
             .
             <p:submenu label="Ordini Clienti">
                <p:menuitem value="Interrogazione ordini"
                            action="#{oci31Controller.cleanFields}" global="false"/>
                <p:menuitem value="Inserimento ordine"
                            action="#{ocm00Controller.cleanFields}" global="false"/>
             </p:submenu>
             .
             .
          </p:menu>
       </h:form>
    </ui:define>
    <ui:define name="right">
       <p:panel header="Utente loggato">
          .
          .
       </p:panel>
    </ui:define>
</ui:composition>
.
.
```

Con il *tag <ui:composition template="main_template.xhtml">* si dichiara l'utilizzo del *file* indicato dall'attributo come modello, sovrascrivendone eventualmente una parte tramite il *tag <ui:define>*. Tale *tag* serve per ridefinire una porzione di pagina del modello: ad esempio in tale pagina si inserisce quanto contenuto all'interno

del *tag* <*ui:define name="left"*> al posto di ciò che era stato definito all'interno del *tag* <*ui:insert name="left"*>*Left*</*ui:insert*> del modello, quindi al posto della stringa *"Left"* che sarebbe il codice visualizzato nel caso non venga ridefinito tale componente di interfaccia.

Tale *file* quindi, come indica anche il suo nome, viene usato per ridefinire il contenuto delle parti esterne della pagina *web*, tipicamente utilizzati come menù. Infatti nel menù di sinistra vengono utilizzati il *tag* <*p:menu*> di *PrimeFaces* ed i suoi sottoelementi per definire il menù dell'applicazione, mentre in quello di destra viene inserito un pannello contenente i dati dell'utente loggato[2].

A questo punto per utilizzare questi due modelli all'interno delle pagine dell'applicazione è sufficiente utilizzare lo stesso meccanismo appena visto, quindi inserendo il componente *composition* (utilizzando come modello il *template* completo di menù, che a sua volta utilizza il modello principale) ed inserendo il contenuto della pagina all'interno del *tag* <*ui:define name="content"*>.

L'esempio di struttura così creata, che sarà utilizzata all'interno di ogni pagina dell'applicazione *web* è visibile in figura 3.5. Tutti i contenuti dell'applicazione, descritti nel seguito della trattazione, saranno quindi visibili nella parte centrale.

Figura 3.5: Struttura delle pagine

3.4.4 Modularizzazione dell'applicazione

Il modulo *EJB* serve per fare tutte le elaborazioni di *business logic* necessarie all'applicazione. Per evitare di inserire centinaia e successivamente migliaia di classi spesso ripetitive si è cercato di creare un meccanismo per uniformare le varie chiamate agli *EJB*, in modo da avere una semplificazione del codice dovuta soprattutto all'uniformità nel meccanismo di chiamata tra le varie operazioni differenti che possono essere effettuate.

[2] Analizzato più approfonditamente nella sezione relativa alla gestione degli accessi

Per spiegare in maniera dettagliata tutto il procedimento sarà preso in considerazione un esempio, senza porre al momento troppo peso sul significato delle varie funzioni, ma soffermandosi esclusivamente sulla struttura e sulla tipologia degli oggetti *Java* utilizzati.

Partiamo dal seguente metodo, che effettua un'interrogazione del *database* per recuperare gli ordini dei clienti:

```
1  public List retrieveData() throws Exception {
2     Oci3e1_request request = new Oci3e1_request();
3     request.setOci3e1ds(oci3e1ds);
4     request.setUser(SessionUtil.getUser());
5     Oci3e1_response response = ejbFacade.execute(request);
6     return response.getOci3e1sds();
7  }
```

Alla linea 2 si definisce un nuovo oggetto di tipo "*Oci3e1_request*": questo oggetto rappresenta la richiesta effettuata e quindi deve essere popolato da tutti i dati che si ritiene debbano essere utilizzati dall'*EJB* che effettua le operazioni di *business logic*. Tale oggetto è un *bean* che contiene al suo interno la dichiarazione di una variabile di tipo "*Oci3e1ds*" che a sua volta è un *bean* che servirà a contenere tutti i dati provenienti dal *form* di richiesta. La classe *Oci3e1_request* inoltre estende, come tutti gli altri oggetti utilizzati per le richieste, la classe "*CmdRequest*", che ha al suo interno un oggetto di tipo "*User*" che serve per la gestione delle informazioni sull'utente loggato.

Alle linee 3 e 4 vengono settate le proprietà del *bean* di richiesta associandogli prima un oggetto che è popolato dai dati provenienti dal *form* al momento della pressione del pulsante di invio dei dati, e poi l'oggetto di tipo *User* memorizzato in sessione in maniera da associare alla richiesta anche le informazioni sull'utente che effettua la richiesta stessa.

A questo punto viene effettuata la chiamata a un metodo dell'*EJB* passandogli come parametro l'oggetto richiesta (linea 5); tale metodo restituirà un oggetto di tipo "*Oci3e1_response*" che, in maniera simile all'oggetto di richiesta, contiene un oggetto che si riferisce ai dati da restituire all'utente (in questo caso un oggetto di tipo "*List<Oci3e1sds>*"), ed estende un oggetto di tipo "*CmdResponse*" che contiene tutti i messaggi associati all'operazione effettuata.

L'*EJB* su cui viene effettuata la chiamata del metodo è costituito da un'interfaccia remota che espone i metodi utilizzati ("*StatelessOci3e1Facade*") e da una classe che li implementa ("*Oci3e1Facade*").

All'interno del metodo "*execute()*" del *Facade* dell'*EJB* è presente il codice per la gestione della connessione al *database* (tramite l'opportuna classe "*DbManager*")

e tutte le operazioni di *business logic* necessarie per ottenere il risultato richiesto, cioè l'oggetto risposta opportunamente popolato.

3.4.5 Gestione dell'accesso al *database* remoto

Nella precedente sottosezione si è fatto riferimento all'utilizzo di una classe *DbManager* per la gestione degli accessi al *database*. Tale classe contiene i metodi necessari per la gestione delle connessioni con il *database* e soprattutto con lo schema desiderato, in base ad esempio all'utente che si connette al sistema (v. 3.4.6).

All'interno di tale classe è presente il metodo per recuperare un oggetto di tipo "*Connection*" utilizzabile per effettuare le *query* al *database*:

```
.
.
public static Connection getJdbcConnection(String dbname, String schema) throws Exception {
   if (dbname.trim().equals("")) {
      return getJdbcConnection();
   }
   Connection conn = getJdbcConnectionByName(dbname);
   return conn;
}
.
.
```

In base ai parametri passati al metodo, viene scelto quale metodo della stessa classe chiamare per la selezione della connessione con i dati indicati o con un eventuale *database* o schema *default.*

Sempre all'interno della classe sono presenti i metodi *standard* per recuperare le informazioni dal *database* ("*getResultSet()*") o per effettuare un aggiornamento dello stesso ("*executeUpdate()*"):

```
.
.
public static ResultSet getResultSet(Connection conn, String sqlstring) throws Exception {
   ResultSet rs;
   Statement stmt;
   stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_READ_ONLY);
   rs = stmt.executeQuery(sqlstring);
   return rs;

}


public static int executeUpdate(Connection conn, String sqlstring) throws Exception {
   int result;
   Statement stmt;
   stmt = conn.createStatement();
   result = stmt.executeUpdate(sqlstring);
```

```
        DBManager.closeStatement(stmt);
        return result;
    }
    .
    .
```

Questi metodi vengono utilizzati quindi a fronte di qualsiasi operazione effettuata sul *database* passando semplicemente come parametri la connessione desiderata e la stringa rappresentante la *query sql* opportunamente costruita.

3.4.6 Gestione degli accessi all'applicazione

Una delle funzionalità che deve avere un'applicazione *web* è la corretta gestione degli accessi al sistema. Oltre ad evitare che persone esterne all'azienda possano accedere a dati sensibili è importante che siano ben definiti i ruoli dei vari dipendenti in base alle diverse operazioni che possono effettuare.

In questa fase si è provveduto semplicemente ad impostare la logica necessaria per tale suddivisione dei ruoli in quanto, trattandosi semplicemente di un prototipo, gli accessi al momento sono comunque limitati all'amministratore del sistema.

3.4.6.1 *Realm* di *Glassfish*

Per gestire l'accesso al sistema ci si è basati sul supporto fornito dall'*application server* utilizzato, che prevede un preciso meccanismo per la gestione degli stessi chiamato "*Realm*".

Per poter utilizzare tale funzionalità è stato necessario innanzitutto provvedere alla creazione di due tabelle di un *database* che contengano i dati relativi agli utenti che possono accedere al sistema e ai loro ruoli.

Non essendo al momento necessario provvedere alla gestione degli accessi dei dipendenti ci si è limitati alla creazione di un *database* gestito in locale, semplicemente per impostare il meccanismo di protezione che sarà poi indispensabile in futuro.

Innanzitutto si è quindi provveduto alla creazione di un *database* all'interno della propria macchina contenente le tabelle degli utenti e dei gruppi. Per fare ciò ci si è avvalsi del *DBMS open source PostegreSql 8.4*. Dalla console di amministrazione del programma si è quindi creato il *database*, lo schema e le tabelle "*users*", contenente *username* e *password* degli utenti e "*groups*", contenente *username* e relativo identificatore del gruppo per definire i ruoli di ogni utente. Per i motivi sopra esposti è stato poi inizialmente inserito nella tabelle un utente con diritti di amministratore,

quindi sono state definite *username* e *password* ed è stato associato ad esso il ruolo "*ADM*".

Una volta provveduto alla creazione della struttura dati è necessario provvedere alla configurazione del *realm* di *Glassfish* per cui si rimanda nuovamente all'appendice B e alla sua registrazione all'interno dell'applicazione.

Per accedere al sistema si è inserita una pagina "*login.html*" all'interno della *root* principale dell'applicazione; all'interno del corpo della stessa è stato inserito il seguente codice:

```
.
.
<form method="POST" action="j_security_check">
   <div class="login_blocco">
       <img src="./resources/images/logo_login_gestionale.gif"/><br/>
       <div class="login">
          Username<br/><input name="j_username" type="text" style="width: 180px;"/>
       </div>
       <div class="login">
          Password<br/><input name="j_password" type="password" style="width: 180px;"/>
       </div>
       <div class="login accedi">
          <input name="bottone" type="submit" value="Log In"/>
       </div>
    </div>
</form>
<iframe src="/JSF2App/faces/gestionale/home.xhtml" frameborder="0" style="border:0px;
        position:absolute;left:-9999px;top:-9999px"/>
.
.
```

Ciò che contraddistingue questa pagina, gestita tramite il *realm* di *Glassfish*, da una pagina gestita senza tale supporto è la presenza dell'attributo del *form* "*action='j_security_check'*" e degli attributi dei campi di *input* "*j_username*" e "*j_password*" che devono avere esattamente questi nomi per essere riconosciuti dall'*application server*.

Per indicare nell'applicazione l'utilizzo di *realm* indicando anche le tipologie di accessi consentiti e le pagine protette va inserito il seguente codice nel *file web.xml*:

```
.
.
<security-constraint>
   <display-name>Protected</display-name>
   <web-resource-collection>
      <web-resource-name>Protected Resources</web-resource-name>
      <url-pattern>/gestionale/*</url-pattern>
      <url-pattern>/faces/gestionale/*</url-pattern>
      <http-method>GET</http-method>
```

```
        <http-method>POST</http-method>
        <http-method>HEAD</http-method>
        <http-method>PUT</http-method>
        <http-method>OPTIONS</http-method>
        <http-method>TRACE</http-method>
        <http-method>DELETE</http-method>
    </web-resource-collection>
    <auth-constraint>
        <description>ADM Users</description>
        <role-name>ADM</role-name>
    </auth-constraint>
</security-constraint>
<login-config>
    <auth-method>FORM</auth-method>
    <realm-name>postgresJdbcRealm</realm-name>
    <form-login-config>
        <form-login-page>/login.html</form-login-page>
        <form-error-page>/error.html</form-error-page>
    </form-login-config>
</login-config>
<security-role>
    <description>Amministratori della Intranet</description>
    <role-name>ADM</role-name>
</security-role>
.
.
```

Innanzitutto all'interno del *tag* <*security-constraint*> si indicano le risorse che si intendono proteggere e i ruoli degli utenti che vi possono accedere. Quindi all'interno del *tag* <*login-config*> si indicano il tipo di autenticazione[3] e le pagine contenenti il *form* di *login* ed il messaggio di errore. Infine vengono dichiarati i vari ruoli riconosciuti all'interno dell'applicazione.

In questo caso quindi solo gli utenti presenti nella tabella del *database* utilizzato con ruolo "*ADM*" possono accedere all'applicazione dopo aver fornito le corrette credenziali. Eventuali altri utenti senza tale ruolo non potrebbero accedere oltre la pagina di *login* illustrata in figura 3.6.

3.4.6.2 Struttura dati memorizzata in sessione

Per poter gestire le informazioni dell'utente loggato è stata creata una struttura dati da mantenere in sessione, e richiamare quando ritenuto opportuno.

Questa funzionalità è stata inserita dopo la prima fase di sviluppo, quando si è sentita la necessità di configurare varie tipologie di accessi:

[3]il valore "*FORM*" indica che la procedura di accesso viene gestita tramite un *form* personalizzato. Le altre opzioni sono "*BASIC*", per indicare l'utilizzo di un *popup* gestito dal *browser*, "*DIGEST*", che è analoga a *BASIC* ma utilizza l'algoritmo crittografico *MD5* e "*CLIENT_CERT*", che si basa sul certificato *SSL client-side*.

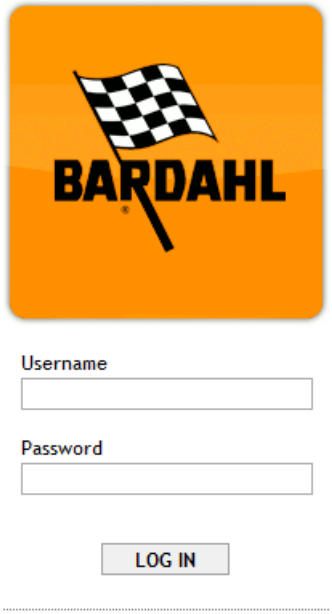


Figura 3.6: Pagina di *login*

- un accesso ai dati contenuti nel proprio *database* locale per le fasi preliminari dello sviluppo;
- un accesso ai dati contenuti nel *database* aziendale in uno schema di test ("*tst60dat*") che replica la struttura dello schema di produzione;
- un accesso ai dati contenuti nel *database* aziendale nello schema utilizzato in produzione ("*mar60dat*").

All'interno del *database* locale contenente le tabelle, sono infatti stati inseriti due ulteriori campi contenenti il nome del "*JNDI name*" relativo al *database* utilizzato dall'utente e il nome dello schema utilizzato. L'intento è quello di mantenere per ogni utente che abbia effettuato il *login* una struttura dati che contenga tutte le informazioni personali necessarie.

Nell'applicazione è stato infatti inserito un *EJB* che si occupa di gestire le informazioni di accesso ed in particolare il recupero delle informazioni relative agli utenti, con il settaggio di eventuali impostazioni di *default*.

Dopo aver effettuato l'accesso all'applicazione è possibile visualizzare sul menù di destra le informazioni relative all'utente loggato, come visibile nella figura 3.7.

Per fare ciò è stato inserito nel *template* che definisce la struttura delle pagine il seguente codice:

.

Figura 3.7: Menù utente loggato

```
.
<ui:define name="right">
   <p:panel header="Utente loggato">
      <h:panelGrid>
         <h:outputText value="Username" style="font-weight: bolder"/>
         <h:outputText value="#{userSession.user.username}"/>
         <br/>
         <h:outputText value="Database utilizzato" style="font-weight: bolder"/>
         <h:outputText value="#{userSession.user.dbname}"/>
         <br/>
         <h:outputText value="Schema utilizzato" style="font-weight: bolder"/>
         <h:outputText value="#{userSession.user.schema_name}"/>
         <br/>
         <h:form>
            <p:commandButton value="Logout" action="#{sessionUtil.logout}"/>
         </h:form>
      </h:panelGrid>
   </p:panel>
</ui:define>
.
.
```

Le espressioni inserite nei campi di *output* si riferiscono ad un *bean* che viene memorizzato in sessione e popolato con le informazioni che vengono recuperate dall'*EJB* che si occupa della gestione degli utenti. Nel *bean* "*UserSession*" è infatti presente un riferimento ad un oggetto *User* definito lato *EJB*, e popolato al momento del recupero delle informazioni.

L'accesso tramite *Expression Language* alla proprietà "*user*" scatena la chiamata del metodo "*getUser()*" così definito:

```
.
.
```

```
public User getUser() throws Exception {
   if (user == null) {
      User_request req = new User_request();
      req.setUserName(MyUtils.getLoggedName());
      user = userFacade.execute(req).getUser();
   }
   return user;
}
.
.
```

Tramite il metodo "*getUser()*" viene recuperato l'oggetto di tipo "*User*" e, se è vuoto come accade al momento dell'accesso dell'utente, viene usato l'*username* utilizzato per l'accesso per effettuare una ricerca, tramite l'opportuno metodo dell'*EJB*, delle informazioni relative a tale utente nella relativa tabella del *database* locale. Il metodo "*execute()*" infatti si occupa di creare un oggetto "*User*", popolandolo con il contenuto della tabella utenti relativa all'utente loggato, e di settare lo schema ed il *datasource* di *default* nel caso tali proprietà non siano indicate.

