Analisi e sviluppo di una web application basata sui framework JPA, JSF, CDI, per l'amministrazione di attività di trasferimento tecnologico

Analysis and development of a web application based on the JPA, JSF, and CDI frameworks for administration of technology transfer activities

Tommaso Levato, Alessio Sarullo, Giulio Galvan

13 novembre 2013

Sommario

Indice

Capitolo 1

Introduzione

Capitolo 2

Analisi

2.1 Modello

Business

In figura 2.1 è mostrato il modello di business.

L'Entità principale del modello è `Contract`, è una classe astratta che viene implementata da `Agreement` e `Funding`. `Agreement`, che rappresenta una Convenzione, e `Funding` che rappresenta il Contributo differiscono fra loro solo per i campi dedicati all'Iva. La classe `Contract` possiede tutti gli attributi comuni alle sue sottoclassi come `wholeTaxableAmount, approvalDate, deadlineDate`, etc.

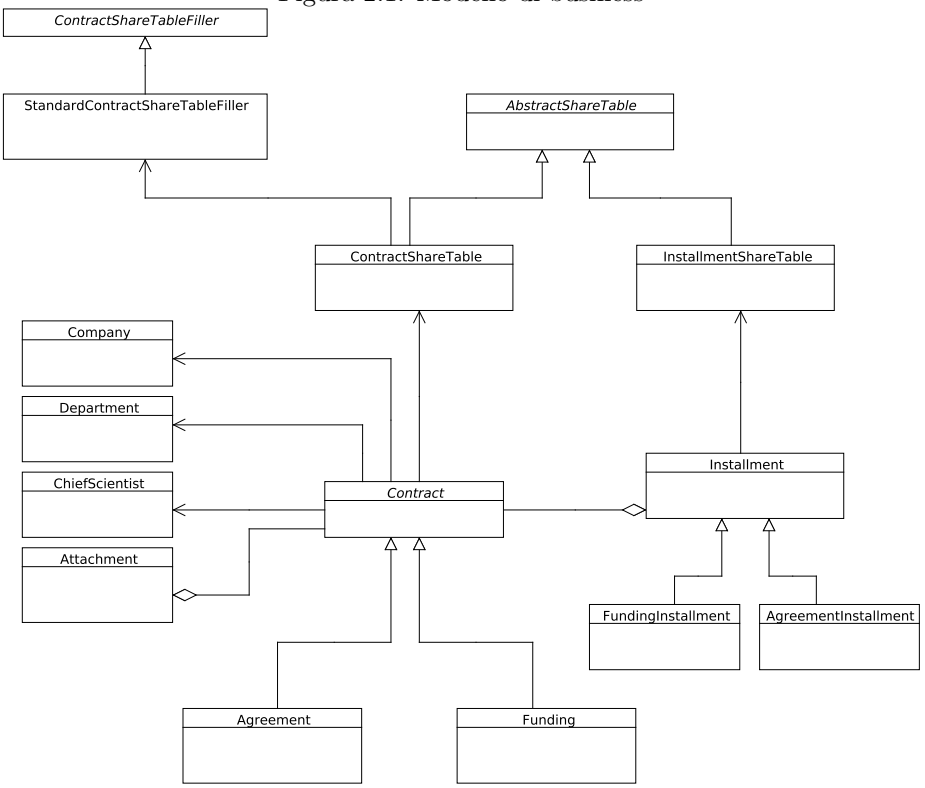
`Contract` ha inoltre un insieme di `Installment`, un insieme di `Attachments`, un `ChiefScientist` ,un `Company` e infine una `ContractShareTable`

`ChiefScientist` rappresenta il Responsabile Scientifico, ogni convenzione/-contributo ha un proprio responsabile. `Company` rappresenta una ditta, la convenzione/contributo ha un riferimento alla dittà con cui ha stipulato l'accordo. `Attachemnt` è la classe che rappresenta un allegato, la convenzione/contributo possiede un insieme di allegati. `ContractShareTable` invece rappresenta la tabella di ripartizione di una convenzione/contributo: i suoi campi indicano come viene ripartito l'importo totale della convenzione/contributo fra il personale, l'ateno, etc. La tabella di ripartizione ha inoltre un riferimento alla classe `StandardContractShareTabelFiller` implementazione concreta di `ContractShareTableFiller` che rappresenta una strategia per il riempimento della tabella in base a normative definite.`Installment` è la classe che rappresenta le rate, una convenzione/contributo ha un insieme di rate; `Installment` ha due implementazioni concrete: `AgreementInstallment` e `FundingInstallment` rispettivamente per convenzioni e contributi. `Installment` ha inoltre un riferimento a `InstallmentShareTable`. `InstallmentShareTable` e `ContractShareTable` estendono la stessa classe base `AbstractShareTable`.

Modello degli utenti

In figura 2.2 è mostrato il modello degli utenti.
La classe centrale di tale modello è chiaramente `User`, che rappresenta un gener-

Figura 2.1: Modello di business



ico utente dell'applicazione. I tipi di utente sono rappresentati dalla enum `Role`, che può avere quattro valori:

- `ADMIN` (*Amministratore*). Rappresenta, appunto, l'amministratore del sistema. È l'unico che può aggiungere altri utenti di tipo Operatore o Docente. Non è afferente ad alcun dipartimento.
- `OPERATOR` (*Operatore*). Un Operatore gestisce le convenzioni del dipartimento a cui afferisce.
- `TEACHER` *Docente*. Il Docente può consultare le proprie convenzioni e, se richiesto, allegare documenti alle stesse.
- `GUEST`. Rappresenta l'utente non ancora loggato. Non possiede alcun permesso.

Le operazioni che un utente può svolgere sono regolate tramite un sistema di permessi, i quali sono rappresentati dalla enum `Permission`: ad ogni `Role` è associata una lista di `Permission`. Per maggiori dettagli riguardo la sicurezza, si rimanda al capitolo INSERIRE RIFERIMENTO A CAPITOLO DELTASPIKE.

Un utente ha quindi un attributo che ne identifica il ruolo. Vi sono poi degli attributi che lo caratterizzano (come ad esempio l'indirizzo e-mail) ed infine un

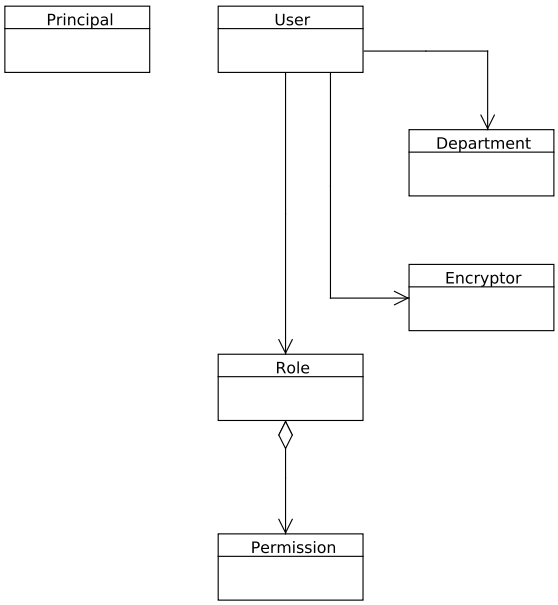


Figura 2.2: Modello degli utenti

attributo di tipo `Encryptor`.
Questa classe racchiude metodi per gestire le diverse codifiche delle password, in modo da poter utilizzare stringhe codificate in maniera differente e rendere indipendente la codifica utilizzata dall'applicazione con quella che adopera il servizio esterno che fornisce le credenziali di Ateneo. Attualmente sono supportate due tipi di codifica: MD5 (la più utilizzata da LDAP) e SHA (adoperata anch'essa da LDAP, ma in misura minore; inoltre rappresenta la codifica di default dell'applicazione).

Parallelamente alla classe `User` vi è la classe `Principal`, che rappresenta un client dell'applicazione. La differenza è sottile, perché rappresentano due aspetti diversi dell'utente: la prima serve per modellare l'utente all'interno dell'applicazione, la seconda serve per gestire la navigazione dell'utente. La classe `Principal` ha quindi due caratteristiche principali:

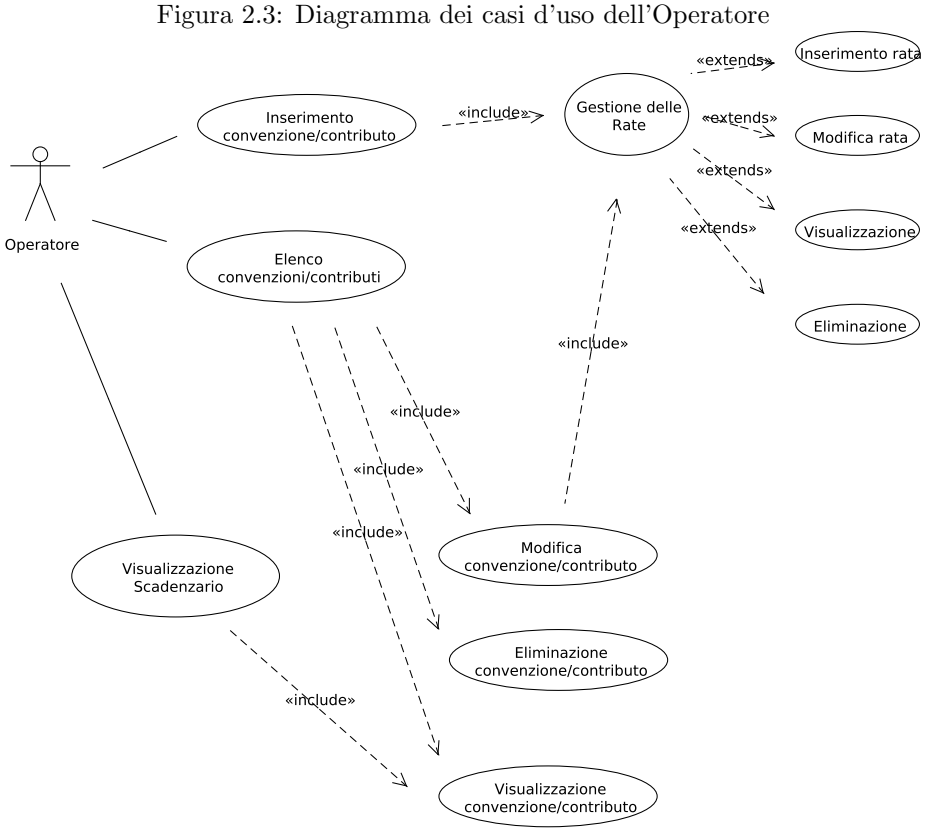
- è disaccoppiata dall'implementazione del modello: gli attributi del `Principal` sono di tipo stringa e ciò consente di cambiare l'implementazione sottostante indipendentemente dal `Principal`
- è sicura, perché non contiene proprietà sensibili dell'utente (come ad esempio la password).

2.2 Casi d'uso

Di seguito elenchiamo i principali casi d'uso per ciascun tipo di utente che interagisce col sistema.

