La tecnologia JavaServer Faces offre un set base di componenti UI riutilizzabili che permettono una costruzione veloce e semplice di interfacce utente per le applicazioni web.

Questi componenti sono per la maggior parte associati uno a uno a elementi di HTML4 ma spesso un’applicazione richiede un componente che abbia funzionalità aggiuntive o componenti completamente nuovi.

La tecnologia JavaServer Faces permette l’estensione dei componenti standard per incrementare le loro funzionalità o per creare componenti personalizzati.

E’stato costruito un ricco campionario di librerie fornite da terze parti per questa capacità di estensibilità ma non verranno analizzate nel libro.

Si consiglia una ricerca web su "JSF Component Libraries" come buon punto di partenza per imparare di più su questo importante aspetto dell’utilizzo della tecnologia JavaServer Faces.

In aggiunta alla ‘estensione di funzionalità dei componenti standard, chi scrive un componente potrebbe voler dare modo all’autore di una pagina di cambiare la visualizzazione del componente sulla stessa o cambiare il comportamento in fase di ascolto (listener behavior).

Oppure chi scrive il componente potrebbe voler renderizzare il componente per differenti tipi di dispositivi client come tablets o smartphones invece che per il desktop di un pc.

Grazie all’architettura flessibile di JavaServer Faces lo scrittore del componente può separare la definizione del comportamento dalla rappresentazione delegando il rendering a un renderer separato.

In questo modo può definire il comportamento di un componente personalizzato una volta ma creare diversi renderers, ognuno dei quali definisce un diverso modo di renderizzare un componente su un particolare tipo di dispositivo.

Un *javax.faces.component.UIComponent* è una classe Java responsabile per la rappresentazione di un pezzo auto-contenuto (self-contained) dell’interfaccia utente durante il ciclo vitale di richiesta-processamento.

Essa è destinata a rappresentare il significato del componente, mentre la rappresentazione grafica è delegata a *javax.faces.render.Renderer*.

Possono esserci più istanze della stessa classe *UIComponent* in ogni view JavaServer Faces, come possono esserci più istanze di ogni classe Java in ogni programma Java.

La tecnologia JavaServer Faces fornisce l’abilità di creare componenti personalizzati estendendo la classe *UIComponent*, che è la classe base di ogni componente UI standard.

Un componente personalizzato può essere usato in qualunque posto in cui un normale componente può essere usato, come ad esempio all’interno di un componente composito.

Un componente *UIComponent* è identificato da due nomi: *component-family* specifica l’obiettivo generale del componente (input o output ad esempio) e *component-type* indica l’obiettivo specifico (come ad esempio un campo di input o un command button).

Un *Renderer* aiuta l’*UIComponent* che ha a che fare con come la classe specifica *UIComponent* dovrebbe apparire in uno specifico tipo di dispositivo client.

I Renderers sono identificati da due nomi: *render-kit e rendered-type.*

*render-kit* è un contenitore in cui uno specifico gruppo di renderers è posizionato ed è identificato da *render-kit-id*.

La maggior parte delle librerie di componenti JavaServer Faces offre il proprio *render-kit*.

Un oggetto *javax.faces.view.facelets.Tag* aiuta l’*UIComponent* e il *Renderer* a permettere all’autore della pagina di includere un’istanza di *UIComponent* in una view JavaServer Faces.

Un tag rappresenta una specifica combinazione di *component-type* e *redered-type*.

Questo capitolo utilizza il componente image map dall’esempio Duke's Bookstore per spiegare come creare dei semplici componenti personalizzati, renderers personalizzati e i tag personalizzati associati e tutti gli altri dettagli connessi all’utilizzo dei componenti e dei renderers nell’applicazione.

Maggiori informazioni nel capitolo 57 dove c’è l’esempio completo.

Inoltre il capitolo descrive come creare altri oggetti personalizzati: convertitori, listeners e validatori, come legare i valori di un componente ai data objects e come legare gli oggetti personalizzati alle proprietà di un managed bean.

## 15.1 Determining Whether You Need a Custom Component or Renderer

L’implementazione di JavaServer Faces supporta un set molto basico di componenti e renderers associati.

Questo capitolo aiuta a decidere quando si possono usare i componenti e i renderers standard in un’applicazione e quando c’è la necessità di crearne di personalizzati.

### 15.1.1 Quando usare un Componente Personalizzato

Una classe componente definisce lo stato e il comportamento di un componente UI.

Il comportamento include la conversione dei valori del componente in un mark-up appropriato, l’accodarsi degli eventi sui componenti, l’esecuzione delle validazioni e ogni altro comportamento relativo a come il componente interagisce col browser e col ciclo di vita del processamento della richiesta.

Si ha bisogno di creare un componente personalizzato nelle seguenti situazioni:

* Si ha bisogno di aggiungere un nuovo comportamento a un componente standard, come generare un tipo aggiuntivo di evento.
* Si ha bisogno di prendere un’azione diversa da quelle disponibili nei componenti standard esistenti durante il processamento del valore della richiesta di un componente.
* Si vuole sfruttare una capacità HTML fornita dal browser ma nessuno dei componenti standard JavaServer Faces la sfrutta nel modo desiderato. La versione attuale non contiene componenti standard per componenti complessi di HTML come i frames.

Comunque, grazie all’estensibilità dell’architettura dei componenti, si può usare JavaServer Faces per creare componenti che li sfruttino.

Nell’esempio Duke’s Bookstore ci sono componenti personalizzati che corrispondono ai tag HTML *map* e *area*.

* Si ha bisogno di renderizzare verso un client non HTML che richiede componenti extra non supportati da HTML (che si spera vengano offerti nel render kit in futuro).

Comunque se si vuole renderizzare verso un client diverso, come un telefono, si potrebbe necessitare della creazione di componenti personalizzati che rappresentano i controlli supportati solamente dal client. Ad esempio alcune architetture di componenti er clients wireless includono il sopporto per orologi e barre di progresso che non sono disponibili su un client HTML.

In questo caso, si potrebbe aver bisogno di un rendered personalizzato insieme al componente, o anche solo del rendered personalizzato.

Non si ha bisogno di creare un componente personalizzato quando:

* Si ha bisogno di aggregare componenti per creare un nuovo componente che ha il proprio comportamento a sé stante. In questo caso si può usare un componente composito per combinare i componenti standard già esistenti. Maggiori informazioni nel capitolo 14.
* Si ha semplicemente bisogno di manipolare dati sul componente o aggiungergli funzionalità specifiche dell’applicazione. In questi casi si dovrebbe creare un managed bean a questo scopo e legarlo al componente standard invece che creare un componente personalizzato.
* Si ha bisogno di convertire i dati di un componente in un tipo non supportato dal suo rendered. Vedere il capitolo 11 sui convertitori per informazioni aggiuntive.
* Si ha bisogno di eseguire la validazione dei dati di un componente. I validatori standard e personalizzati possono essereaggiunti alla pagina usando i tag dei validatori.
* Si ha bisogno di registrare event listeners sui componenti. Si può farlo o usando i tags *f:valueChangeListener* o *f:actionListener* o si può puntare a un metodo di processamento di un evento su un managed bean utilizzando gli attributi *actionListener* o *valueChangeListener*.

### 15.1.2 Quando usare un rendered personalizzato

Un renderer, che genera il mark-up per visualizzare un componente su una pagina web, permette di separare la semantica di un componente dalla sua rappresentazione.

Mantenendoli separati, si può offrire supporto a diversi tipi di dispositivi client.

Si può immaginare un rendered come a un “adattatore per il client”.

Esse produce output adatto al consumo e alla visualizzazione sul client e accetta input dal client quando l’utente interagisce con il componente.

