Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Gestione dei flussi applicativi con Spring Web-Flow

Laureando

Andrea Lovato

Relatore

Prof. Giorgio Maria Di Nunzio

Anno Accademico 2011/2012

Ai miei genitori,ad Alessia, ad Elisa,
ed a tutti coloro che mi hanno aiutato,
supportato e reso possibile questo percorso.
Senza di voi, non starei scrivendo queste righe.

Grazie.

Andrea Lovato

Indice

Elenco delle figure

1 Introduzione

L'evoluzione delle tecnologie ha permesso la produzione di software maggiormente orientato allo sfruttamento delle risorse di Internet permettendo la nascita di applicazioni interattive web-based. I nuovi e sempre più innovativi servizi che queste web application forniscono sono oggi utilizzabili da un qualunque terminale dotato di una connessione web e di un browser.
Fino ad oggi il pattern MVC era alla base della grande maggioranza di queste applicazioni sviluppate in ambito Java, grazie all'obiettivo di scomporre l'applicativo in tre strati: Model, View e Controller. Spesso però ci si e trovati di fronte al problema di gestione del flusso applicativo, ovvero delle pagine che determinano l'applicazione.
È proprio questo problema che Spring Web Flow vuole risolvere, permettendo di codificare e configurare i flussi applicativi in modo centralizzato e armonico.
Lo scopo di tale tesi è quello di studiare tale framework attraverso la comprensione delle tecniche e dei moduli che esso mette a disposizione, cominciando con un'analisi teorica e seguendo con una pratica.
Si parte, nel primo capitolo, con una descrizione di tutte le componenti che hanno reso Spring il framework più popolare per lo sviluppo di applicazioni JEE.
Successivamente si introduce il concetto di Model-View-Controller ed il confronto con Spring MVC, di cui Spring Web Flow è un sotto progetto.
Nel terzo capitolo si è entrati nel dettaglio del framework, spiegando cos'è un flusso e definendo i concetti che ne stanno alla base.
È stato sperimentato, nel quarto capitolo, tale framework attraverso l'implementazione di un piccolo modulo da inserire in una web application esistente, che permette di cercare e modificare le anagrafiche dei clienti di un'azienda.

2 Spring Framework

2.1 Introduzione

Quando si progetta e si sviluppa un'applicazione software, spesso ci si ritrova a scrivere del codice per delle situazioni comuni, compiti già eseguiti in altri progetti o da altri sviluppatori. I framework sono nati con lo scopo di evitare queste situazioni, fornendo librerie di codici utilizzabili per aumentare la velocità dello sviluppo del prodotto finito, nonchè ottimizzare e rendere stabile tale software.

Fig. 1 : *Spring Logo*

Nato come codice allegato al libro *"Expert One-on-One J2EE Design and Development"*, Spring ha il chiaro intento di gestire la complessità legata allo sviluppo di applicazioni enterprise.
La sua prima apparizione risale al 2003 sotto licenza Apache, ma fu solo nel marzo del 2004 il primo importante rilascio, ovvero la versione 1.0; attualmente è giunto alla versione 3.1.2 e viene definito come il più popolare framework per lo sviluppo di applicazioni Java Enterprise.[1] Visto come una valida, e ormai ben solida alternativa al modello basato su *Enterprise Javabeans*, [2] questo framework supera l'appesantimento portato dall'utilizzo forzato di interfacce EJB di tipo home e remote, troppo invasive nel codice scritto, pur mantenendo la gestione del deployment descriptions

[1] www.springsource.org
[2] en.wikipedia.org/wiki/Enterprise_JavaBeans

in XML grazie a nuovi ed innovativi modelli di programmazione, come l'*Aspect Oriented Programming (AOP)* e l'*Inversion of Control (IoC)*.
Spring è inoltre un framework leggero: grazie alla sua struttura estremamente modulare è possibile utilizzarlo nella sua interezza o solo in parte, senza stravolgere l'architettura del progetto; questa peculiarità permette una facile integrazione anche con altri framework già esistenti, come Struts[3] o Hibernate.[4] Fornisce, inoltre, una serie completa di strumenti per gestire la complessità dello sviluppo software, fornendo un approccio semplificato sia più comuni problemi di sviluppo (accesso ai database, gestione delle dipendenze, etc.) che di testing.

2.1.1 Aspect Oriented Programming

Il paradigma di *Aspect Oriented Programming* è nato come naturale conseguenza del paradigma di *Object Oriented Programming*.[5] Nello sviluppo di un applicativo, infatti, esistono molti comportamenti che, per la loro natura, non sono facilmente isolabili in moduli distinti, pur essendo logicamente indipendenti; un valido esempio può essere l'attività di logging, compito che, spesso, non può essere modellato come uno o un insieme di oggetti, semplicemente perchè interessa l'applicazione nel suo insieme.
Il concetto base del paradigma AOP è quello di modularizzare le varie componenti del software, separando questi *cross-cutting concerns*,[6] ponendo il codice applicativo del tutto all'oscuro della loro esistenza.
I punti fondamentali sul quale il paradigma AOP si basa sono:

- **Aspect**
  Gli aspect appresentano la modularizzazione delle caratteristiche indipendenti di un sistema, che altrimenti invaderebbero in modo "trasversale" l'intera applicazione.

[3] struts.apache.org
[4] www.hibernate.org
[5] en.wikipedia.org/wiki/Object-oriented_programming
[6] en.wikipedia.org/wiki/Cross-cutting_concern

- **Joint Point**
  I Joint Point sono un punto preciso nell'esecuzione del programma, e può essere rappresentato dall'esecuzione di un metodo, da un'eccezione, da un'inizializzazione statica o dinamica o da accessi di lettura e scrittura; in Spring un Joint Point rappresenta sempre l'esecuzione di un metodo.

- **Pointcut**
  Descrive un determinato set di regole con cui associare l'esecuizione di un Advice ad un determinato Joint Point.

- **Advice**
  Rappresenta il codice che viene eseguito da un Aspect quando un determinato Joint Point viene raggiunto. A differenza di un metodo che deve essere esplicitamente invocato, un Advice viene eseguito automaticamente ogni volta che ad un determinato evento (Joint Point) si verifica una particolare condizione (Pointcut).

- **Target**
  Chiamato anche Advised Object, rappresenta l'oggetto sul quale agisce l'Advice

2.1.2 Una tecnica di IoC: Dependency Injection

Il pattern *Inversion of Control* sta alla base di Spring. Grazie a questa pratica di programmazione si riesce a minimizzare le dipendenze che intercorrono tra gli oggetti, ciò significa rendere l'applicazione più robusta, modulare e facilitarne il riutilizzo dei componenti. Con IoC si inverte letteralmente il controllo del flusso (Control Flow) rispetto alla programmazione tradizionale, la quale lascia al programmatore definire la logica di tale flusso. Cioè non sarà più lo sviluppatore che si farà carico di inizializzare ed invocare i metodi degli oggetti coinvolti nel flusso applicativo, ma bensì il framework, che inietterà le dipendenze direttamente nelle classi.

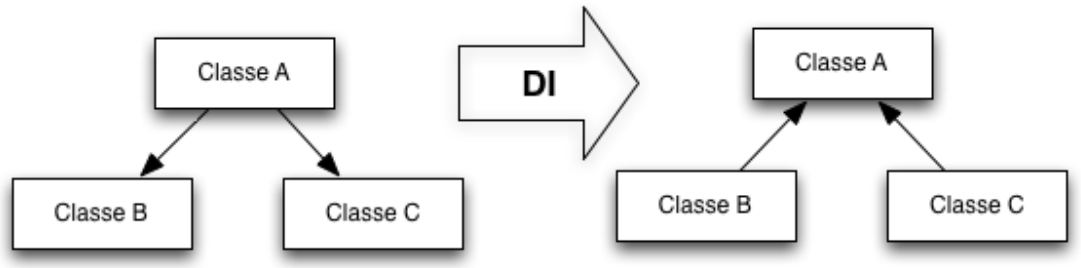

Fig. 2 : *È possibile vedere, nel primo diagramma, le dipendenze della classe A verso le classi B e C. Grazie alla Dependency Injection le dipendenze vengono "iniettate" nella classe A da un container.*

Una delle tecniche con le quali si può attuare l'IoC è la Dependency Injection. Grazie a questa, gli oggetti che verranno utilizzati nel flusso applicativo saranno creati da un componente esterno (Container), che si occuperà di creare l'oggetto stesso, le relative dipendenze e di assemblarle attraverso l'uso dell'injection.
I modi con cui il framework gestisce l'injection sono:

- **Constructor Injection** dove le dipendenze vengono iniettate attraverso l'argomento del costruttore;
- **Setter Injection** dove le dipendenze vengono iniettate attraverso un metodo "set".