```
.
.
public User_response execute(User_request req) throws Exception {
   User_response res = new User_response();
   User user = UserDAO.getUsers_item(req.getUserName());
   user.setSchema_name((user.getSchema_name() == null) ? DBManager.schema_dft :
                                                         user.getSchema_name());
   user.setDbname((user.getDbname() == null) ? DBManager.dataSourceName_dft :
                                               user.getDbname());
   user.setSession(req.getSessionId());
   res.setUser(user);
   return res;
}
.
.
```

L'utilizzo di questo meccanismo può sembrare a prima vista piuttosto complesso rispetto alla semplice operazione da effettuare, ma permette una manutenibilità massima, in quanto nel caso in futuro vengano effettuate delle modifiche alla tabella contenente gli utenti, magari aggiungendo proprietà che inizialmente non sembravano rilevanti, è sufficiente aggiungere una proprietà del *bean* per avere a disposizione tale proprietà anche all'interno delle pagine *web* o in qualsiasi altro punto dell'applicazione.

3.4.7 *Bundle.properties*

Per consentire una maggior manutenibilità dell'applicazione è stato inserito un *file* contenente varie associazioni tra stringhe e valori di testo. Questo *file* è solitamente inserito per ragioni di internazionalizzazione.

Supponiamo infatti di voler cambiare agevolmente la lingua della nostra applicazione (ad esempio nelle etichette dei campi o dei pulsanti dei *form*). Senza tale *file* andrebbe cambiata a mano ogni etichetta presente nella pagina, e risulta facilmente comprensibile come questa operazione non sia effettuabile nella pratica viste le migliaia di etichette presenti nell'applicazione.

Al contrario tramite l'utilizzo dei *file* di *properties* si ha una dichiarazione univoca all'interno della pagina *web* relativa ad una particolare etichetta tramite *Expression Language*:

```
<p:commandButton value="#{bundle.SearchButton}"/>
```

In questo modo viene associato come etichetta del pulsante il valore della stringa "*SearchButton*" presente all'interno del *file* di *properties* riconosciuto dal *framework* in base al locale utilizzato[4]. E' sufficiente perciò definire un valore per la stringa *"SearchButton*" all'interno di ogni *file* di *properties* (ognuno relativo ad una lingua) in modo che tale etichetta venga visualizzata nella lingua corretta per l'utente dell'applicazione.

Oltre al problema dell'internazionalizzazione l'utilizzo di un *file* di *properties* per le stringhe di caratteri che caratterizzano le etichette di ogni componente offre il vantaggio di una maggior velocità di aggiornamento del sistema nel caso si vogliano effettuare piccoli cambiamenti. Nel caso in cui si associ ad ogni pulsante di ricerca il codice sopra illustrato, è sufficiente cambiare il contenuto della stringa "*SearchButton*" nel *file* di *properties* per cambiare ogni rifermento ad esso, ad esempio se si decidesse di inserire l'etichetta "Cerca" a tale pulsante al posto dell'etichetta "Ricerca".

3.4.8 Briciole di pane

Per facilitare la navigazione all'interno delle pagine dell'applicazione è stato inserito il meccanismo delle cosiddette "briciole di pane", ovvero quel componente che consente di tracciare il percorso effettuato dalla pagina iniziale dell'applicazione ed eventualmente ritornare ad una pagina da cui si è già passati.

[4]Quindi in base all'indirizzo IP della macchina che utilizza l'applicazione

PrimeFaces mette a disposizione un componente apposito inseribile nell'applicazione tramite il *tag* <*p:breadCrumb*>. Ad esempio il codice seguente produce quanto visibile in figura 3.8 o in figura 3.9 a seconda del valore di una variabile denominata "*formFunction*" contenuta nell"'*Ocm00Controller*":

```
.
.
<p:breadCrumb rendered="#{ocm00Controller.formFunction eq 'manage'}" preview="true"
              expandEffectDuration="200" previewWidth="30"  style="width: 99%">
   <p:menuitem url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}#{bundle.HomePagePath}"/>
   <p:menuitem value="#{bundle.Oci3e1dsUrl}"
               url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}
                    #{bundle.InterrogazioneOrdiniPath}#{bundle.Oci3e1dsPagePath}"/>
   <p:menuitem value="#{bundle.Oci3e1sdsUrl}"
               url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}
                    #{bundle.InterrogazioneOrdiniPath}#{bundle.Oci3e1sdsPagePath}"/>
   <p:menuitem value="#{bundle.CustomerOrderManageUrl}"
               url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}
                    #{bundle.GestioneOrdiniPath}#{bundle.CustomerOrderPath}"/>
</p:breadCrumb>
<p:breadCrumb rendered="#{ocm00Controller.formFunction eq 'insert'}" preview="true"
              expandEffectDuration="200" previewWidth="30"  style="width: 99%">
   <p:menuitem url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}#{bundle.HomePagePath}"/>
   <p:menuitem value="#{bundle.CustomerOrderInsertUrl}"
               url="#{bundle.FacesPath}#{bundle.GestionaleFolderPath}
                    #{bundle.GestioneOrdiniPath}#{bundle.CustomerOrderPath}"/>
</p:breadCrumb>
.
.
```

Figura 3.8: "Briciole di pane" in gestione di un ordine

Figura 3.9: "Briciole di pane" in inserimento di un ordine

I due componenti "*p:breadCrumb*" sono visualizzati alternativamente in base al valore di una variabile che indica la tipologia di utilizzo del *form* utilizzato. Il *form* da cui è stato preso lo spezzone di codice infatti è presente nella pagina "*CustomerOrder.xhtml*" a cui si può accedere sia al momento della modifica di un ordine esistente che a quello dell'inserimento di un nuovo ordine. Tale variabile viene settata al valore opportuno nel momento in cui l'utente esegue determinate azioni come ad esempio la selezione di un ordine.

Con un semplice *click* sull'etichetta che segnala il percorso effettuato si è immediatamente reindirizzati alla pagina desiderata.

Capitolo 4

Modulo di gestione degli ordini

Il modulo più importante sviluppato e presentato in questa tesi è sicuramente quello della gestione degli ordini. L'azienda ha infatti come necessità primaria la possibilità della gestione degli ordini dei clienti sul *web* quindi è stata data priorità allo sviluppo di tale modulo.

La gestione degli ordini si divide in due parti principali: l'interrogazione degli ordini e la manutenzione degli stessi.

4.1 Interrogazione degli ordini

Dal menù principale dell'applicazione è possibile accedere alla schermata di interrogazione degli ordini cliccando su "Ordini Clienti" -> "Interrogazione Ordini". Si accede così ad una pagina *web* contenente un *form* con numerosi campi per raffinare la ricerca di un ordine o di un gruppo di ordini in base a determinati criteri come si può vedere in figura 4.1 e in figura 4.2.

Nello scegliere il *layout* di tali schermate si è cercato di distaccarsi il meno possibile dalla visualizzazione del sistema originale, illustrata in figura 4.3 e in figura 4.4 cercando solamente di apportare dei miglioramenti ove necessario.

Lo sviluppo di tale applicazione deve avere infatti lo scopo di semplificare la gestione dell'azienda, non certo di complicarla, quindi si è cercato di mantenere, dove possibile, l'impostazione del *form* originale in maniera da annullare il tempo di apprendimento del nuovo sistema da parte dei dipendenti dell'azienda.

Per ottenere la grafica illustrata non è stato necessario effettuare particolari modifiche personalizzate del *layout* dei componenti, in quanto la maggior parte dei

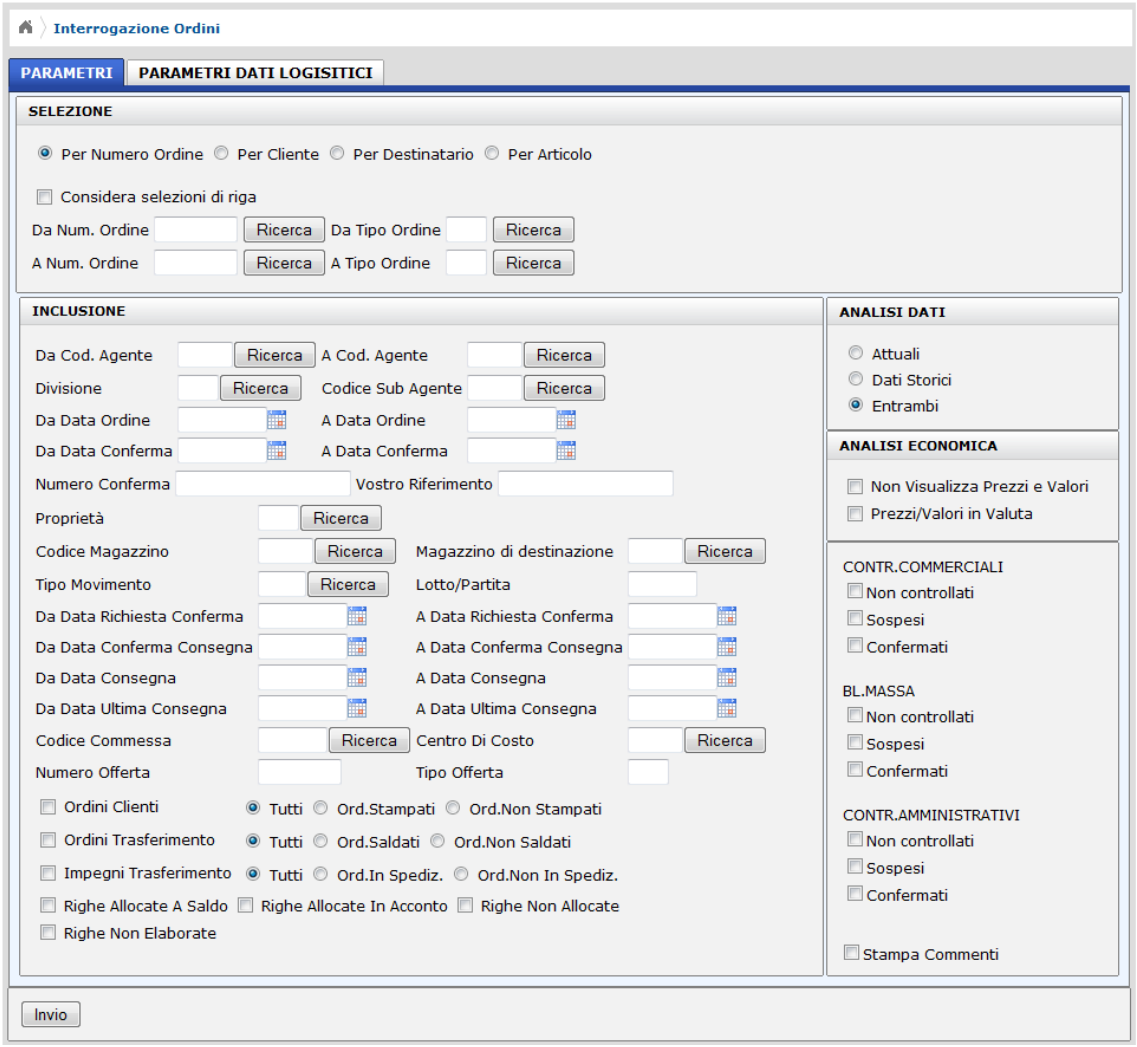


Figura 4.1: Schermata Interrogazione Ordini (1)

componenti utilizzati sono stati presi dalla libreria di *PrimeFaces* che garantisce di default un *layout* molto *user-friendly*.

Prima di analizzare le funzionalità di interrogazione e decodifica dei dati verranno adesso analizzati i componenti principali del *form*, illustrando le funzionalità che offrono.

4.1.1 Suddivisione grafica della schermata di interrogazione

Per modularizzare l'applicazione si è ritenuto opportuno suddividerla in parti distinte per riuscire ad ottenere *file* più compatti e di conseguenza più manutenibili.

Le due schermate illustrate in figura 4.1 e in figura 4.2 sono in realtà all'interno della stessa pagina *XHTML* e più precisamente all'interno dello stesso *form* di

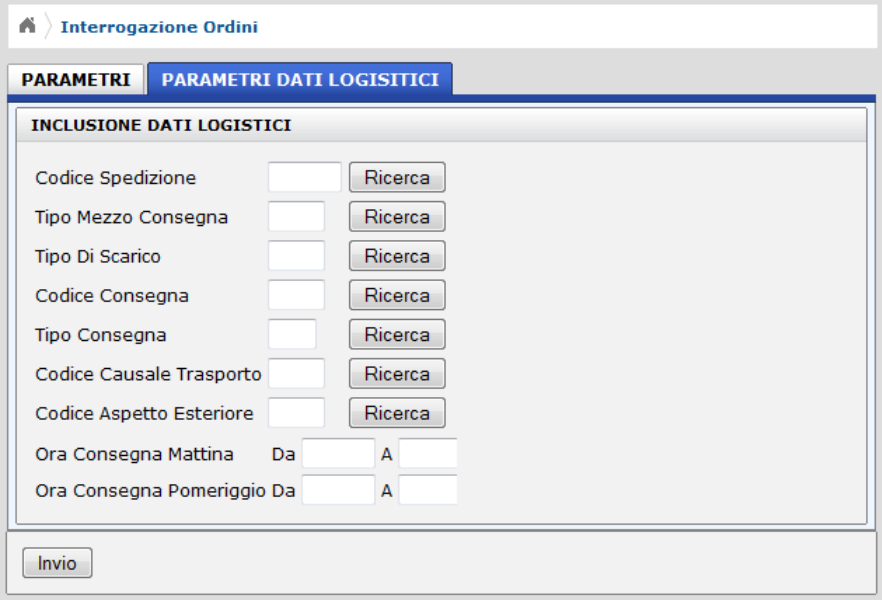


Figura 4.2: Schermata Interrogazione Ordini (2)

immissione dati come risulta più chiaro dal codice seguente che rappresenta proprio tale pagina:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:p="http://primefaces.prime.com.tr/ui">
   <ui:composition template="/gestionale/template/main_with_menu.xhtml">
      <ui:define name="content">
         .
         .
         <h:form id="oci3e1dsForm" styleClass="jsfcrud_list_form">
            <p:tabView>
               <p:tab title="PARAMETRI">
                  <ui:include src="panel/selectionPanel.xhtml"/>
                  <h:panelGrid columns="2" width="100%">
                     <ui:include src="panel/includePanel.xhtml"/>
                     <h:panelGroup>
                        <ui:include src="panel/dataAnalysisPanel.xhtml"/>
                        <ui:include src="panel/economicAnalysisPanel.xhtml"/>
                        <ui:include src="panel/externalPanel.xhtml"/>
                     </h:panelGroup>
                  </h:panelGrid>
               </p:tab>
               <p:tab title="PARAMETRI DATI LOGISTICI">
                  <ui:include src="panel/logisticDataPanel.xhtml"/>
               </p:tab>
            </p:tabView>
            <p:panel>
               <p:commandButton value="Invio" action="#{oci31Controller.prepareResult}"/>
            </p:panel>
```

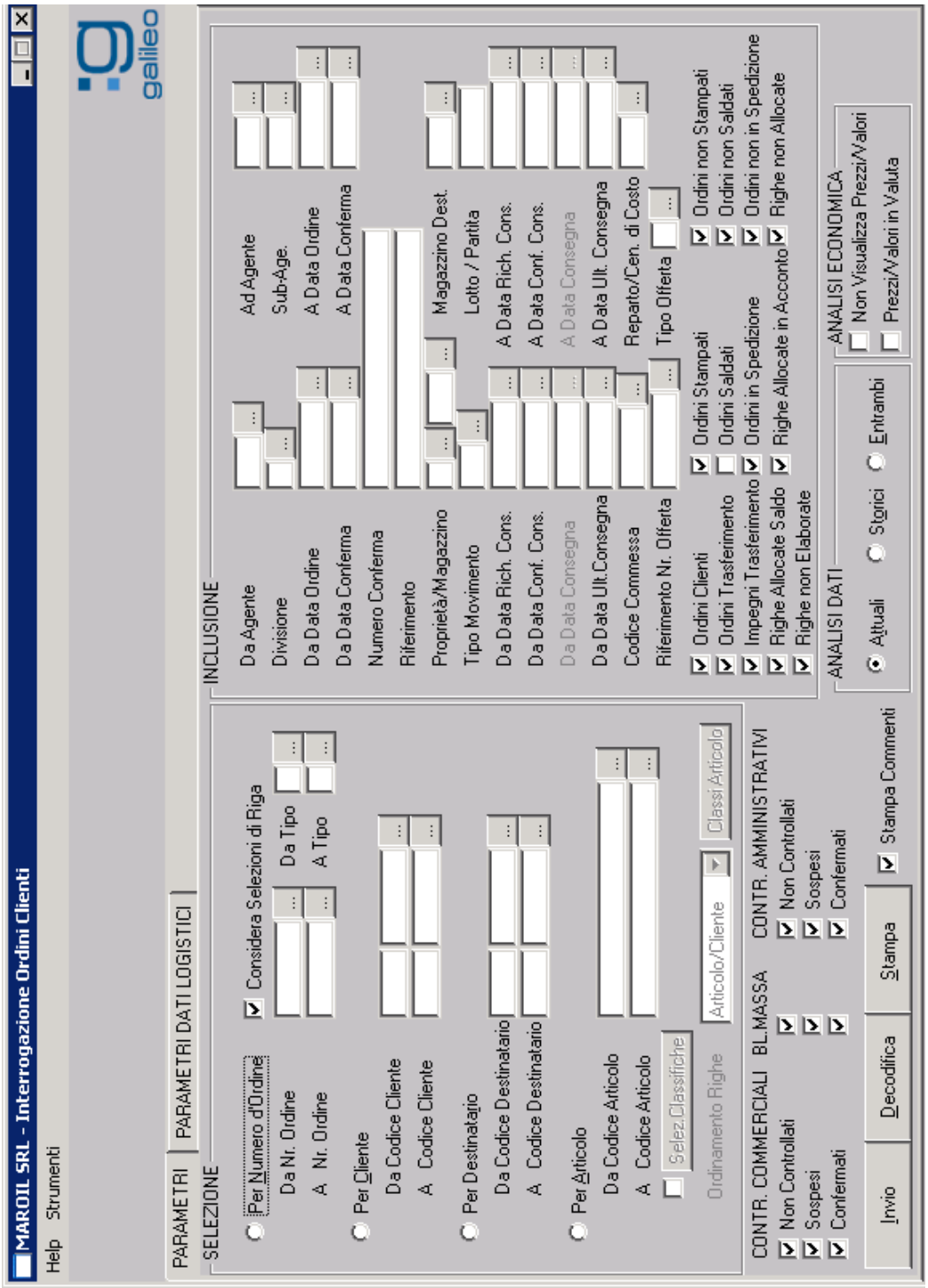


Figura 4.3: Schermata Interrogazione Ordini in sistema originale (1)