Operatore Una rappresentazione grafica dei casi d'uso dell'Operatore è disponibile in figura 2.3

Figura 2.3: Diagramma dei casi d'uso dell'Operatore



1. Inserimento di una nuova Convenzione/Contributo

   Percorso base: l'operatore, una volta effettuato il login, clicca su "Crea una convenzione/contributo"; viene visualizzata una schermata suddivisa in varie schede, ognuna corrispondente ad un passo della procedura. E' possibile passare da una vista all'altra mediante i pulsanti "Avanti" e "Indietro". I passi sono:

   (a) Inserimento dei dati della convenzione/contributo

   In questa scheda sono elencati tutti i campi necessari per la definizione di una convenzione/contributo, che l'Operatore deve compilare. Tali campi sono:

   - Il titolo
   - Il numero di protocollo
   - L'UAR
   - La tipologia

- Il responsabile scientifico
  Per selezionare un responsabile scientifico è possibile usare l'apposito menù a tendina o, in alternativa, qualora la persona cercata non sia nell'elenco, aggiungerla cliccando sul pulsante "Aggiungi".
- Il referente
- La dittà
  Per selezionare una dittà si può usare l'apposito menù a tendina o, se la dittà cercata non fosse presente nell'elenco, aggiungerne una nuova cliccando sul pulsante "Aggiungi".
- Il nome del progetto CIA
- Il Repertorio
- Il totale imponibile
- L'Iva
- La data di approvazione
- La data di inizio
- La data di scadenza

Nota : i campi riguardanti l'Iva non sono presenti nel caso del contributo.

(b) Inserimento della tabella di ripartizione

Questa scheda contiene le voci della tabella di ripartizione. L'Operatore può modificare alcuni valori percentuali in base ai quali dividere l'importo totale. Le voci non modificabili sono calcolate in relazione ai campi modificabili facendo riferimento alle norme di ateneo. Le voci modificabili sono:

- Personale: stabilisce la quota destinata al personale; è l'unico campo principale che l'operatore può modificare e in base al quale vengono calcolati gli altri.
- Missioni, Materiale di consumo, etc. sono sottocampi di Beni e Servizi e servono per meglio specificare come verrà ripartita la quota destinata a "Beni e Servizi".

(c) Gestione delle rate

Questo passo della procedura è facoltativo: viene data la possibilità all'operatore di inserire delle rate per la convenzione/contributo che sta creando. Una volta inserite una o più rate queste vengono visualizzate in una tabella e l'Operatore ha la possibilità di modificare, visualizzare, eliminare una rata cliccando sui tasti "Modifica","Visualizza" ed "Elimina" che compaiono sulla destra, nella riga della tabella corrispondente alla rata in questione. Per i dettagli riquardo alle operazioni sulle rate si rimanda ai corrispettivi casi d'uso.

(d) Inserimento della documentazione relativa alla convenzione

Questa scheda elenca i documenti allegati alla convenzione. L'Operatore può aggiungere o eliminare un documento cliccando sugli appositi tasti. Premendo il tasto "Salva" la convenzione viene salvata e la procedura termina. Si ritorna alla schermata precedente.

Percorso alternativo: Durante uno qualsiasi dei passi, l'Operatore può cliccare il tasto "Annulla", che comporta, a seguito di una conferma, il ritorno alla schermata precedente senza che la convenzione/contributo venga inserita o i cambiamenti effettuati salvati. Se l'Operatore clicca sul tasto "Salva" senza aver compilato dei campi obbligatori, o avendo inserito dei valori non consentiti, viene visualizzato un messaggio di errore e il documento non viene salvato. La schermata non viene cambiata, in modo che l'Operatore possa procedere alla correzione.

2. Visualizzazione delle convenzioni/contributi

   Percorso base: l'operatore clicca su "Visualizza contratti"; viene mostrata una lista delle convenzioni/contributi correntemente stipulate in riferimento al dipartimento di afferenza dell'Operatore, con opportuni filtri per agevolare la ricerca (tra cui un filtro per data di scadenza) e ordinabili secondo la data. L'Operatore può selezionare una convenzione e compiere 3 azioni, per cui si rimanda ai corrispettivi casi d'uso, mediante gli appositi pulsanti, che appaiono sulla destra portando il cursore su una convenzione:

   - visualizzarla
   - modificarla
   - eliminarla

   Percorso alternativo: Si può tornare alla pagina iniziale con l'apposito tasto.

3. Modifica di una convenzione

   Percorso base: l'Operatore a partire dalla schermata "Visualizza Contratti" clicca sul pulsante "Modifica" che appare sulla destra nella riga della tabella corrispondente alla convenzione/contributo desiderata. Viene visualizzata una schermata suddivisa analoga a quella descritta nel caso della "Creazione di una convenzione/contributo" con la differenza che in questo caso è possibile spostarsi da una scheda all'altra cliccando sulla scheda stessa. Inoltre è presente la scheda aggiuntiva "Riepilogo" che contiene alcune informazioni di riepilogo come il residuo totale della convenzione/contributo o il totale fatturato. L'Operatore effettua i cambiamenti desiderati quindi clicca sul pulsante "Salva".

   Percorso alternativo: l'Operatore può cliccare sul tasto Annulla in qualsiasi momento per tornare alla schermata Visualizzazione delle convenzioni senza salvare le modifiche effettuate. Se l'Operatore clicca su Salva ma alcuni valori immessi non sono corretti, viene visualizzato un messaggio di errore e non si torna alla schermata Visualizzazione delle convenzioni. I cambiamenti effettuati non vengono (ovviamente) salvati.

4. Visualizzazione di una convenzione/contributo

   Del tutto analogo a “Modifica di una convenzione/contributo” con la differenza che in questo caso non è possibile modificare i dati della convenzione/contributo.

5. Inserimento di una rata
   Percorso base: l’operatore può inserire una rata sia in fase di creazione della convenzione/contributo sia in fase di modifica; in entrambi i casi dopo aver raggiunto la scheda “Rate” l’Operatore clicca sul pulsante “Aggiungi una rata”. Viene visualizzata una finestra di dialogo suddivisa in varie schede, ognuna corrispondente ad un passo della procedura. E’ possibile passare da una scheda all’altra mediante i pulsanti Avanti e Indietro. I passi sono:

   (a) Inserimento dei dati della rata

   L’operatore inserisce i seguenti campi

   - Importo
   - Iva
   - Data
   - Numero Reversale
   - Data Reversale
   - Numero di Sospeso
   - Numero di fatturato
   - Data fatturata
   - E’stata pagata la fattura?
   - Deve essere allegata la fattura?
   - Note

   Nota : i campi riguardanti l’Iva non sono presenti nel caso del contributo.

   (b) Inserimento della tabella di ripartizione

   Il procedimento è del tutto analogo a quello descritto nel caso d’uso Inserimento della tabella di ripartizione per l’inserimento di una nuova convenzione/contributo.

   Le modifiche vengono salvate cliccando sul pulsante Salva. Si ritorna alla schermata precedente;

   Percorso alternativo: l’Operatore clicca sul pulsante “Annulla”, viene chiusa la finestra di dialogo senza che la rata sia stata inserita.

6. Modifica di una rata

   Percorso base: l’Operatore accede alla schermata “Visualizza contratti” cliccando sul pulsante apposito nella pagina iniziale, quindi seleziona la convenzione/contributo alla quale la rata appartiene e clicca sul pulsante

"Modifica" che compare all'interno della riga selezionata. Viene così visualizzata la schermata "Modifica di una convenzione/contributo"; l'Operatore raggiunge la scheda "rate" e clicca sul pulsante "Modifica" che compare selezionando la riga della tabella corrispondente alla rata desiderata. Viene visualizzata una finestra di dialogo composta di varie schede analoghe a quelle descritte nel caso dell'inserimento. E' possibile passare da una scheda all'altra cliccando su di esse. Dopo avere effettuato le modifiche richieste l'Operatore clicca sul pulsante "Salva", la convenzione/contributo viene aggiornata e viene visualizzata la schermata precedente.

Percorso alternativo: l'Operatore clicca sul pulsante "Annulla", le modifiche vengono scartate e si torna alla schermata precedente.

7. Visualizzazione di una rata

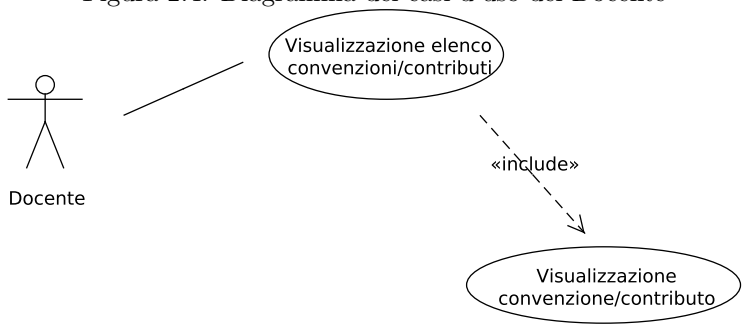
   Una volta raggiunta la scheda "rate" relativa alla convenzione/contributo di interesse l'Operatore, dopo aver selezionato la rata desiderata, clicca sul pulsante "Visualizza". Viene presentata una finestra di dialogo analoga a quella descritta nel caso della modifica con la differenza che i campi non sono modificabili. E' possibile tornare alla schermata precedente cliccando sul pulsante "Indietro".

8. Eliminazione di una rata

   Una volta raggiunta la scheda "rate" relativa alla convenzione/contributo di interesse l'Operatore, dopo aver selezionato la rata di interesse, clicca sul pulsante "Elimina'; appare una finestra di dialogo che chiede di confermare l'eliminazione, l'Operatore clicca "Sì", la rata viene eliminata.

Docente I casi d'uso del Docente sono rappresentati in figura 2.4

Figura 2.4: Diagramma dei casi d'uso del Docente



1. Visualizzazione dell'elenco delle convenzioni/contributi

   Il docente, dopo aver effettuato il login, può cliccare sul pulsante "Visualizzazione della lista delle convenzioni/contributi"; la schermata che viene visualizzata contiene una tabella che elenca le convenzioni/contributi del

docente. E' possibile filtrare le convenzioni/contributi secondo vari criteri(data, tipo, scadenze più vicine, ...).Inoltre è possibile visualizzare i dettagli di una convenzione/contributo cliccando sul pulsante "Visualizza" che appare posizionando il puntatore su una riga della tabella.

2. Visualizzazione di una convenzione/contributo di cui il docente è responsabile scientifico

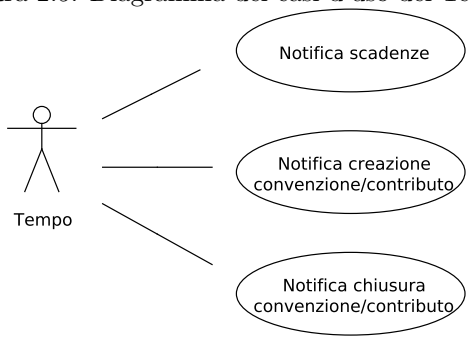
   Il docente dalla schermata "Lista delle convenzioni/contributi" può cliccare sul pulsante "Visualizza" relativo ad una convenzione/contributo; compare una schermata suddivisa in schede analoga a quella della modifica/creazione della convenzione. Il docente può navigare fra le schede cliccandoci sopra. Non è permessa nessuna modifica ai dati della convenzione/contributo tuttavia il docente può inserire degli allegati dalla scheda "Allegati". Cliccando su "Salva" gli allegati inseriti dal docente vengono memorizzati, al contrario cliccando su "Indietro" le modifiche vegono scartate.