Grazie a dei file di configurazione XML, il Container inietta le dipendenze direttamente nei bean (che in Spring può essere rappresentato da una qualunque classe Java) e ne gestisce l'intero ciclo di vita.

2.2 La Struttura di Spring

Oltre alle peculiarità già introdotte, dietro a Spring c'è un vasto ecosistema che si estende dalle Web Applications all'interazione con i Database. Come già accennato, uno dei principali vantaggi offerti da questo framework è quello di utilizzare solo le

parti necessarie all'applicazione, escludendo e non integrando le altre; ciò è garantito dalle caratteristiche di "disaccoppiamento" permesse dall' IoC e AOP.

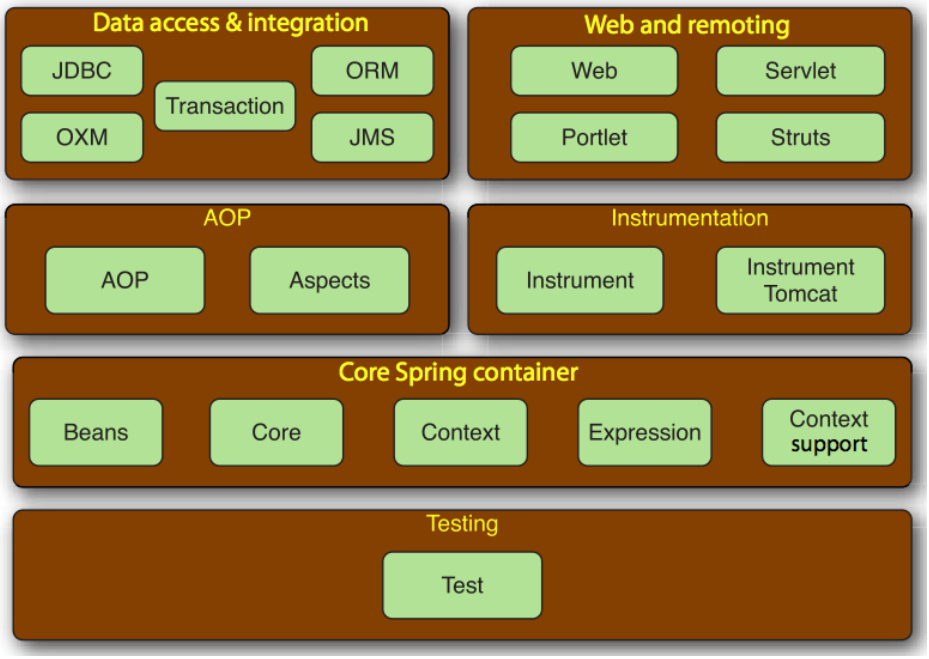

Fig. 3 : *La struttura a moduli di Spring Framework*

- **Modulo Core**

  Il modulo Core rappresenta la parte principe, la base su cui tutto il framework è costruito. In esso si trova Spring bean factory, la porzione di Spring che provvede alle funzionalità di Inversion of Control e Dependency Injection, ovvero di come i beans vengono creati, configurati e gestiti. Inoltre il modulo Context estende le funzionalità tipiche di un moderno framework, fra cui troviamo JNDI (Java Naming and Directory Interface),[7] EJB (Enterprise JavaBean), JMX (Java Management Extension),[8] I18n (internazionalizzazione)[9] ed il supporto agli eventi.

[7] en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface
[8] en.wikipedia.org/wiki/JMX
[9] en.wikipedia.org/wiki/I18n

- **Modulo AOP**
  Questo modulo provvede a supportare tutti gli aspetti della programmazione Aspect Oriented.

- **Modulo Data Access & Integration**
  Fornisce un livello di astrazione per l'accesso ai dati mediante tecnologie come JDBC o Hibernate. Grazie a questo modulo si riesce a nascondere le complessità delle API di accesso ai dati, semplificando e uniformando quelle che sono le problematiche legate alla gestione delle connessioni, transazioni ed eccezioni. Notevole attenzione è stata data all'integrazione con i principali ORM in circolazione (JPA, JDO, iBatis), nonché fornire un supporto per le implementazioni Object/XML Mapper come JAXB[10], Castor[11] e XMLBeans[12].

- **Modulo Web and Remoting**
  È attraverso il modulo Web and Remoting che questo framework mette a disposizione un implementazione del pattern Model-View Controller (MVC),[13] con Spring MVC Framework. Oltre alle web-application, offre metodi per costruire applicazioni che interagiscono tra loro.

- **Testing**
  Riconoscendo l'importanza dei test sul codice scritto, Spring mette a disposizione un ambiente molto potente per il test, grazie anche alla sua integrazione con JUnit e TestNG e alla presenza di Mock Object per il testing del codice in isolamento.

[10] en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding
[11] en.wikipedia.org/wiki/Castor_(software)
[12] en.wikipedia.org/wiki/XMLBeans
[13] en.wikipedia.org/wiki/Model-view-controller

3 Web Application

3.1 Il Pattern MVC

Spesso, quando si implementa un'applicazione Web utilizzando la tecnologia Java, a causa dell'utilizzo di un protocollo "povero" come HTTP si tende ad usare in modo indiscriminato Servlet[14] e pagine JSP(JavaServer Pages).[15] Se questo da un lato dà la possibilità di una maggiore flessibilità e libertà di programmazione, dall'altro può andare contro alle norme della qualità del software, come riusabilità, portabilità e manutenibilità.

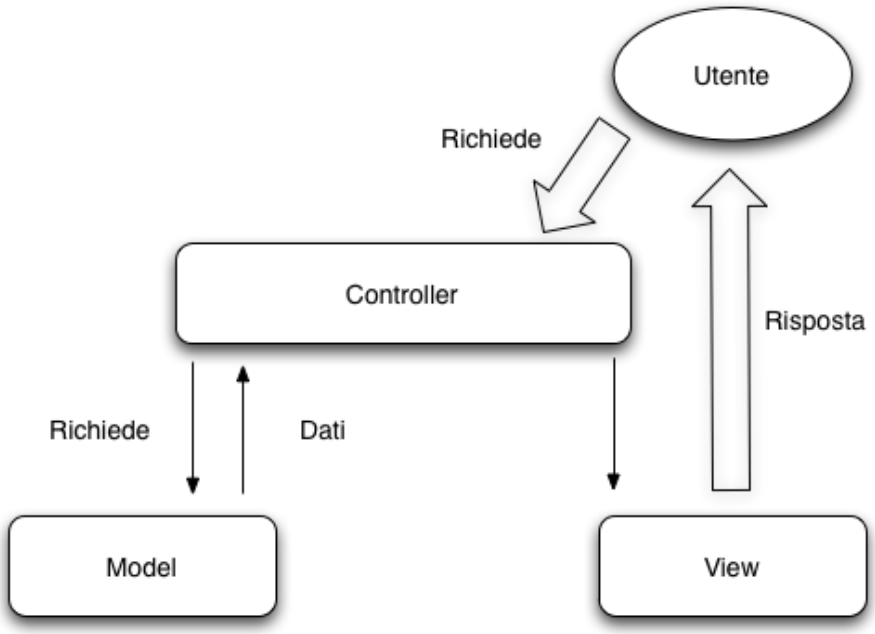

Fig. 4 : *Schema di funzionamento del pattern MVC*

[14] en.wikipedia.org/wiki/Servlet
[15] en.wikipedia.org/wiki/JavaServer_Pages

Per sopperire a tali esigenze, si è designata un'architettura per le Web application chiamata Model-View-Controller.
Tale è stata introdotta per la prima volta da Trygve Reenskaug nel tardo 1970, ma diventerà una pietra miliare per l'architettura di tutte le future applicazioni Web-based, poichè riesce brillantemente a separare tutta la logica business e la rappresentazione delle informazioni dalle interazioni utente con quest'ultime.
Tale modello è costituito sostanzialmente da 3 strati: la parte business formata da Model e Controller, e l'interfaccia utente (View). Nel dettaglio:

- **Model**
  Implementa la logica di business, fornendo i metodi utili per l'accesso ai dati dell'applicazioneed ha la responsabilità della gestione del database.