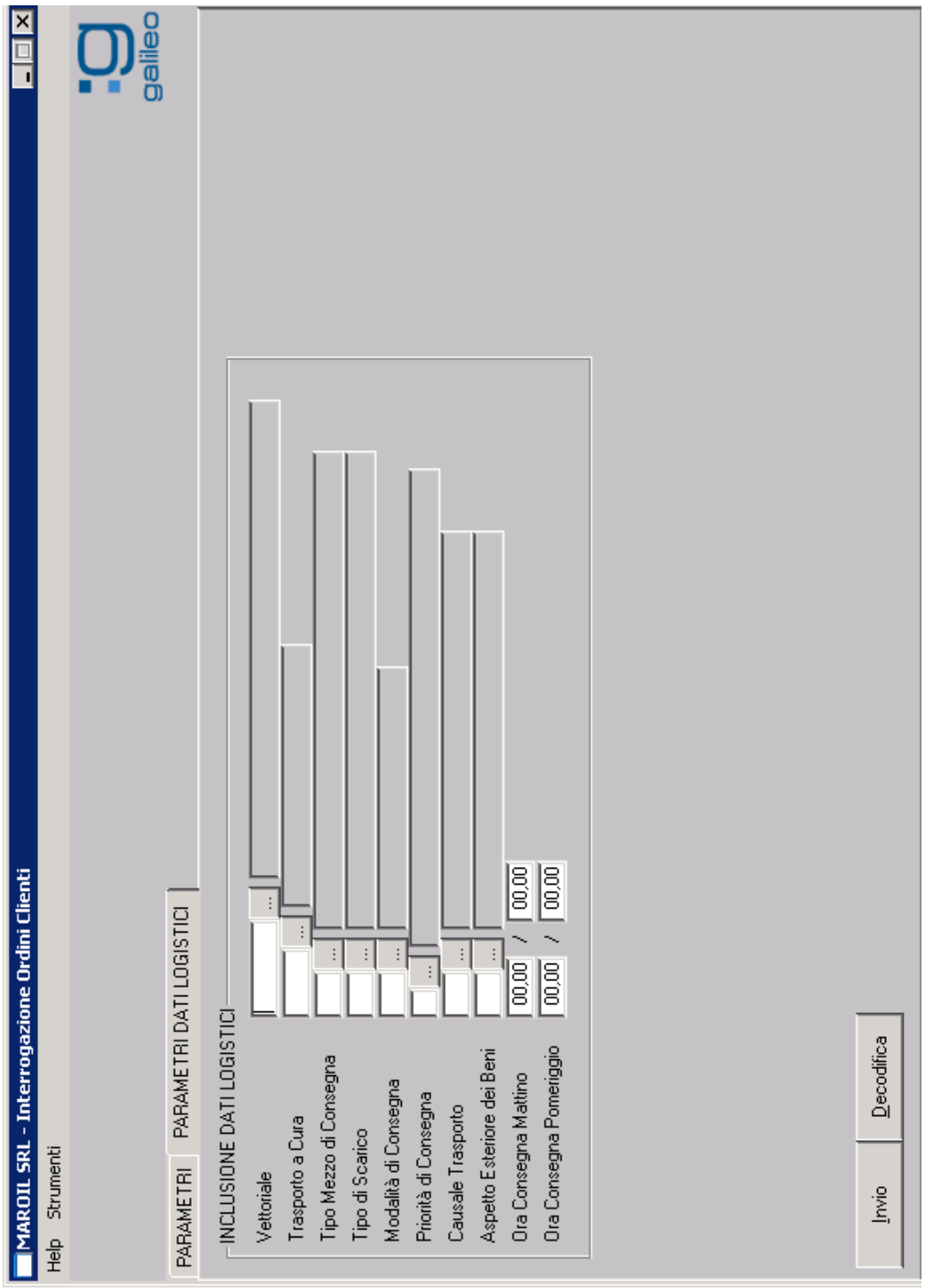


Figura 4.4: Schermata Interrogazione Ordini in sistema originale (2)

```
        </h:form>
      </ui:define>
    </ui:composition>
</html>
```

Come si può notare il *tag JSF* "*<h:form>*" racchiude al suo interno il *tag* di *PrimeFaces* "*<p:tabView>*" che definisce appunto la struttura di una *tab* (la classica struttura "a linguette") con le due linguette "PARAMETRI" e "PARAMETRI DATI LOGISTICI". Per permettere una più pulita gestione, la pagina è stata suddivisa in più *file* che vengono inclusi all'interno della pagina principale tramite il *tag* "*<ui:include>*". Per facilitare anche all'utente la gestione dei vari campi del *form* tale suddivisione è stata mantenuta anche a livello visivo, in quanto ogni *file* è stato associato ad un pannello (tramite il *tag* "*<p:panel>*").

Per la formattazione dei *form* si è fatto un largo uso dei componenti di *JSF* "*<h:panelGrid>*" e "*<h:panelGroup>*".

Un *panelGrid* è praticamente una griglia su cui possono esser posizionati vari oggetti. Il funzionamento è concettualmente molto semplice: una volta definite il numero di colonne, gli oggetti da inserire all'interno di tale griglia vengono posizionati, a partire dalla prima riga, su ogni riga fino all'esaurimento delle colonne. In pratica in una *panelGrid* con 2 colonne e 4 elementi, tali elementi devono essere inseriti in sequenza al suo interno; il *framework* provvederà a posizionare i primi due elementi sulla prima riga, dopodiché, essendo terminate le colonne, provvederà a posizionare i restanti due elementi sulla seconda riga.

Un *panelGroup* è utile proprio all'interno di un *panelGrid* quando si vuole che un gruppo di elementi vengano considerati come un unico elemento all'interno della griglia.

Nel *file* sopra descritto si sono utilizzati questi due componenti per posizionare i pannelli, infatti dopo il primo pannello si è inserita una *panelGrid* con due colonne inserendo nella prima colonna il pannello più grande, e nella seconda colonna gli altri tre pannelli, includendoli appunto in un *panelGroup* per inserirli tutti nella stessa colonna.

4.1.1.1 Pannello di selezione

Una prima modifica che si può notare a prima vista rispetto al sistema originale è rappresentata dal cambiamento del pannello di selezione, che consente di raffinare la ricerca per tipologia di codice. Per aumentare la leggibilità delle scher-

mate del sistema, piuttosto bassa nella versione originale, si è scelto di limitare la visualizzazione dei campi strettamente indispensabili.

Essendo in presenza in questo caso di una scelta tra possibili valori di ricerca, si è ritenuto opportuno inserire un componente che preveda la selezione preliminare di una tipologia di ricerca e, tramite la tecnologia *AJAX* che permette l'aggiornamento di un componente senza ricaricare necessariamente tutta la pagina, la visualizzazione soltanto dei campi relativi a tale selezione:

```
.
.
<p:panel header="SELEZIONE">
   <h:selectOneRadio value="#{oci31Controller.oci3e1ds.ordin}" onchange="submit()">
      <f:selectItem itemLabel="Per Numero Ordine" itemValue="O"/>
      <f:selectItem itemLabel="Per Cliente" itemValue="C"/>
      <f:selectItem itemLabel="Per Destinatario" itemValue="D"/>
      <f:selectItem itemLabel="Per Articolo" itemValue="A"/>
   </h:selectOneRadio>
   <h:panelGroup rendered="#{oci31Controller.oci3e1ds.ordin eq 'O'}">
      <h:panelGrid columns="2">
        <cu:nOci3e1checkbox id="fselr" value="#{oci31Controller.oci3e1ds.fselr}"/>
         <h:outputText value="Considera selezioni di riga"/>
      </h:panelGrid>
      <h:panelGrid columns="6">
        <h:outputText value="Da Num. Ordine"/>
        <h:inputText value="#{oci31Controller.oci3e1ds.nrord}" size="7" maxlength="7"/>
        .
        .
      </h:panelGrid>
   </h:panelGroup>
   .
   .
</p:panel>
.
.
```

Nella pagina è stato inserito un componente "*<h:selectOneRadio/>*", il classico *radio button*, che in base alla scelta effettuata setta una variabile "*ordin*".

All'interno del pannello sono presenti 4 componenti *<h:panelGroup/>*[1], che vengono visualizzati in base al valore della variabile settata. In questo modo l'utente può visualizzare solo i campi che effettivamente gli interessano.

4.1.2 Componenti personalizzati

Poiché alcune piccole funzionalità che si sono rese necessarie nello sviluppo del sistema non erano disponibili né tra i componenti forniti con l'implementazione

[1]Nel codice illustrato è stato inserito soltanto il primo, gli altri sono stati omessi per questione di leggibilità in quanto analoghi ad esso.

standard di *JSF* né con quella di *PrimeFaces* è stato necessario ridefinire alcuni componenti.

4.1.2.1 *<cu:sOci3e1checkbox>*

Nella sottosezione relativa al pannello di selezione dell'interrogazione degli ordini è stato inserito un componente personalizzato, relativo ad una *checkbox*.

Per *default* una *checkbox* riconosce soltanto alcuni valori booleani[2]. Per l'interfacciamento con il *database* attuale è stato necessario avere a disposizione una *checkbox* che riconoscesse anche altri valori, quali ad esempio "S", "N", "X", ecc...

Poiché *JavaServer Faces* non mette a disposizione la possibilità di modificare i valori di *default* si è provveduto a definire un componente personalizzato che estendesse quello predefinito.

Per fare ciò è stato necessario innanzitutto definire il componente stesso tramite il seguente codice *Java*:

```
@FacesComponent(value = "sOci3e1checkbox")

public class UISOci3e1CheckBox extends UIComponentBase {
   @Override

   public String getFamily() { return "Oci3e1_CHECKBOX_FAMILY"; }
   @Override
   public void encodeEnd(FacesContext context) throws IOException {
      ResponseWriter responseWriter = context.getResponseWriter();
      responseWriter.startElement("input", null);
      responseWriter.writeAttribute("type", "checkbox", "checkbox");
      responseWriter.writeAttribute("id", getClientId(context), "id");
      responseWriter.writeAttribute("name", getClientId(context), "clientId");
      Object currentValue = getCurrentComponent(context).getAttributes().get("value");
      if (formatValue(currentValue)) {
         responseWriter.writeAttribute("checked", "checked", "checked");
      }
      responseWriter.endElement("input");

   }
   @Override
   public void decode(FacesContext context) {
      try {
         ExternalContext externalContext = context.getExternalContext();
         Map requestMap = context.getExternalContext().getRequestParameterMap();
         String clientId = getClientId(context);
         String string_submit_val = (String) requestMap.get(clientId);
         HttpSession session = (HttpSession) externalContext.getSession(true);
         Oci31Controller oci31controller =
                        (Oci31Controller) session.getAttribute("oci31Controller");
         Oci3e1ds oci3e1ds = oci31controller.getOci3e1ds();
         String attrName = clientId.split(":")[1];
```

[2] "*Y*", "*true*", "*yes*" e "*N*", "*false*", "*no*"

```
            String methodName = "set" + attrName.substring(0, 1).toUpperCase() +
                                attrName.substring(1).toLowerCase();
            Class[] argTypes = new Class[]{String[].class};
            argTypes[0] = String.class;
            Method myMethod = oci3e1ds.getClass().getDeclaredMethod(methodName, argTypes);
            if (string_submit_val == null) { myMethod.invoke(oci3e1ds, " "); }
            if ("on".equals(string_submit_val)) { myMethod.invoke(oci3e1ds, "S"); }
         } catch (Exception ex) {
            Logger.getLogger(UISOci3e1CheckBox.class.getName()).log(Level.SEVERE, null, ex);
         }

      }
      private boolean formatValue(Object currentValue) {
         String curVal = (String) currentValue;
         if (curVal.equals("S")) {
            return true;
         } else {
            return false;
         }
      }
   }
```

Nel codice sopra si può notare innanzitutto la notazione "*@FacesComponent(value = "sOci3e1checkbox")*" che indica il nome del componente per il *framework*. La classe estende la "*UIComponentBase*" che contiene i componenti base del linguaggio e sovrascrive i metodi di codifica e decodifica che vanno ad effettuare la conversione del valore proveniente o diretto al *model* per presentarlo nella *view*. In particolare questo componente viene ridefinito per riconoscere nella *form* in cui è utilizzato il valore "S" ad indicare il settaggio della *checkbox* ed il valore " " ad indicare il contrario.

Per poter utilizzare il componente all'interno di qualche pagina *web* è necessario dichiararlo, e quindi definire una nuova *taglib* con un *file .xml* che comprenda la definizione dei vari nuovi *tag* definiti. Essendo necessaria la creazione di vari *tag* di *checkbox* personalizzate si è chiamato il *file* semplicemente "*checkbox-taglib.xml*". Al suo interno sono stati definiti tutti i *tag* di *checkbox* personalizzati che si è deciso di utilizzare:

```
facelet-taglib xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                                   http://java.sun.com/xml/ns/javaee/
                                          web-facelettaglibrary_2_0.xsd"
               version="2.0">
   <namespace>http://javaserverfaces.dev.java.net/demo/custom-taglib</namespace>
   <tag>
      <tag-name>xCstabcheckbox</tag-name>
      <component>
         <component-type>xCstabcheckbox</component-type>
```

```
      </component>
   </tag>
   <tag>
      <tag-name>s0ci3e1checkbox</tag-name>
      <component>
          <component-type>s0ci3e1checkbox</component-type>
      </component>
   </tag>
   <tag>
      <tag-name>n0ci3e1checkbox</tag-name>
      <component>
         <component-type>n0ci3e1checkbox</component-type>
      </component>
   </tag>
</facelet-taglib>
```

Soffermandoci solo sul componente precedentemente descritto si può notare come in tale *file* venga dichiarato il nome del *tag*, che sarà utilizzato nelle pagine *web*, ed il tipo del componente, che è il nome con cui lo stesso è stato dichiarato nella classe dove è stato definito.

Nel *file* è indicata anche la dichiarazione del nome del *namespace* che deve essere inserito come attributo del *tag HTML* nelle pagine in cui si intende utilizzare il *tag* personalizzato.

E' inoltre necessario inserire nel descrittore di *deployment* dell'applicazione le seguenti linee di codice per indicare dove si trova la *taglib* personalizzata:

```
.
.
<context-param>
   <param-name>javax.faces.FACELETS_LIBRARIES</param-name>
   <param-value>/WEB-INF/checkbox-taglib.xml</param-value>
</context-param>
.
.
```

4.1.3 Selezione delle date

Un componente fondamentale in tutte le occasioni in cui si devono in qualche modo gestire le date è il calendario. Tale componente fornisce un campo di testo dove deve essere inserita una data affiancato da un pulsante per la selezione della data stessa da un calendario, come visibile in figura 4.5.

In questo modo è possibile selezionare agevolmente qualunque data si desideri senza perder tempo nella scrittura a mano di codice *Javascript*, in quanto il componente stesso fornisce tutte le possibili opzioni di utilizzo di cui potremmo aver bisogno.

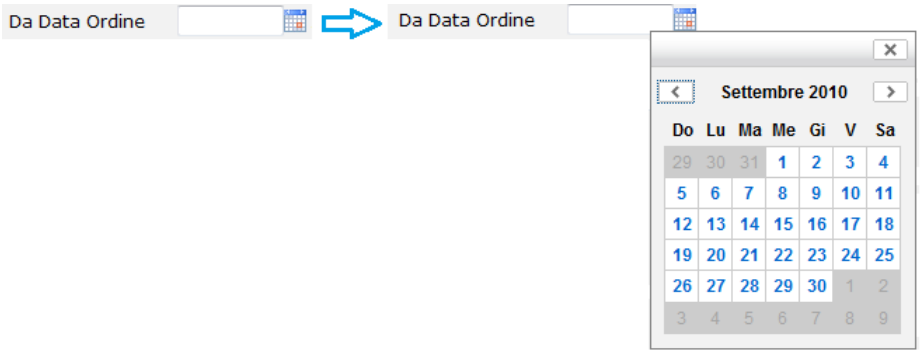


Figura 4.5: Componente p:calendar

L'unico codice da inserire nella pagina per ottenere quanto descritto in figura è il seguente:

```
<h:outputText value="Da Data Ordine"/>
<p:calendar value="#{Filter.dtOrdDate}" pattern="dd/MM/yyyy" inputStyleClass="inputCalendar"/>
```

4.1.4 Gestione delle ricerche

Una funzionalità fondamentale utilizzata sia nell'interrogazione che nella gestione degli ordini, ma che si prevede sarà utilizzata molto spesso anche nel resto dell'applicazione e quella della ricerca di un dato all'interno di una tabella del *database*.

Proprio perché utilizzata pesantemente si è ritenuto opportuno studiare una soluzione parametrica che permetta di gestire le ricerche in maniera immediata e funzionale senza dover scrivere migliaia di linee di codice inutili.

4.1.4.1 Parte di presentazione

Per spiegare dettagliatamente questo meccanismo è necessario partire dalla pagina *web* in cui si vuole inserire un campo derivante da una ricerca in una tabella del database. Prendiamo come esempio la ricerca del codice di un agente:

```
.
.
<h:inputText id="cdagd" value="#{oci31Controller.oci3e1ds.cdagd}" size="3" maxlength="3"/>
<p:commandButton value="Ricerca" type="button" immediate="true"
                 action="#{searchBean.search('/gestionale/xtabag/List.xhtml','oci3e1dsForm',
                                             'cdagd','xtabagList','xtabagValue')}"
                 oncomplete="window.open('#{myUtils.getContextPath()}
                                         #{myUtils.getSearchPagePath()}', '', 'width=800,
                                         height=600,left=0,top=0,resizable=no,menubar=yes,
                                         toolbar=yes, scrollbars=no,locations=no,
                                         status=no');"/>
```

.
.

Nella pagina è presente il campo da aggiornare con il valore proveniente dalla tabella con un attributo "*id*" ed il pulsante con cui si richiama la ricerca. Alla pressione di tale pulsante viene innanzitutto chiamato il metodo "*search()*" del *managedBean* di nome "*searchBean*", cui vengono passati:

- il *path* della pagina da richiamare per effettuare la ricerca (che include al suo interno la tabella contente il *mapping* dei dati contenuti nella tabella del *database*);
- l'identificatore del *form* in cui è presente il campo da aggiornare;
- l'identificatore del campo da aggiornare;
- l'identificatore del *form* contenuto nella pagina indicata nel primo argomento,
- l'identificatore del componente in cui viene inserito il valore desiderato al momento della selezione dalla tabella visualizzata[3].

Il metodo del *bean* in questione non fa altro che memorizzare i 5 parametri passati come argomento all'interno delle 5 proprietà del *bean* che ha visibilità di sessione e quindi sarà accessibile anche dalla pagina aperta dal pulsante.

Il pulsante infatti non fa altro che aprire sempre la solita pagina utilizzata come modello (*search_template.xhtml*) in una nuova finestra del *browser* tramite il codice *Javascript* inserito nell'attributo "*oncomplete*":

```
.
.
<h:head>
   .
   .
   <script type="text/javascript">
      function updateSearch(){
         window.opener.document.forms['#{searchBean.parentForm}']
                                    ['#{searchBean.parentForm}:
                                      #{searchBean.parentField}'].value=
                                    document.forms['#{searchBean.childForm}']
                                                  ['#{searchBean.childForm}:
                                                    #{searchBean.childField}'].value;
         document.forms['#{searchBean.childForm}'].submit();
         window.opener.document.forms['#{searchBean.parentForm}'].submit();
         window.close();
```

[3]Questo passaggio maggiormente chiaro più avanti.

```
      }
   </script>
</h:head>
<h:body>
   .
   .
   <ui:include src="#{searchBean.searchPage}"/>
</h:body>
.
.
```

All'interno della sezione "*<h:head>*" di tale *file* è contenuta la dichiarazione della funzione *Javascript* "*updateSearch()*" utilizzata per memorizzare il campo selezionato dalla tabella che rappresenta i risultati della ricerca nel campo della pagina chiamante associato al pulsante di ricerca che è stato premuto.

La funzione *Javascript* è stata inserita in questa pagina in quanto, come è visibile all'interno di "*<h:body>*", tale pagina include semplicemente quella indicata nella proprietà del *searchBean* che quindi avrà a disposizione la funzione *Javascript* da chiamare.