Se si sta creando un componente personalizzato, si ha bisogno di assicurarsi, tra le altre cose, che il componente esegua queste operazioni fondamentali per renderizzare il componente:

* **Decoding**: conversione dei parametri della richiesta in entrata ai valori locali del componente.
* **Encoding**: conversione del valore locale del componente nel corrispettivo mark-up che lo rappresenta nella risposta.

La specifica JavaServer Faces supporta due modelli di programmazione per gestire il decoding e l’encoding:

* Implementazione diretta: converte i parametri in entrata della richiesta al valore locale del componente.
* Implementazione delegata: il componente delega l’implementazione di encoding e decoding a un renderer separato.

Delegando le operazioni a un renderer si ha la possiilità di associare il proprio componente personalizzato con diversi renderers in modo da poter renderizzare il componente su diversi clients.

Se non si ha bisogno di farlo è più semplice lasciare la gestione del rendering al componente ma in ogni caso la separazione del renderer permette di dividere la semantica dalla visualizzazione.

L’applicazione Duke’s Bookstore separa i renderers dai componenti.

### 15.1.3 Component, Renderer, and Tag Combinations

Quando si crea un componente personalizzato si può creare un renderer personalizzato che lo accompagni.

Per associare il componente al renderer e riferirsi al componente dalla pagina, si ha bisogno anche di un tag personalizzato.

Sebbene ci sia il bisogno di scrivere il componente e il renderer personalizzato, non c’è alcun bisogno di scrivere codice per il tag personalizzato (chiamato tag handler).

Se si specifica la combinazione di renderer e componente Facelets crea il tag handler automaticamente.

In alcune rare situazioni, si potrebbe voler usare un renderer personalizzato insieme a un componente standard invece che a un componente personalizzato o si potrebbe voler usare un tag personalizzato senza un renderer o un componente.

Questo capitolo fornisce degli esempi di queste situazioni e riassume cosa è obbligatorio per un componente, un renderer o un tag personalizzato.

Si potrebbe voler usare un renderer personalizzato senza un componente personalizzato se si vuole aggiungere qualche validazione lato-client su un componente standard.

In questo caso si dovrebbe implementare il codice di validazione con un linguaggio di scripting lato-client, come JavaScript, e quindi renderizzare il JavaScript con il renderer personalizzato.

In questa situazione si avrebbe bisogno di un Tag personalizzato per il renderer in modo che il tag handler possa registrare il renderer sul componente standard.

I componenti personalizzati, come i renderers personalizzati, necessitano di essere associati a tag personalizzate. Comunque si può avere un tag personalizzato senza un renderer o un componente personalizzato.

Ad esempio supponiamo di dover creare un validatore personalizzato che necessiti di attributi extra sul proprio tag.

In questo caso il tag personalizzato corrisponderebbe al validatore personalizzato e non a un componente o renderer personalizzato.

In ogni caso continuerebbe a esserci la necessità di associare il tag personalizzato all’oggetto lato-server.

La tabella sotto riassume cosa si può o deve associare a un componente, tag o renderer personalizzato:



## 15.2 Understanding the Image Map Example

Duke’s Bookstore incude un componente image map personalizzato nella pagina *index.xhtml*.

Questo image map mostra una selezione di sei titoli di libri.

Quando l’utente clicca uno dei titoli l’applicazione va in una pagina che mostra il titolo del libro selezionato insieme a informazioni a proposito di un libro correlato.

La pagina permette all’utente di aggiungere altri libri al carrello della spesa.

### 15.2.1 Perché usare la tecnologia JavaServer Faces per implementare un Image Map?

Questa tecnologia è un framework ideale per l’implementazione di questo tipo di image map perché può eseguire il lavoro da fare sul server senza richiedere la creazione di un’image map a lato server.

In genere, le image-map lato-client sono preferite a quelle lato-server per molte ragioni.

Una di queste è che un’image map lato-client permette al browser di fornire feedback immediato quando un utente posiziona il mouse su un hotspot.

Un’altra ragione è che l’image map lato-client è più performante perché non richiede di andare anti e indietro dal server.

Comunque, in alcune situazioni, l’image map dovrebbe poter recuperare i dati o cambiare la visualizzazione di controlli nonform, cosa che un’image map lato-client non può fare.

Dato che il componente personalizzato image map utilizza la tecnologia JavaServer Faces, riesce a ottenere il meglio da entrambi questi stili di image map: può gestire le parti dell’applicazione che hanno bisogno di essere eseguite sul server e permette alle altre parti di essere eseguite lato-client.

### 15.2.2 Understanding the Rendered HTML

Sotto è riportata una versione abbreviata della parte form di una pagina HTML che l’applicazione ha bisogno di renderizzare:

*<form id="j\_idt13" name="j\_idt13" method="post"*

*action="/dukesbookstore/index.xhtml" ...>*

*...*

*<img id="j\_idt13:mapImage"*

*src="/dukesbookstore/javax.faces.resource/book\_all.jpg?ln=images"*

*alt="Choose a Book from our Catalog"*

*usemap="#bookMap" />*

*...*

*<map name="bookMap">*

*<area alt="Duke"*

*coords="67,23,212,268"*

*shape="rect"*

*onmouseout="document.forms[0]['j\_idt13:mapImage'].src='resources/images/book\_all.jpg'"*

*onmouseover="document.forms[0]['j\_idt13:mapImage'].src='resources/images/book\_201.jpg'"*

*onclick="document.forms[0]['bookMap\_current'].value='Duke'; document.forms[0].submit()"*

*/>*

*...*

*<input type="hidden" name="bookMap\_current">*

*</map>*

*...*

*</form>*

Il tag *img* associa un’immagine (book\_all.jpg) con l’image map referenziata nel valore dell’attributo *usemap*.

Il tag *map* specifica l’image map e contiene un set di tags *area*.

Ogni tag *area* specifica una regione dell’image map.

Gli attributi *onmouseover*, *onmouseout*, e *onclick* definiscono quale codice JavaScript viene eseguito quando capitano questi eventi.

Quando l’utente muove il mouse su una regione, la funzione *onmouseover* associata a quella regione mostra la mappa con quella regione evidenziata.

Quando l’utente muove il mouse fuori dalla regione, la funzione *onmouseout* mostra di nuovo l’immagine originale.

Se l’utente clicca su una regione, la funzione *onclick* setta il valore del tag *input* all’ID dell’area selezionata e fa il submit della pagina.

Il tag *input* rappresenta un controllo nascosto che conserva il valore dell’area selezionata correntemente, tra gli scambi lato-client, in modo che il componente lato-server possa recuperarlo.

Gli oggetti lato-server recuperano il valore di *bookMap\_current* e settano quello locale nell’istanza *javax.faces.context.FacesContext* in base alla regione selezionata.

### 15.2.3 Understanding the Facelets Page

Sotto è presentato un form abbreviato di una pagina Facelets che il componente image map utilizza per generare la pagina HTML mostrata nel capitolo precedente.

Utilizza le tags personalizzate *bookstore:map* e *bookstore:area* per rappresentare i componenti personalizzati:

*<h:form>*

*...*

*<h:graphicImage id="mapImage"*

*name="book\_all.jpg"*

*library="images"*

*alt="#{bundle.ChooseBook}"*

*usemap="#bookMap" />*

*<bookstore:map id="bookMap"*

*current="map1"*

*immediate="true"*

*action="bookstore">*

*<f:actionListener*

*type="dukesbookstore.listeners.MapBookChangeListener" />*

*<bookstore:area id="map1" value="#{Book201}"*

*onmouseover="resources/images/book\_201.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage" />*

*<bookstore:area id="map2" value="#{Book202}"*

*onmouseover="resources/images/book\_202.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage"/>*

*...*

*</bookstore:map>*

*...*

</h:form>

L’attributo *alt* del tag *h:graphicImage* è associato alla stringa "Choose a Book from our Catalog".