- **Controller**
  Implementa la logica di controllo, riceve i comandi dell'utente (in genere attraverso lo strato View) e li attua modificando lo stato degli altri due componenti.

- **View**
  Implementa la logica di presentazione, interpretando i risultati ottenuti da una dal Model e gestendo l'interazione con gli utenti.

È importante far notare che sia lo strato View che lo strato Controller dipendono direttamente dal Model, il quale non dipende dagli altri. Questo è uno dei fattori più importanti di questa architettura, poichè permette al modello di essere implementato e testato *indipendentemente* dallo strato di visualizzazione.
Grazie a questi accorgimenti, molteplici sono i benefici derivati: grazie a tale approccio, è possibile implementare viste multiple, permettendo l'esposizione degli stessi dati allo stesso instante in modi diversi. Si riduce la complessità di sviluppo, e conseguentemente il costo di aggiornamento, permettendo la manutenzione ad uno degli strati senza coinvolgerne altri.

3.2 Spring MVC

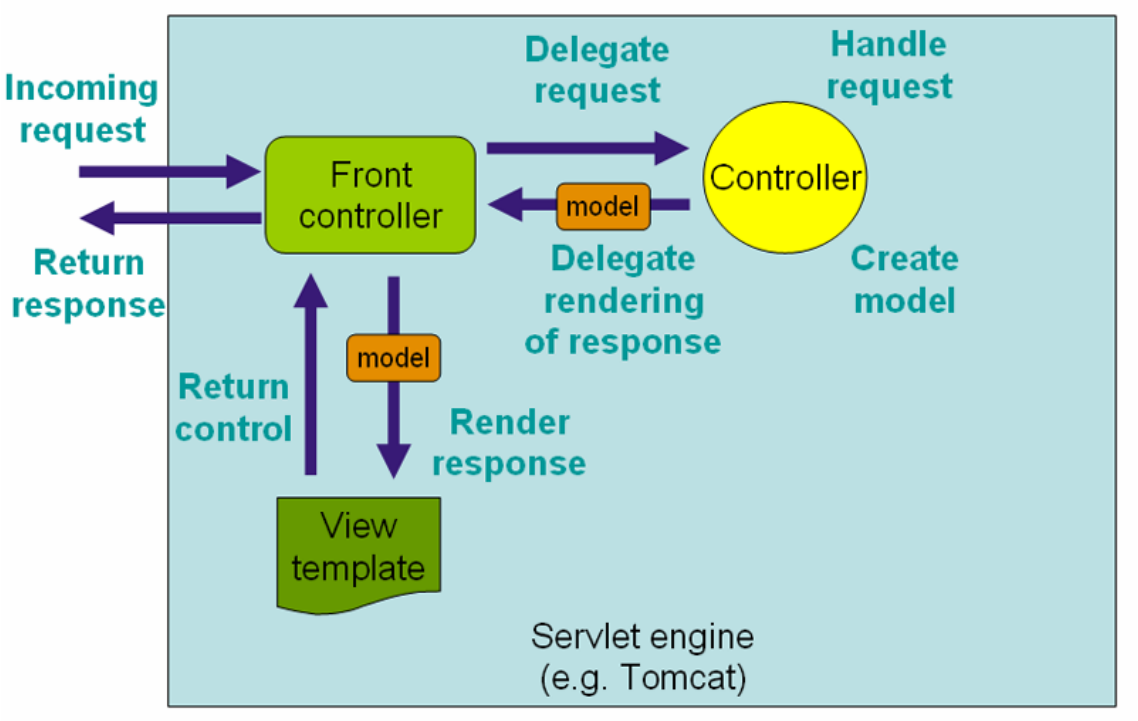

Fig. 5 : *Il flusso di richieste in Spring MVC*

In Spring, il modello Model-View Controller è progettato attorno ad un front controller chiamato *DispatcherServlet*, unica vera variante rispetto al modello "classico", il quale ha il compito di inoltrare le richieste alle varie interfacce attive durante una fase `HTTP request`.
I moduli più importanti di Spring MVC sono:

- **HandlerMapping**
  Seleziona quale Controller dovrà gestire la richiesta in arrivo secondo i suoi attributi.

- **HandlerAdapter**
  Permette l'esecuzione della logica del Controller selezionato dal HandlerMap-

ping.

- **Controller**
  Nel mezzo tra Model e View, amministra le richieste in arrivo e reindirizza le risposte al View.

- **View**
  È responsabile di far dialogare l'applicazione con l'utente, gestendo le risposte che possono provenire dal Controller direttamente o dal Model.

3.3 Gestire una richiesta con Spring MVC

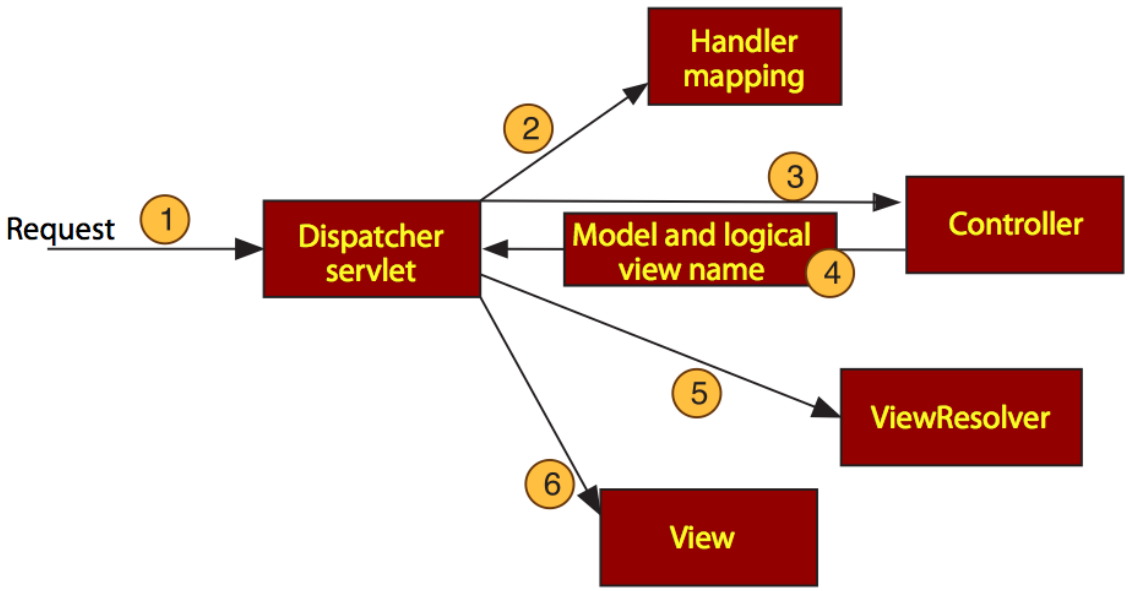

Fig. 6 : *Gestione di una richiesta*

Ogni volta che un utente clicca su un link o dà il comando di invio in un determinato form, effettua una richiesta alla web application.
Con Spring MVC, tale richiesta viene letta inizialmente dal DispatcherServlet, il quale, una volta averla ricevuta, delega la responsabilità della sua gestione al Controller

appropriato. Solitamente in una web application esso non è unico, così il DispatcherServlet si affida all'Handler Mapping per riconoscere il destinatario, secondo una mappa che associa l'URL della richiesta ai controller. Una volta identificato, il DispatcherServlet gli invia la richiesta.
Arrivata a destinazione, la richiesta viene eseguita interagendo con il Model, producendo spesso informazioni che dovranno poi essere visualizzate dall'utente. Quindi il controller restituirà tali informazioni, sotto forma di un oggetto ModelAndView, al DispatcherServlet, che le inoltra prima al ViewResolver, il quale sceglierà la pagina corretta, poi allo strato View che restituirà tali informazioni all'utente in un formato user-friendly (come HTML).