Come accennato precedentemente la pagina "*List.xhtml*" selezionata include a sua volta un'altra pagina "*Datatable.xhtml*" che contiene la tabella con il recupero dei dati; questo duplice passaggio è stato mantenuto per permettere in futuro di riutilizzare la pagina stessa nel caso in cui si voglia preliminarmente effettuare un filtraggio della ricerca stessa. Analizziamo quindi direttamente la pagina contenente la tabella:

```
.
.
<h:inputHidden id="xtabagValue" value="#{xtabagController.xtabag.xcodtb}"/>
<p:dataTable value="#{xtabagController.items}" var="item" selectionMode="single"
             selection="#{xtabagController.xtabag}" dblClickSelect="true"
             update="xtabagValue" onselectComplete="updateSearch()"
             emptyMessage="#{bundle.NoResultFound}" paginator="true" rows="10"
             scrollable="true" width="100%" styleClass="datatable">
   <p:column resizable="true" sortBy="#{item.xcodtb}">
      <f:facet name="header">
         <h:outputText value="Cod."/>
      </f:facet>
      <h:outputText value="#{item.xcodtb}"/>
   </p:column>
   <p:column resizable="true" sortBy="#{item.tdsage}">
      <f:facet name="header" >
         <h:outputText value="Descrizione"/>
      </f:facet>
      <h:outputText value="#{item.tdsage}"/>
   </p:column>
   .
```

```
    .
</p:datatable>
.
.
```

Per poter recuperare i valori della tabella e selezionarne uno è necessario utilizzare i vari attributi del componente "*<p:datatable>*"[4]:

- *value = "#{xtabagController.items}"* - indica le righe con cui deve essere popolata la tabella della pagina, in questo caso quindi all'interno del metodo "*getItems()*" del *controller* "*xtabagController*" viene effettuata la chiamata all'*EJB* che si occuperà del recupero dei dati;

- *var = "item"* - variabile utilizzata all'interno del *datatable* su cui vengono chiamati i vari campi dell'oggetto recuperato[5] ;

- *selectionMode = "single"* - indica la modalità di selezione delle righe, in questo caso singola perché interessa recuperare un dato univoco contenuto all'interno di una sola riga[6];

- *selection = "#{xtabagController.xtabag}"* - indica l'oggetto che sarà popolato al momento della selezione di una riga; in questo caso all'interno del *controller* utilizzato è presente un oggetto di tipo "*xtabag*" che rappresenta il *mapping* dei campi presenti nella tabella, che vengono quindi popolati con i valori della riga selezionata;

- *update = "xtabagValue"* - indica l'identificatore del componente da aggiornare alla selezione della riga

- *dblClickSelect = "true"* - per indicare che la selezione viene effettuata al doppio *click* su una riga;

- *onselectComplete = "updateSearch()"* - dopo aver effettuato il doppio *click* sulla riga scelta viene chiamata la funzione *Javascript* dichiarata nell'intestazione della pagina modello, che si occupa di copiare il contenuto del campo nascosto nella pagina, che è stato appena aggiornato con il valore desiderato della riga

[4]Sono descritti soltanto gli attributi concernenti il funzionamento, tralasciando quelli riguardanti il *layout*

[5]Non si ritiene al momento opportuno analizzare nel dettaglio il componente per cui si rimanda alla guida di *PrimeFaces* [14] reperibile sul sito ufficiale o a qualsiasi guida *JSF* in quanto il componente "*datatable*" delle due librerie è molto simile nel funzionamento di base.

[6]Trattandosi tra l'altro del campo chiave della tabella

selezionata, nel campo di testo della pagina chiamante, e di chiudere la finestra con la tabella di ricerca.

Il campo di testo nascosto "*<h:hiddenInput/>*" è stato necessario per permettere il funzionamento corretto della funzione *Javascript*, in quanto si rende necessario aggiornare prima tale campo con il valore desiderato della riga selezionata (invece di passare alla pagina chiamante tutti i valori della riga) e poi passare tale campo alla pagina chiamante facendo il "*submit*" di entrambe le pagine.

4.1.4.2 Parte di *business logic*

Visto il notevole utilizzo delle ricerche semplici in tabella all'interno dell'applicazione è opportuno soffermarsi anche ad analizzare come sono state implementate tali ricerche anche dal punto di vista della logica di *business*.

Per ottimizzare ulteriormente il sistema, anche le chiamate all'*EJB* che si occupa delle ricerche è stato parzialmente parametrizzato in maniera da evitare la proliferazione inutile di classi *Java* che devono effettuare operazioni molto simili da un caso all'altro.

Come visto precedentemente il recupero dei dati viene effettuato nella pagina contenente la tabella che deve visualizzarli tramite la chiamata del metodo *"getItems()*" del *controller* opportuno. Avendo trattato nel paragrafo precedente la selezione del codice degli agenti analizziamo adesso proprio l'"'*XtabagController*":

```
@ManagedBean(name = "xtabagController")
@SessionScoped
public class XtabagController implements Serializable {
   @EJB
   private StatelessGenericDAOFacade genericDAOFacade;
   private Xtabag xtabag;
   private DataModel items = null;
   private PaginationHelper pagination;

   public DataModel getItems() {
      if (items == null) {
         items = getPagination().createPageDataModel();
      }
      return items;
   }
   public PaginationHelper getPagination() {
      if (pagination == null) {
         pagination = new PaginationHelper(500) {
            .
            .
            @Override
            public DataModel createPageDataModel() {
```

```
                DataModel result = null;
                try {
                   GenericListDAO_request request = new GenericListDAO_request();
                   request.setUser(SessionUtil.getUser());
                   request.setTable(new Xtabag());
                   GenericListDAO_response response =
                                    genericDAOFacade.getGenericDAO_List(request);
                   result = new ListDataModel((List<Xtabag>) response.getList());
                } catch (Exception ex) { . . . }
                return result;
             }
          };
       }
       return pagination;
    }
    .
    .
}
```

Il metodo "*getItems()*", essendo "*items*" inizialmente con valore "*null*" chiama il metodo "*createPageDataModel()*" per creare l'oggetto "*DataModel*" utilizzato all'interno del *datatable*. All'interno di tale metodo viene usato un oggetto "*GenericListDAO_request*" che rappresenta la richiesta generica relativa alle ricerche semplici in tabella. Tale richiesta estende la *CmdRequest* contenente l'oggetto *User* e un *Object* "*table*" che è destinato a contenere il *bean* su cui effettuare il *mapping* della tabella da interrogare.

Dopo averne settato gli opportuni campi, la richiesta viene passata come argomento nella chiamata del metodo "*getGenericDAO_List()*" dell'*EJB* "*GenericDAOFacade*", effettuata ovviamente tramite la sua interfaccia remota. Tale chiamata restituirà un oggetto di tipo "*GenericListDAO_response*" che conterrà un oggetto di tipo "*List*" popolato opportunamente con i valori in tabella che, dopo le opportune operazioni di conversione saranno memorizzati nell'oggetto di tipo "*DataModel*" che sarà utilizzato nella tabella da visualizzare.

A questo punto andiamo ad analizzare cosa succede all'interno dell'*EJB* analizzando soltanto i passaggi principali:

```
@Stateless
public class GenericDAOFacade implements StatelessGenericDAOFacade {

   @Override
   public GenericListDAO_response getGenericDAO_List(GenericListDAO_request req)
                                                                  throws Exception {
      GenericListDAO_response res = new GenericListDAO_response();
      DBManagerConn conn = new DBManagerConn(...);
      .
      .
```

```
            res.setList(GenericDAO.getList(req.getTable(), req.getFilter(), req.getOrderby(), conn));
            DBManager.closeConnection(conn);
            return res;
        }
    }
```

Dopo aver creato l'oggetto che sarà restituito come risposta ed aver creato la connessione al *database*, viene chiamato il metodo "getList()" del "*GenericDAO*" passandogli come parametri la connessione appena creata contenente i dati dell'utente, e le proprietà della richiesta relativa alla ricerca da effettuare, quindi l'oggetto che rappresenta la tabella, eventuali filtri o condizioni di ordinamento non settati nel caso in questione.

Tale metodo, senza scendere troppo nel dettaglio del codice in quanto siamo in presenza a questo punto di una normalissima interrogazione del *database*, utilizza i dati in richiesta per creare la *query* di interrogazione del *database* ed effettuarla, recuperando i dati richiesti, a questo punto individuati in maniera univoca.

I dati recuperati dal metodo in questione vengono settati nella risposta che vengono restituiti al chiamante, e quindi come visto al *controller* che provvede a visualizzarli nella pagina *web*, dopo aver chiuso la connessione con il *database*.

A questo punto, una volta popolata la tabella visualizzata all'utente è la parte di presentazione che si occupa come visto della gestione delle ultime fasi della ricerca tramite la funzione *Javascript.*

4.1.5 Logica di *business* dell'interrogazione degli ordini

Dopo aver analizzato alcune delle funzionalità più generali dell'applicazione, che saranno utilizzate anche in altri moduli, verranno adesso illustrate le funzionalità direttamente collegate all'interrogazione degli ordini.

Come accennato precedentemente nella schermata di interrogazione degli ordini è presente un *form* contenente dei campi per effettuare una selezione delle caratteristiche su cui si intende effettuare la ricerca di un ordine.

Se non si seleziona nessuna ricerca specifica e si lascia tutte le selezioni di *default* viene effettuata una ricerca che restituisce tutti gli ordini memorizzati nel *database*. Questa, oltre ad essere un'operazione abbastanza pesante visto l'elevato numero di dati presenti, risulta nella maggior parte dei casi essere alquanto inutile, in quanto difficilmente un utente desidera visualizzare davvero tutti gli ordini.

Per effettuare l'interrogazione degli ordini, dopo aver raffinato la ricerca in base ai parametri selezionati, è sufficiente premere il tasto "Invio" visualizzato nella pagina contenente il *form*.

Alla pressione di tale pulsante si viene indirizzati alla pagina "*Oci3e1sds.xhtml*" contenente la tabella in cui saranno visualizzati i dati recuperati. Tale pagina ha quindi al suo interno un componente "*<p:datatable>*" così strutturato:

```
.
.
<p:dataTable id="oci3e1sds_datatable" value="#{oci31Controller.items}" var="item"
             selectionMode="single" selection="#{ocm00Controller.oci3e1sds}"
             dblClickSelect="true" update="oci3e1sds_datatable"
             onselectComplete="location.href='/JSF2App/faces/gestionale/ocm00/
                               CustomerOrder.xhtml'"
             emptyMessage="#{bundle.NoResultFound}" paginator="true" rows="10"
             styleClass="datatable" scrollable="true" width="100%">
   <p:column>
   .
   .
   </p:column>
   .
   .
</p:datatable>
.
.
```

Il meccanismo del *datatable* è analogo a quella illustrato nel paragrafo dedicato alla gestione delle ricerche, con la differenza che alla selezione di una riga si viene indirizzati alla pagina di gestione dell'ordine selezionato che sarà analizzata nella sezione dedicata.

Per il resto l'attributo "*value*" indica che i dati da caricare saranno recuperati dal metodo "*getItems()*" dell"'*Oci31Controller*" che, analogamente a quanto visto per le ricerche quando la variabile "*items*" è nulla[7], richiama il metodo "*retrieveData()*" contenente la logica per recuperare i dati:

```
public List retrieveData() throws Exception {
   Oci3e1_request request = new Oci3e1_request();
   request.setOci3e1ds(oci3e1ds);
   request.setUser(SessionUtil.getUser());
   Oci3e1_response response = ejbFacade.execute(request);
   return response.getOci3e1sds();
}
```

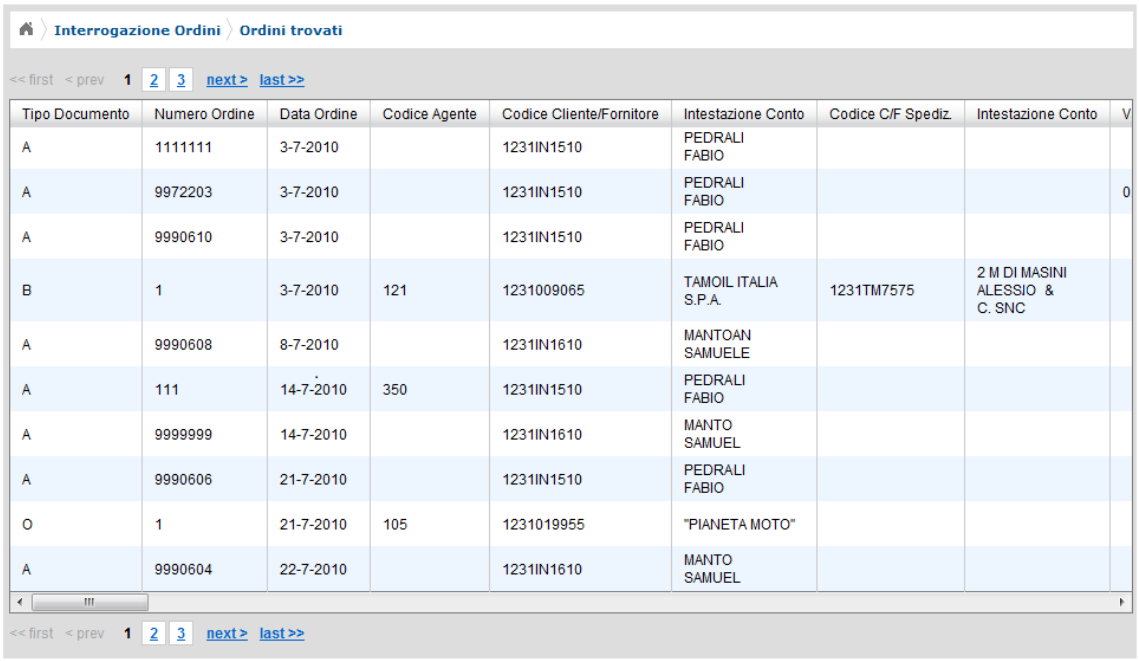
In tale metodo di provvede a creare un oggetto "*Oci3e1_request*" e a popolarlo con i dati dell'utente loggato e dei dati inviati con il *form*, per poi chiamare il metodo

[7]Questo avviene in quanto al momento della pressione del pulsante "Invio" nella pagina del *form* precedente si provvede a resettare per sicurezza tale variabile

"*execute()*" del relativo *EJB* per recuperare una risposta con i dati necessari per popolare la tabella.

All'interno di tale metodo viene composta ed eseguita la *query* necessaria per il recupero dei dati traducendo quanto scritto in codice *RPG* nel sistema originale, provvedendo quindi ad inserire via via nella *query* soltanto le condizioni inserite dall'utente.

Una volta eseguita la *query* viene utilizzato il risultato di essa per popolare la risposta contenente il *bean* da cui saranno prelevati i dati necessari a riempire le colonne della tabella degli ordini come illustrato in figura 4.6.



Tipo Documento	Numero Ordine	Data Ordine	Codice Agente	Codice Cliente/Fornitore	Intestazione Conto	Codice C/F Spediz.	Intestazione Conto	V
A	1111111	3-7-2010		1231IN1510	PEDRALI FABIO			
A	9972203	3-7-2010		1231IN1510	PEDRALI FABIO			0
A	9990610	3-7-2010		1231IN1510	PEDRALI FABIO			
B	1	3-7-2010	121	1231009065	TAMOIL ITALIA S.P.A.	1231TM7575	2 M DI MASINI ALESSIO & C. SNC	
A	9990608	8-7-2010		1231IN1610	MANTOAN SAMUELE			
A	111	14-7-2010	350	1231IN1510	PEDRALI FABIO			
A	9999999	14-7-2010		1231IN1610	MANTO SAMUEL			
A	9990606	21-7-2010		1231IN1510	PEDRALI FABIO			
O	1	21-7-2010	105	1231019955	"PIANETA MOTO"			
A	9990604	22-7-2010		1231IN1610	MANTO SAMUEL			

Figura 4.6: Risultato interrogazione ordini

A questo punto è sufficiente effettuare un doppio *click* con il mouse su una riga della tabella contenente un ordine per visualizzare il dettaglio di tale ordine, effettuando poi eventualmente le operazione ritenute necessarie che saranno analizzate nella prossima sezione.

4.2 Gestione degli ordini

La gestione degli ordini è sicuramente il cuore del sistema informativo di un'azienda che gestisca la commercializzazione di un prodotto. Gli ordini degli articoli, siano

essi effettuati da clienti dell'azienda che da fornitori, devono essere correttamente inseriti, aggiornati e cancellati.

4.2.1 Inserimento di un nuovo ordine

Accedendo dal menù principale dell'applicazione alla voce "Ordini Clienti" -> "Inserimento Ordine" si accede ad una pagina, visibile in figura 4.7, contenente due sezioni, una per scegliere il tipo ed il numero di un ordine da inserire o da modificare, ed una per effettuare la copia dei campi di un altro ordine nel caso in cui si voglia velocizzare la procedura di popolamento dell'ordine nel caso in cui molti campi siano uguali a quelli di un altro ordine.

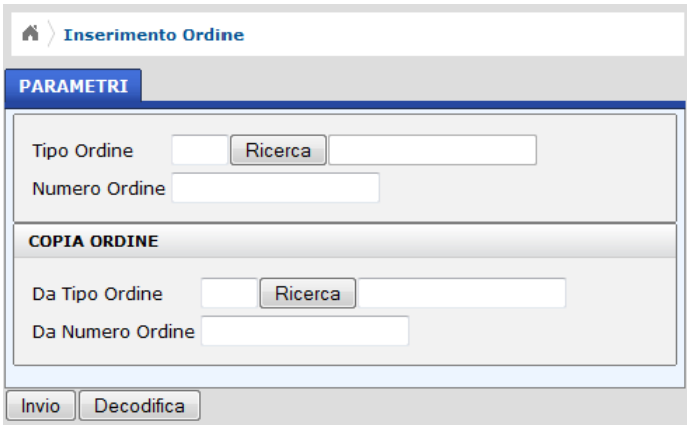


Figura 4.7: Inserimento Ordine - selezione parametri

La pagina visualizzata è strutturata con lo stesso meccanismo visto nel caso della pagina iniziale dell'interrogazione degli ordini. Si ha quindi una pagina contenitore (*CustomerOrder.xhtml*) con il seguente contenuto:

```
.
.
<h:form id="ordini">
   <p:tabView activeIndex="#{ocm00Controller.tabIndex}">
      <ui:include src="tab/ParametersTab.xhtml"/>
      <ui:include src="tab/CustomerDataTab.xhtml"/>
      <ui:include src="tab/CommercialDataTab.xhtml"/>
      <ui:include src="tab/RowsTab.xhtml"/>
   </p:tabView>
   <p:messages showDetail="true"/>
   .
   .
</h:form>
.
.
```

Come si può notare all'interno della prima parte della pagina è presente un componente "*<p:tabView>*" che definisce una sezione a linguette che include al suo interno quattro *file*. Ognuno di tali *file* include semplicemente un componente "*<p:tab>*" che rappresenta una specifica linguetta con i campi desiderati al suo interno.