Amministratore

1. Inserimento di un nuovo utente
2. Visualizzazione della lista degli utenti

Tempo I casi d'uso del Tempo sono rappresentati in figura 2.4

Figura 2.5: Diagramma dei casi d'uso del Tempo



1. Notifica delle scadenze

   Ad intervalli periodici stabiliti, i docenti che hanno convenzioni/contributi attive con rate in scadenza ravvicinata, vengono avvertiti tramite posta elettronica.

2. Notifica della creazione di una nuova convenzione/contributo

   Al momento del completamento della creazione di una nuova convenzione/contributo viene inviata una email all'indirizzo di posta elettronico del responsabile scientifico indicato per la convenzione/contributo.

Tale email contiene le informazioni principali che caratterizzano la convenzione/contributo.

3. Notifica della chiusura di una convenzione/contributo

   Al momento della chiusura di una convenzione/contributo (ovvero quando il fatturato è pari all'importo totale) viene inviata una email all'indirizzo di posta elettronico del responsabile scientifico che notifica la chiusura della convenzione/contributo riportando alcuni dati di questa.

Capitolo 3

Tecnologie

3.1 JPA

3.1.1 Introduzione

Le applicazioni di tipo Enterprise necessitano di raccogliere e persistere grandi quantità di informazioni, la soluzione che più si è affermata per risolvere questo problema è il database relazionale. Nell'ambiente Enterprise , in particolare per la piattaforma Java, si è sentita l'esigenza di soluzioni che permettessero l'integrazione della piattaforma Java con database di tipo relazionale, in modo naturale.

3.1.2 Mapping fra modello Relazione e a Oggetti

In un modello a oggetti,come il modello di dominio di un applicativo Java, le entità che popolano il modello sono Classi. Le relazioni fra le classi del modello sono espresse tramite riferimenti, gli attributi di una classe. Nel modello relazionale invece le entità sono chiamate Relazioni e sono collegate fra loro mediante il concetto di chiave. Effettuare un mapping fra i due modelli significa quindi avere un modo per trasferire i concetti da un modello all'altro in modo da ridurre le distanze fra i due modelli. In particolare una soluzione per il mapping dovrebbe avere le seguenti caratteristiche:

- le applicazioni dovrebbero essere scritte e pensate secondo il modello di dominio e senza avere legami col modello relazione del database; dovrebbe essere possibile recuperare informazioni dal database senza dover scrivere espressioni che coinvolgano tabelle o chiavi primarie tipiche di un database.
- la soluzione dovrebbe essere non intrusiva: sebbene sia impensabile di realizzare una soluzione che renda la persistenza completamente trasparente, si può richiedere che la persistenza non "invada" il modello di domionio. In concreto le classi del dominio non devono essere obbligate ad implementare interfacce o estendere classi per poter essere rese persistenti.
- dovrebbe essere possibile integrarsi con database preesistenti

3.1.3 JPA

Nel tempo sono state sviluppate e proposte varie soluzioni, a partire da ODBC passando per JDBC e EJB fino ad arrivare a *JPA*. JPA (Java Persistence API) è una specifica per il mapping fra modello a oggetti e modello relazionale per applicazioni Java che risolvendo i problemi degli standard precedenti raggiunge gli obiettivi sopra formulati. Esistono vari prodotti che implementano JPA fra cui *Hibernate* che è stato scelto come implementazione di riferimento.

Funzionamento

Entity Un' unità che possegga una stato e che possa essere persistita viene detta Entità. Le classi Java possono essere facilmente trasformate in entità semplicemente annotando la classe e alcuni dei suoi attributi, o alternativamente fornendo dei descrittori xml. L'unico requisito che la classe deve rispettare è che deve possedere un costruttore senza parametri: questo serve affinchè Hibernate (o qualsiasi altro provider) possa ricreare l'oggetto una volta interrogato il database. Come si può notare, non è possibile rendere persistibile una classe in modo del tutto trasparente, ma di sicuro si può affermare che questo avvenga in modo non intrusivo. Allo stesso modo, sempre usando delle annotazioni o dei file di descrizione xml, è possibile mappare le relazioni che sussistono fra le entità del modello.

Entity Manager e Peristence Context Le Entità sono gestite da un Entity Manager. Un entity Manager è in grado di persistere un entità nonchè di eliminarla o recuperarla dal database. Ogni Entity Manager è associato ad un Persistence Context. Un Persistence Context è un insieme di istanze di entità. Una entità si dice "managed" se è contenuta in un Persistence Context. Se un Persistence Context, tramite un Entity Manager, partecipa ad una transazione lo stato delle entità "managed" contenuto in memoria centrale viene salvato sul database. Le entità non "managed" sono chiamate "detached" e il loro stato in memoria non viene sincronizzato col database in nessuna transazione. Come è possibile ottenere un Entity Manager? Ci sono vari tipi di Entity Manager ognuno dei quali legato a diverse esigenze applicative. Si è scelto di includere nella presente trattazione solo una sottocategoria: le Entity Manager di tipo "Container-Managed".Questa scelta è dovuta al fatto che le Entity Manager "Container-Managed" sono quelle che sono state usate nella realizzazione dell'applicativo nonchè le tipologie preferite per l'ambiente Java EE. Un' Entity Manager di questo tipo è ottenuta dal container mediante iniezione di dipendenza. Le Entity Manager di tipo "Container-Managed" sono di due tipi:

- Transaction Scoped
  Una Entity Manager di questo tipo è *stateless*, ovvero lavora con un Persistence Context che viene costruito ogni volta che comincia una transazione (JTA) e termina il proprio ciclo di vita al termine della transazione
- Extended
  Una Entity Manager di tipo Extended è usata in coppia con un bean(vedi 3.2) di tipo *statefull*. In questo caso l'Entity Manager lavora con un singolo

Persistence Context il cui ciclo di vita è legato a quello del bean che potenzialmente sopravvive a più di una transazione.

L'uso del corretto tipo di Entity Manager dipende dal contesto: per esempio sarà conveniente usare un'Entity Manager di tipo transaction-scoped nel caso dell'eliminazioni di un entità dal database mentre è forse più conveniente usare un'Entity Manager di tipo extended per recuperare dal database un oggetto il cui stato viene modificato più volte e salvato solo alla fine della sessione corrente.

Query Il linguaggio in cui sono espresse le query è chiamato JPQL. Questo linguaggio ha due caratteristiche principali che vanno nella direzione indicata negli obiettivi presentati:

- Il linguaggio è indipendente dal database sottostante. Questo significa che l'applicazione non è dipendente dal particolare database usato ma al contrario è possibile con facilità migrare da una soluzione all'altra senza dover cambiare il codice.
- Sebbene JPQL sia un linguaggio dichiarativo che rassomiglia molto da vicono SQL non usa tabelle e colonne per esprimere i propri criteri di ricerca ma usa le entità del modello di dominio e i loro attributi.

3.2 CDI

3.2.1 Introduzione

La specifica *Contexts and Dependecy Injection* (o, più semplicemente, *CDI*) è stata introdotta nella versione 6 di Java EE per unificare il livello applicazione (che fa uso degli *Enterprise Java Bean*, o *EJB*) e il livello web (con particolare riferimento alla tecnologia JSF, che fa uso dei *Managed bean*).
Esistono attualmente varie versioni di CDI. Nello sviluppo dell'applicazione è stata usata *JBoss Weld*, che è quella di riferimento ed è inoltre inclusa di default in *JBoss AS*.
In seguito, verranno descritti in modo sufficientemente approfondito le caratteristiche principali di CDI. Per una trattazione più dettagliata e completa, si rimanda alla specifica INSERIRE RIFERIMENTO A SPECIFICA e alla documentazione dell'implementazione di riferimento INSERIRE RIFERIMENTO A WELD.

3.2.2 Caratteristiche

Come dice il nome stesso, i principali servizi offerti da CDI sono due:

- **Contesto**: CDI è in grado di conferire agli oggetti Java un ciclo di vita legato ad un particolare contesto.
- **Iniezione di dipendenza**: tramite CDI è possibile delegare al framework il compito di istanziare gli oggetti, creando così un meccanismo di iniezione di dipendenza. Tale operazione può essere eseguita sia durante lo sviluppo che durante la fase di *deploy* dell'applicazione. Questi oggetti gestiti dal framework invece che direttamente dal programmatore vengono denominati *bean*.

Inoltre CDI:

- consente l'integrazione con l'*Expression Language* (o *EL*), permettendo di far riferimento agli oggetti dalle pagine JSF in modo diretto.
- favorisce il disaccoppiamento sia al livello client-server, permettendo varie implementazioni lato server attraverso l'utilizzo dei *qualifier*, sia per quanto riguarda il ciclo di vita dei bean, attraverso una sua gestione mediante l'utilizzo di vari contesti.
- permette un forte controllo sui tipi, eliminando la necessità di identificatori di tipo stringa per il collegamento tra i vari bean tramite l'utilizzo delle annotazioni Java.

3.2.3 Bean

Definizione

Come risulta ormai chiaro, alla base di CDI vi sono oggetti particolari chiamati *bean*.
Un bean, nella sua accezione più ampia, è un "componente software riusabile che può essere gestito dal container" (adattamento libero della definizione fornita dalla specifica *JavaBeans*, consultabile all'indirizzo INSERIRE QUI RIFERIMENTO BIBLIOGRAFICO AL LINK PAG. 33 DI CoreJSF).
I bean di CDI rispettano questa specifica, ma hanno un'importante proprietà aggiuntiva: lo *scope*, che li lega ad uno specifico contesto, il quale ne determina il ciclo di vita e la visibilità ai client. CDI mette a disposizione quattro *scope*:

- *Request scope* Il tempo di vita del bean equivale ad una singola richiesta HTTP
- *Conversation scope* Una *conversazione* di CDI può essere di due tipi: *transient* e *long-running*. Normalmente, una conversazione è *transient* e ha la stessa durata di una richiesta HTTP. Tuttavia, una conversazione *transient* può divenire *long-running* tramite una chiamata al metodo `begin()`, e rimane tale finché non viene chiamato il metodo `end()`, riportandola allo stato *transient*.
- *Session scope* È legato alla sessione HTTP. Un bean *session scoped* rimane quindi attivo attraverso più richieste HTTP ed è visibile tra più viste che condividono la stessa sessione.
- *Application scope* Un bean *application scoped* viene creato una volta sola per tutta la durata dell'applicazione.

Oltre agli *scope* 'classici' appena elencati, ve ne sono altri chiamati *pseudo-scope*. Tra questi, il più importante è lo *pseudo-scope dependent*: un bean *dependent* viene istanziato per servire un solo client o bean, ed il suo ciclo di vita è quindi legato a quello del client/bean. Se ad un bean non viene assegnato esplicitamente uno *scope*, viene considerato *dependent*.

Un bean CDI non è solo definito dal suo *scope*: di seguito sono elencati gli altri principali attributi di un bean CDI.

- Un insieme (non vuoto) di *bean type*. Un *bean type* è un tipo che è visibile dal client. A livello pratico, quasi tutti i tipi Java possono essere *bean type*.
- Un insieme (non vuoto) di *qualifier*. I *qualifier* sono annotazioni particolari che definiscono ulteriormente un bean, e sono in genere usati per distinguere tra varie implementazioni di una stessa interfaccia.
- (Opzionale) Un nome EL. Questo nome serve per far riferimento al bean tramite EL, (ad esempio, da una pagina JSF). È possibile specificare il nome desiderato tramite l'annotazione `@Named`; alternativamente, viene usato un nome di default scelto dal frame (di solito, il nome della classe con l'iniziale minuscola).