4 Spring Web Flow

4.1 Introduzione

Spesso nello sviluppo di web application con un'elevata interazione con l'utente, una delle maggiori difficoltà è gestire la linearità dell'applicazione, a causa dell'elevato numero di pagine che interagiscono tra loro.
In framework che implementano l'architettura MVC si è visto come si possa coordinare gli strati di controllo, vista e modello, ma non offre nessun tipo di supporto per quanto riguarda il flusso operativo dell'applicazione.

Fig. 7 : *Logo Spring Web Flow*

Spring Web Flow è un'estensione di Spring MVC la quale sopperisce a questa mancanza, mettendo a disposizione un'infrastruttura mirata alla gestione centralizzata e uniforme di questi aspetti.
I punti chiave su cui si basa tutta la filosofia di Spring Web Flow sono:

- C'è sempre un punto di inizio e uno di fine, e questi sono ben definiti.
- Una volta entrati nel flusso, l'utente deve compiere un insieme di passi in un ordine ben definito.
- I cambiamenti e le scelte apportati durante il flusso non saranno definitivi fino all'ultimo step.

- Una volta terminata tale procedura, non sarà più possibile rieseguire la transazione accidentalmente.

4.2 Definizione di flusso applicativo

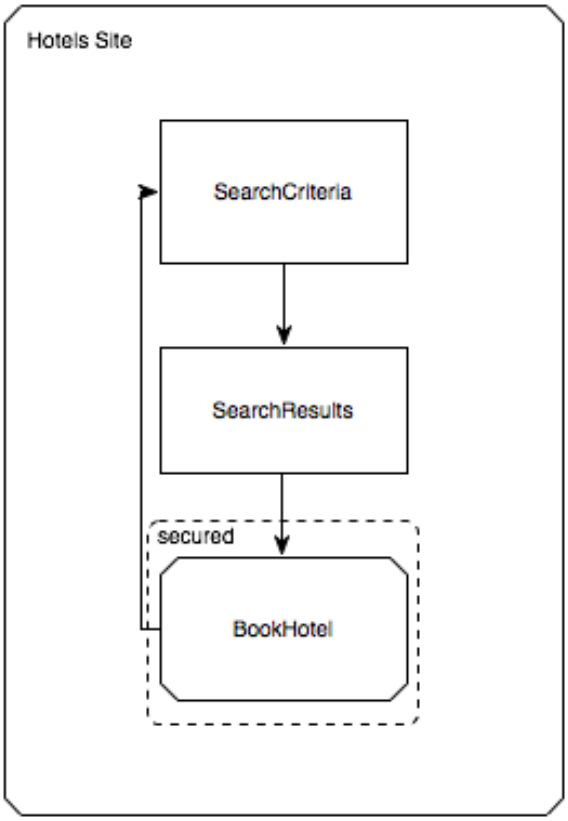

Fig. 8 : *Flusso per il processo di prenotazione in un hotel*

Un "flusso" generalmente è una sequenza di passi atti a guidare l'utente attraverso l'esecuzione di un determinato compito. In esso si possono verificare più richieste HTTP, può avere delle transazioni, essere riusabile ma sopratutto può essere dinamico.
In Spring Web Flow, un flusso è un insieme di passi chiamato "states", i quali pos-

sono essere eseguiti in diversi contesti. Entrando in uno "state" solitamente si ha come risposta una pagina che deve essere visualizzata dall'utente, con la quale può interagire.
L'interazione con la pagina genera degli eventi, che innescano delle transizioni (transitions) ad altri stati, con la logica conseguenza di avere un navigazione tra pagine.

4.3 Concetti di base

In Spring Web Flow la definizione di flusso è normalmente codificata in un file XML, dove il framework è in grado di creare un'istanza dell'interfaccia `FlowDefinition`. Tre sono gli elementi primari: States (o stati), Transitions (o transizioni) e le variabili.

4.3.1 States

SWF definisce 5 diversi tipi di States, mettendo a disposizione ogni tipo di funzionalita per una web application interattiva

- **Action**
  Gli Action sono gli States dove la logica dell'applicazione prende atto, permettendo di eseguire del codice applicativo nel momento in cui vi si accede, ma non implica nessuna interazione con l'utente. In genere, esse sono definite dal linguaggio SpEL (Spring Expression Language),[16] ma è possibile utilizzare Unified EL (Expression Language)[17] oppure OGNL (Object-Graph Navigation Language).[18]

Esempio di Action state

```
1 <action-state id="saveOrder">
```

[16] Introdotto dalla versione 2.1.0, SWF usa di default SpEL
[17] en.wikipedia.org/wiki/Unified_Expression_Language
[18] en.wikipedia.org/wiki/OGNL

```
        <evaluate expression="pizza.saveOrder(order)" />
        <transition to="thankYou" />
</action-state>
```

Com'è possibile vedere dall'esempio riportato, questa Action richiama il bean di ID `pizza` e invoca il suo metodo `saveOrder()`.
Spesso però è necessario eseguire delle operazioni prima altri tipi di stato, come caricare dal database dei dati da visualizzare nella pagina che verrà visualizzata in un *viewstate*. In questi casi Spring permette, in determinati punti dell'esecuzione dello state, di eseguire del codice sempre definito dal linguaggio SpEL.
Questi punti sono:

Tag	Descrizione
<on-start>	ogni volta che il flusso comincia la sua esecuzione
<on-entry>	quando si entrando in uno stato
<on-exit>	quando si sta uscendo da uno stato
<on-end>	quando uno stato termina la sua esecuzione (non permesso nell'endstate)
<on-render>:	prima della visualizzazione della pagina
on-transition	durante una transizione tra due stati

Tranne che per l'ultimo punto, dove verrà definita tra l'elemento <transition>, è stato definito il tag dove verrà inclusa l'azione da eseguire.

- **Decision**
  È possibile designare un flusso puramente lineare, passando da uno state ad un altro senza mai poter prendere strade alternative. Ma spesso si ha la necessità di farlo variare, per renderlo dinamico o per poter gestire gli eventi riscontrati.
  I Decision State permettono al flusso di porsi ad un bivio e, dopo la valutazione

di un'espressione booleana (grazie ad `evaluate`), sarà il flusso a scegliere con quale delle transizioni scegliere.

Esempio di Decision state

```
<decision-state id="checkDeliveryArea">
        <if test="pizza.checkDeliveryArea(customer.zipCode)"
                then="addCustomer"
                else="deliveryWarning" />
</decision-state>
```

Il vero cuore del decision state è l'elemento `if`. Una volta valutata l'espressione, il flusso converge ad uno dei due stati; se l'espressione risulta `true` converge allo stato identificato dal `then`, altrimenti da `else`. È evidente la somiglianza con i più comuni linguaggi di programmazione per quanto riguarda il costrutto `if`.

- **Subflow**
  Subflow state permette di suddividere il flusso in sotto-flussi, permettendo di richiamarne uno nuovo da uno esistente, in maniera del tutto analoga a quando si richiama un metodo da un altro metodo.

Esempio di Subflow

```
<subflow-state id="order" subflow="pizza/order">
        <input name="order" value="order"/>
        <transition on="orderCreated" to="payment" />
</subflow-state>
```

L'elemento `input` passa l'oggetto `order` al sottoflusso chiamato `order` e collocato nella cartella `pizza`. Nel momento in cui nel sottoflusso si verificherà un evento di ID `orderCreated`, si avrà un passaggio al flusso originale con una transizione allo stato `payment`, state appartenente al sottoflusso invocato.

- **End**
  End state è l'ultimo stato a cui il flusso arriva e nel quale esso in genere termina.

  Esempio di End state

```
1 <end-state id="customerReady" />
```

  Quando lo si raggiunge, ciò che veramente succede dipende da vari fattori:

  – Se il flusso che sta per terminare è un `subflow`, il controllo ritorna al flusso originario da cui il sottoflusso è stato chiamato.
  – Se il flusso ha una sua pagina da visualizzare, tale verrà visualizzata. Essa può essere una pagina esterna al flusso, raggiunta tramite il prefisso `externalRedirect:`, oppure può essere la prima di un nuovo flusso; in tal caso il prefisso sarà `flowRedirect:`
  – Nel caso non ci siano attributi `view` specificati, esso semplicemente finisce, lasciando al browser l'URL iniziale dov'era stato invocato dal DispatcherServlet. Senza alcun flusso attivo, ne verrà creata una nuova istanza, e si ricomincerà dal primo state.