Per consentire una guida più agevole al popolamento di tale *file* si è cercato di gestire la visualizzazione di solo alcune parti alla volta, in maniera da nascondere all'utente i campi e i pulsanti con cui inizialmente non deve interagire.

Nella pagina inizialmente è infatti visibile una sola linguetta. Questo perché le altre linguette sono visualizzate solo nel caso in cui venga inserito un tipo di ordine corretto.

Esistono infatti varie tipologie di ordini contenute in una tabella apposita del *database* e sia per inserire un nuovo ordine che per modificarne uno esistente è necessario inserire una tipologia di ordine corretta.

Tramite il pulsante di ricerca viene effettuata la solita ricerca nella tabella del *database* per selezionare la tipologia di ordine desiderata. Poiché è possibile inserire anche manualmente il valore nel campo di testo è necessario effettuare un controllo apposito su tale valore.

Per far sì che la pagina in questione abbia il comportamento desiderato, cioè che nasconda inizialmente le altre linguette finché non venga selezionato un tipo di ordine corretto, sono state innanzitutto inseriti nei componenti *<p:tab>* che si intende inizialmente nascondere degli attributi "*rendered*" con valori "*#{ocm00Controller.tabsEnabled == true}*". Con tale controllo si rendono visualizzabili tali linguette solamente nel caso in cui la variabile "*tabsEnabled*" all'interno dell"'*Ocm00Controller*", inizialmente con valore "*false*" abbia assunto valore "*true*".

All'interno del componente *<p:tab>* visualizzato inizialmente, quello del pannello "PARAMETRI", è presente il campo di testo relativo alla tipologia di ordine da inserire che ha il seguente codice:

```
.
.
<h:inputText id="tdocoo" value="#{ocm00Controller.ocmov00f.S }" size="2"
             valueChangeListener="#{ocm00Controller.enableTabs}" onchange="submit();"/>
.
.
```

Rispetto ad altri campi del *form* ha due attributi in più: "*valueChangeListener*" e "*onchange*". Il primo serve per indicare un metodo da richiamare al cambiamento del valore del campo, il secondo serve per indicare l'azione da eseguire al momento del cambiamento del campo, quindi l'invio delle informazioni di tale campo.

Il metodo "*enableTabs()*" è definito come segue:

```
public void enableTabs(ValueChangeEvent e) {
   XcstabgoController xcstabgoController =
                    (XcstabgoController) MyUtils.getManagedBean("xcstabgoController");
   Iterator it = xcstabgoController.getItems().iterator();
   while (it.hasNext()) {
      Xcstabgo xcstabgo = (Xcstabgo) it.next();
      String dbValue = ((String) xcstabgo.getXcodtb()).trim();
      if (dbValue.equals(e.getNewValue().toString().trim())) {
          setTabsEnabled(true);
          return;
      }
   }
   MessagesBean mb = (MessagesBean) MyUtils.getManagedBean("messagesBean");
   mb.addMessages("Tipo Ordine non corretto");
}
```

Il metodo prende automaticamente in ingresso le informazioni relative al campo di *input* cui tale metodo è associato. Tramite una funzione di utilità creata *ad hoc* ed utilizzata più volte all'interno dell'applicazione viene recuperato dalla sessione l'"'*XcstabgoController*" per chiamare il metodo "*getItems()*" che, come in altri casi, permette di recuperare i valori presenti nella tabella opportuna, in questo caso quella contenente l'elenco di tutte le tipologie di ordine.

Una volta recuperate tali informazioni viene effettuata un'iterazione su di esse e viene effettuato un controllo tra il codice della tipologia di ordine presente in tabella e il valore inserito nel campo; nel caso in cui ci sia una corrispondenza tra tali valori viene settata a "*true*" la variabile utilizzata come *flag* per la visualizzazione delle altre linguette della pagina, mentre nel caso in cui non venga trovata alcuna corrispondenza di valori viene utilizzato un *bean* di nome "*messagesBean*" per la gestione degli errori da visualizzare nella pagina tramite il componente "*<p:messages/>*" (v.4.2.1.2).

Se il confronto ha avuto successo è quindi possibile visualizzare il resto della pagina per la gestione dell'ordine in cui adesso è possibile inserire ogni informazione sul nuovo ordine che si intende gestire tramite i due *form* visibili in figura 4.8 e in figura 4.9.

Nei *form* illustrati è possibile notare come alcuni dei campi siano disabilitati in scrittura, infatti si riferiscono esclusivamente al dettaglio dei campi che invece è possibile inserire.

Infatti se si inserisce dei valori nei campi di *input* e si preme il tasto "Decodifica", viene effettuata una ricerca nel *database* delle descrizioni dei vari valori inseriti per effettuare una verifica della correttezza delle informazioni stesse.

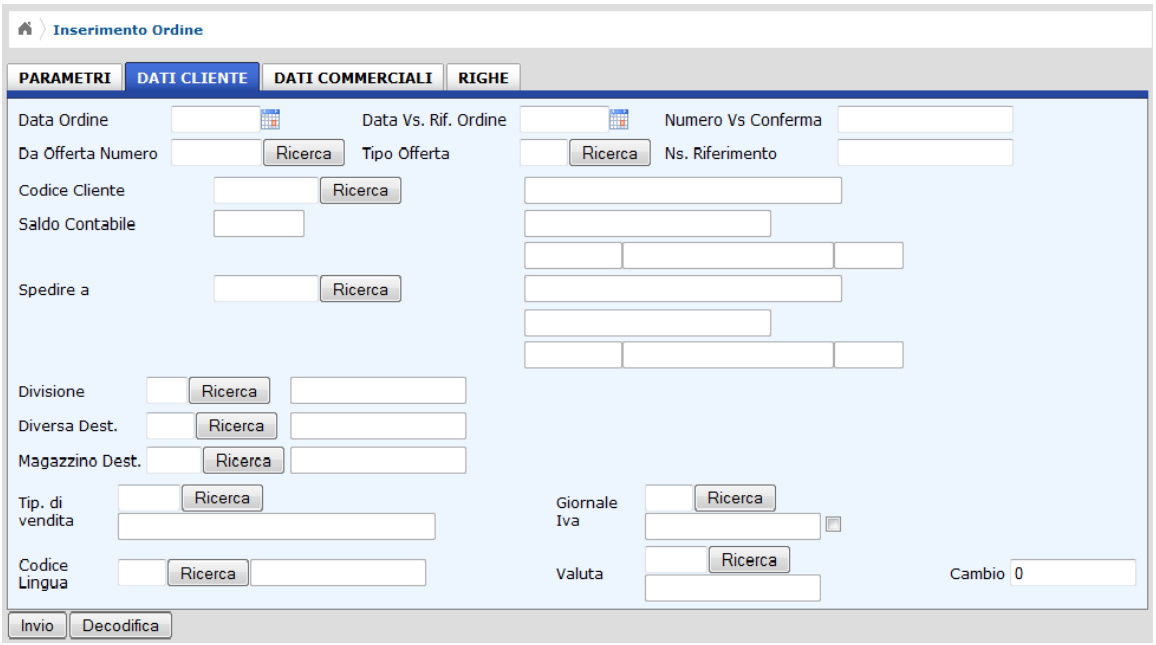


Figura 4.8: Inserimento ordini - Form 'Dati cliente'

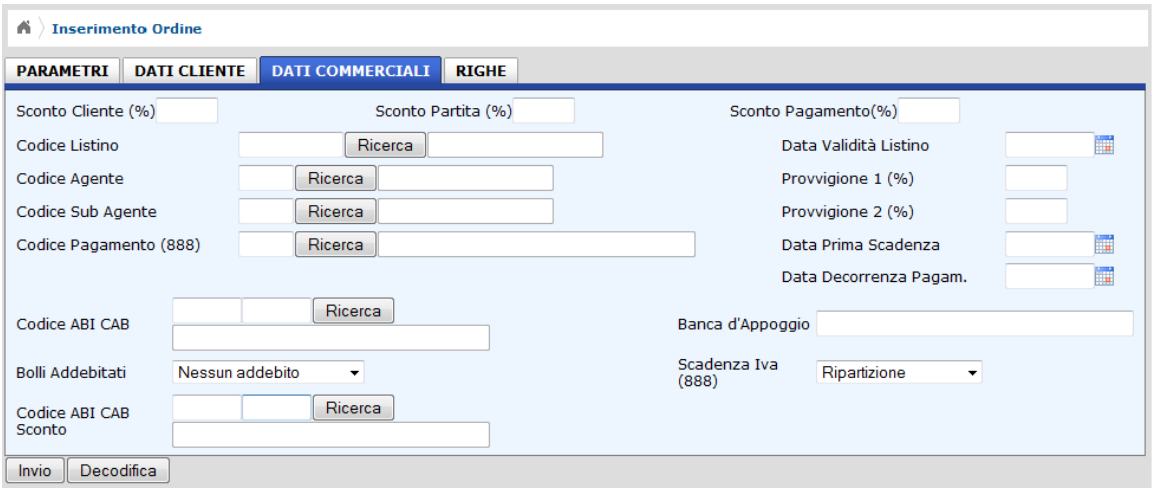


Figura 4.9: Inserimento ordini - Form 'dati commerciali'

Ad esempio se si inserisce il codice della divisione nel campo apposito, dopo la pressione del pulsante "Decodifica" viene visualizzato, se presente, il dettaglio della divisione (in questo caso la semplice descrizione) nel campo a lato come mostrato in figura 4.10.

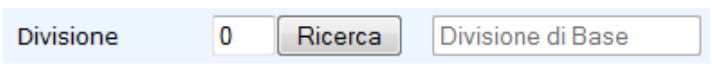


Figura 4.10: Dettaglio Divisione

A tale pulsante è associata infatti l'esecuzione del metodo "*check()*" dell"'*Ocm00-Controller*", che chiama a sua volta il metodo "*checkOrder()*", incaricato di effettuare la richiesta si decodifica, prima di tornare a visualizzare la pagina tramite il meccanismo della navigazione implicita tipico di JavaServer Faces 2:

```
.
.
public String check() throws Exception {
   checkOrder();
   return "CustomerOrder";

}
public Ocm00c_response checkOrder() throws Exception {
   Ocm00c_request c_request = new Ocm00c_request();
   c_request.setOcmov00f(ocmov00f);
   c_request.setFunction("C");
   c_request.setUser(SessionUtil.getUser());
   Ocm00c_response c_response = ocm00cFacade.execute(c_request);
   setOcmov00vds(c_response.getOcmov00vds());
   setOcmov00f(c_response.getOcmov00vds().getOcmov00f());
   MyUtils.composeMessages(c_response.getExceptions());
   return c_response;
}
.
.
```

L'utilizzo di due metodi diversi è dovuto al fatto che il metodo "*check()*" viene richiamato anche in altri casi[8], senza tornare necessariamente alla pagina "*CustomerOrder.xhtml*".

Visto che il procedimento di chiamata è molto simile ai precedenti, andremo adesso ad analizzare ciò che è contenuto nell'oggetto restituito come risposta. Per il popolamento della pagina viene utilizzata la struttura di due *bean* : il primo ("*Ocmov00f*") per i dati legati ai campi di *input* da inviare nella richiesta, ed il secondo ("*Ocmov00vds*") per i dati provenienti dalla risposta che andranno a popolare i campi di descrizione.

Nella risposta è appunto contenuto un *bean* del secondo tipo che sarà quindi memorizzato nel *bean* all'interno del controllore in maniera da poterlo utilizzare all'interno della pagina.

Quando viene premuto il pulsante di decodifica viene effettuata la codifica di tutti i dati inviati, quindi nel caso in cui venga inviato un dato non corrispondente ad alcun elemento nel *database*[9] viene restituito nella risposta un messaggio di errore. Questo e tutti i possibili altri messaggi sono contenuti all'interno della variabile

[8] Ad esempio al momento della scrittura della riga di un ordine come sarà chiaro più avanti

[9] Ad esempio un codice cliente inesistente

"*exceptions*" che contiene una lista di tutti i possibili messaggi con i relativi dettagli che possono essere contenuti in una risposta[10] e che viene utilizzata per popolare il *MessagesBean* utilizzato poi nella pagina per visualizzare i messaggi (v. 4.2.1.2).

L'ultima linguetta si riferisce alle righe presenti in un ordine. Inizialmente sono ovviamente vuote, ma tramite il pulsante "Aggiungi Riga" è possibile aggiungere i dettagli di una riga dell'ordine[11].

Con la pressione del tasto "Invio" i dati vengono memorizzati nel *database* tramite la chiamata del metodo "*write()*", e quindi anche del metodo "*writeOrder()*"[12], dell'*"Ocm00Controller*":

```
.
.
public String write() throws Exception {
   writeOrder();
   setRowsEnabled(true);
   setTabIndex(3);
   items = null;
   return "CustomerOrder";

}
public void writeOrder() throws Exception {
   Ocm00w_request w_request = new Ocm00w_request();
   w_request.setOcmov00f(ocmov00f);
   w_request.setUser(SessionUtil.getUser());
   ocm00wFacade.execute(w_request);
}
.
.
```

Nel metodo "*writeOrder()*" vengono al solito settati nell'oggetto richiesta i *bean* relativi ai dati inseriti nel *form* che vengono poi utilizzati dal metodo "*execute()*" del relativo *EJB* per effettuare la scrittura nel *database* dopo la creazione della *query SQL* di inserimento.

4.2.1.1 Gestione delle ricerche complesse

Nei *form* appena descritti sono presenti alcune ricerche più complesse rispetto a quelle descritte nella sottosezione 4.1.4. In tale caso infatti sono state descritte ricerche di elementi provenienti da un'unica tabella, mentre nel caso del codice

[10] Tale variabile è infatti contenuta nella *CmdResponse* che come abbiamo visto è estesa da tutte le altre risposte utilizzate

[11] Tale funzionalità sarà analizzata più avanti quando si tratterà dell'aggiornamento di un ordine

[12] Vengono utilizzati due metodi per lo stesso motivo analizzato nel caso della funzione di decodifica

cliente, da ricercare nel *form* dei dati cliente, è necessario effettuare una ricerca complessa su più tabelle.

Il meccanismo di ricerca è lo stesso dal punto di vista dell'interfaccia utente, mentre ciò che cambia è la parte di logica. Invece di un "*GenericDAOFacade*", viene utilizzato l"'*AccountChartSearchFacade*", che è appunto l'*EJB* che si occupa di tale operazione. In tale *EJB* viene infatti fatta un'operazione di "*left join*" tra due tabelle da cui vengono estratte le informazioni di interesse, visualizzabili e selezionabili con lo stesso meccanismo visto precedentemente, cioè con un componente "*p:datatable*" che provvederà a gestire anche la selezione di una riga della tabella visualizzata con la memorizzazione del campo chiave nel campo del *form* padre.

Poiché è possibile in questo caso raffinare la ricerca è stato creato un meccanismo a linguette, che permette di visualizzare in un *form* i parametri di ricerca e nell'altro il risultato della ricerca, come visibile in figura 4.11 e in figura 4.12.

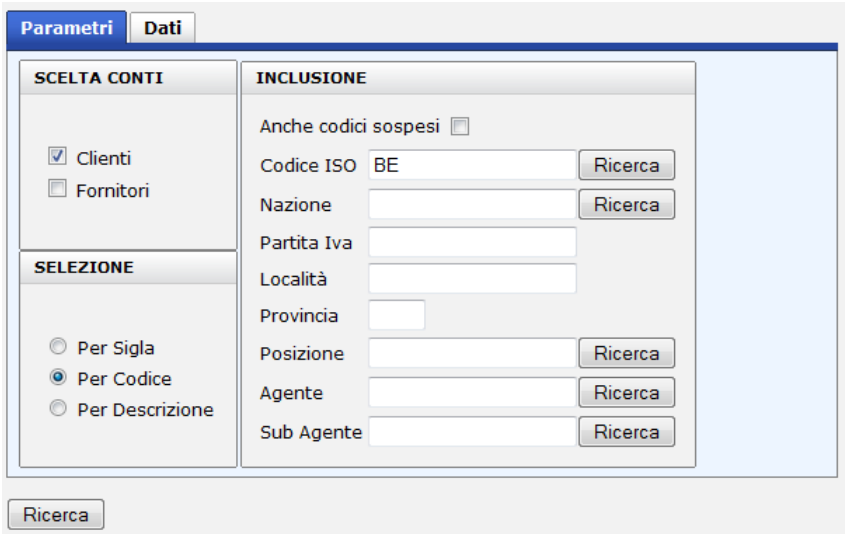


Figura 4.11: Ricerca Codice Cliente - Parametri

Un'altra particolarità di questo caso particolare di ricerca è il fatto che all'interno della finestra visualizzata siano presenti ulteriori pulsanti di ricerca. Il meccanismo per effettuare le nuove ricerche è quello classico con la particolarità che in questo caso deve essere usato un ulteriore *bean* di ricerca diverso dal "*SearchBean*" utilizzato precedentemente perché si è all'interno della stessa sessione ed i dati contenuti nel precedente *bean* devono essere ancora memorizzati nel *form* di partenza.

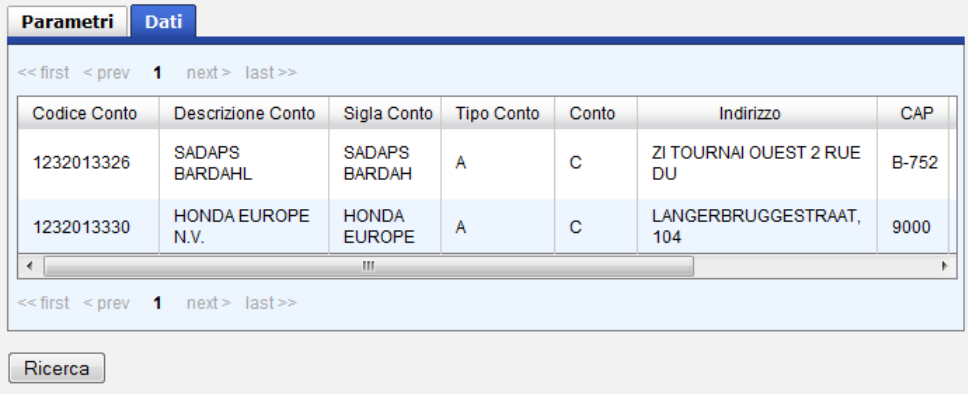


Figura 4.12: Ricerca Codice Cliente - Dati

4.2.1.2 Gestione dei messaggi

PrimeFaces mette a disposizione un apposito *tag* per la gestione dei messaggi con un *layout* predefinito molto immediato. Tramite il *tag* "*<p:messages/>*" possono essere infatti visualizzati i messaggi in un formato molto semplice e per questo molto chiaro come visibile in figura 4.13 dove è illustrato un pannello di errore contenente due specifici messaggi.