Iniezione di un bean

Il meccanismo che sta alla base di CDI è l'*iniezione di dipendenza*, che costituisce a una forma di "inversione di controllo"(in inglese, *inversion of control*): non è più il programmatore a controllare gli oggetti da istanziare, bensì il framework (come approfondimento si consiglia la lettura di INSERIRE RIFERIMENTO ARTICOLO MARTIN FOWLER IoC).
Per eseguire l'iniezione di dipendenza in CDI, dichiarando un attributo di una classe e delegando al framework la responsabilità di inizializzarlo, bisogna innanzitutto inserire le classi che definiscono i bean in un archivio(`jar`, `war` etc) che contiene il file `META-INF/beans.xml`. A questo punto, è sufficiente annotare l'attributo con `@Inject`: il *container* all'interno del quale viene eseguita l'applicazione cercherà - nel contesto appropriato - un bean dello stesso tipo dell'attributo dichiarato e provvederà all'inizializzazione.

Qualifier* e *alternatives Nel caso in cui il tipo dell'attributo non sia concreto potrebbero esistere diversi bean che lo implementano e che possono essere iniettati. Per ovviare a questo problema, CDI consente al programmatore di specificare quale bean utilizzare mediante annotazioni personalizzate chiamate *qualifier*. Definire un *qualifier* è molto semplice: è un'annotazione annotata a sua volta con `@Qualifier`. Annotando un bean di implementazione di un'interfaccia con un *qualifier* consente quindi di decidere, durante lo sviluppo dell'applicazione, quale classe concreta adoperare.
Le *alternative* (*alternatives*) consentono invece di effettuare questa scelta durante il *deploy* dell'applicazione. Per dichiarare un bean come una 'alternativa' di un'interfaccia è sufficiente annotarlo con `@Alternative`. È poi possibile scegliere quale alternativa utilizzare modificando il file di configurazione `META-INF/beans.xml`.

Metodi *producer* Un metodo *producer* è un metodo che genera un oggetto che può essere iniettato. Per definire un metodo *producer* è sufficiente annotarlo con `@Producer`. L'oggetto così prodotto è a tutti gli effetti un bean CDI, ed è pertanto possibile caratterizzarlo con le annotazioni menzionate in precedenza, come ad esempio `@Named` o i *qualifier*.
I metodo *producer* hanno due importanti caratteristiche:

1. rendono possibile stabilire a runtime l'implementazione di un *bean type*, a differenza dei *qualifier* o delle alternative

2. consentono di trattare come bean CDI qualunque classe Java; ad esempio, consentono di utilizzare l'iniezione di dipendenza con le *entità* di JPA.

Tipi di bean

Come già detto, la definizione generale di bean è piuttosto ampia e per questo molte specifiche ne adottano una propria. CDI non supporta soltanto i bean che rispecchiano le caratteristiche precedentemente elencate, ma anche quelli definiti da altre specifiche. I principali sono due: i *managed bean* e i *session bean*.

Managed bean Una classe Java non nidificata è un *managed bean* se è definito tale dalla specifica di una delle tecnologie di Java EE (ad esempio, dalla specifica di JSF risulta che una classe è un *managed bean* se è annotata con l'annotazione `@ManagedBean`) oppure se rispecchia le seguenti condizioni:

- È una classe interna (*inner class*) non statica
- È una classe concreta o è annotata con `@Decorator`
- Non è annotata con un'annotazione che definisce un EJB
- Non è dichiarata essere un bean EJB nel file `ejb-jar.xml`
- Ha un costruttore che non riceve parametri o che è annotato con `@Inject`

La semantica e il ciclo di vita di un *managed bean* sono descritti nella rispettiva specifica, consultabile INSERIRE RIFERIMENTO.

Session bean Un *session bean* è un particolare tipo di EJB che incapsula logica di business, nascondendo così ai client dettagli implementativi del servizio offerto; in altre parole, i client si limitano ad invocare i metodi del *session bean*, ignorando cosa accade all'interno del server.
Un *session bean* può essere di tre tipi:

- *stateful* Uno *stateful session bean* rimane in vita fino a quando il client mantiene un riferimento allo stesso, memorizzando lo stato di quella specifica sessione client-bean (per via della natura 'interattiva' di tali bean, talvolta questo stato viene detto *conversational state*). Uno *stateful session bean* non viene condiviso fra più client.

- *stateless* Uno *stateless session bean* non memorizza un *conversational state* con il client, ma mantiene il suo stato solo per la durata dell'invocazione del metodo chiamato dal client.

- *singleton* Un *singleton session bean* esiste per tutto il ciclo di vita dell'applicazione e viene istanziato una sola volta. I *singleton session bean* sono analoghi agli *stateful session bean* nel senso che mantengono il loro stato tra invocazioni del client, ma se ne differenziano perché sono condivisi tra i client, che vi accedono in modo concorrente.

Sebbene i *session bean* possano rispettare le caratteristiche dei *managed bean*, non lo sono, in quanto il loro ciclo di vita è differente da quello descritto nella specifica di questi ultimi.

3.3 JSF

3.3.1 Introduzione

Il framework *JSF* (acronimo per *Java Server Faces*) fa parte delle tecnologie standard della piattaforma *Java EE* e il suo scopo è quello di facilitare lo sviluppo dell'interfaccia utente di un'applicazione web. È un framework *component-based*: consente allo sviluppatore di costruire una pagina web focalizzandosi sugli oggetti che la compongono, piuttosto che sul codice HTML che la genera, permettendo pertanto un livello di astrazione maggiore.
Ne esistono diverse implementazioni; nello sviluppo della applicazione è stata utilizzata quella di riferimento, *Mojarra* (versione 2.1), nonché la libreria *PrimeFaces* (versione 3.5) per estenderne le funzionalità.
Nei prossimi paragrafi si esamineranno un po' più in dettaglio le caratteristiche ed il funzionamento di JSF.

3.3.2 Caratteristiche

Panoramica

La tecnologia JSF è composta essenzialmente da due componenti:

- Un insieme di API che consentono di:
  - rappresentare ed utilizzare i componenti
  - gestire gli eventi
  - effettuare la conversione dei dati
  - eseguire una validazione lato server dei dati
  - definire le regole di navigazione fra le pagine
  - creare applicazioni multi-lingua
- Librerie per la gestione dei tag e la connessione dei componenti ad oggetti Java lato server.

Inoltre, la struttura del framework è tale da consentire l'estensione delle funzionalità e l'aggiunta di nuovi componenti.

Architettura del framework

L'architettura di JSF (illustrata in Figura 3.1) segue il classico schema *Model-View-Controller*: JSF consente di collegare il modello (*model*) con l'interfaccia grafica (*view*), fornendo gli strumenti necessari per poter controllare e processare le azioni degli utenti e aggiornare il modello sottostante di conseguenza (*controller*). Tali strumenti realizzano principalmente tre tipi di servizi:

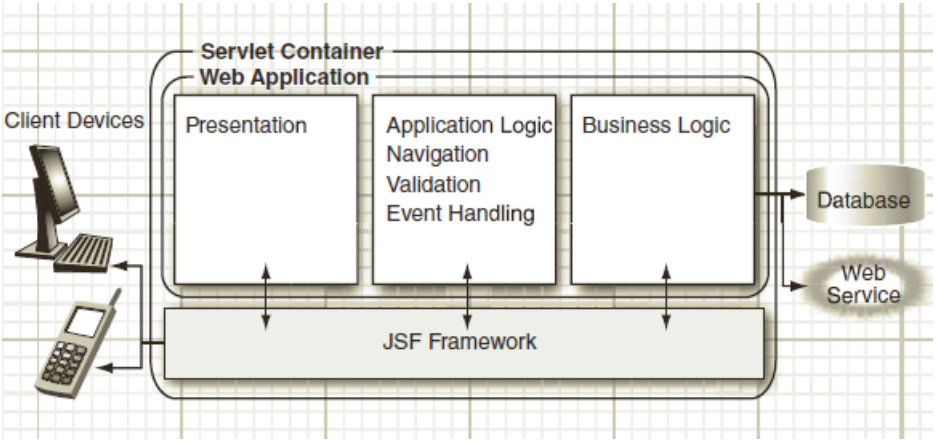


Figura 3.1: Architettura di JSF

- **gestione degli eventi**: interagendo con la pagina web, l'utente può compiere una grande varietà di azioni; esse vengono rilevate dal browser, che 'lancia' il relativo evento. JSF consente di elaborare la risposta del programma tramite un meccanismo molto semplice: è possibile associare ad un dato evento un metodo di un oggetto Java lato server che si occupa della sua gestione.
- **conversione dei dati**: nel web, i dati vengono immessi, visualizzati e scambiati sotto forma di stringa; il modello dei dati, invece, è composto da oggetti Java. JSF permette in modo agevole di effettuare le conversioni necessarie per consentire l'interazione tra questi due mondi: oltre a fornire dei convertitori per i tipi basilari (come ad esempio numeri o stringhe), offre anche la possibilità di definire convertitori personalizzati.
- **validazione dei dati**: viene realizzata in modo simile alla conversione: lo sviluppatore può decidere di utilizzare i *validator* standard o implementarne di nuovi. Tramite questo meccanismo si evita che gli errori dell'utente pregiudichino la validità del modello.

Bean

Il livello *controller* di JSF fa un uso intensivo dei *bean*. Un bean è un oggetto Java gestito dal framework e non in maniera diretta dal programmatore.
Esistono vari tipi di bean; quelli a cui si può accedere direttamente da una pagina JSF sono detti *managed bean*. Per essere tale, un bean deve avere un nome ed uno *scope*, ossia un contesto in cui il bean è 'visibile' ed utilizzabile dall'applicazione.
Per maggiori dettagli, si rimanda al capitolo INSERIRE RIFERIMENTO A CDI QUI.

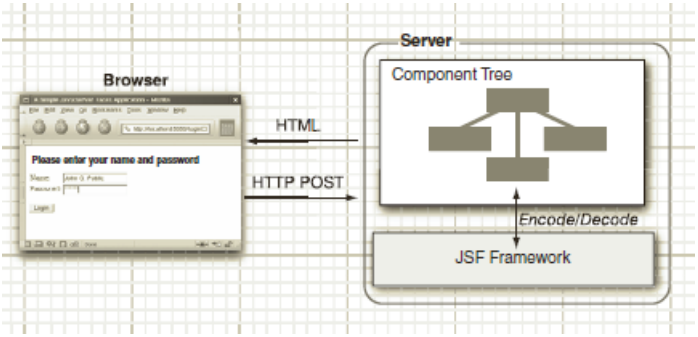


Figura 3.2: *Encoding* e *decoding*

3.3.3 Funzionamento

Rendering delle pagine

Ci sono diversi modi per creare una pagina JSF. Quello standard fa uso della tecnologia *Facelets*, che è stata sviluppata proprio per essere utilizzata da JSF; per creare una pagina Facelets è infatti sufficiente scrivere un documento XHTML che fa uso dei tag di JSF, con la possibilità di utilizzare l'*Expression Language* (*EL*) per consentire la comunicazione tra il livello presentazione e il livello applicazione. Ciascun tag è associato ad una classe che lo gestisce e le istanze di queste classi sono dette *tag handler*: quando la pagina viene letta, i *tag handler* vengono eseguiti e costruiscono un albero dei componenti della pagina (*component tree*). Ad ogni tag corrisponde un nodo dell'albero. Ogni componente è inoltre associato ad un oggetto *renderer*, che produce codice HTML in relazione allo stato del componente stesso (processo che viene detto *encoding*).
La pagina così prodotta arriva all'utente. Quando l'utente invia dati al server, questo processa la richiesta e produce una serie di coppie ID/valore che viene memorizzata in una tabella hash. Ciascun componente può quindi consultarla e stabilire quali sono i valori relativi al tag ad esso associato, salvandoli come *valori locali* (o *local values*). Questa operazione è chiamata *decoding*.