  È importante far notare che è possibile avere più di un End state, in merito al percorso effettuato dal flusso per raggiungerlo; in questo modo, sarà possibile richiamare transizioni a pagine o a flussi diversi a seconda dell'ID raggiunto.

- **View**
  Lo state View è solitamente lo state più utilizzato in un flusso, e viene utilizzato per mostrare le informazioni provenienti dal flusso all'utente e permettergli interagire con esse.

  Esempio di View state

```
1 <view-state id="welcome" view="greeting" />
```

L'attributo `view` presente indirizza il flusso alla visualizzazione della pagina da lui definita, chiamata `greeting`. In questo caso l'applicazione cercherà un qualsiasi file così chiamato e lo inoltrerà allo strato *view*, indipendentemente dall'estensione, permettendo un'elevata compatibilità sia con linguaggi di visualizzazione di pagine web dinamiche (come JSP o ASP) sia con tecnologie come AJAX.
Nel caso non sia definita alcuna proprietà `view`, il flusso cercherà il file con lo stesso nome associato all'ID dello state.

Esempio di View state con form

```
1 <view-state id="takePayment" model="flowScope.paymentDetails"/>
```

Nel caso in cui sia presente un form nella pagina per l'immissione di dati, attraverso l'attributo `model` si potrà descrivere quale sarà la variabile a cui i dati immessi potranno essere associati.

4.3.2 Transitions

Com'è possibile intuire, Transitions (o transizioni) connettono i diversi state nel flusso. Ad eccezione dell' `endstate`, essi hanno almeno un elemento transition che permette al flusso continuare la sua esecuzione una volta che lo state attuale è terminato.
Le transizioni vengono definite dall'elemento <`tranistion`> e vengono definite come figli dell'elemento state. Nella sua forma più semplice, identifica il prossimo step nella successione del flusso.

```
1 <transition to="customerReady" />
```

Quando nella dichiarazione è presente solamente l'attributo `to`, essa sarà la transizione, verso lo state di ID definito da tale attributo, di default, da essere intrapresa se nessun altra è applicabile.
Molto più comune è una transition in cui, oltre all'elemento `to`, viene definito anche

l'elemento `on`; questo accompagna l'ID dell'evento che dovrà verificarsi perché la transizione abbia atto.

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

Nel caso in cui si voglia effettuare un passaggio da uno state ad un altro in risposta ad una eccezione, è possibile sostituire l'elemento `on` con `on-exception` seguita dall'eccezione che deve manifestarsi per effettuare tale passaggio.

4.3.3 Variabili e contesti di visibilità

Nel flusso è possibile istanziare e dichiarare variabili in modi diversi. Uno di questi è attraverso l'elemento `var`.

Esempio di creazione di una variabile

```
<var name="customer" class="com.pizza.Customer"/>
```

Oppure è possibile, attraverso l'attributo `evaluate` o `set`, creare ed istanziare una variabile in seguito alla risoluzione di un'espressione (SpEL).

Esempio di creazione di una variabile con evaluate

```
<evaluate result="viewScope.toppingsList"
        expression="T(com.pizza.Topping).asList()" />
```

Esempio di creazione di una variabile con set

```
<set name="flowScope.pizza"
        value="new com.pizza.Pizza()" />
```

Per riguarda la visibilità, SWF definisce 5 contesti, che verranno scelti precedendo il nome della variabile dal suo *scope*.

- **conversationscope**
  Una variabile verrà creata al momento della creazione del flusso principale e distrutta quando questo termina. Verrà vista da tutto il flusso e dai suoi sottoflussi.

- **flowscope**
  Creata quando il flusso comincia il suo percorso e distrutta al momento della sua terminazione. Sarà visibile solamente al flusso da cui è stata creata.

- **requestscope**
  Istanziata durante il flusso, terminerà con esso. È visibile in tutto il flusso.

- **flashscope**
  Creata quando il flusso comincia e distrutta al momento della sua terminazione. Il suo valore verrà cancellato ad ogni `view` state raggiunto.

- **viewscope**
  Creata quando si entra in un `view` state e termina quando si transita in un altro stato, è visibile solo dallo stato stesso.

5 Un caso reale: ColorProject

Per approfondire lo studio di tale framework si è realizzato un piccolo modulo di gestione dei clienti per ColorProject SRL, azienda che da anni opera nel mondo dei colori sublimatici utilizzando la tecnologia SWF.
Durante l'anno accademico in corso si è implementata una piccola web application come progetto didattico del corso di Basi di Dati; essa si interfaccia ad un database per la gestione di tutte le operazione logistiche, contabili e tecniche che l'azienda svolge quali: forniture ingredienti, tracciatura ordini e fatturazione.
Quindi ci si limiterà ad integrare tale applicazione con un modulo creato grazie a Spring Web Flow che gestisca le credenziali dei clienti presenti nel database, che ne permetta ricerca ed eventualmente l'immissione di un nuovo nominativo.

5.1 Struttura dell'applicazione

È necessario, innanzitutto, creare un flusso che l'applicazione dovrà seguire.
Per prima cosa sarà necessario ricercare un eventuale cliente: per fare ciò dovremo mettere a disposizione dell'utente un form dove sia possibile inserire i dati (Nome dell'azienda o del cliente, Partita Iva o Codice Fiscale), che permettano di effettuare la ricerca.
Nel caso la ricerca dia esito positivo, l'applicazione avrà il compito di far visualizzare tutte le credenziali salvate nel database e di mettere l'utente in condizione di effettuare altre operazioni, come effettuare una nuova ricerca o modificare tali credenziali. La funzione di cancellazione dei nominativi non è stata presa in considerazione poichè, per scelte aziendali, è stato deciso di mantere nel database tutte le anagrafiche dei clienti.
Nel caso la ricerca dia esito negativo, si dovrà portare a conoscenza l'utente di tale situazione, permettendogli di rieffettuare la ricerca oppure di aggiungere un nuovo cliente.

5.2 La classi Java

Per poter utilizzare appieno la potenza di Spring Web Flow, è stato necessario definire un bean `Cliente`, con il quale poter definire agevolmente tutti i valori ottenibili dal form o dal database. Inoltre si è implementata una classe, `ClienteManager`, la quale estende le funzionalità del bean ed è stata utilizzata una classe, `FormAction`, implementata di default da Spring.

5.2.1 Classe Cliente

Esso raccoglie in sè il nome del cliente (sia esso una persona fisica o legale), la sua partita iva, un recapito telefonico, un indirizzo e-mail e la sede Legale. Sono stati implementati i metodi `set` e `get` per la gestione dell'anagrafica.

Firma dei metodi getNome e setNome

```
1 public String getNome();
2 public void setNome(String nome);
```

Firma dei metodi getSedeleg e setsedeleg

```
1 public String getSedeleg();
2 public void setSedeleg(String sedeLeg);
```

Firma dei metodi getPiva e setPiva

```
1 public String getPiva();
2 public void setPiva(String piva);
```

5.2.2 Classe ClienteManager

Tale classe estende le funzionalità del bean `Cliente`.

Firma del metodo exists

```
1 public boolean exists(String nome, String PIva);
```

Grazie ai suoi metodi è possibile ricercare l'esistenza di un dato cliente sia per nome che per partita Iva(attraverso il metodo booleano `exists`).

Firma dei metodi getCliente

```
public Cliente getCliente(String nome);
public Cliente getCliente(String PIva);
```

Con il metodo `getCliente` è possibile ricercare un set di clienti il cui nome è, o assomiglia, al nome dell'argomento. Del tutto analoga sarà la ricerca effettuata tramite la partita Iva.