Il tag *f:actionListener* all’interno del tag *bookstore:map* punta a una classe listener per l’evento azione.

Il metodo *processAction* del listener posiziona l’ID del libro della map area selezionata all’interno della session map.

Il modo in cui è gestito questo evento è spiegato nel capitolo 15.7.

L’attributo *action* del tag *bookstore:map* specifica un outcome logico di tipo *String*, “*bookstore*”, che invia tramite regole di navigazione implicite l’applicazione alla pagina *bookstore.xhtml*.

Per maggiori informazioni consultare il capitolo 16.9.

L’attributo *immediate* del tag *bookstore:map* è settato su true, quindi l’implementazione standard di *javax.faces.event.ActionListener* deve essere eseguita nella fase di Apply Request Values invece di aspettare la fase Invocake Application.

Dato che la richiesta risultante dal click del mouse non necessita di alcuna validazione, conversione o aggiornamento dei dati su oggetti lato-server, è sensato saltare direttamente alla fase Invoke Application.

L’attributo *current* del tag *bookstore:map* è settato sull’area di default, che è *map1*.

Notare che il tag *bookstore:area* non contiene alcuno dei JavaScript, coordinate o modelli di dati shape data) che è visualizzato nella pagina HTML.

Il JavaScript è generato dalla classe *dukesbookstore.renderers.AreaRenderer*.

l valore degli attributi *onmouseover* e *onmouseout* indica l’immagine da caricare quando questi eventi hanno luogo. Come vengono generati i JavaScript è spiegato meglio in Performing Encoding (15.4.2).

La dati di coordinata, shape e testo alternativo sono ottenuti attraverso l’attributo *value*, tali valori fanno riferimento a un attributo nello scope dell’applicazione.

Il valore di tali attributi è un bean che conserva i dati *coords, shape* e *alt*.

Come vengono conservati tali bean nello scope dell’applicazione è spiegato nel prossimo capitolo.

### 15.2.4 Configuring Model Data

In un’applicazione JavaServer Faces, dati quali le coordinate di un hotspot o un’image map vengono recuperati dall’attributo *value* attraverso un bean.

Comunque lo shape e le coordinate di un hotspot dovrebbero essere definite insieme perché le coordinate vengono interpretate diversamente in base a di quale shape sia l’hotspot.

Dato che il valore del componente può essere legato solo a una proprietà, l’attributo *value* non può riferirsi sia allo shape che alle coordinate.

Per risolvere questo problema, l’applicazione incapsula tutte queste informazioni in un set di oggetti *ImageArea*.

Tali oggetti sono inizializzati dentro lo scope dell’applicazione dall’infrastruttura di creazione del managed bean (vedere capitolo 16.4.1).

Riportiamo la parte della dichiarazione del managed bean per il bean *ImageArea* corrispondente all’hotspot del sud america:

*<managed-bean eager="true">*

*...*

*<managed-bean-name>Book201</managed-bean-name>*

*<managed-bean-class>*

*javaeetutorial.dukesbookstore.model.ImageArea*

*</managed-bean-class>*

*<managed-bean-scope>application</managed-bean-scope>*

*<managed-property>*

*...*

*<property-name>shape</property-name>*

*<value>rect</value>*

*</managed-property>*

*<managed-property>*

*...*

*<property-name>alt</property-name>*

*<value>Duke</value>*

*</managed-property>*

*<managed-property>*

*...*

*<property-name>coords</property-name>*

*<value>67,23,212,268</value>*

*</managed-property>*

*</managed-bean>*

Per maggiori informazioni sull’inizializzazione dei managed beans capitolo 16.2.

Gli attributi *value* dei tags *bookstore:area* fanno riferimento ai beans nello scope dell’applicazione, come mostrato nel tag *bookstore:are* preso da *index.xhtml*:

*<bookstore:area id="map1" value="#{Book201}"*

*onmouseover="resources/images/book\_201.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage" />*

Il riferimento ai valori del bean model object *ImageArea* si ha con l’implementazione di un metodo *getValue* nella classe del componente.

Questo metodo chiama *super.getValue*. La superclasse ha un metodo *getValue* che trova gli oggetti *ImageArea* associati a *AreaComponent*.

La classe *AreaRenderer*, che necessita di renderizzare i valori di *alt*, *shape* e *coords* dall’oggetto *ImageArea*, chiama il metodo *getValue* di *AreaComponent* per recuperare l’oggetto *ImageArea*.

*ImageArea iarea = (ImageArea) area.getValue();*

*ImageArea* è un bean semplice, quindi si può accedere ai valori di *shape, coord* e *alt* chiamando i metodi accessori appropriati di *ImageArea*.

Il capitolo 15.5.1 spiega come far ciò nella classe *AreaRenderer*.

### 15.2.5 Summary of the Image Map Application Classes



## 15.3 Passaggi per creare un componente personalizzato

Si possono seguire i seguenti steps nello sviluppo del proprio componente personalizzato:

1. Creare la classe del componente personalizzato che fa le seguenti cose:
2. Override del metodo *getFamily* per far ritornare la famiglia del componente, che è usato per cercare i renderers che possono renderizzare il componente.
3. Include il codice di rendering o lo delega a un renderer
4. Abilita gli attributi dei componenti ad accettare espressioni
5. Mette in coda un evento sul componente se il componente genera un evento
6. Salva e riprista lo stato del componente
7. Delegare il rendering al renderer se non deve essere gestito dal componente, facendo:
8. Creare la classi di un renderer personalizzato estendendo *javax.faces.render.Renderer*.
9. Registrare il renderer in un render kit.
10. Registrare il componente
11. Creare un event handler se il componente genera eventi
12. Creare un tag library descriptor (TLD) che definisce i tag personalizzati.

## 15.4 Creare le classi dei componenti personalizzati

Come spiegato in “Quando usare un componente personalizzato”, la classe di un componente definisce lo stato e il comportamento di un componente UI.

Le informazioni di stato includono il tipo, l’identificatore e il valore locale.

Il comportamento definito dalla classe del componente include:

* Decoding (conversione dei parametri della richiesta in valori locali del componente)
* Encoding (conversione del valore locale nel corrispondene mark-up)
* Salvataggio dello stato del componente
* Update del valore del bean con il valore locale
* Processamento della validazione del valore locale
* Accodamento eventi

La classe *javax.faces.component.UIComponentBase* definisce il comportamento di default di una classe componente.

Tutte le classi rappresentanti componenti standard estendono *UIComponentBase*.

Queste classi aggiungono le proprie definizioni di comportamento, come farebbe un classe component.

La classe componente personalizzata deve estendere *UIComponentBase* direttamente o estendere una classe che rappresenta uno dei componenti standard . Queste classi sono posizionate nel package *javax.faces.component* e i loro nomi iniziano con UI.

Se un componente personalizzato ha lo stesso obiettivo di un componente standard, si può estendere il componente standard invece che estendere direttamente *UIComponentBase*.

Ad esempio supponiamo di voler creare un componente che sia un menu modificabile.

E’ sensato far estendere a questo componente *UISelectOne* invece che *UIComponentBase* perché è possibile riutilizzare il comportamento già definito in *UISelectOne*.

L’unica nuova funzionalità da definire è rendere il menu modificabile.