Comunicazione client-server

La comunicazione tra client e server può avvenire essenzialmente in due modi, entrambi standard: tramite richieste *POST* e *GET*. JSF inoltre supporta nativamente ed in modo trasparente le richieste AJAX, consentendo di creare pagine web dinamiche. La comunicazione asincrona tramite AJAX è uno strumento molto efficace in vari contesti; la figura 3.3 mostra come possa ad esempio essere utilizzata per la gestione degli eventi e la validazione dei dati.

Ciclo di vita

Ciò che viene eseguito tra una richiesta HTTP e la relativa risposta è chiamato dalla specifica JSF *ciclo di vita* (in inglese *life cycle*). Di seguito vengono

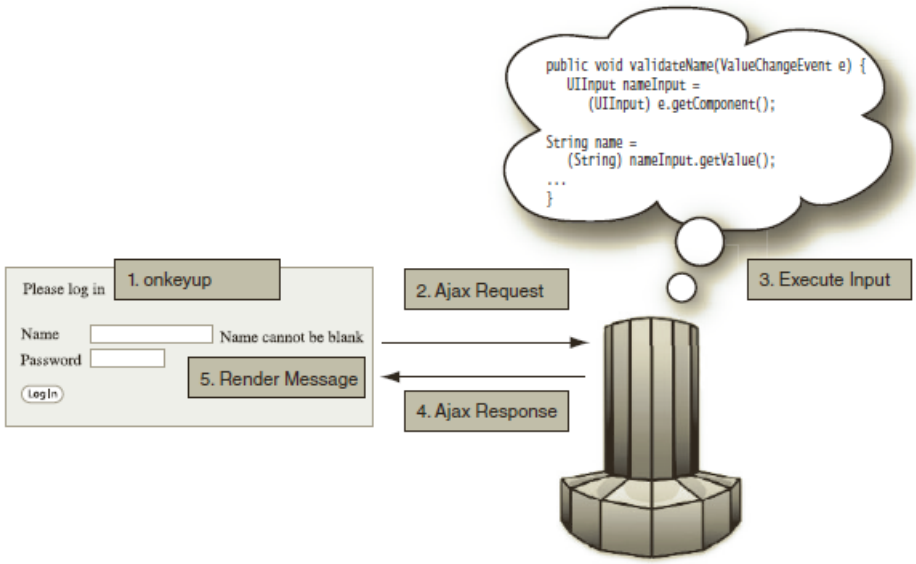


Figura 3.3: Richiesta AJAX per la validazione di un input

riportate le sei fasi del ciclo di vita di JSF (raffigurate nella figura 3.4), come stabilito dalla specifica:

1. ***Restore view***: è la fase in cui viene costruito l'albero dei componenti, o viene recuperato nel caso di pagina mostrata precedentemente.
2. ***Apply request values***: in questa fase, la richiesta HTTP che l'utente invia al server viene processata, eseguendo l'operazione di *decoding*.
3. ***Process validations***: viene eseguita la conversione e la validazione dei dati dell'utente.
4. ***Update model values***: durante questa fase viene aggiornato il modello dei dati.
5. ***Invoke application***: è la fase in cui il *navigation handler* di JSF decide quale pagina visualizzare in base alle decisioni prese dal controllore.
6. ***Render response***: viene eseguito l'*encoding*, producendo una pagina HTML che viene poi inviata al browser tramite la risposta del server.

Non sempre vengono eseguite tutte le fasi del ciclo di vita. Alcuni esempi:

- Nel caso in cui la richiesta HTTP non contenga valori, a seguito della fase *Restore view* viene eseguita immediatamente la fase *Render response*. Ciò accade, ad esempio, quando una pagina viene visualizzata per la prima volta.
- Se si riscontrano errori di conversione o validazione durante la fase *Process validations*, si passa alla fase *Render response*: viene visualizzata nuovamente la stessa pagina, mostrando i relativi messaggi di errore (se previsti nella creazione della pagina stessa da parte dello sviluppatore).

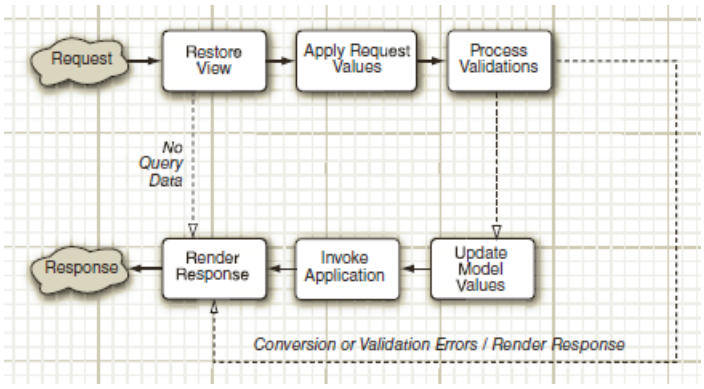


Figura 3.4: Ciclo di vita di JSF

- In una richiesta AJAX vengono indicati quali componenti processare e quali aggiornare. Per i primi vengono eseguite le prime cinque fasi del ciclo di vita e successivamente viene eseguito il *rendering* solo per i componenti da aggiornare.

3.4 Deltaspike

3.4.1 Introduzione

Deltaspike è un *framework* che estende le funzionalità di *CDI*. Il suo utilizzo in Jama è la gestione della sicurezza: associare ad ogni utente un insieme di permessi che determinano le azioni che può effettuare. La caratteristica fondamentale di Deltaspike è l'essere un *framework annotation-based*: funziona tramite le *Annotation* di Java - che d'ora in poi verranno chiamate semplicemente annotazioni - il che lo rende facile da utilizzare e poco invasivo.

3.4.2 Caratteristiche

Il cuore della sicurezza in Deltaspike è la classe *Authorizer*: è un *bean* di CDI - tipicamente *Application Scoped* - definito dallo sviluppatore che definisce i metodi che verranno invocati durante il controllo sui permessi di un utente. Questi metodi devono essere annotati con l'annotazione *@Secures* definita da Deltaspike, che indica che essi sono i metodi da invocare per il controllo sui permessi. Non basta: devono essere annotati anche con un'altra annotazione, una *custom annotation* - cioè definita dallo sviluppatore - che associa il metodo in questione ai metodi su cui effettuare il controllo dei permessi. Quest'ultima deve essere a sua volta annotata con l'annotazione *@SecurityBindingType* di Deltaspike.
Nonostante già il controllo sui metodi possa essere sufficiente per gestire l'intera sicurezza di un'applicazione, una funzionalità interessante e molto utile di Deltaspike è la possibilità di effettuare un controllo a livello di pagina: ad esempio, far sì che solo un Amministratore possa visitare la pagina di creazione di un utente, o che solo un Operatore Amministrativo possa visitare la pagina di creazione di una convenzione. Inoltre, qualora il controllo non vada a buon fine,

si può specificare una pagina di errore diversa per ogni pagina, ed un messaggio di errore che verrà visualizzato a video dopo il redirect.
Per concludere quest'introduzione e visione d'insieme delle funzionalità di Deltaspike con gli autori hanno avuto a che fare, si ritiene comunque importante far notare che, nonostante sia un framework già usabile e utile, non sia ancora completamente maturo: la documentazione è scarsa - per non dire di peggio - e alcune funzionalità utili sono mancanti o non funzionanti - ad esempio, il redirect ad una pagina di errore anche per la sicurezza sui metodi.
Si discute adesso in dettaglio come utilizzare Deltaspike.

3.4.3 Funzionamento

Rendere sicuro un metodo Per spiegare come si rende sicuro un metodo, si prenderà come esempio un problema affrontato nello sviluppo di Jama: far sì che solo un Operatore Amministrativo possa eliminare una convenzione. L'eliminazione di una convenzione, in Jama, consiste in sostanza nell'invocazione di un metodo di uno dei *bean* dell'applicazione, quindi il problema si risolve impedendo l'invocazione di tale metodo da parte di utenti che non siano un Operatore Amministrativo.
Il primo passo è il definire un'annotazione, chiamata per esempio *@DeleteContractsAllowed*:

```
Retention(value = RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.METHOD })
@Documented
@SecurityBindingType
public @interface DeleteContractsAllowed {}
```

Si procede dunque ad annotare il nostro metodo che elimina una convenzione con l'annotazione appena definita:

```
...

@DeleteContractsAllowed
public void deleteContract() {
   //Elimina una convenzione.

   ...
}
```

L'ultima cosa da fare è fornire l'Authorizer di un metodo annotato *@Secures* e *@DeleteContractsAllowed*:

```
public class Authorizer {
   @Secures
   @DeleteContractsAllowed
   public boolean canDeleteContracts {
      //Verifica che l'utente possa eliminare una convenzione.
      //Restituisce true in caso affermativo, altrimenti false.

      ...
```

```
    }
}
```

Il metodo appena definito verrà invocato da Deltaspike in maniera automatica tutte le volte che si invocherà *deleteContract()*. Nel caso *canDeleteContracts()* restituisca *true*, l'eliminazione può proseguire, altrimenti viene generata un'eccezione.

Rendere sicura una pagina La sicurezza su base pagina si implementa definendo interfacce e classi che rappresentano rispettivamente cartelle e pagine. Ad esempio, per rendere sicura la pagina *home.xhtml* che si trova dentro la cartella *pages*, si definisce un interaccia *Pages* con all'interno una classe *Home*; le classi così definite devono implementare l'interfaccia *ViewConfig* di Deltaspike. Il *path* è relativo alla cartella *webapp* dell'applicazione, quindi la classe Home contenuta nell'interaccia Pages fa riferimento alla pagina *webapp/pages/home.xhtml*. I nomi sono *case insensitive*: la classe Home fa riferimento alle pagine home.xhtml e Home.xhtml; stesso discorso vale per le interfacce.
Il metodo più facile per rendere sicuro un insieme di più pagine è creare un file Java e di definire all'interno di quest'ultimo tutta la gerarchia di interfacce e di classi - ricordando di omettere il *modifier public* per ognuna delle interfacce/-classi, altrimenti si otterrebbe un errore di compilazione.
Ognuna delle classi definite deve essere annotata con l'annotazione *@Secured*, completata con l'attributo *value* che specifica una classe definita dallo sviluppatore che implementa l'interfaccia *AccessDecisionVoter* di Deltaspike; questa classe deve essere un *bean* di CDI, tipicamente *Application Scoped* . La classe specificata deve implementare il metodo *public Set<SecurityViolation> checkPermission()*, che viene invocato da Deltaspike ogni volta che si tenta di accedere alla pagina.
Il metodo restituisce un Set di *SecurityViolation*, una *Anonymous Inner Class* definita da Deltaspike. Nel caso esso sia vuoto, l'accesso viene consentito, altrimenti viene impedito.
Come abbiamo visto nell'introduzione, una delle peculiarità più interessanti della sicurezza su base pagina è il poter specificare una pagina di errore. Per fare ciò, si arricchisce l'annotazione *@Secured* delle pagine con l'attributo *errorView*, indicando una pagina definita secondo la solita convenzione spiegata ad inizio paragrafo. Nel caso venga specificata una pagina di errore, le SecurityViolation contenute nel Set restituito da checkPermission() vengono aggiunte al *bundle* di messaggi della pagina, ovvero all'interno dell'area definita dal tag *<h:messages>* della pagina - detta in maniera meno tecnica, verrà visualizzato un messaggio a video nella pagina di errore per ogni SecurityViolation contenuta.
Il messaggio da visualizzare viene specificato nel metodo *public String getReason()* di ogni SecurityViolation; ricordando che quest'ultima è una Anonymous Inner Class, il metodo getReason() viene obbligatoriamente ridefinito ad ogni SecurityViolation creata, permettendo così di specificare il messaggio di errore più adatto in ogni occasione.