Firma del metodo createCliente

```
public Cliente createCliente(String nome,
                String PIva , String sedeleg ,
                String telefono, String fax, String mail);
```

Si ha la creazione di un nuovo cliente grazie al metodo `createCliente` che crea un nuovo cliente con le date credenziali.
La modifica dell'anagrafica è permessa dal metodo `modifyCliente`, che ricerca e modifica un cliente con partita Iva pari a `PIva`

Firma del metodo modifyCliente

```
public Cliente modifyCliente(String PIva,
                String nome, String newPIva, String sedeleg,
                String telefono, String fax, String mail);
```

5.2.3 Classe FormAction

La classe `formAction` è uno Spring bean che implementa l'interfaccia `org.springframework.action.FormAction`, ed è una delle funzionalità di default più importanti di Spring Web Flow.
Grazie a questa classe, è possibile collegare i campi del form a della variabili `flowscope`.
Attraverso il metodo `setupForm`, prepariamo Spring ad occuparsi dei valori immessi

nel form; con il metodo `bindAndValidate` imponiamo di controllare la correttezza di tali valori (grazie ad un `validator` implementato nella classe `ClienteManager`) e di salvarli in un bean.
Nonostante sia di default implementata, tale classe va importata in Spring Web Flow come un bean tra i file di configurazione XML.

Configurazione formAction

```
<bean id="formAction"
        class="org.springframework.webflow.action.FormAction">
        <property name="formObjectClass"
                value="com.gestcolor.app.ClienteManager"/>
        <property name="validator">
                <bean class="com.gestcolor.app.ClienteManagerValidator"/>
        </property>
</bean>
```

É definito nel bean `formAction` sia il `validator`, sia `formObjectClass`, ovvero la classe a cui Spring si occuperà di associare i valori prelevati dal form una volta che sono stati approvati dal `validator`.

5.3 Flusso applicativo

Data la struttura indicata in Figura 9, il primo passo sarà acquisire tramite un form i dati del cliente. Tale acquisizione verrà fatta da una pagina JSP chiamata `dataCliente`. Al termine dell'immissione dei dati, l'utente tramite il tasto "Conferma" creerà un evento (chiamato `submit`) che il flusso interpreterà cambiando state da `dataCliente` a `searchCliente`.
In questo action state, l'applicazione controllerà nel database la presenza del nominativo precedentemente immesso. In caso di esito positivo, il flusso transiterà nello state `showCliente`, che permetterà di visualizzare i dati trovati, altrimenti transiterà nello stato `notFoundCliente`.
Qui il flusso permetterà di effettuare una nuova ricerca, ritornando nello state

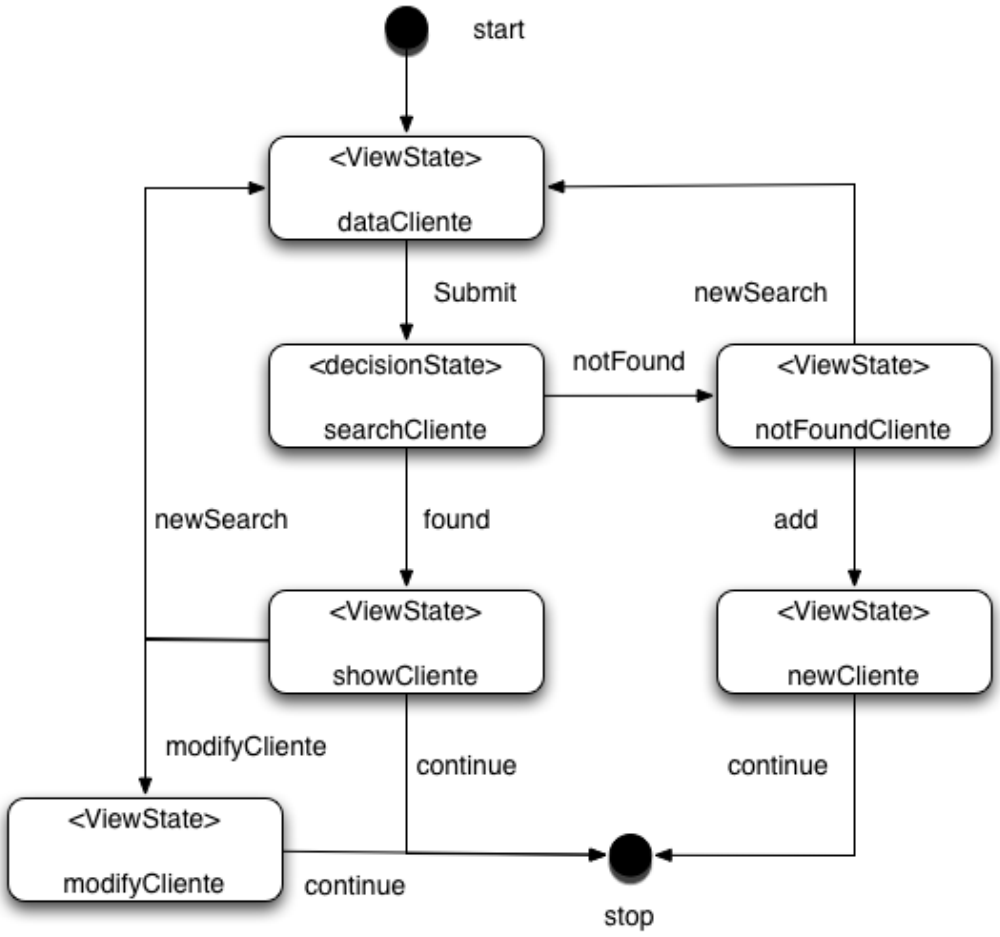

Fig. 9 : *Flusso dell'applicazione commisionata da ColorProject*

`dataCliente` o l'aggiunta di nuove credenziali nello state `newCliente`. Una volta terminate queste operazioni, il flusso terminerà permettendo una nuova istanza del flusso (in questo caso di studio), oppure il proseguire della web-application.
In questo flusso, da qualsiasi state di tipo view è possibile, attraverso l'evento `cancel` raggiungere l'end state del flusso con la conseguente terminazione, ma, per chiarezza dell'immagine, tali connessioni sono state volutamente omesse.
Per permettere a Spring Web Flow di interagire con i dati che verranno immessi nel form, verranno utilizzate due classi Java, rispettivamente `Cliente` e `ClienteManager`.

Insieme all'interfaccia `FormAction` implementata di default da SWF, queste classi verranno utilizzate come Spring-bean nel flusso che verrà definito.

5.3.1 View State dataCliente

Il primo passo che il flusso incontra è lo state di tipo view dataCliente. In questo stato sono due i possibili eventi che il flusso dovrà gestire: `submit` che porterà alla transizione verso lo stato `searchCliente` e l'evento `cancel` che porta al termine del flusso.

View State dataCliente

```
<view-state id="dataCliente">
        <render-action>
                <action bean="formAction" method="setupForm">
        </render-action>
        <transition on="submit" to="searchCliente">
                <action bean="formAction" method="bindAndValidate">
        </transition>
        <transition on="cancel" to="endFlow"/>
</view-state>
```

Grazie alla classe `FormAction` implementata da Spring Web Flow, utilizziamo il metodo `setupForm` che permette a Spring di trattare tutti i campi del form come variabili di tipo `flowscope`.
Nel momento in cui l'utente genera l'evento `submit`, il metodo `bindAndVallidate` della classe `formAction` collegherà tali valori a quelli presenti nel bean.

5.3.2 Decision State searchCliente

Nel momento in cui l'utente ha inserito i dati relativi al cliente da cercare, il flusso transiterà al `decision state searchCliente`. Il compito di questo stato è semplicemente di assicurarsi che esista nel database un cliente che corrisponda ai dati

precedentemente inseriti.
Questo controllo è effettuato dal metodo booleano `exists(String nome, String PIva)` della classe `ClienteManager` inglobato da `formAction`.

Action State searchCliente

```
<decision-state id="searchCliente">
        <if test="ClienteManager.exists(flowscope.nome,flowscope.piva)"
                then="showCliente"
                else="notFoundCliente" />
</decision-state>
```

5.3.3 View State notFoundCliente

Si arriva in questo state nel caso in cui non esista il cliente con le generalità precedentemente immesse. A tal proposito, sarà possibile effettuare una nuova ricerca oppure procedere per l'immissione di un nuovo nominativo.