Figura 4.13: Messaggi di errore

Per visualizzare il messaggio è sufficiente inserire all'interno della pagina il codice seguente nel punto esatto in cui si vuole che tali messaggi vengano visualizzati (quando presenti):

```
<p:messages showDetail="true"/>
```

Per popolare i messaggi nelle varie pagine in maniera personalizzata è stato utilizzato un *bean* di supporto denominato "*MessagesBean*", definito come segue:

```
@ManagedBean(name = "messagesBean")
@RequestScoped

public class MessagesBean {
   public void addErrorMessage(String message) {
      FacesContext.getCurrentInstance().addMessage(null,
                     new FacesMessage(FacesMessage.SEVERITY_ERROR, "Attenzione", message));

   }
   public void addInfoMessage(String message) {
      FacesContext.getCurrentInstance().addMessage(null,
```

```
                    new FacesMessage(FacesMessage.SEVERITY_INFO, "Informazione", message));
   }
}
```

Questo *bean* non fa altro che utilizzare le *API JSF* per memorizzare nel contesto i messaggi passati come parametro al metodo "*addErrorMessage()*" in maniera da visualizzarli ogni volta che nell'albero dei componenti è presente il componente "*<p:messages>*".

Si noti come lo *scope* del *bean* sia in questo caso di richiesta perché solitamente un messaggio deve essere visualizzato immediatamente a seguito di una specifica richiesta.

Nel caso in cui si voglia sfruttare un oggetto di tipo "*List<ExceptionRes>*"[13], è sufficiente chiamare il metodo statico di utilità "*composeMessages()*" che effettua proprio un'iterazione su tale oggetto per estrarre i messaggi di errore e aggiungerli al contesto tramite il "*MessagesBean*":

```
public static void composeMessages(List<ExceptionRes> exceptions) {
   MessagesBean mb = (MessagesBean) MyUtils.getManagedBean("messagesBean");
   Iterator it = exceptions.iterator();
   while (it.hasNext()) {
      ExceptionRes er = (ExceptionRes) it.next();
      mb.addMessages(er.getMessage());
   }
}
```

4.2.1.3 Gestione della visualizzazione dei pulsanti

Dalle figure relative al *form* di inserimento ordini, si può notare come alcuni pulsanti non siano utilizzabili.

La motivazione di questa scelta implementativa è relativa al fatto che il *form* appena analizzato è utilizzato anche per la visualizzazione e la modifica di ordini già presenti, come sarà più chiaro nella sottosezione 4.2.2.

All'interno della pagina sono presenti i seguenti pulsanti:

```
.
.
<p:commandButton value="Invio" action="#{ocm00Controller.write}" ajax="false"
                 disabled="#{ocm00Controller.disabledFields}"/>
<p:commandButton value="Decodifica" action="#{ocm00Controller.check}" ajax="false"
                 disabled="#{ocm00Controller.disabledFields}"/>
<p:commandButton value="Modifica" action="#{ocm00Controller.setDisabledFields('false')}"
```

[13]Tale oggetto è presente all'interno delle risposte provenienti dalle chiamate *EJB* che provvedono a popolarlo opportunamente con le informazioni sugli errori riscontrati in seguito alla chiamata del metodo specifico.

```
                ajax="false" disabled="#{ocm00Controller.disabledFields eq 'false'}"
                rendered="#{ocm00Controller.formFunction eq 'manage'}"/>
<p:commandButton value="Elimina" action="#{ocm00Controller.delete}" ajax="false"
                rendered="#{ocm00Controller.formFunction eq 'manage'}">
   .
   .
</p:commandButton>
.
.
```

Tramite l'attributo "*disabled*" si è fatto in modo che ogni pulsante possa essere premuto soltanto in alcuni casi, in base all'operazione che si sta compiendo.

Il valore della variabile booleana "*disabledFields*" viene aggiornato al verificarsi di determinate azioni. Ad esempio selezionando un ordine dopo un'interrogazione è possibile solamente premere il tasto "Modifica" o il tasto "Elimina", in quanto si è in fase di visualizzazione di un ordine.

Nel caso si intenda modificare il contenuto dell'ordine, tramite il pulsante "Modifica" viene settata tale variabile a "*false*", in modo tale che si abilitino i pulsanti "Invio" e "Decodifica" che possono essere utili dopo un'eventuale modifica della testata dell'ordine stesso.

L'attributo "*rendered*" è invece utilizzato per la gestione della visualizzazione dei pulsanti. Come detto infatti il *form* è utilizzato sia per l'inserimento che per la gestione di un ordine, quindi nel caso dell'inserimento di un ordine è inutile visualizzare i pulsanti di modifica e di eliminazione.

4.2.2 Aggiornamento di un ordine esistente

Oltre ad inserire un nuovo ordine è fondamentale consentire la modifica dell'ordine stesso.

Per scegliere l'ordine da modificare è necessario innanzitutto cercarlo nel *database* tramite l'interrogazione degli ordini analizzata precedentemente. Effettuando il doppio *click* sulla riga della tabella relativa all'ordine scelto si può visualizzare il *form* di gestione degli ordini popolato con i dati dell'ordine selezionato grazie all'utilizzo dei soliti attributi del *datatable*:

```
.
.
<p:dataTable id="oci3e1sds_datatable" value="#{oci31Controller.items}" var="item"
             selectionMode="single" selection="#{ocm00Controller.oci3e1sds}"
             dblClickSelect="true" update="oci3e1sds_datatable"
             onselectComplete="location.href='/JSF2App/faces/gestionale/ocm00/
             CustomerOrder.xhtml'" emptyMessage="#{bundle.NoResultFound}" paginator="true"
```

```
            rows="10" scrollable="true" width="100%" styleClass="datatable">
.
.
</p:datatable>
.
.
```

In questo caso però il comportamento è leggermente diverso per permettere durante questa operazione di fare ulteriori operazioni supplementari. Al momento della selezione di una riga della tabella infatti i relativi dati vengono utilizzati per popolare l'oggetto definito dal valore dell'attributo "*selection*". Per fare ciò viene in realtà chiamato il metodo "*setter*" di tale oggetto contenuto all'interno dell"'*Ocm00Controller*" e per effettuare le relative operazioni aggiuntive ritenute necessarie in questa operazione si è provveduto proprio a ridefinire tale metodo nella maniera seguente:

```
.
.
public void setOci3e1sds(Oci3e1sds oci3e1sds) throws Exception {
   items = null;
   Ocmov00f ocmov00f_app = new Ocmov00f();
   ocmov00f_app.setCddtoo("01");
   ocmov00f_app.setTdocoo(oci3e1sds.getTdoc());
   ocmov00f_app.setNroroo(oci3e1sds.getNror().intValue());
   Ocm00c_request req = new Ocm00c_request();
   req.setOcmov00f(ocmov00f_app);
   req.setUser(SessionUtil.getUser());
   req.setFunction("RC");
   Ocm00c_response res = ocm00cFacade.execute(req);
   setOcmov00f(res.getOcmov00vds().getOcmov00f());
   setOcmov00vds(res.getOcmov00vds());
   setDisabledFields(true);
   setTabsEnabled(true);
   setRowsEnabled(true);
   setFormFunction("manage");
   this.oci3e1sds = oci3e1sds;
}
.
.
```

Il *bean* contenente i dati visualizzati nella tabella contiene solo una parte dei dati che saranno necessari per popolare il *form* di gestione ed effettuare le successive operazioni. Di conseguenza prima di procedere con la visualizzazione indicata dall'attributo *onselectComplete = "location.href='/JSF2App/faces/gestionale/ocm00/CustomerOrder.xhtml'"* del *datatable* si procede nell'ordine:

1. a resettare il contenuto della variabile "*items*" per effettuare un nuovo caricamento della tabella con le righe;

2. a creare un oggetto di tipo "*ocvmov00f*" popolandolo con i dati relativi all'ordine;

3. a settare un oggetto richiesta con i valori necessari per effettuare la chiamata;

4. ad utilizzare tale oggetto per richiamare la procedura di decodifica della testata dell'ordine in modo da recuperare anche i campi descrittivi tramite una chiamata all'*EJB* "*Ocm00cFacade*";

5. a settare dei *bean* utilizzati per i popolamento della pagina con i dati contenuti nella risposta della chiamata all'*EJB*;

6. a settare un *flag* che disabilita la scrittura dei campi di *input*;

7. a settare il *flag* di visualizzazione delle linguette in quanto avendo recuperato un ordine da una lista si è sicuramente in presenza di un ordine valido, quindi non è necessario effettuare nuovamente il controllo sulla validità del tipo di ordine;

8. a settare una variabile utilizzata per la creazione delle briciole di pane della pagina;

9. ad effettuare l'operazione di settaggio della variabile prevista inizialmente dal metodo.

Inizialmente i dati ed i pulsanti possono essere soltanto visualizzati senza possibilità di interazione con essi, come visibile in figura 4.14, ma tramite la semplice pressione del pulsante "Modifica" è possibile attivare la modifica dei campi di *input* del *form*.

Dopo aver effettuato le modifiche ritenute opportune e le eventuali operazioni di decodifica è possibile memorizzare tali cambiamenti con la semplice pressione del tasto "Invio", che provvede, come visto precedentemente, a scrivere nel *database* facendo l'aggiornamento di tale ordine.

4.2.2.1 Gestione delle righe di un ordine

Oltre alle informazioni generali relative all'ordine è possibile gestirne il dettaglio delle righe. Andando nell'ultima linguetta relativa alla gestione delle righe viene visualizzata una schermata tipo quella illustrata in figura 4.15.

Effettuando un doppio *click* sulla riga desiderata viene aperta una nuova finestra per la gestione della riga stessa dell'ordine con i campi del *form* già popolati con i dati relativi alla riga dell'ordine selezionata, come visibile in figura 4.16, 4.17 e 4.18.

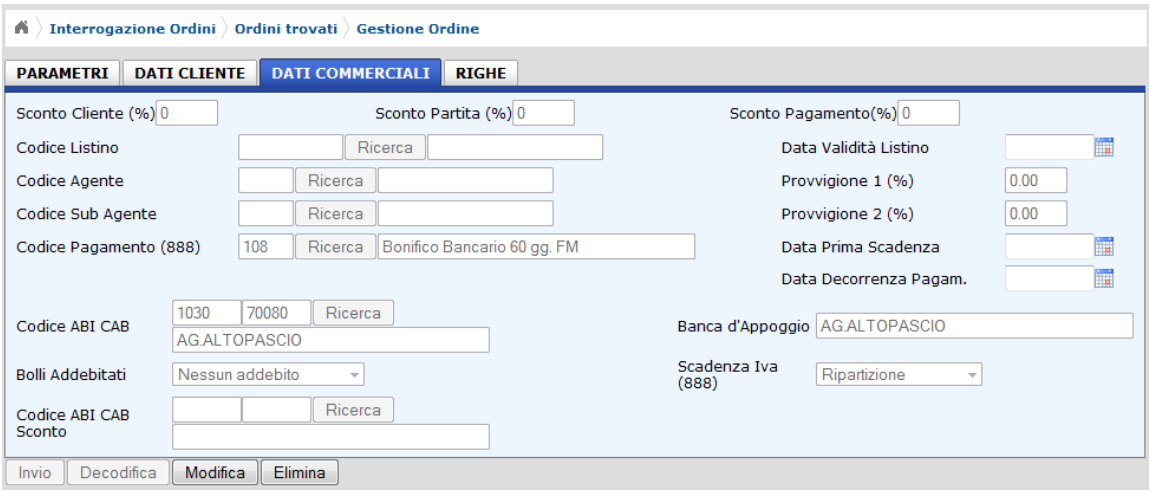


Figura 4.14: Esempio di form di sola visualizzazione

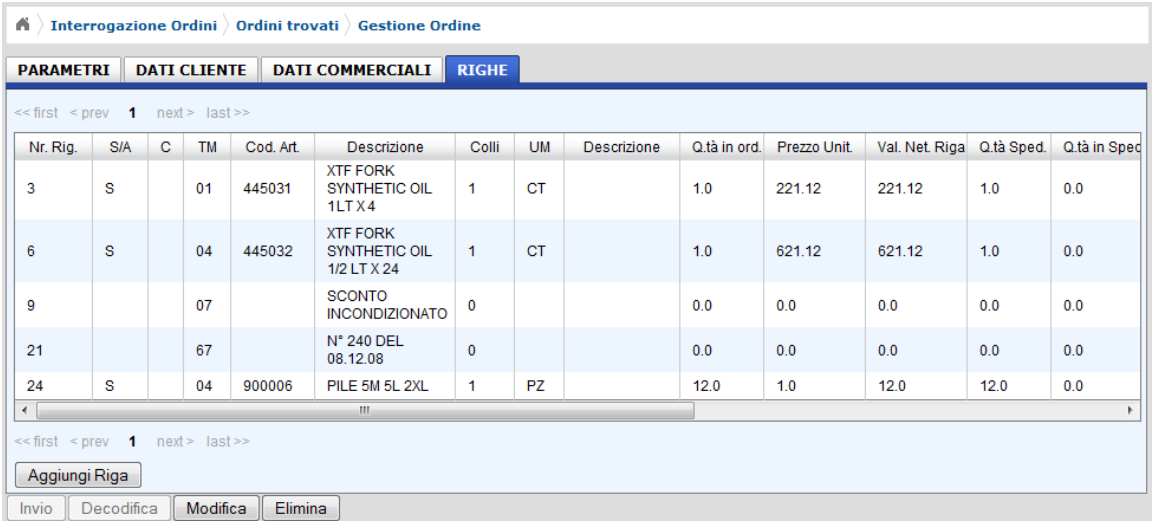


Figura 4.15: Aggiornamento Ordini - Tabella 'Righe'

Per ottenere questo comportamento è necessario utilizzare il solito meccanismo di selezione delle righe analizzato precedentemente, quindi utilizzare un componente <*p:datatable*> con gli attributi opportuni:

```
.
.
<p:dataTable id="rowsDatatable" value="#{ocm00Controller.items}" var="item"
             selectionMode="single" selection="#{ocm01Controller.ocm01lds}"
             update="rowsDatatable" dblClickSelect="true" onselectComplete="window.open(
             '#{myUtils.getContextPath()}/faces/gestionale/ocm01/RowsDetailOrder.xhtml',
             '', 'width=800,height=600,left=0,top=0,resizable=no,menubar=yes,toolbar=yes,
             scrollbars=no,locations=no,status=no');" emptyMessage="#{bundle.NoResultFound}"
             paginator="true" rows="10" scrollable="true" width="100%"
             styleClass="datatable">
```

Figura 4.16: Aggiornamento Ordini - Dettaglio Righe - Parametri

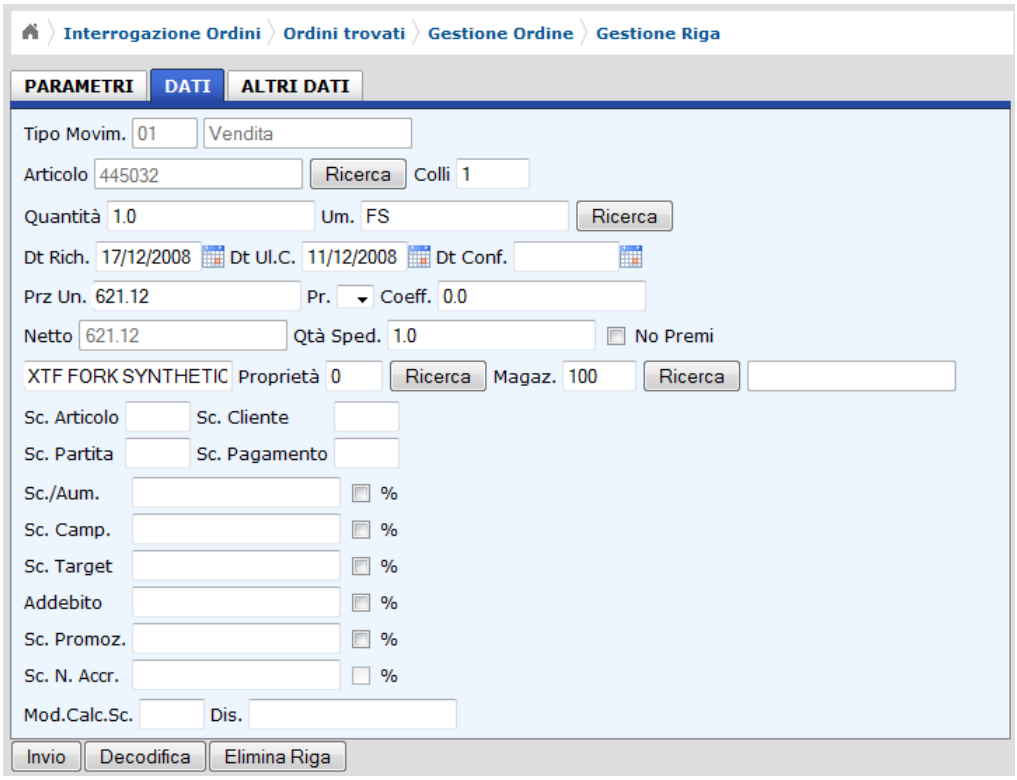


Figura 4.17: Aggiornamento Ordini - Dettaglio Righe - Dati

```
.
.
</p:datatable>
.
.
```

Come al solito l'attributo "*onselectComplete*" indica l'azione da eseguire a selezione avvenuta (in questo caso l'apertura della finestra contente il *form* del dettaglio delle

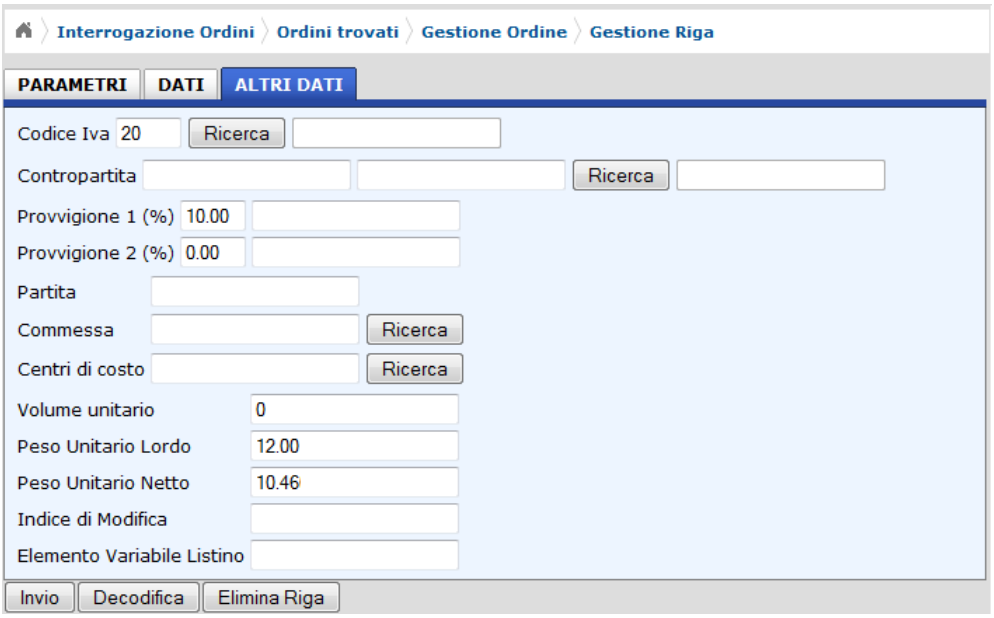


Figura 4.18: Aggiornamento Ordini - Dettaglio Righe - Altri Dati

righe), mentre l'attributo "*selection*" indica l'oggetto da popolare con i valori della riga selezionata (in questo caso l'oggetto "*ocm01lds*").