Per fissare meglio le idee, è opportuno mostrare un esempio. Per prima cosa, si creano le classi relative alle pagine da rendere sicure:

```
interface Pages {

   @Secured(value = { ViewHomeAccessDecisionVoter.class }, errorView = L
       ogin.class)
   class Home implements ViewConfig {}


   class Login implements ViewConfig {}

}
```

Successivamente si definisce l'AccessDecisionVoter; non viene mostrata nel dettaglio nella logica che esegue il controllo, verrà invece usato uno pseudo-codice che renda l'idea di come deve funzionare la classe:

```
@ApplicationScoped
public class ViewHomeAccessDecisionVoter implements AccessDecisionVoter {
   private SecurityViolation violation = new SecurityViolation() {
      public String getReason() {
         return "Non sei autorizzato";
      };

   public Set<SecurityViolation> checkPermission(
           AccessDecisionVoterContext accessDecisionVoterContext) {

      Set<SecurityViolation> violations = new HashSet<>();

      if( user cannot see home )
         violations.add(violation);
      return violations;
   }
}
```

Il gioco è fatto: chiunque tenti di visitare la pagina *home* senza averne i permessi - ad esempio, un utente che senza fare il login tenta di andare subito alla home - viene reindirizzato alla pagina di login, dove potrà vedere un bel messaggio che recita: *Non sei autorizzato!*

3.5 Altre librerie

3.5.1 JLDAP

LDAP è un protocollo per l'accesso a cartelle. Definisce un meccanismo in cui i client mandano richieste e ricevono risposte da server LDAP. *JLDAP* è una libreria sviluppata da *Novell* che permette l'accesso al servizio di gestione di cartelle LDAP.

LDAP viene utilizzato dall'applicativo per recuperare, tramite un server LDAP di SIAF, informazioni riguardo agli utenti al momento del login. In

particolare consente ai docenti di ateneo di loggarsi con le proprie credenziali uniche di ateneo senza che ci sia bisogno di inserirli a mano in qualche database interno.

Funzionamento L'operazione centrale è la ricerca dell'utente tramite matricola nel server LDAP. Il metodo che consente di effettuare questa operazione è:

```
LDAPSearchResults search(String base, int scope, String filter, S
    tring[] attrs, boolean typesOnly)
```

I parametri più interessanti di questo metodo sono:

- *base*
  Serve per specificare il punto di partenza della query, ovvero la cartella a partire dalla quale effettuare la ricerca. Per l'applicativo tale parametro è impostato a "ou=people,dc=dinfo,dc=unifi,dc=it";
- *filter*
  Serve per specificare i criteri di ricerca: si specificano i valori di alcuni attributi dell'entità cercata. Nel caso della ricerca per matricola occore impostare "uid=¡matricola¿".

Il risultato di questa operazione è una entità rappresentata così:

```
dn: uid=D064678,ou=docenti,ou=personale,ou=people,dc=unifi,dc=it
objectClass: unifiPersonale
cn: NOMECOGNOME
gidNumber: 513
uid: D064678
uidNumber: 800064678
employeeType: Professori Associati
givenName: NOME
mail: nome.cognome@unifi.it
sn: COGNOME
userPassword:: {md5}e33ENX1ZdUkzaXNIMSthVEVFejErZUVoM1VRPT1
```

Una volta ottenuta l'entità è possibile effettuare il login come spiegato in INSERIRE RIFERIMENTO A LOGIN.

3.5.2 FreeMarker

TODO

Capitolo 4

Utilizzo

Capitolo 5

Dietro le quinte

5.1 Login

La schermata di login si presenta in maniera molto semplice: è costituita da un messaggio di benvenuto e da un form in cui inserire le proprie credenziali per poter accedere. I dati inviati dal form vengono processati dal bean `UserManager`, che si occupa di eseguire l'autenticazione.
Quando l'utente invia il form viene chiamato il metodo `login` dello `UserManager`, listato qui di seguito:

`INSERIRELISTATO`

Per prima cosa, il sistema recupera l'utente associato alla matricola inserita tramite il DAO `UserDaoBean`. Esso esegue un'interrogazione al database locale; se non trova nessuna corrispondenza, il compito di cercare la matricola viene delegato allo `LdapManager`, il quale effettua una ricerca nel sistema di autenticazione di Ateneo utilizzando il protocollo LDAP per la comunicazione. Se anche in questo caso non viene trovato un riscontro, l'utente viene avvertito con un messaggio di errore.
Se invece viene trovato un utente, tutte le informazioni ad esso relative vengono salvate in un oggetto `User` che viene restituito allo `UserManager`. A questo punto viene verificata la validità della password. Questo compito è delegato alla classe `User` stessa, la quale contiene al suo interno un attributo di tipo `Encryptor` che specifica l'algoritmo di criptazione utilizzato per la propria password. La password inserita viene quindi criptata e confrontata con quella memorizzata. Chiaramente, si procede solo se risultano uguali.
A questo punto lo `UserManager` crea un nuovo `Principal` con i dati dell'utente e viene mostrata la home dell'applicazione: il login è stato effettuato con successo!

5.2 Lista convenzioni

5.2.1 Livello presentazione

All'interno dell'applicazione, ricoprono un ruolo centrale le schermate di visualizzazione delle convenzioni/contributi. Gli oggetti che consentono il funzionamento di una schermata di visualizzazione della convenzione sono i seguenti:

- il file `.xhtml` che produce la pagina web
- un bean *controllore di pagina*
- un *lazy model*, che rappresenta i dati visualizzati
- un DAO che recupera i dati da visualizzare

Di seguito, segue una spiegazione di ognuno di questi componenti.

Pagina web

Una schermata di visualizzazione delle convenzioni/contributi è costituita essenzialmente da una tabella, che viene creata mediante il tag `pdata:Table` di PrimeFaces. Poiché dovrà gestire grandi quantità di dati, è stata utilizzata la paginazione unita al *lazy loading*: quando viene caricata una pagina, vengono mantenuti in memoria soltanto i contratti che vengono effettivamente visualizzati. Ciò è reso possibile (o, perlomeno, molto più semplice) grazie ai due attributi della `dataTable` di PrimeFaces `paginator` e - soprattutto - `lazy`; sono entrambi attributi booleani che specificano se utilizzare la relativa tecnica.

Controllore di pagina

Associata ad ogni vista, vi è un bean detto *controllore di pagina*. Esso si occupa di gestire la conversazione necessaria al corretto funzionamento della pagina e di fare da intermediario tra la vista ed il modello, eseguendo le operazioni necessarie per eseguire la navigazione verso le varie schermate dell'applicazione raggiungibili. Il controllore di pagina ha due attributi importanti: il *lazy model*, a cui si può far riferimento dalla pagina (ovviamente), e un `ContractManager` che invece si occupa della logica di business, gestendo i contratti (e che non è direttamente accessibile dalla pagina).

Lazy model

Per creare una tabella *lazy-loaded*, è necessario implementare un *lazy model*, che fornirà la lista di oggetti che verranno infine visualizzati. Il *lazy model* deve inoltre gestire gli eventuali filtri ed ordinamenti richiesti.
Un *lazy model* è una classe Java che estende la classe di PrimeFaces `LazyModel<T>`, dove `T` è il tipo di dati da visualizzare (in questo caso, quindi, bisognerà estendere `LazyModel<Contract>`). Il metodo principale di questa classe è `load`, che ha il compito di caricare i dati dal modello e passarli alla tabella. La sua signature è:

```
public List<T> load(int first, int pageSize, String sortField, SortOrder
    sortOrder, Map<String, String> filters)
```

Questo metodo viene chiamato ogni volta che bisogna aggiornare la tabella. I parametri formali rappresentano:

1. `first` e `pageSize`: la prima riga da visualizzare (partendo da 0) e la dimensione di una pagina, rispettivamente. Queste due informazioni possono essere combinate per sapere quale pagina è visualizzata correntemente: è infatti il quoziente della divisione intera di `first` per `pageSize`.

2. `sortField, sortOrder`: sono informazioni sull'ordinamento della tabella. Il primo indica in base a quale campo ordinare, il secondo è un'enumerazione che indica se l'ordinamento deve essere ascendente o discendente.

3. `filters`: è una mappa di coppie in cui la chiave è il campo su cui filtrare e il valore è, appunto, il valore del campo.

Affinché la paginazione sia effettivamente *lazy*, però, è necessario che la query effettuata per ottenere le informazioni sia 'mirata'. Per maggiori dettagli su come questo avvenga, si rimanda a INSERIRE RIFERIMENTO A PAGER HIBERNATE.

Se la tabella prevede anche che una riga possa essere selezionata, un *lazy model* deve anche esporre i metodi `getRowData` e `getRowKey`, che consentono, rispettivamente, di estrarre il dato relativo ad una riga e la riga relativa ad un dato. Entrambi utilizzano una chiave fornita dal programmatore per distinguere le righe della tabella (come ad esempio un identificativo) che deve essere esplicitata nell'attributo `rowKey` della `p:dataTable`.