View State notFoundCliente

```
<view-state id="notFoundCliente">
        <transition on="search" to="dataCliente"/>
        <transition on="new" to="newCliente"/>
        <transition on="cancel" to="endFlow"/>
</view-state>
```

5.3.4 View State newCliente

Se dovesse essere necessario immettere nel database le generalità per un nuovo cliente, l'utente si troverà ad interagire con una pagina del tutto simile alla pagina dataCliente, con la differenza che dovrà inserire tutti i dati di un cliente (nome, partita iva, numero di telefono, sede legale, etc.).
Data la filosofia di Spring, che permette di rendere definitivi i vari cambiamenti apportati solo alla fine del flusso, la vera aggiunta dei dati sul database si avranno

solo quando verrà lanciato l'evento `continue`, che permetterà al flusso di transitare verso `end state`, nonostante si arrivi alla fine anche grazie all'evento `cancel`.

View State newCliente

```
<view-state id="newCliente">
        <render-action>
                <action bean="formAction" method="setupForm">
        </render-action>
        <transition on="continue" to="endFlow">
                <evaluate expression="ClienteManager.createCliente(
                        flowscope.nome, flowscope.piva,
                        flowscope.sedeleg, flowscope.telefono,
                        flowscope.fax, flowscope.mail)"/>
        </transition>
        <transition on="cancel" to="endFlow"/>
</view-state>
```

5.3.5 View State showCliente

Nel caso in cui si abbia un riscontro dei dati inseriti nel form dall'utente con i dati trovati nel database, i dati verranno semplicemente mostrati a video.

View State showCliente

```
<view-state id="showCliente">
        <render-action>
                <bean-action bean="ClienteManager" method="getCliente">
                <method-arguments>
                        <argument expressions="flowscope.nome" />
                </method-arguments>
        </render-action>
        <transition on="continue" to="endFlow" />
        <transition on="cancel" to="endFlow"/>
```

```
</view-state>
```

In questo caso, prima del caricamento della pagina l'applicazione si incarica di lancare il metodo `getCliente(long PIva)` che restituisce un oggetto di tipo `Cliente` con tutte le generalità.

5.3.6 View State modifyCliente

L'ultimo view state che il flusso può incontrare è quello relativo alla modifica dei nominativi di un dato cliente. In questo step, il flusso dovrà prima far visualizzare i valori da modificare (in maniera del tutto analoga a `showCliente`) e permetterne l'inserimento dei nuovi valori (come `newCliente`).

View State modifyCliente

```
<view-state id="showCliente">
        <render-action>
                <action bean="formAction" method="setupForm">
                <bean-action bean="ClienteManager" method="getCliente">
                <method-arguments>
                        <argument expressions="flowscope.piva" />
                </method-arguments>
        </render-action>
        <transition on="continue" to="endFlow">
                <evaluate expression="ClienteManager.modifyCliente(
                                flowscope.piva,flowscope.nome,
                                flowscope.newpiva, flowscope.sedeleg,
                                flowscope.telefono, flowscope.fax,
                                flowscope.mail)"/>
        </transition>
        <transition on="cancel" to="endFlow"/>
</view-state>
```

Però non tutti i valori visualizzati saranno sempre modificati: sarà la pagina stessa che inserirà nelle caselle di testo del form i nominativi da modificare e controllerà le eventuali variazioni da parte dell'utente; in questo caso il database verrà aggiornato, altrimenti non verrà eseguita alcuna modifica.

5.4 Configurazione dell'applicazione

5.4.1 Web.xml

Spring Web Flow è stata definita come un'estensione di Spring MVC, e pertanto su essa si basa; come tutte le applicazioni spring-based, le richieste passeranno attraverso il *DispatcherServlet*.
Come una qualsiasi JEE web application con architettura MVC, tutta la configurazione si trova nel file `web.xml`; quindi, dopo il reperimento delle librerie Spring Web Flow attraverso il sito ufficiale o Maven[19], configuriamo tale servlet.

```
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
<servlet>
        <servlet-name>clienteMVC</servlet-name>
        <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>
                1
        </load-on-startup>
</servlet>
<servlet-mapping>
```

[19] maven.apache.org/

```
16          <servlet-name>
17                  clienteMVC
18          </servlet-name>
19          <url-pattern>
20                  /flows/*
21          </url-pattern>
22  </servlet-mapping>
```

Inizialmente entra in gioco `ContextLoaderListener`, il quale inizializza Spring all'avvio dell'applicazione.
La servlet, com'è possibile vedere dalla classe importata, è un *DispatcherServlet* in tale occasione chiamato `clienteMVC`; esso controllerà tutti gli indirizzi delle richieste che perverranno alla web application, ma interverrà solo a quelli in cui comparirà */flows/* subito dopo al base URL dell'applicazione.

5.4.2 Flow Executor

Come il nome stesso può suggerire, `flowExecutor` guida l'esecuzione del flusso. Esso è responsabile della sua esecuzione e di gestire gli eventi che l'utente produce nel suo corso.

Creazione di un flowExecutor

```
1           <flow:flow-executor
2           id="flowExecutor"
3           flow-registry="flowRegistry" />
```

Sebbene sia il responsabile della creazione e dell'esecuzione del flusso, non lo è per quanto riguarda il reperimento della sua definizione. `FlowExecutor` necessita infatti di un registro, `flowRegistry`, il quale gli passa l'URL della posizione dove potrà trovare la definizione del flusso da eseguire.

5.4.3 Flow Registry

Il compito del `flowRegistry` è quello di rendere visibile il flusso al flowExecutor.

Flow Registry

```
<flow:flow-registry id="flowRegistry"
                    base-path="/WEB-INF/flows">
          <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

Grazie a questo tipo do configurazione, `flowRegistry` controllerà la presenza del flusso nella directory `/WEB-INF/flows`. Ogni file presente in tale cartella il cui nome del file termina per *-flow.xml* è considerato dal `flowRegistry` un possibile flusso. In maniera del tutto alternativa è possibile definire esplicitamente il file di definizione del flusso:

Definizione alternativa del file di definizione del flusso

```
<flow:flow-registry id="flowRegistry"
                    path="/WEB-INF/flows/cliente-flow.xml">
</flow:flow-registry>
```

.

5.4.4 FlowHandlerMapping

Per come è stata configurata l'architettura di Spring MVC, il `DispatcherServlet` inoltra direttamente ogni richiesta pervenutagli ai vari controller di competenza. Ma per quanto riguarda i flussi, abbiamo bisogno di `FlowHandlerMapping` che aiuta la servlet a riconoscere le varie richieste provenienti da essi e dirottarle verso Spring Web Flow.

FlowHandlerMapping

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
 <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

È possibile vedere che, nel bean, è presente una dipendenza verso `flowRegistry`; grazie a ciò, `FlowHandlerMapping` mappa gli URL che arrivano al `DispatcherServlet`

e riconosce le richieste provenienti dai flussi.
I parametri riconosciuti dal `FlowHandlerAdapter` sono:

Nome	Descrizione
`_flowId`	L'id del flusso, necessario per lanciare l'esecuzione di un nuovo flusso.
`_flowExecutionKey`	L'id di esecuzione del flusso, necessaria per riprendere o ricaricare l'esecuzione di un flusso esistente.
`_eventId`	L'id di un evento lanciato, utile per riprendere l'esecuzione di un flusso.

5.4.5 FlowHandlerAdapter

Quando `FlowHandlerMapping` riconosce una richiesta proveniente da un flusso, esso deve dirottarla verso `flowExecutor`.
Per questo compito esiste `FlowHandlerAdapter`, ed è l'equivalente di un controller per le richieste provenienti dai flussi, e come tale le elabora.

FlowHandlerAdapter

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
        <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Grazie alla dipendenza presente verso `flowExecutor`, esso funge da ponte tra il `DispatcherServlet` e `flowExecutor`: una volta che `FlowHandlerMapping` riconosce una richiesta da un flusso, esso la inoltra a `flowExecutor` che lo gestirà in merito alla richiesta ricevuta.

5.5 Le pagine JSP

L'unica maniera che l'utente ha per interagire con il flusso è quella data dagli state `view`, che permettono la visualizzazione e l'interazione con le informazioni prove-

niente dall'applicazione grazie a delle pagine JSP.
Queste pagine dovranno produrre delle richieste indirizzate al flusso, che verranno riconosciute dal `FlowHandlerMapping` e grazie al `FlowHandlerAdapter` passate dal `DispatcherServlet` al `flowExecutor` per la transizione da una state ad un altro.
Si prenda in esempio la pagina JSP `dataCliente.jsp`; essa contiene un form dove l'utente inserirà i dati per la ricerca del cliente. Al momento della conferma, l'utente premerà il tasto Invia, il quale produrrà l'evento `submit`, oppure il tasto Annulla, terminando l'esecuzione del flusso.
Da notare l'URL nell'immagine, il quale ha fatto una richiesta all'applicazione *gestcolor*, di ottenere il flusso `cliente-flow` posto nella cartella *flows*. L'applicazione pone il suffisso `?execution=e2s1`, dove e sta per `execution`, s per `step`. Ovvero, in questo esempio, l'ID dell'esecuzione del flusso è 2, step 1.

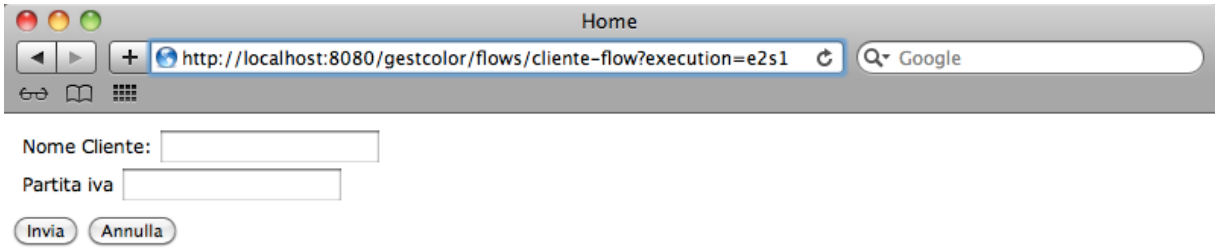

Fig. 10 : *Form per la ricerca di un cliente.*

Codice dei tasti Invia e Annulla