Quando si decide se avere un componente che estende *UIComponentBase* o un componente standard, si potrebbe voler anche implementare su tale componente una o più interfacce di comportamento definite nel package *javax.faces.component* che includono:

* ***ActionSource***che indica che un componente può causare un *javax.faces.event.ActionEvent.*
* ***ActionSource2***che estende *ActionSource* e permette alle proprietà del componente di riferirsi a metodi che gestiscono gli eventi usando le method expressions definite dall’EL.
* ***EditableValueHolder*** che estende *ValueHolder* e specifica caratteristiche aggiuntive per componenti modificabili, come la validazione e l’emissione di eventi value-change.
* ***NamingContainer*** che affida a ogni componente radicato nel componente un ID univoco.
* ***StateHolder*** che denota che un componente ha uno stato che deve essere salvato tra una richiesta e l’altra
* ***ValueHolder***che indica che un componente mantiene un valore locale e le opzioni per accedere ai dati nel model tier.

Se il proprio componente estende *UIComponentBase,* esso implementa automaticamente solo *StateHolder*.

Dato che tutti i componenti estendono direttamente o indirettamente *UIComponentBase*, tutti implementano *StateHolder*.

Ogni componente che implementa *StateHolder* implementa anche l’interfaccia *StateHelper* che esende *StateHolder* e definisce un contratto tipo Mappa (Map-like contract) che rende semplice per i componenti salvare e ripristinare uno stato paziale di una view.

Se il componente estende uno degli altri componenti standard, potrebbe implementare altre interfacce di comportamento in aggiunta a *StateHolder*.

Ad esempio se estende *UICommand* implementerà automaticamente *ActionSource2*.

Se il componente estende *UIOutput* o una classe componente che lo estende, implementerà in automatico *EditableValueHolder* e *ValueHolder*.

Si può inoltre avere un’implementazione diretta si un’interfaccia di comportamento su un componente che non estende automaticamente un particolare componente standard.

Ad esempio se si ha un componente che estende *UIInput* e si vuole che esegua delle azioni, si deve implementare esplicitamente *ActionSource2* perché il componente *UIInput* non implementa questa interfaccia in automatico.

L’image map nell’esempio Duke’s Bookstore ha due classi componenti: *AreaComponent* e *MapComponent*.

La classe *MapComponent* estende *UICommand* e implementa anche *ActionSource2*, il che significa che può eseguire azioni evento quando un utente clicca sulla mappa.

La classe *AreaComponent* estende il componente standard *UIOutput*.

L’annotazione *@FacesComponent* registra i componenti con l’implementazione di JavaServer Faces:

*@FacesComponent("DemoMap")*

*public class MapComponent extends UICommand {...}*

*@FacesComponent("DemoArea")*

*public class AreaComponent extends UIOutput {...}*

La classe *MapComponent* rappresenta il componente corrispondente al tag *bookstore:map*:

*<bookstore:map id="bookMap"*

*current="map1"*

*immediate="true"*

*action="bookstore">*

*...*

*</bookstore:map>*

La classe *AreaComponent* rappresenta il componente corrispondente al tag *bookstore:area*:

*<bookstore:area id="map1" value="#{Book201}"*

*onmouseover="resources/images/book\_201.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage"/>*

*MapComponent* ha uno o più istanze di *AreaComponent* come children.

Il suo comportamento consiste delle seguenti azioni:

* Recupera il valore dell’area selezionata
* Definisce le proprietà corrispondenti ai valori del componente
* Genera un evento quando l’utente clicca sull’image map
* Mette in coda l’evento
* Salva il suo stato
* Renderizza i tags *map* e *input* di HTML

*MapComponent* delega il rendering dei tags *map* e *input* alla classe *MapRenderer*.

*AreaComponent* è legato a un bean che conserva shape e coordinate della regione dell’image map.

Vedremo come accedere a questi dati tramite value expression nel capitolo 15.5.1.

Il comportamento di *AreaComponent* consiste di:

* Recupero dei dati shape e coordinate dal bean
* Settaggio del valore del tag nascosto all’*id* del suo componente
* Renderizzazione del tag *area*, incluso il JavaScript per le funzioni *onmouseover, onmouseout*, e *onclick*

Dato che questi compiti sono attualmente eseguiti da *AreaRenderer, Areacomponent* deve delegarli a *AreaRenderer*.

Il capitolo 15.7 mostra i dettagli di come *MapComponent* gestisca gli eventi.

### 15.4.1 Specifying the Component Family

Se la classe componente personalizzata delega il rendering, necessita di fare l’override dek metodo *getFamily* del componente *UIComponent* per ritornare l’identificatore di un **component family** usato per riferirsi a un componente o a un set di componenti che può essere renderizzato da un renderer o da un set di renderers.

Il component family è usato insieme al tipo del renderer per trovare i renderers che possono renderizzare il componente:

*public String getFamily() {*

*return ("Map");*

*}*

L’identificatore del component family, *Map*, deve coincidere con quello definito dagli elementi *component-family* inclusi nelle configurazioni del componente e del renderernel file di configurazione dell’applicazione.

Il capitolo 16.10 spiega come definire il component family nella configurazione del renderer.

Il capitolo 16.11 spiega come definire il component family nel componente di configurazione.

### 15.4.2 Performing Encoding

Durante la fase di Render Response, l’implementazione di JavaServer Faces processa i metodi di encoding di tutti i componenti e dei loro renderers associati nella view.

I metodi di encoding convertono il valore locale corrente del componente nel mark-up corrispondente che lo rappresenta nella risposta.

La classe *UiComponentBase* definisce un set di metodi per il mark-up di rendering: *encodeBegin, encodeChildren* e *encodeEnd*.

Se il componente ha dei componenti figli, potrebbe essere necessario usare più di uno di questi metodi per renderizzare il componente, altrimenti tutte le renderizzazioni vengono effettuate in *encodeEnd*.

In alternativa si può usare il metodo *encodeAll* che comprende tutti i metodi.

Dato che *MapCompontent* è un componente padre di *AreaComponent*, il tag *area* deve essere renderizzato dopo il tag *map iniziale* e prima del tag *map* finale.

Per far ciò la classe *MapRender* renderizza il tag *map* iniziale in *encodeBegin* e il resto del tag *map* in *encodeEnd*.

L’implementazione di JavaServer invoca automaticamente il metodo *encodeEnd* del renderer *AreaComponent* dopo di che invoca il metodo *encodeBegin* di *MapRenderer* prima di cui invoca il metodo *encodeEnd* del *MapRenderer*.

Se un componente necessita di eseguire il rendering per i suoi figli, lo fa usando il metodo *encodeChildren*.

Sotto sono riportati i metodi *encodeBegin* e *encodeEnd* di *MapRenderer*:

*@Override*

*public void encodeBegin(FacesContext context, UIComponent component)*

*throws IOException {*

*if ((context == null)|| (component == null)) {*

*throw new NullPointerException();*

*}*

*MapComponent map = (MapComponent) component;*

*ResponseWriter writer = context.getResponseWriter();*

*writer.startElement("map", map);*

*writer.writeAttribute("name", map.getId(), "id");*

*}*

*@Override*

*public void encodeEnd(FacesContext context, UIComponent component)*

*throws IOException {*

*if ((context == null) || (component == null)){*

*throw new NullPointerException();*

*}*

*MapComponent map = (MapComponent) component;*

*ResponseWriter writer = context.getResponseWriter();*

*writer.startElement("input", map);*

*writer.writeAttribute("type", "hidden", null);*

*writer.writeAttribute("name", getName(context,map), "clientId");*

*writer.endElement("input");*

*writer.endElement("map");*

*}*

Si noti che che *encodeBegin* renderizza solo il tag *map* iniziale.

Il metodo *encodeEnd* renderizza il tag *input* e il tag *map* finale.

I metodi di encoding accettano un *UIComponent* e un *javax.faces.context.FacesContext* un come argomento.