***Lazy model* delle liste di contratti** Molte liste di contratti, da quelle visualizzate dall'Operatore a quella a cui ha accesso il Docente, condividono varie operazioni in comune e sono state perciò create classi astratte per il riuso del codice.
La classe di base è `ContractTableLazyDataModel`, che è un *lazy model* e implementa tutte le operazioni precedentemente descritte, consentendo perciò il normale funzionamento dell'interfaccia. Tuttavia, tramite una lista di contratti è possibile svolgere alcune operazioni che comportano la navigazione ad un'altra pagina web (un esempio è la visualizzazione della convenzione). Quando l'utente ritorna alla lista, la pagina viene creata *ex-novo* e la tabella non mantiene perciò i cambiamenti effettuati dall'utente, come ad esempio il filtraggio dei risultati in base alla data.
La classe `ContractTableLazyDataModel` risolve questi problemi, esponendo dei metodi per impostare o estrarre lo stato della tabella. L'intero procedimento avviene in questo modo:

1. l'utente esegue un'operazione su una convenzione che comporta il cambio di pagina, come la visualizzazione o la modifica

2. viene chiamato il rispettivo metodo del controllore di pagina, che estrae lo stato della tabella come una stringa di coppie chiave/valore e lo passa al bean del livello sottostante che si occupa della gestione dei contratti (il `ContractManager`)

3. la conversazione attuale viene chiusa

4. il `ContractManager` inizia una nuova conversazione
5. la nuova pagina viene visualizzata; a questo punto il controllore di pagina precedente è *out of scope*
6. l'utente esegue alcune operazioni nella nuova pagina e poi clicca sul pulsante per tornare alla lista, causando la chiusura della conversazione
7. all'URL prodotto per tornare alla tabella vengono accodati i parametri passati precedentemente al `ContractManager` come stringa
8. prima che la pagina sia caricata, il bean controllore di pagina viene inizializzato dal container: la conversazione comincia e il *lazy model* viene creato
9. il *lazy model* viene inizializzato con i parametri contenuti nell'URL
10. la pagina viene caricata e la tabella ritorna così allo stato in cui si trovava prima della visualizzazione/modifica.

Pager e Criteria API

Le classi che si occupano del recupero delle informazioni dal database sono `ContractSearchService` e `DeadLineSerachService` la prima è usata per costruire lo scadenzario mentre la seconda è usata per la lista delle convenzioni/contributi. Queste classi si differenziano solo per il tipo di query che eseguono ma sono simili per struttura, per cui di qui in avanti ci si riferirà solo a `ContractSearchService`. Lo scopo di queste classi è effettuare un ricerca paginata sul database secondo vari criteri di ricerca quali responsabile scientifico, data, dittà etc. Il codice che si occupa della paginazione è stato incapsulato nella classe `ResultPager`. `ResultPager` viene costruito col metodo

```
ResultPagerBean(int currentPage, int pageSize, TypedQuery<T> query, Typed
    Query<Long> countQuery)
```

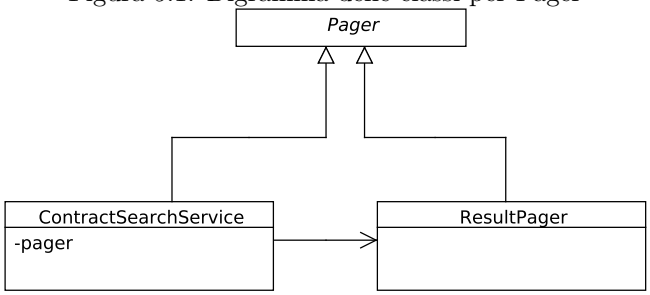
Come si può vedere è possibile impostare il numero di risultati per ogni pagina, la pagina corrente e naturalmente la query che si vuole eseguire. Per ottenere una pagina di risultati è sufficiente chiamare `getCurrentResults()` mentre è possibile spostarsi da una pagina all'altra tramite `next()` e `previous()`

`ContractSearchService` viene specializzato, per poter eseguire la paginazione, tramite la composizione con un oggetto di tipo `Pager`, l'interfaccia che `ContractSearchService` e `ResultPager` implementano, come illustrato in figura 5.1

La query è invece costruita utilizzando le Criteria API di JPA, un modo alternativo per costruire query che usa una API java invece di un linguaggio come JPQL. Questo tipo di costruzione è specialmente indicata per query dinamiche, ovvero quando non si conoscono i criteri di ricerca fino al runtime. Il caso che si sta analizzando è proprio quello di una query dinamica: l'utente tramite l'interfaccia messa a disposizione filtra le convenzioni/contributi in base a vari tipi di criteri.

Volendo usare JPQL si sarebbe dovuto costruire la stringa di definizione della query al runtime, passarla al metodo `createQuery` di un `Entity Manager` il quale

Figura 5.1: Digramma delle classi per Pager



avrebbe parsato la stringa e restituito un oggetto `Query` da cui sarebbe stato possibile estrarre i risultati. Ciò significa che ogni volta che si vuole eseguire query è necessario parsare una stringa, costruendo una rappresentazione interna della query per poi generare il codice SQL da eseguire sul database. Criteria API consente di eliminare l'overhead dovuto al parsing e costruire i vari criteri di ricerca utilizzando una API Java invece di stringhe. Di seguito è mostrato un estratto di codice usato in `ContractSearchService` per definire la query.

```
TypedQuery<Contract> query;

CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Contract> c = cb.createQuery(Contract.class);
Root<? extends Contract> agr = c.from(contractClass);

c.select(agr).distinct(true);

List<Predicate> criteria = new ArrayList<Predicate>();

if (companyId != null) {

   ParameterExpression<Integer> p = cb.parameter(Integer.class,
        "companyId");
   criteria.add(cb.equal(agr.get("company").get("id"), p));

}

...


c.where(cb.and(criteria.toArray(new Predicate[0])));

query = em.createQuery(c);

if (companyId != null) {
   query.setParameter("companyId", companyId);
   countQuery.setParameter("companyId", companyId);

}
```

...

Come si può vedere dall'estratto, per prima cosa si crea una query Root, questa gioca il ruolo di una variabile identificativa in una clausola FROM in JPQL e indica a quale tipo di schema si è interessati; quindi con l'operazione `select(agr)` si indica quale è il risultato della query. La clausola WHERE è invece ottenuta componendo una serie di predicati, per esempio con l'espressione `cb.equal(agr.get("company").get("id"), p)` si richiede che l'id della dittà sia uguale ad un certo valore.

5.3 Creazione e Modifica di Entità

Il processo di creazione di un entità del modello coinvolge vari componenti:

- una pagina web che viene utilizzata nella creazione dell'interfaccia grafica
- (opzionale) un bean di presentazione, che garantisce un corretto funzionamento della pagina
- un controllore, che fa da intermediario tra il livello di presentazione e il modello di business
- un DAO, che si occupa della comunicazione con il database

5.3.1 Livello presentazione

Quando una schermata di creazione viene visualizzata, per prima cosa viene istanziato dal *container* il controllore che si occupa della creazione dell'entità, il quale a sua volta inizia una conversazione e crea l'oggetto che conterrà i valori inseriti tramite interfaccia dall'utente.
Graficamente le pagine web di questo tipo sono costituite semplicemente da alcuni campi di input in cui inserire i dati, inseriti all'interno di un wizard se il numero di valori da inserire è troppo elevato e renderebbe troppo carica un'unica schermata; in questo caso, si rende a volte necessario un bean di presentazione per controllare il flusso del wizard. Normalmente, i form non sono inseriti direttamente nella pagina; piuttosto, si utilizzano vari componenti personalizzati, i quali vengono inclusi nella pagina per ottenere l'effetto desiderato. In questo modo, lo stesso componente può essere utilizzato in più contesti. Questo è reso possibile grazie al tag `composite` di JSF, che consente di creare componenti ed inserirle all'interno di un namespace personalizzato.
La pagina è poi ovviamente corredata di pulsanti per salvare i dati inseriti o per tornare alla vista precedente ignorando i cambiamento. Nel caso in cui l'utente decida di salvare le modifiche effettuate, il controllore provvede ad aggiornare il database per mezzo del DAO opportuno. In entrambi i casi, la conversazione viene chiusa e si ritorna alla schermata precedente.

5.3.2 Edit Session e Extended Persistence Context

Si descrive la struttura di un bean, da qui in avanti `Manager`, che possa essere usato dallo strato di presentazione per la creazione e modifica di una entità di business.

Caratteristiche del Manager `Manager` avrà un riferimento all'entità di business che stiamo creando che possa essere riempita in base ai campi riempiti dall'utente a video. Come si è detto la creazione/modifica di una entità di business in generale è realizzata attraverso più passi di una procedura, e quindi non può essere confinata in una sola request. Questo suggerisce che `Manager` debba essere *reques scoped*. Si sottolinea che non sarebbe possibile usare un bean di tipo *session scoped* perchè questo comporterebbe la condivisione del bean fra due tab del browser: l'utente che crede di creare due entità in parallelo sta invece modificando la stessa!

Direttamente collegato alla questione dello scope c'è il problema del detachment: se il Persistence Context che viene usato ha il proprio ciclo di vita legato alla transazione, necessariamente dopo il recupero dal database l'entità diventerà detached. Come si è spiegato in 3.1 lo stato di un entità detached non verrà scritto sul database in nessuna transazione. Sebbene sia possibile riportare un' entità da detached a managed tramite l'operazione di `merge()`, in generale è preferibile non farlo perchè la gestione dei riferimenti dell'entità è problematica. Per ovviare a questo problema possiamo optare per un Entity Manager di tipo extended. Questo ci garantisce che durante tutto il ciclo di vita del bean `Manger`, ovvero per tutta la conversation, avremmo un solo Persistence Context, e di conseguenza l'entità che stiamo modificando non sarà mai detached.

Un'altra problematica che si deve affrontare è prevedere la possibilità di annullare la creazione/modifica in qualsiasi momento della procedura. Per risolvere questo problema dobbiamo gestire le transazioni del container. Un modo elegante per farlo è annotare il bean con l'annotazione

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
```

e il metodo che conclude la procedura salvando i dati con

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
```

Il bean `Manager` può quindi essere strutturato come segue:

```
@ConversationScoped
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class Manager implements Serializable {

   private static final long serialVersionUID = -4966124878956728047L;
   @Inject
   private Conversation conversation;

   private Entity entity;

   @PersistenceContext(unitName = "primary", type = PersistenceContext
       Type.EXTENDED)
   private EntityManager em;


   public UserEditorBean() {
      super();
```

```
    }


    private void begin() {

        conversation.begin();
    }


    @Remove
    private void close() {

        conversation.end();

    }


    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public String save() {

        em.persist(entity);

        close();

        return "home";
    }


    public String cancel() {
        close();
        return "home";
    }



    public String createUser() {
        begin();
        currentUser = new User();
        return "wizard";
    }


    public String editEntity( int id) {

        begin();

        entity = em.find(id);
        return "wizard";
    }


    }

}
```

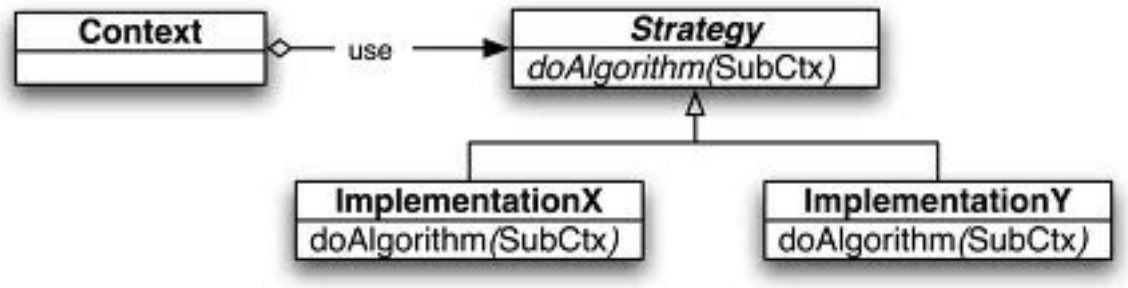


Figura 5.2: Il pattern Strategy

Ricapitolando il bean così strutturato ha un tempo di vita pari alla durata della conversazione, il persistence context associato all'entity manager lo stesso, quindi nessuna entità sarà mai detached, inoltre viene effettuata una transazione solo chiamando il metodo `save()`: in questo modo è possibile annullare le modifche semplicemnte chiamando il metodo `cancel()`.