```
<input type="hidden" name="_flowExecutionKey"
                value="${flowExecutionKey}"/>
<input type="submit" class="button"
                name="_eventId_search" value="Invia"/>
<input type="submit" class="button"
```

```
6                    name="_eventId_cancel" value="Annulla"/>
```

Prima dei tasti, è stato inserito nel form un attributo nascosto, `_flowExecutionKey`, necessario per la ripresa dell'esecuzione del flusso.
L'attributo `_eventId` è invece codificato, assieme al suo valore `submit`, nella sintassi del nome del bottone, così come per l'attributo `cancel`.
Nel caso di pagine, come `showCliente`, dove non è necessario la presenza di form (suddetta pagina mostra soltanto i valori anagrafici dei clienti trovati dalla ricerca effettuata) è possibile inviare un evento all'applicazione attraverso un comune link HTML piuttosto di un form.

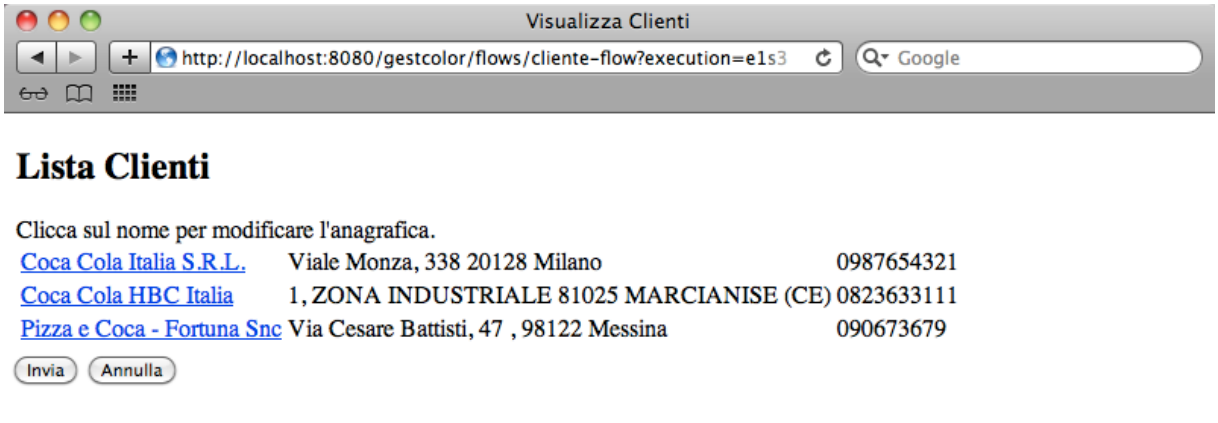

Fig. 11 : *Visualizzazione e selezione per la modifica dei dati.*

Codice del link per la modifica dell'anagrafica

```
1 <a href="cliente-flow?_flowExecutionKey=${flowExecutionKey}
2                  &_eventId=modify&piva=${ClienteManager.piva}">
3         ${ClienteManager.nome}
4 </a>
```

In questo caso si lancia, nel momento in cui l'utente preme il link, un evento all'applicazione del tipo `modify`, che porterà il flusso allo state `modifyCliente`. Questo state richiede l'attributo `ClienteManager.piva` per la visualizzazione dei

dati da modificare del cliente scelto; anche quest'ultimo lo si trova nel URL della richiesta.

6 Conclusioni

In questa tesi si è voluto studiare Spring Web Flow partendo dal framework Spring, di cui ne è una parte, per comprendere meglio i fattori che lo hanno reso così famoso come la Dependency Injection e l'Aspect Oriented Programming.
Si è studiato il pattern MVC, prima a livello puramente teorico, riportandone i punti di forza e gli strati che lo compongono, ed in seguito come questo framework ha implementato tale architettura, mostrando come una richiesta viene gestita da Spring MVC.
Nel quarto capito s'è affrontato lo studio di Spring Framework, partendo inizialmente dalla definizione di flusso applicativo nel suo insieme, soffermandoci su come SWF ne codifica ogni singolo passo e le interazioni che ne permettono un avanzamento.
Si termina con un piccolo esempio pratico, in cui si mette all'opera tale framework mostrando che è possibile, grazie ad una classe precedentemente creata (classe `Cliente`)ed a poche righe di codice, implementare un piccolo modulo completamente funzionante.
Per dimostrarne la versatilità, l'applicazione di gestione clienti qui riportata è stata propositamente implementata senza l'utilizzo di alcun controller. Il motivo principale di questa scelta è legata all'apprendimento della gestione delle richieste provenienti da un flusso, permettendo di studiare ed usare delle funzionalità che, con un controller, non sarebbero state utilizzate.
Nonostante ciò è evidente che, un'applicazione implementata con l'ausilio di un controller sarebbe stato il modo ideale e più corretto per l'implementazione di tale modulo, portando ad una vera separazione dei moduli che in questa applicazione non è avvenuta.
Spring Web Flow è un potente mezzo che permette in modo semplice e veloce sia di sopperire alle limitazioni dei più comuni framework MVC, come la gestione della logica di navigazione in una web application, sia il riutilizzo di codice con il minimo sforzo.

Riferimenti bibliografici

[1] Craig Walls Spring in Action, third edition ed. Manning, 2011;

[2] Erwin Vervaet The Definitive Guide to Spring Web Flow ed. Apress, 2008;

[3] Atzeni, Ceri, Paraboschi, Torlone Basi di Dati Modelli e Linguaggi di Programmazione, ed. McGraw-Hill;

[4] Elmasri, Navathe Sistemi di Basi di Dati ed. Pearson, 2011;

[5] Wikipedia: `http://www.wikipedia.com`;

[6] Java: `http://www.java.com`;

[7] Spring Web-Flow `http://www.springsource.org/spring-web-flow`;

[8] Spring Web-Flow Reference Guide: `http://static.springsource.org/spring-webflow/docs/2.3.x/reference/html/index.html`;

[9] Maven: `http://maven.apache.org/`;

[10] Spring Tool Suite: `http://www.springsource.org/sts`;

[11] OmniGraffle: `http://www.omnigraffle.com`;