L’istanza di *FacesContext* contiene tutte le informazioni associate con la richiesta corrente.

Il componente *UIComponent* è il componente che necessita di essere renderizzato.

Il resto del metodo renderizza il mark-up dell’istanza *javax.faces.context.ResponseWriter* che scrive il mark-up alla risposta corrente.

Ciò consiste nel passare i nomi dei tags e degli attributi HTML all’istanza di *ResponseWriter* come stringa, recuperando i valori degli attributi dei componenti e passando tali valori all’istanza di *ResponseWriter*.

Il metodo *startElement* prende una *String*(il nome del tag)e il componente a cui corrisponde (in questo caso *Map*), passando tali informazioni all’istanza di *ResponseWriter* aiuta gli strumenti a design time design time tools) a sapere quale parte del mark-up generato è correlato a quali componenti.

Dopo la chiamata di *startElement* si può scrivere *writeAttribute* per renderizzare gli attributi del tag.

Il metodo *writeAttribute* prende il nome dell’attributo, il suo valore e il nome di una proprietà o attributo del componente container corrispondente all’attributo.

L’ultimo parametro può essere nullo, in tal caso non sarà renderizzato.

Il valore *name* dell’attributo del tag *map* è recuperato tramite il metodo *getId* di *UIComponent* che ritorna l’identificatore univoco del componente.

Il valore dell’attributo *name* del tag *input* viene recuperato dal metodo *getName(FacesContext, UIComponent)* di *MapRenderer*.

Se vuoi che il componente esegua la propria renderizzazione se non c’è un renderer, ma che lo deleghi se ce n’è uno, si devono includere le seguenti linee di codice nel metodo di encoding per verificare se esiste un renderer associato a tale componente:

*if (getRendererType() != null) {*

*super.encodeEnd(context);*

*return;*

*}*

Se esiste un renderer disponibile, questo metodo invoca il metodo *encodeEnd* della superclasse, che trova il renderer.

La classe *MapComponent* delega tutte le renderizzazioni al *MapRenderer*, in modo che non necessiti di cercare renderers disponibili.

In alcune classi di componenti personalizzati che estendono componenti standard, si potrebbe necessitare dell’implementazione di altri metodi in aggiunta a *encodeEnd*.

Ad esempio se ci fosse la necessità di recuperare il valore del componente dai parametri della richiesta, di dovrebbe implementare anche il metodo *decode*.

### 15.4.3 Performing Decoding

Durante la fase Apply Request Values, l’implementazione di JavaServer Faces processa i metodi *decode* di tutti i componenti nell’albero.

Il metodo *decode* estrae un valore locale del componente dai parametri di una richiesta in entrata e utilizza un’implementazione di *javax.faces.convert.Converter* per convertire il valore nel tipo accettabile dalla classe del componente.

Una classe componente personalizzato o il suo renderer deve implementare il metodo *decode* solo se deve recuperare il valore locale o se necessita di accodare degli eventi.

Il componente accoda degli eventi tramite la chiamata di *queueEvent*.

Ecco il metodo *decode* di *MapRenderer*:

*@Override*

*public void decode(FacesContext context, UIComponent component) {*

*if ((context == null) || (component == null)) {*

*throw new NullPointerException();*

*}*

*MapComponent map = (MapComponent) component;*

*String key = getName(context, map);*

*String value = (String) context.getExternalContext().*

*getRequestParameterMap().get(key);*

*if (value != null)*

*map.setCurrent(value);*

*}*

*}*

Il metodo *decode* per prima cosa prende il nome del campo *input* nascosto chiamando *getName(FacesContext, UIComponent)*.

Quindi questo usa il nome come key per la mappa del parametro richiesto per recuperare il valore corrente del campo *input*.

Questo valore rappresenta l’area correntemente selezionata.

Infine setta il valore dell’attributo *current* della classe *MapComponent* al valore del campo *input*.

### 15.4.4 Abilitare le proprietà del componente ad accettare espressioni

Quasi tutti gli attributi dei tags standard di JavaServer Faces accettano le espressioni, se sono value o method expressions.

E’ raccomandabile abilitare gli attributi del componente ad accettare le espressioni perché ciò fornisce più flessibilità quando si scrivono pagine Facelets.

Per abilitare gli attributi ad accettare espressioni, la classe del componente deve implementare i metodi getter e setter delle proprietà del componente.

Tali metodi possono utilizzare i servizi offerti dall’interfaccia *StateHelper* per archiviare e recuperare non solo i valori per queste proprietà ma anche lo stato dei componenti attraverso richieste multiple.

Dato che *MapComponent* estende la classe *UICommand*, *UICommand* prende le istanze *ValueExpression* e *MethodExpression* associate a ogni ottributo che supporta.

In modo simile, la classe *UIOutput*, che *AreaComponent* estende, ottiene le istanze *ValueExpression* per gli attributi supportati.

Per entrambi i componenti, i semplici metodi *getter* e *setter* archiviano e recuperano i valori chiave e lo stato degli attributi, come mostrato nel frammento di codice seguente di *AreaComponent*:

*enum PropertyKeys {*

*alt, coords, shape, targetImage;*

*}*

*public String getAlt() {*

*return (String) getStateHelper().eval(PropertyKeys.alt, null);*

*}*

*public void setAlt(String alt) {*

*getStateHelper().put(PropertyKeys.alt, alt);*

*}*

*...*

Comunque se si ha una classe componente personalizzato che estende *UIComponentBase*, si potrebbe aver bisogno di implementare i metodi che prendono le istanze di *ValueExpression* e *MethodExpression* associate a quegli attributi che sono abilitati ad accettare espressioni.

Ad esempio si potrebbe includere un metodo che prenda l’istanza di *ValueExpression* per l’ettributo *immediate*:

*public boolean isImmediate() {*

*if (this.immediateSet) {*

*return (this.immediate);*

*}*

*ValueExpression ve = getValueExpression("immediate");*

*if (ve != null) {*

*Boolean value = (Boolean) ve.getValue(*

*getFacesContext().getELContext());*

*return (value.booleanValue());*

*} else {*

*return (this.immediate);*

*}*

*}*

Le proprietà corrispondenti agli attributi del componente che accettano le method expressions devono accettare e ritornare un oggetto *MethodExpression*.

Ad esempio se *MapComponent* estendesse *UIComponentBase* invece di *UICommand,* esso necessiterebbe di offrire una proprietà *action* che ritorni e accetti un oggetto *MethodExpression*:

*public MethodExpression getAction() {*

*return (this.action);*

*}*

*public void setAction(MethodExpression action) {*

*this.action = action;*

*}*

### 15.4.5 Salvare e ripristinare lo stato

Come spiegato nel capitolo precedente, si usano i servizi dell’interfaccia *StateHelper* per permettere di salvare lo stato del componente e allo stesso tempo di settare e recuperare i valori della proprietà.

L’implementazione di *StateHelper* permette il salvataggio dello stato parziale che salva solo i cambiamenti nello stato dalla richiesta iniziale, non l’intero stato, perché lo stato generale può essere ripristinato

durante la fase Restore View.

Per le classi componenti che implementano *StateHolder* potrebbe essere preferibile l’implementazione dei metodi *saveState(FacesContext)* e *restoreState(FacesContext, Object)* per aiutare l’implementazione di JavaServer Faces a salvare e ripristinare lo stato dei componenti attraverso richieste multiple.

Per salvare un set di valori si può implementare il metodo *saveState(FacesContext)*.

Questo metodo viene chiamato durante la fase Render Response, in cui lo stato della risposta viene salvato per processare le richieste successive.