Poiché per popolare l'oggetto viene utilizzato il relativo metodo "*setter*" contenuto nell"'*Ocm01Controller*", è sufficiente inserire la logica da eseguire all'interno di tale metodo per eseguire il popolamento dell'oggetto utilizzato per la gestione del *form*:

```
.
.
public void setOcm01lds(Ocm01lds ocm01lds) throws Exception {
   Ocm00Controller ocm00Controller =
                        (Ocm00Controller) MyUtils.getManagedBean("ocm00Controller");
   if (ocm01lds == null || ocm00Controller.getFormFunction().equals("insert")) {
      return;
   }
   ocmov00f = ocm00Controller.getOcmov00f();
   Ocm01c_request c_request = new Ocm01c_request();
   Ocmov01f ocmov01f_app = new Ocmov01f();
   ocmov01f_app.setCddtoo(ocmov00f.getCddtoo());
   ocmov01f_app.setTdocoo(ocmov00f.getTdocoo());
   ocmov01f_app.setNroroo(ocmov00f.getNroroo());
   ocmov01f_app.setNrrgoo(ocm01lds.getNrrgoo());
   ocmov01f_app.setNsrgoo(ocm01lds.getNsrgoo());
   c_request.setOcmov00f(ocmov00f);
   c_request.setOcmov01f(ocmov01f_app);
   c_request.setUser(SessionUtil.getUser());
   c_request.setFunction("RC");
   Ocm01c_response c_response = ocm01cFacade.execute(c_request);
   setOcmov01f(c_response.getOcmov01f());
```

```
        setOcmov01vds(c_response.getOcmov01vds());
        this.ocm01lds = ocm01lds;
    }
    .
    .
```

In tale metodo, dopo un controllo necessario per fare in modo che il metodo gestisca esclusivamente la parte di gestione dell'ordine e non di inserimento, viene settato l'oggetto richiesta con le informazioni relative all'ordine selezionato e alla riga selezionata per effettuare un recupero delle informazioni relative alla riga tramite il metodo "*execute()*" dell"'*ocm01cFacade*".

Le informazioni recuperate dall'oggetto "*Ocm01c_response*" vengono utilizzate per settare gli oggetti di tipo "*Ocmov01f*" e di tipo "*Ocmov01vds*" che verranno utilizzati all'interno della pagina *web* contenente il *form* del dettaglio righe per recuperare il valore dei vari campi.

Nel caso in cui si intenda invece aggiungere una nuova riga relativa all'ordine che si sta gestendo è necessario premere sul pulsante "Aggiungi Riga" per fare modo che venga aperta una nuova schermata identica a quella che viene aperta in caso di selezione di una precisa riga per la modifica, con la differenza che in questo caso la maggior parte dei campi del *form* sono vuoti in quanto devono appunto essere inseriti, come visibile in figura.

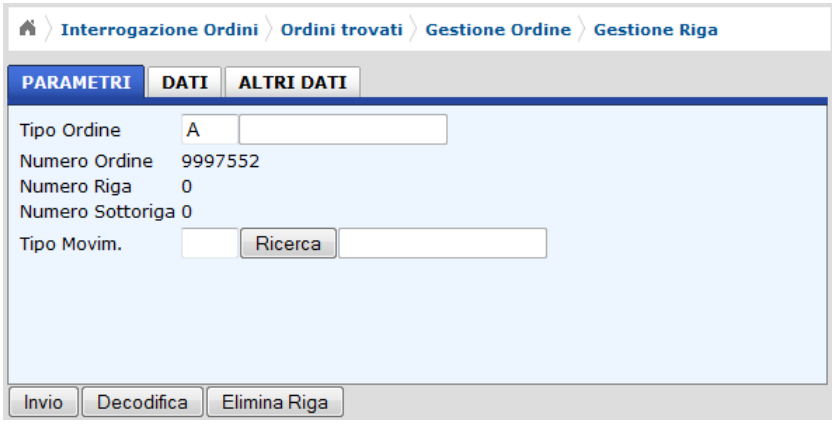


Figura 4.19: Inserimento riga ordine - tab 'Parametri'

Come si può vedere sono caricati solo i campi relativi all'ordine selezionato. Questo è effettuato tramite la chiamata al metodo "*prepareAddRow()*" dell"'*Ocm01-Controller*" che provvede anche ad effettuare l'azzeramento degli eventuali oggetti memorizzati in sessione:

```
    .
```

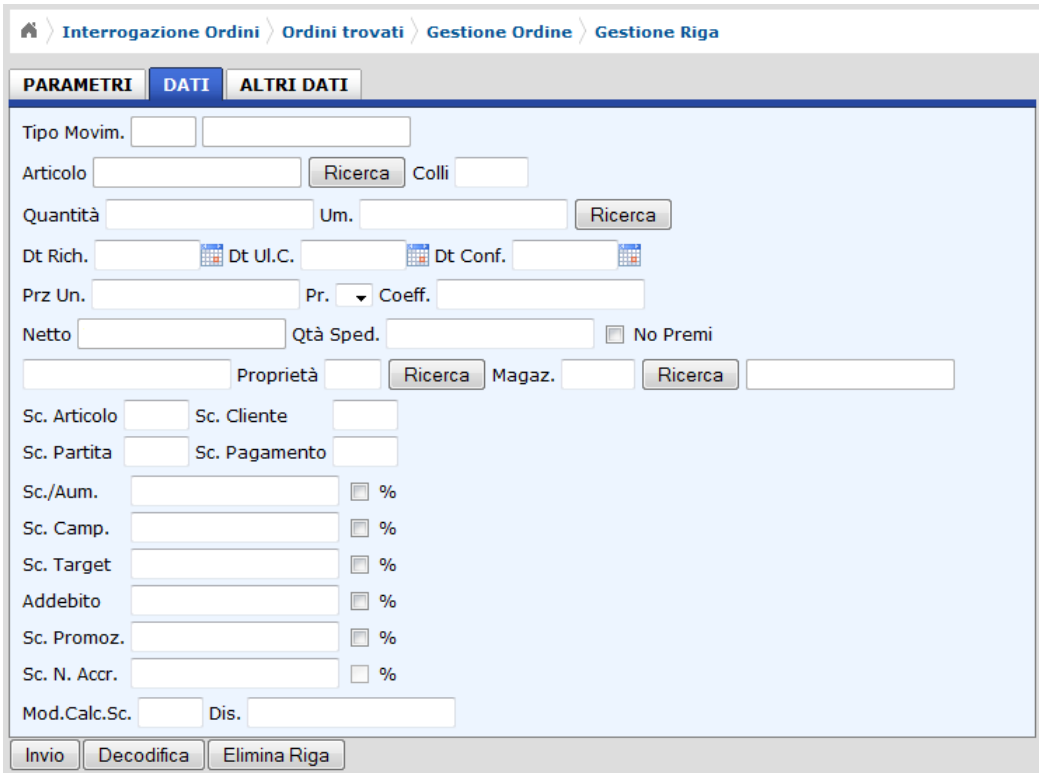


Figura 4.20: Inserimento riga ordine - tab 'Dati'

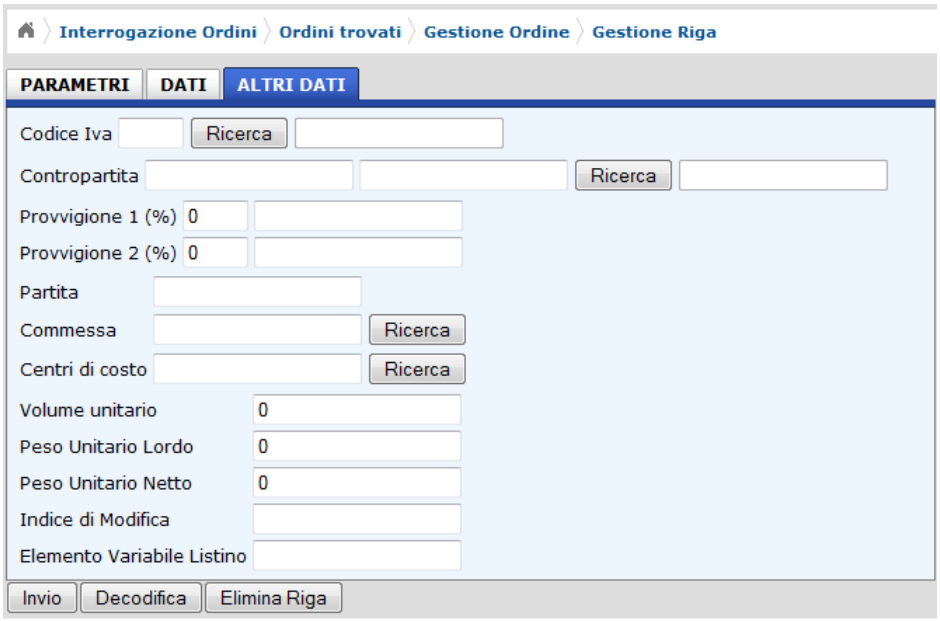


Figura 4.21: Inserimento riga ordine - tab 'Altri Dati'

```
.
public void prepareAddRow(){
   Ocm00Controller ocm00controller =
```

```
                        (Ocm00Controller) MyUtils.getManagedBean("ocm00Controller");
   Ocm00c_response ocm00c_response = ocm00controller.checkOrder();
   if (ocm00c_response.getExceptions().size() == 0) {
      ocm00controller.writeOrder();
      setRowTabIndex(1);
      ocmov01f = new Ocmov01f();
      ocmov01vds = new Ocmov01vds();
      getOcmov01f().setTdocoo(getOcmov00f().getTdocoo());
      getOcmov01f().setNroroo(getOcmov00f().getNroroo());
      return "/gestionale/ocm01/RowsDetailOrder.xhtml";
   }
   return "CustomerOrder";
}
.
.
```

Come si può vedere dal codice al momento della chiamata di tale metodo vengono anche effettuati:

- la decodifica dell'ordine per verificare, nel caso in cui nel frattempo l'ordine sia stato modificato, se sono stati inseriti dati corretti (in caso contrario si rimane nella schermata dell'ordine);
- eventualmente l'aggiornamento dell'ordine per fare in modo che la nuova riga sia aggiunta su un ordine correttamente aggiornato.

A questo punto è possibile effettuare varie operazioni sulla riga visualizzata: la decodifica dei dati, l'invio dei dati e la cancellazione dell'intera riga dall'ordine.

Tramite la pressione del tasto "Decodifica" viene chiamato il metodo "*checkRow*()" dell"'*Ocm01Controller*" che effettua operazioni analoghe a quanto visto nel caso di decodifica della testata di un ordine:

```
.
.
public String checkRow() throws Exception {
   Ocm01c_request c_request = new Ocm01c_request();
   c_request.setFunction("C");
   c_request.setUser(SessionUtil.getUser());
   c_request.setOcmov01f(ocmov01f);
   c_request.setOcmov00f(ocmov00f);
   Ocm01c_response c_response = ocm01cFacade.execute(c_request);
   setOcmov01vds(c_response.getOcmov01vds());
   MyUtils.composeMessages(c_response.getExceptions());
   return "RowsDetailOrder";
}
.
.
```

Effettuati i necessari inserimenti nel *form* di gestione delle righe è necessario procedere con la memorizzazione di tali dati nel *database*, sia che si tratti di un nuovo inserimento che di un aggiornamento di una riga esistente. Per fare ciò è sufficiente premere il pulsante "Invio" con il quale viene chiamato il metodo "*writeRow()*" dell'"'*Ocm01Controller*":

```
.
.
public void writeRow() throws Exception {
   Ocm00Controller ocm00controller =
                          (Ocm00Controller) MyUtils.getManagedBean("ocm00Controller");
   Ocm00c_response ocm00c_response = ocm00controller.checkOrder();
   if (ocm00c_response.getExceptions().size() == 0) {
      ocm00controller.writeOrder();
      setRowTabIndex(1);
      Ocm01w_request w_request = new Ocm01w_request();
      w_request.setOcmov01f(ocmov01f);
      w_request.setUser(SessionUtil.getUser());
      Ocm01w_response w_response = ocm01wFacade.execute(w_request);
   }
}
.
.
```

Tale metodo effettua innanzitutto effettua innanzitutto la decodifica e l'aggiornamento dei dati dell'ordine controllando che non ci siano errori, e successivamente procede con la chiamate del metodo "*execute()*" dell'*EJB* "*Ocm01Facade*" che si occupa della scrittura delle informazioni nelle opportune tabelle del *database*.

E' possibile, dopo averne visualizzato i dettagli, decidere di eliminare un'intera riga già presente in un ordine.

Per fare ciò è necessario procedere con la pressione del tasto "Elimina Riga", che comporta l'apertura di una finestra di richiesta di conferma, come visibile in figura 4.22, trattandosi di un'operazione irreversibile.

Per ottenere questo comportamento è sufficiente inserire il componente *<p:confirmDialog>* nel *file* "*RowsDetailOrder.xhtml*" come illustrato di seguito:

```
.
.
<p:commandButton value="Elimina Riga" action="#{ocm01Controller.deleteRow}">
   <p:confirmDialog message="#{bundle.ConfirmDeleteRow}" header="#{bundle.AdvertenceMessage}"
                    fixedCenter="true" yesLabel="#{bundle.YesLabel}"/>
</p:commandButton>
.
.
```

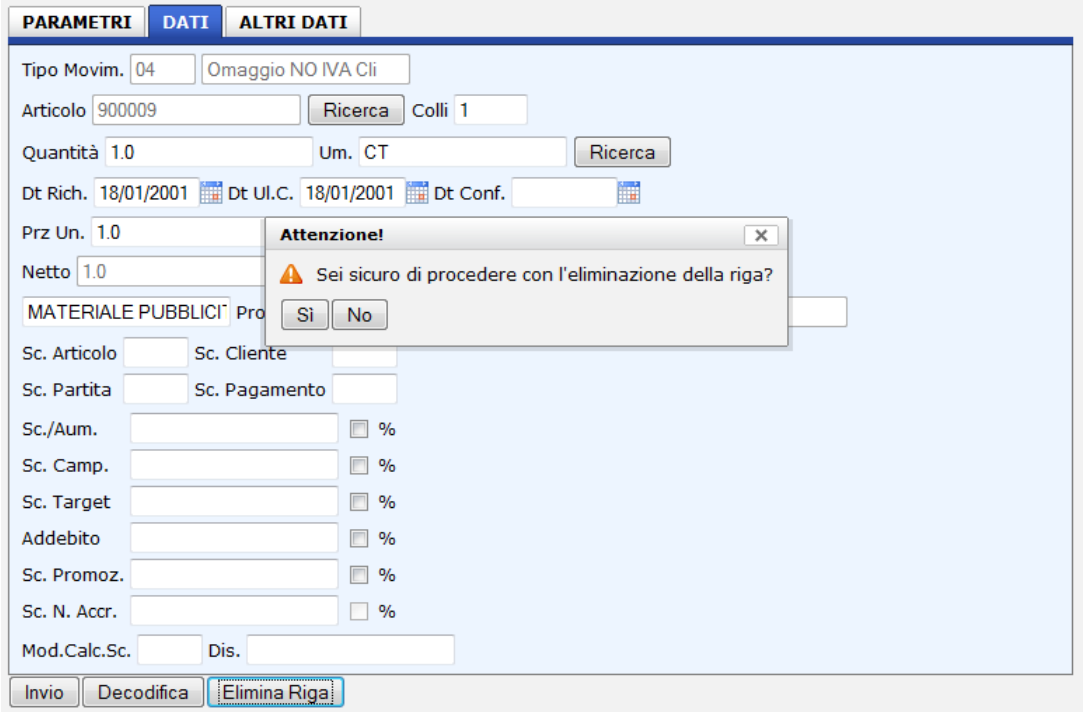


Figura 4.22: Finestra di conferma eliminazione riga

Prima di essere chiamato il metodo identificato dal valore dell'attributo "*action*" viene visualizzata la finestra di conferma, con cui l'utente può interagire confermando o meno l'operazione.

Nel caso in cui l'utente decida di non procedere con l'eliminazione della riga e quindi risponda in maniera negativa alla domanda che gli viene posta, la finestra viene chiusa e non viene eseguita alcuna azione quindi si rimane nella situazione precedente alla pressione del pulsante "Elimina Riga".

Se invece l'utente decide di confermare l'operazione viene ugualmente chiusa la finestra, ma in questo caso il flusso di esecuzione passa in mano al metodo indicato dall'attributo "*action*", quindi il metodo "*deleteRow()*" dell"'*Ocm01Controller*":

```
.
.
public String deleteRow() throws Exception {
   Ocm01d_request d_request = new Ocm01d_request();
   d_request.setUser(SessionUtil.getUser());
   d_request.setOcmov01fPkey(new Ocmov01fPkey(ocmov01f.getCddtoo(), ocmov01f.getTdocoo(),
                                              ocmov01f.getNroroo(), ocmov01f.getNrrgoo(),
                                              ocmov01f.getNsrgoo()));
   ocm01dFacade.execute(d_request);
   Ocm00Controller ocm00Controller =
                          (Ocm00Controller) MyUtils.getManagedBean("ocm00Controller");
   ocm00Controller.setItems(null);
```

```
        ocm00Controller.setTabIndex(3);
        MessagesBean mb = (MessagesBean) MyUtils.getManagedBean("messagesBean");
        mb.addInfoMessage("Riga eliminata con successo");
        return "/gestionale/ocm00/CustomerOrder";
    }
    .
    .
```

In tale metodo si provvede a creare la chiave della tabella relativa alla riga da eliminare tramite il contenuto dell'oggetto "*ocmov01f*", popolato al momento della selezione della riga, e a settare un oggetto di tipo "*Ocm01d_request*" contenente appunto la chiave della tabella e le solite informazioni sull'utente loggato.