5.4 Filler Vari ed Eventuali

Come già spiegato precedentemente, durante la creazione di una convenzione/-contributo deve essere inserita una suddivisione dell'importo concordato, chiamata *ripartizione*. La ripartizione, in generale, non è totalmente libera: alcuni valori possono essere immessi e sulla base di questi ne vengono calcolati altri secondo opportuni schemi. Ad esempio, attualmente in una convenzione va al *Fondo Comune di Ateneo* il 2.5% dell'ammontare totale della convenzione.
Il meccanismo in base al quale calcolare le quote 'derivate' a partire da quelle 'libere' può subire delle modifiche nel tempo. Per questo motivo, si è scelto di adottare il pattern *Strategy* per consentire di cambiare in un secondo momento l'algoritmo da utilizzare.

Il pattern *Strategy*

Nel pattern Strategy, l'algoritmo è incapsulato in una classe separata rispetto al contesto in cui deve essere usato. Inoltre, tutti gli algoritmi (o meglio, tutte le classi che rappresentano un algoritmo) implementano un'interfaccia comune. L'oggetto all'interno del quale si deve adoperare un algoritmo di quel tipo avrà un riferimento all'interfaccia, rimanendo così slegata dalle classi concrete: questo rende semplice modificare in fase di manutenzione l'implementazione da utilizzare. Lo schema del pattern è raffigurato in figura 5.2; per maggiori dettagli, si consiglia INSERIRE RIFERIMENTO A DESIGN PATTERN.

Filler

La classe che rappresenta genericamente l'algoritmo è chiamata `ContractShare TableFiller`; per brevità, si farà di seguito riferimento ad essa con il termine

filler.
Attualmente, è prevista un'unica implementazione del filler, chiamata `Standard ContractShareTableFiller` (o *filler standard*). Questa implementazione prevede tutti gli altri attributi della tabella di ripartizione vengano calcolati sulla base della quota relativa al personale. In figura 5.3 è mostrato il pattern Strategy applicato al caso concreto.
Le percentuali sulla base delle quali eseguire il calcolo sono definibili attraverso gli opportuni file di configurazione, che si trovano all'interno della sotto-cartella `aliquoteDipartimenti` della cartella di configurazione (path relativo alla directory di jBoss: `standalone/deployments/Jama.war/WEB-INF/classes/config`).

In realtà, in questa cartella non sono presenti file di configurazione, ma altre sotto-cartelle. Infatti, è possibile specificare aliquote diverse per ogni dipartimento, ognuno dei quali ha la propria directory, riconoscibile dall'identificativo del dipartimento stesso. Questo però ha effetti anche sul codice: non è possibile utilizzare lo stesso filler per tutti i dipartimenti. È questo il motivo per cui è stato utilizzato il pattern *Abstract factory* per incapsulare la responsabilità di istanziare il filler appropriato. Quando è necessario ottenere un filler per un dato contratto, è sufficiente chiamare il metodo `getFiller` della factory passandogli il dipartimento del docente che lo ha stipulato.
Chiaramente, anche per la factory è necessario stabilire quale implementazione utilizzare, essendo questa legata al filler da produrre. Il meccanismo è sempre lo stesso: non si fa riferimento alla classe concreta, ma a quella di base. Se si dovesse cambiare tipo di filler, è sufficiente implementare la relativa factory ed aggiornare il resto di conseguenza. L'aggiornamento è inoltre molto più semplice nel caso della factory, perché esse non sono entità di JPA (ossia, non sono annotate `@Entity`), ma sono bean di CDI. Questo consente di utilizzare le *alternatives* (si veda il capitolo 3.2 per maggiori dettagli) e quindi basta modificare il file `bean.xml` specificando il bean da utilizzare.

Le cose sono in realtà leggermente più complicate di così. Infatti, bisogna tener conto che le variazione delle quote o del metodo di calcolo stesso non devono essere retroattive. Ad esempio, se nel 2013 al Fondo Comune di Ateneo spetta il 2.5% del totale della convenzione, mentre nel 2014 l'1% sulla quota relativa al personale, le convenzioni stipulate nel 2013 dovranno mantenere la quota precedente (2.5% sul totale), che siano ancora attive nel 2014 o che non lo siano. Da ciò nasce l'esigenza di aggiungere nel contratto un attributo che rappresenta il filler utilizzato e di salvare questa informazione nel database; inoltre, è necessario anche poter effettuare query sul filler stesso. Per questi motivi, nell'implementazione del pattern Strategy si è deciso di utilizzare una classe astratta invece di un'interfaccia.
In figura 5.4 è mostrato lo schema completo.

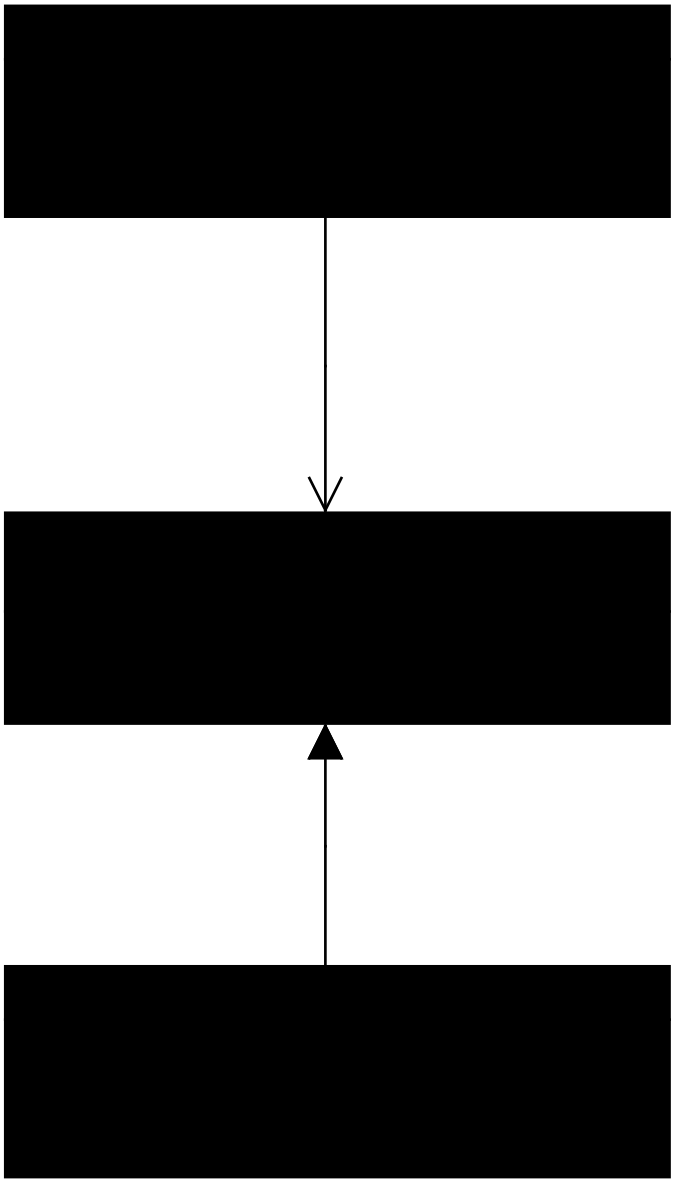


Figura 5.3: Il pattern Strategy applicato ai filler

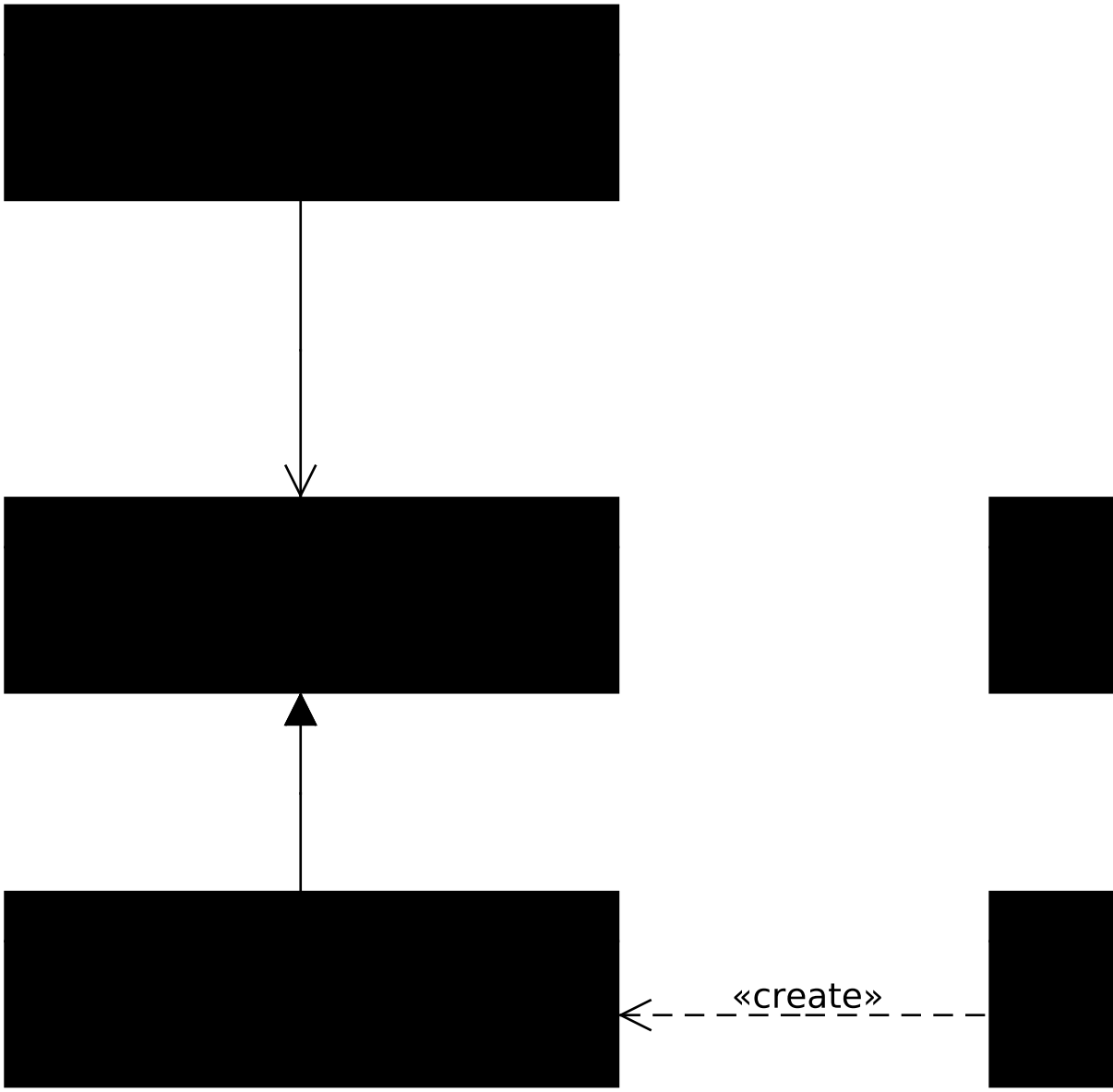


Figura 5.4: Il pattern Strategy applicato ai filler