Ecco un ipotetico metodo da *MapComponent*, che ha un solo attributo:*current*:

*@Override*

*public Object saveState(FacesContext context) {*

*Object values[] = new Object[2];*

*values[0] = super.saveState(context);*

*values[1] = current;*

*return (values);*

*}*

Questo metodo inizializza un array, che conserverà lo stato salvato e andrà poi a salvare tutti gli stati associati al componente.

Un componente che implementa *StateHolder* può anche fornire un’implementazione per *restoreState(FacesContext, Object)* che ripristini lo stato del componente a quello salvato con il metodo *saveState(FacesContext)*.

Il metodo *restoreState(FacesContext, Object)* viene chiamato durante la fase Restore View, durante cui l’implementazione di JavaServer Faces controlla se esiste uno stato che sia stato salvato durante l’ultima fase di Render Response che necessiti di essere ripristinato in preparazione del prossimo postback.

Ecco un ipotetico metodo *restoreState(FacesContext, Object)* per *MapComponent*:

*public void restoreState(FacesContext context, Object state) {*

*Object values[] = (Object[]) state;*

*super.restoreState(context, values[0]);*

*current = (String) values[1];*

*}*

Questo metodo prende l’istanza di un *FacesContext* e di un *Object*, che rappresenta l’array che conserva lo stato del componente e setta le proprietà al valore salvato nell’array *Object*.

Sia che si implementino questi metodi nella classe componente sia che non lo si faccia, si può usare il parametro di contesto *javax.faces.STATE\_SAVING\_METHOD* per specificare nel deployment descriptor dove si vuole che venga salvato lo stato: sul client o sul server.

Se lo stato è salvato sul client, lo stato dell’intera view viene renderizzato in un campo nascosto sulla pagina.

Per default è salvato sul server.

L’applicazione web d’esempio Duke’s Forest salva lo stato della propria view sul client.

Salvando lo stato sul client si usa un maggior dispendio di banda e più risorse client, salvandolo sul server si usano più risorse server.

Si potrebbe preferire il salvataggio dello stato sul client se ci si aspetta che gli utenti disabilitino le cookies.

## 15.5 Delegare il rendering a un renderer

Sia *MapComponent* che *AreaComponent* delegano tutte le proprie renderizzazioni a un renderer separato.

Il capitolo 15.4.2 spiegava come il *MapRenderer* esegue l’encoding per *MapComponent*.

Questo capitolo spiega nel dettaglio il processo di delega del rendering usando *AreaRenderer*, che esegue il rendering per *AreaComponent*:

Per delegare il rendering bisogna eseguire questi passaggi:

* Creare una classe *Renderer*
* Registrare il renderer con un render kit usando l’annotazione *@FacesRenderer* (o usando il rsource file di configurazione dell’applicazione come spiegato nel capitolo 16.10).
* Identificare il tipo del renderer con l’annotazione *FacesRenderer*.

### 15.5.1 Creare la classe Renderer

Quando si delega il rendering a un renderer, si possono delegare tutti gli encoding e i decoding al renderer o si può decidere di fare una parte di questi processi nella classe component.

La classe *AreaComponent* delega l’encoding alla classe *AreaRenderer*.

La classe renderer inizia con l’annotazione *@FacesRenderer*:

*@FacesRenderer(componentFamily = "Area", rendererType = "DemoArea")*

*public class AreaRenderer extends Renderer {*

L’annotazione *@FacesRenderer* registra la classe renderer con l’implementazione JavaServer Faces come una classe renderer.

L’annotazione identifica la famiglia e il tipo del renderer.

Per eseguire il rendering di *AreaComponent*, *AreaRenderer* deve implementare un metodo *encodeEnd*.

Questo metodo recupera lo shape, le coordinate *e* il valore del testo alternativo archiviato nel bean *ImageArea* che è legato ad *AreaComponent*.

Supponiamo che il tag *area* attualmente in uso venga renderizzato come un attributo *value* di “book203”.

Le seguenti linee di codice di *encodeEnd* prendono il valore dell’attributo “book203” dall’istanza di *FacesContext*:

*ImageArea ia = (ImageArea)area.getValue();*

L’attributo *value* è l’istanza del bean *ImageArea* che contiene i valori *shape, coords* e *alt* associati con l’istanza *AreaComponent* “book203”.

Il capitolo 15.2.4 descrive come l’applicazione immagazzini questi valori.

Dopo aver recuperato l’oggetto *ImageArea* il metodo renderizza i valori *shape, coords* e *alt* semplicemente chiamando i metodi accessori associati e passando i valori di ritorno all’istanza di *ResponseWriter*, come mostrato dalle seguenti linee di codice:

*writer.startElement("area", area);*

*writer.writeAttribute("alt", iarea.getAlt(), "alt");*

*writer.writeAttribute("coords", iarea.getCoords(), "coords");*

*writer.writeAttribute("shape", iarea.getShape(), "shape");*

Il metodo *encodeEnd* renderizza anche lo JavaScript per gli attributi *onmouseout, onmouseover* e *onclick*.

La pagina Facelets ha bisogno di offrire solo un path alle immagini che vengono caricate durante le azioni *onmouseover* e *onmouseout*:

*<bookstore:area id="map3" value="#{Book203}"*

*onmouseover="resources/images/book\_203.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage"/>*

La classe *AreaRenderer* prende in carico la generazione del JavaScript per queste azioni, come mostrato nel seguente codice da *encodeEnd*.

Il JavaScript che viene generato da *AreaRenderer* per l’azione *onclick* setta il valore del campo nascosto al valore dell’ID del componente dell’area correntemente selezionata e fa il submit della pagina:

*sb = new StringBuffer("document.forms[0]['").append(targetImageId).*

*append("'].src='");*

*sb.append(*

*getURI(context,*

*(String) area.getAttributes().get("onmouseout")));*

*sb.append("'");*

*writer.writeAttribute("onmouseout", sb.toString(), "onmouseout");*

*sb = new StringBuffer("document.forms[0]['").append(targetImageId).*

*append("'].src='");*

*sb.append(*

*getURI(context,*

*(String) area.getAttributes().get("onmouseover")));*

*sb.append("'");*

*writer.writeAttribute("onmouseover", sb.toString(), "onmouseover");*

*sb = new StringBuffer("document.forms[0]['");*

*sb.append(getName(context, area));*

*sb.append("'].value='");*

*sb.append(iarea.getAlt());*

*sb.append("'; document.forms[0].submit()");*

*writer.writeAttribute("onclick", sb.toString(), "value");*

*writer.endElement("area");*

Facendo il submit, questo codice causa il ritorno alla fase Restore View del ciclo vitale.

Questa fase salva ogni informazione di stato, inclusi i valori nel campo nascosto, in modo che un nuovo component tree sia costruito per una nuova richiesta.

Questo valore è recuperato dal metodo *decode* della classe *MapComponent*.

Questo metodo viene chiamato dll’implementazione di JavaServer Faces durante la fase Apply Request Values che segue quella di Restore View.

In aggiunta al metodo *encodeEnd, AreaRenderer* contiene un costruttore vuoto che viene usato per creare un’istanza di *AreaRenderer* che può essere aggiunta al render kit.

L’annotazione *@FacesRenderer* registra la classe renderer tramite l’implementazione di JavaServer Faces come una classe renderer.

Questa annotazione identifica la famiglia del componente oltre al tipo del renderer.

### 15.5.2 Identificare il tipo del renderer

Durante la fase di Render Response, JavaServer Faces chiama il metodo *getRendererType* del tag handler del componente per determinare quale renderer invocare, se ne esiste uno.

Si identifica il tipo associato al renderer nell’elemento *rendererType* dell’annotazione *@FacesRenderer* di *AreaRenderer* così come nell’elemento *rendererType* del descrittore della tag library.