Tale richiesta viene utilizzata dall'*EJB* che si occupa dell'eliminazione della riga da tutte le tabelle coinvolte; infine viene effettuata un'operazione di pulizia dei delle righe memorizzate in sessione e creato un messaggio da visualizzare nella pagina.

La pulizia dell'oggetto "*items*" è effettuata per consentire la corretta visualizzazione dell'elenco aggiornato delle righe dell'ordine.

Nel momento in cui viene ricaricata la pagina contente la tabella delle righe viene chiamato il metodo "*getItems()*" per recuperare le righe stesse; a questo punto, poiché "*items*" è stato precedentemente azzerato viene effettuata nuovamente la *query* che consente di recuperare l'elenco delle righe dell'ordine, che sarà simile a quello precedentemente visualizzato con l'assenza però della riga appena eliminata.

4.2.3 Cancellazione di un ordine

Dopo aver analizzato l'inserimento e la modifica è opportuno vedere anche come un ordine venga cancellato.

Per fare ciò è necessario innanzitutto selezionare l'ordine che si intende eliminare con la procedura vista precedentemente (v. sezione relativa all'interrogazione degli ordini). Dopo aver effettuato sulla riga della tabella degli ordini che identifica l'ordine che si intende eliminare viene aperto il solito *form* contenente i dettagli dell'ordine.

Tra i vari pulsanti disponibili ce ne è uno denominato "Elimina" che è proprio quello che permette di eliminare l'ordine visualizzato. Cliccando su tale pulsante viene aperta innanzitutto una finestra (figura 4.23) in cui si richiede la conferma dell'operazione all'utente, come nel caso relativo all'eliminazione di una riga dell'ordine.

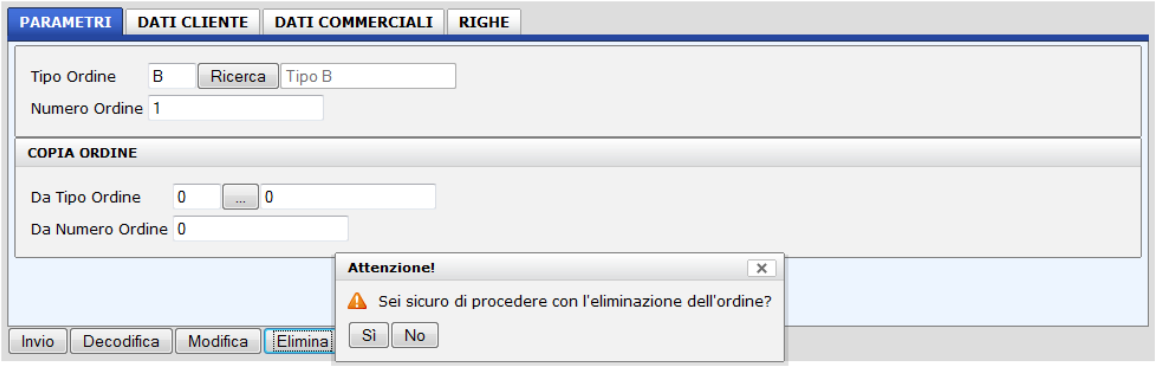


Figura 4.23: Finestra di conferma eliminazione ordine

Il comportamento è analogo al caso relativo all'eliminazione di una riga, con la chiamata al metodo "*delete()*" dell"'*ocm00Controller*" nel caso in cui l'utente decida di procedere con l'eliminazione:

```
.
.
public String delete() throws Exception {
   Ocm00d_request d_request = new Ocm00d_request();
   Ocmov00fPkey ocmov00fkey = new Ocmov00fPkey(ocmov00f.getCddtoo(), ocmov00f.getTdocoo(),
                                                ocmov00f.getNroroo());
   d_request.setOcmov00fPkey(ocmov00fkey);
   d_request.setUser(SessionUtil.getUser());
   ocm00dFacade.execute(d_request);
   return "/gestionale/oci3e1/Oci3e1ds.xhtml";
}
.
.
```

Dopo le operazioni di eliminazione si ritorna alla pagina contenente il *form* di interrogazione degli ordini popolato con le informazioni relative all'ultima ricerca in modo che, tramite la immediata pressione del tasto "Invio" è possibile visualizzare il nuovo contenuto del *database*, dove spiccherà l'assenza dell'ordine appena eliminato.

Capitolo 5

Conclusioni e sviluppi futuri

Scopo della tesi è stata la reingegnerizzazione dell'applicativo gestionale della ditta *Maroil-Bardhal* Italia, derivante dal bisogno di un ammodernamento del sistema informativo aziendale.

Il lavoro svolto rappresenta soltanto l'inizio di tale attività; il modulo di gestione degli ordini rappresenta il punto principale da cui si è partiti, ma ci sono moltissime altre funzionalità da implementare per replicare il sistema attualmente utilizzato in azienda: gestione del piano dei conti, del magazzino, delle spedizioni, degli acquisti e dei pagamenti, funzionalità relative alla contabilità e al budget, sono solo alcune di esse.

Vista la complessità del sistema risulta evidente il perché delle varie scelte orientate alla massima modularità e manutenibilità del sistema. Si prevede infatti che lo sviluppo del resto dell'applicazione avvenga in maniera graduale perciò è fondamentale che le principali funzionalità siano utilizzabili fin da subito e soprattutto che le successive aggiunte da effettuare siano completamente trasparenti agli utilizzatori dei moduli già implementati.

Questa precisa scelta di sviluppo consente inoltre di rimanere più al passo con le nuove tecnologie, in quanto la sostituzione di un componente o il cambiamento di una funzionalità comporta uno sforzo minimo, grazie soprattutto alla scelta di utilizzare *framework* a componenti come *JavaServer Faces* 2 e *PrimeFaces*.

Il mondo dei *framework Java* è in continuo aggiornamento quindi è molto importante stare al passo con le nuove tecnologie, perché l'eventuale utilizzo di *framework* ormai obsoleti, o magari proprio il mancato utilizzo di un *framework*, avrebbe comportato sicuramente un allungamento dei tempi di sviluppo del sistema, legati ad una maggior complessità delle soluzioni da implementare, ma anche una maggior

difficoltà nel portare avanti tutte le modifiche previste in futuro.

L'approccio utilizzato per lo sviluppo del nuovo sistema gestionale è inoltre orientato al *web*, visto che come detto precedentemente, la possibilità di usufruire dei servizi via *internet* è un punto fondamentale su cui l'azienda ha deciso di investire. La struttura della nuova applicazione infatti permette indifferentemente l'accesso sia da rete locale che dalla rete *internet*, quindi non appena si intende usufruire delle funzioni offerte via *web* è sufficiente predisporre semplicemente l'infrastruttura necessaria per far girare l'applicazione sul *web* invece che su un *application server* in funzione su una macchina della rete locale.

Un possibile sviluppo futuro previsto per l'applicativo è l'introduzione dell'utilizzo dei *Web Service* per esportare alcune funzionalità che possano essere utilizzate da sistemi differenti.

Una caratteristica fondamentale di un *Web Service* è infatti quella di offrire un'interfaccia *software* (descritta ad esempio tramite il "*Web Services Description Language*") utilizzando la quale altri sistemi possono interagire con il *Web Service* stesso attivando le operazioni descritte nell'interfaccia tramite appositi "messaggi" trasportati tramite il protocollo *HTTP* e formattati secondo lo *standard XML*. Un'architettura basata sui *Web Service* è chiamata "*Service Oriented Architecture*" (*SOA*).

L'infrastruttura *software* utilizzata per la gestione di *SOA* complesse è la *Enterprise Service Bus* (*ESB*). Un'*ESB* fornisce strumenti di supporto per la realizzazione e la gestione delle *SOA* come ad esempio il supporto alla sicurezza o la messaggistica.

Recentemente la *Sun* ha proposto una versione di *Glassfish* con tali funzioni, chiamata appunto *Glassfish ESB*.

Appendice A

As/400

Il sistema *AS/400* (*Application System*/400) è un *server* sviluppato dall'*IBM* come supporto del sistema informativo gestionale nel giugno 1988 e ancora oggi in produzione. Il suo successo è stato determinato dal fatto di avere un costo relativamente limitato, più di 2500 applicazioni *software* disponibili ed una grande stabilità sia in termini di sistema operativo che di *hardware*.

L'architettura del sistema può essere rappresentata col classico modello a strati tipico dei computer:

- al livello più basso troviamo l'*hardware*. Il sistema *AS/400* utilizza processori *Power*, mentre in precedenza usava processori *RISC* (dopo il 1998) e processori *CISC*;
- al livello immediatamente superiore, si trova uno strato *software* chiamato *Machine Interface*, che collega l'*hardware* al vero e proprio Sistema Operativo. La *Machine Interface* ha lo scopo di permettere al produttore l'aggiornamento dell'*hardware* senza per questo dover modificare il Sistema operativo;
- al livello superiore troviamo il sistema operativo chiamato in origine *os/400*, ed attualmente (dal 2004) *i5/OS*;
- al di sopra del Sistema Operativo ci sono i cosiddetti "Prodotti programma" forniti da *IBM*, ovvero tutte le utilità e gli strumenti per la gestione e l'utilizzo del sistema, ad esempio i compilatori dei linguaggi implementati (*RPG/RPG ILE*, *C*, *C++*, *Fortran*, *Pascal*, *Java*, etc.), gli strumenti per la programmazione (*SDA*, *SEU*, *RLU*, *PDM*), il *database* integrato nel sistema operativo (caratteristica unica di questa macchina), gli strumenti per la gestione dei dati (*DFU*, *SQL*, *QUERY / QUERY MANAGER*) ed altro;

- ancora sopra c'è lo strato finale, quello dei programmi applicativi

Oggi un sistema *AS/400* è in grado di fare tutto quello che può essere fatto su un comune *server web*: produrre stampe di qualità, interfacciamento ai *web service*, integrazione con le applicazioni *legacy*, utilizzo di portali o prodotti *open-source* ecc.

Appendice B

Configurazione di *Glassfish*

B.1 Gestione dell'accesso al *database*

Per avviare l'applicazione *web* con l'*application server* è necessario effettuare alcuni passi:

La prima cosa da fare è inserire le librerie per il collegamento all'*As/400* su cui sono memorizzati i dati, quindi occorre inserire la libreria "*jt400.jar*" nella sottocartella "*lib*" dell'installazione di *Glassfish* utilizzata.

A questo punto, dopo aver avviato il *server*, è necessario accedere alla console di amministrazione di *Glassfish* per effettuare la configurazione. Dal menù accedere a "*JDBC -> Connection Pools*" e cliccare sul pulsante "*New*". Viene aperta una schermata in cui vanno inseriti i seguenti dati per la creazione del *connection pool*:

- *JNDI name*: *Maroil AS-400*
- *Resource Type*: *javax.sql.DataSource*
- *Datasource classname*: *com.ibm.as400.access.AS400JDBCDataSource*

Cliccare poi sul *tab Additional Properties* inserendo le seguenti proprietà:

- *databaseName*: *tst60dat*
- *driverClass*: *com.ibm.as400.access.AS400JDBCDriver*
- *URL: jdbc:as400://192.168.0.1/tst60dat*[1]
- *serverName*: 192.168.0.1

[1]L'indirizzo indicato è quello in cui è in ascolto l'*As/400* nella *VPN* utilizzata per l'accesso ai suoi dati

- *user*: prova
- *password*: prova

A questo punto occorre accedere dal menù a "*JDBC -> JDBC Resources*" cliccando sul pulsante "*New*", inserendo un *JNDI Name* (es.*Maroil-AS400*) e selezionando il *connection pool* precedentemente creato dal menù a tendina.

Per la connessione in locale è necessario seguire lo stesso procedimento inserendo ovviamente i dati del *database* locale che si intende utilizzare e non dimenticando di inserire tra le librerie dell'*application server* la libreria del *database* utilizzato.

B.2 Gestione degli accessi

Per la configurazione del *Realm* di *Glassfish* è necessario innanzitutto accedere alla console di amministrazione dell'*application server*, selezionando dal menù principale la voce "*Security -> Realms -> New*" ed inserendo i dati inseriti in figura B.1 in cui è illustrata la configurazione relativa alle tabelle utilizzate.

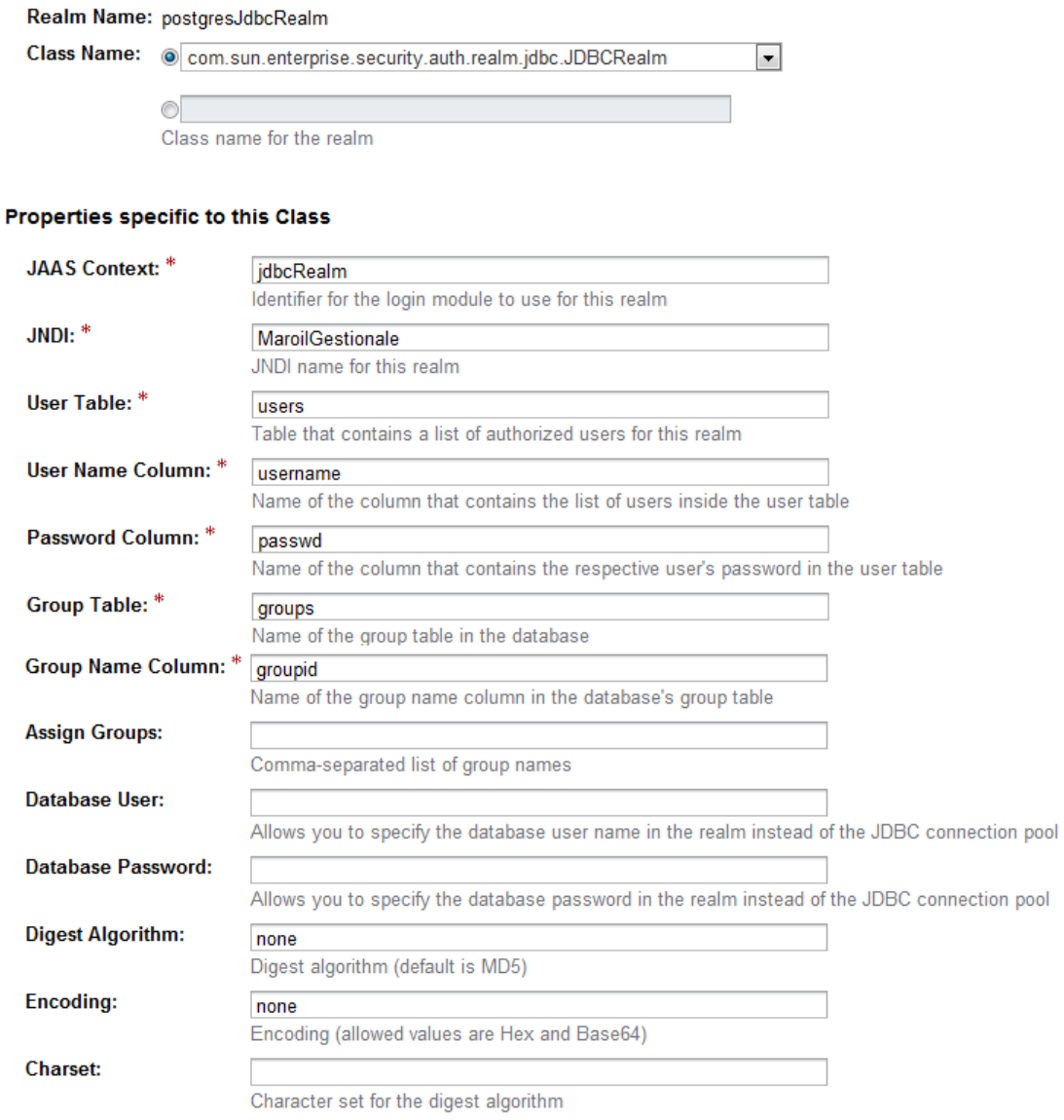


Figura B.1: Configurazione Realm in Glassfish

Bibliografia

[1] Singh, Stearns, Johnson & The Enterprise Team, << Designing Enterprise Applications with the J2EE(TM) Platform (2nd Edition) >>, 2002

[2] http://struts.apache.org

[3] Ian Roughley, <<Pratical Apache Struts 2 Web 2.0 Projects>>, Apress 2007

[4] http://www.springframework.org

[5] Seth Ladd with Darren Davison, Steven Devijver, and Colin Yates, <<Expert Spring MVC and Web Flow>>, Apress 2006

[6] https://javaserverfaces.dev.java.net

[7] Ed Burns, Chris Schalk, <<JavaServer Faces 2.0 - The Complete Reference>>, McGraw Hill 2010

[8] Kent Ka Iok Tong, <<Beginning JSF 2 APIs and JBoss Seam>>, Apress 2009

[9] http://wicket.apache.org

[10] Kent Tong, <<Enjoying Web Development With Wicket>>, TipTec Development 2007

[11] http://vaadin.com

[12] Marko Grönroos, Book of Vaadin

[13] http://www.primefaces.org

[14] PrimeFaces User's guide

Ringraziamenti

Desidero ringraziare innanzitutto i miei relatori per il supporto nella realizzazione di questo lavoro. In particolare la professoressa Bernardeschi che ha sempre mostrato la sua cortesia e disponibilità non solo durante lo svolgimento di questa tesi, ma anche durante le lezioni e gli esami, e l'ingegner Barbagli, non solo per i preziosi consigli per la realizzazione della tesi, ma anche per l'entusiasmo e la voglia che mi ha trasmesso in questi mesi. Vorrei ringraziare inoltre Carlo Soldani di Shinteck e Iacopo Pecchi di Sintra Consulting per i consigli ed il supporto tecnico che mi hanno fornito, e la Maroil Bardhal Italia, per le informazioni che mi hanno permesso di inserire in tesi.

Vorrei ringraziare soprattutto i miei genitori per tutto ciò che hanno fatto per farmi raggiungere questo importante traguardo. Anche nei momenti più difficili non mi hanno fatto mai mancare il loro appoggio e non mi hanno fatto mai pesare nulla, dimostrandomi la totale fiducia che hanno in me e che mi ha permesso di andare avanti fino al raggiungimento di questo obiettivo.

Desidero poi ringraziare Karina, per l'amore che mi ha in questi ultimi anni e la pazienza con cui ha atteso il raggiungimento di questo mio traguardo, Veronica per avermi sempre fatto da guida con affetto e non esser stata mai avara di consigli così come Maurizio, Matilde per lo slancio di entusiasmo che mi ha messo addosso, e Vittorio e Ida per l'affetto con cui mi hanno accompagnato in questo cammino.

Vorrei poi ringraziare i miei compagni di casa a Pisa, in particolare Pietropaolo e Andrea, e il mio ormai inossidabile compagno di studi Donatello.