## 15.6 Implementare un Event Listener

La tecnologia JavaServer Faces supporta gli eventi azione e value-change per i componenti.

Gli eventi azione accadono quando l’utente attiva un componente che implementa *javax.faces.component.ActionSource.*

Questi eventi sono rappresentati dalla classe *javax.faces.event.ActionEvent*.

Gli eventi value –change invece accadono quando un utente attiva un componente che implementa *javax.faces.component.EditableValueHolder*.

Questi eventi sono rappresentati dalla classe *javax.faces.event.ValueChangeEvent*.

Questo capitolo spiega come implementare le due classi listener.

Per gestire eventi generati da componenti personalizzati, bisogna implementare un event listener e un event handler e accodare manualmente l’evento sul componente.

Più informazioni nel capitolo 15.7.

Nota: Non c’è bisogno di creare un’implementazione di *ActionListener* per gestire eventi che risultano semplicemente di navigazione verso una paginaa senza eseguire altri processi specifici dell’applicazione.

Vedere il capitolo 12.3.1 per informazioni sulla gestione della navigazione.

### 15.6.1 Implementare listeners Value-Change

Un implementazione di *javax.faces.event.ValueChangeListener* deve includere un metodo *processValueChange(ValueChangeEvent)*.

Questo metodo processa l’evento value-change specificato e viene invocato dall’implementazione di JavaServer Faces quando c’è un evento value-change.

L’istanza di *ValueChangeEvent* conserva sia i nuovi che i vecchi valori del componente che ha avviato l’evento.

Nell’esempio Duke’s Bookstore, l’implementazione del listener *NameChanged* è registrata nello stesso componente *UIInput* della pagina *bookcashier.xhtml*.

Questo listener conserva nello session scope il nome inserito dall’utente nel campo corrispondente al componente name.

La pagina *bookreceipt.xhtml* di conseguenza recupera il nome dallo session scope:

*<h:outputFormat title="thanks"*

*value="#{bundle.ThankYouParam}">*

*<f:param value="#{sessionScope.name}"/>*

*</h:outputFormat>*

Quando la pagina viene caricata mostra il nome dentro il messaggio:

*"Thank you, {0}, for purchasing your books from us."*

Ecco parte dell’implementazione di *NameChanged*:

*public class NameChanged extends Object implements ValueChangeListener {*

*@Override*

*public void processValueChange(ValueChangeEvent event)*

*throws AbortProcessingException {*

*if (null != event.getNewValue()) {*

*FacesContext.getCurrentInstance().getExternalContext().*

*getSessionMap().put("name", event.getNewValue());*

*}*

*}*

*}*

Quando l’utente inserisce il nome nel campo, si genera il value-change event e il metodo *processValueChange(ValueChangeEvent)* dell’implementazione del listener *NameChanged* viene invocato.

Questo metodo prima prende l’ID del componente che ha generato l’evento dall’oggetto *ValueChangeEvent* e mette il valore, insieme al nome di un attributo, nella session map dell’istanza di *FacesContext*.

Il capitolo 11.2 spiega come registrare questo listener nel proprio componente.

### 15.6.2 Implementare un action listener

Un’implementazione di *javax.faces.event.ActionListener* deve includere un metodo *processAction(ActionEvent).*

Questo metodo processa l’evento azione specificato e viene evocato dall’implementazione di JavaServer Faces quando accade un evento azione.

L’esempio Duke’s Bookstore utilizza due implementazioni di *ActionListener*: *LinkBookChangeListener* e *MapBookChangeListener*.

Vedi 15.7 per dettagli su *MapBookChangeListener* e 11.2.2 per una spiegazione su come registrarlo su un componente.

## 15.7 Gestire eventi per componenti personalizzati

Come spiegato prima, gli eventi vengono automaticamente accodati sui componenti standard che li generano.

Un componente personalizzato, invece, deve accodarli manualmente tramite il metodo *decode*.

Il capitolo 15.4.3 spiega come accodare un evento su *MapComponent* usando il metodo *decode*.

Questo capitolo spiega come scrivere la classe che rappresenta l’evento di cliccare sulla mappa e come scrivere il metodo che processa tale evento.

Come spiegato nei capitoli sui Facelets, l’attributo *actionListener* del tag *bookstore:map* punta alla classe *MapBookChangeListener*.

Il metodo *processAction* della classe listener, processa l’evento del click sull’image map.

Sotto è riportato il metodo:

*@Override*

*public void processAction(ActionEvent actionEvent)*

*throws AbortProcessingException {*

*AreaSelectedEvent event = (AreaSelectedEvent) actionEvent;*

*String current = event.getMapComponent().getCurrent();*

*FacesContext context = FacesContext.getCurrentInstance();*

*String bookId = books.get(current);*

*context.getExternalContext().getSessionMap().put("bookId", bookId);*

*}*

* Quando l’implementazione di JavaServer Faces chiama questo metodo, esso passa in un oggetto *ActionEvent* che rappresenta l’evento generato dal click.
* Poi lo converte in un oggetto *AreaSelectEvent*.
* Poi il metodo prende il *MapComponent* associato all’evento.
* In seguito prende il valore dell’attributo *current* dell’oggetto *MapComponent* per avere il valore dell’ID del libro da un oggetto *HashMap* che è costruito da qualche altra parte nella classe *MapBookChangeListener*.
* Infine il metodo piazza l’ID ottenuto dall’oggetto *HashMap* nella session map dell’applicazione.

In aggiunta al metodo che processa l’evento, si ha bisogno della classe dell’evento.

Questa classe è semplice da scrivere: bisogna estendere *ActionEvent* e fornire un costruttore che prenda il componente su cui l’evento è accodato e un metodo che ritorni il componente.

Sotto c’è la classe *AreaSelectedEvent* usata con l’image map:

public class AreaSelectedEvent extends ActionEvent {

public AreaSelectedEvent(MapComponent map) {

super(map);

}

public MapComponent getMapComponent() {

return ((MapComponent) getComponent());

}

}

Come spiegato nel capitolo sulle classi personalizzate, per far sì che *MapComponent* generi l’evento deve implementare *ActionSource*.

Dato che *MapComponent* estende *UICommand*, esso implementa anche *ActionSource*.

## 15.8 Defining the Custom Component Tag in a Tag Library Descriptor

Per usare un tag personalizzato bisogna dichiararlo in un Tag Library Descriptor (TLD).

Il file TLD definisce come il tag personalizzato viene usato in una pagina JavaServer Faces.

Il web container usa il TLD per validare il tag.

Il set di tags che fanno parte del render kit HTML sono definite nell’HTML\_BASIC TLD, disponibile nella standard HTML tag library di JavaServer Faces.

Il nome del file TLD deve finire con *taglib.xml*.

Nell’esercizio Duke’s Bookstore i tags personalizzati *area* e *map* sono definiti nel file *web/WEB-INF/bookstore.taglib.xml*.

Tutte le definizioni dei tag devono essere annidate dentro l’elemento *facelet-taglib* nel TLD.

Ogni tag è definito da un elemento *tag*.

Sotto è riportata la definizione di *area* e *map*:

*<facelet-taglib xmlns="http://xmlns.jcp.org/xml/ns/javaee"*

*...>*

*<namespace>http://dukesbookstore</namespace>*

*<tag>*

*<tag-name>area</tag-name>*

*<component>*

*<component-type>DemoArea</component-type>*

*<renderer-type>DemoArea</renderer-type>*

*</component>*

*</tag>*

*<tag>*

*<tag-name>map</tag-name>*

*<component>*

*<component-type>DemoMap</component-type>*

*<renderer-type>DemoMap</renderer-type>*

*</component>*

*</tag>*

*</facelet-taglib>*

L’elemento *component-type* specifica il nome definito nell’annotazione *@FacesComponent* e l’elemento *renderer-type* specifica il *rendererType* definito nell’annotazione *@FacesRenderer*.

L’elemento *facelet-taglib* deve includere un elemento *namespace* che definisce il namespace da specificare nelle pagine che utilizzano il componente personalizzato.

Vedere il capitolo successivo sull’utilizzo di un componente personalizzato per informazioni su come specificare il namespace nelle pagine.

Il file TLD è posizionato nella directory *WEB-INF*.

In più viene incluso un entry nel web deployment descriptor (*web.xml*) per identificare la libreria di tag personalizzati nel dile descriptor, come segue:

*<context-param>*

*<param-name>javax.faces.FACELETS\_LIBRARIES</param-name>*

*<param-value>/WEB-INF/bookstore.taglib.xml</param-value>*

*</context-param>*

## 15.9 Usare un componente personalizzato

Per usare un componente personalizzato in una pagina si aggiunge il tag personalizzato associato con il componente sulla pagina.

Come spiegato nel capitolo precedente, bisogna assicurarsi che il TLD che definisceogni tag personalizzato sia impacchettato nell’applicazione se si intende usare i tags nelle proprie pagine.

I file TLD sono archiviati nella directory *META-INF* o in una sub-directory del file WAR o nella directory *META-INF* o in una sub-directory di una tag library nel file JAR.

Bisogna inoltre includere la dichiarazione del namespace nella pagina in modo che la pagina abbia accesso ai tags.

I tags personalizzati sono definiti in *bookstore.taglib.xml* nell’esempio Duke’s Bookstore.

Il tag *ui:composition* della pagina *index.xhtml* dichiara il namespace definito nel tag library:

*<ui:composition xmlns="http://www.w3.org/1999/xhtml"*

*xmlns:ui="http://xmlns.jcp.org/jsf/facelets"*

*xmlns:h="http://xmlns.jcp.org/jsf/html"*

*xmlns:f="http://xmlns.jcp.org/jsf/core"*

***xmlns:bookstore="http://dukesbookstore"***

*template="./bookstoreTemplate.xhtml">*

Infine per usare un componente personalizzato nella pagina, si aggiunge il tag del componente alla stessa.

In Duke’s Bookstore è incluso un elemento image map personalizzato nella pagina *index.xhtml*.

Questo componente permette di selezionare un libro cliccando su una regione dell’image map:

*...*

*<h:graphicImage id="mapImage"*

*name="book\_all.jpg"*

*library="images*

*alt="#{bundle.chooseLocale}"*

*usemap="#bookMap" />*

*<bookstore:map id="bookMap"*

*current="map1"*

*immediate="true"*

*action="bookstore">*

*<f:actionListener*

*type="javaeetutorial.dukesbookstore.listeners.MapBookChangeListener" />*

*<bookstore:area id="map1" value="#{Book201}"*

*onmouseover="resources/images/book\_201.jpg"*

*onmouseout="resources/images/book\_all.jpg"*

*targetImage="mapImage" />*

*...*

*<bookstore:area id="map6" value="#{Book207}"*

*onmouseover="resources/images/book\_207.jpg"*

*onmouseout="resources/images//book\_all.jpg"*

*targetImage="mapImage" />*

*</bookstore:map>*

Il tag standard *h:graphicImage* associa un’immagine (*book\_all.jpg*) con l’mage map referenziata dal valore dell’attributo *usemap*.

Il tag personalizzato *bookstore:map* rappresenta il componente personalizzato, *mapComponent*, specifica l’image map e contiene un set di tags *bookstore:area*.

Ognuno di questi tags *bookstore:area* rappresenta un *AreaComponent* personalizzato e specifica una regione dell’image map.

Sulla pagina gli attributi *onmouseover* e *onmouseout* specificano l’immagine che viene visualizzata quando un utente esegue le azioni dascritte da tali attributi.

Il renderer personalizzato renderizzaun attributo *onclick*.

Nella pagina HTML renderizzata gli attributi *onmouseover*, *onmouseout* e *onclick* definiscono quale codice JavaScript viene eseguito quando ci sono questi eventi.

Quando l’utente muove il mouse su una regione la funzione *onmouseover* associata a tale regione mostra la mappa con la regione evidenziata.

Quando l’utente muove il mouse fuori dalla regione la funzione *onmouseout* mostra di nuovo l’immagine originale.

Quando l’utente clicca con il mouse una regione la funzione onclick setta il valore di un tag *input* nascosto all’ID dell’area selezionata e fa il submit.

Quando un renderer renderizza questi attributi in HTML, esso renderizza anche il codice JavaScript.

Il renderer personalizzato renderizza anche l’intero attributo *onclick* invece che lasciare all’autore della pagina di settarlo.

Il renderer personalizzato che renderizza il tag HTML *map* renderizza anche un componente *input* nascosto che conserva l’area corrente.

Gli oggetti lato-server recuperano il valore del campo *input* nascosto e setta il locale nell’istanza *FacesContext* in base alla regione selezionata.

## 15.10 Creare e usare un convertitore personalizzato

Un classe convertitore in JavaServer Faces converte stringhe in oggetti e oggetti in stringhe quando se ne ha bisogno.

Molti convertitori standard sono offerti da JavaServer Faces a questo scopo.

Vedere il capitolo 11.1 per informazioni sui convertitori standard.

Come già spiegato, se i convertitori standard offerti non possono eseguire la conversione dei dati nel modo di cui si necessita, si può creare un convertitore personalizzato per eseguire questo tipo di conversioni.

Questa implementazione deve almeno definire come convertire dati in entrambe le direzioni tra le due views dei dati descritte nel capitolo 7.4.3.

Tutti i convertitori personalizzati devono implementare l’interfaccia *javax.faces.convert.Converter*.

Questo capitolo spiega come implementare questa interfaccia per eseguire la conversione personalizzata dei dati.

L’esempio Duke’s Bookstore usa un’implementazione personalizzata di *Converter* per convertire i dati inseriti nel campo del numero di carta di credito sulla pagina *bookcashier.xhtml*.

Il convertitore toglie gli spazi e le barrette (-) dal testo della stringa e lo formatta in modo che ci sia uno spazio bianco ogni quattro caratteri.

Un altro caso comune per un convertitore personalizzato è in una lista di oggetti di tipo non-standard.

Ad esempio in Duke’s Tutoring le entità *Student* e *Guardian* richiedono un convertitore personalizzato in modo da poter convertire a e per un componente input *UISelectItems*.

### 15.10.1 Creare un convertitore personalizzato

La classe convertitore personalizzata *CreditCardConverter* è costruita come segue:

*@FacesConverter("ccno")*

*public class CreditCardConverter implements Converter {*

*...*

*}*

L’annotazione *@FacesConverter* registra la classe convertitore personalizzata come convertitore con il nome di *ccno* nell’implementazione JavaServer Faces.

In alternativa si può registrare il convertitore inserendolo nell’ application configuration resource file, come mostrato nel capitolo 16.7.

Per definire come i dati sono convertiti dalla view di rappresentazione alla model view, il *Converter* deve implementare il metodo *getAsObject(FacesContext, UIComponent, String)* dall’interfaccia *Converter*.

Sotto è riportata l’implementazione di questo metodo preso da *creditCardConverter*:

*@Override*

*public Object getAsObject(FacesContext context,*

*UIComponent component, String newValue)*

*throws ConverterException {*

*if (newValue.isEmpty()) {*

*return null;*

*}*

*// Since this is only a String to String conversion,*

*// this conversion does not throw ConverterException.*

*String convertedValue = newValue.trim();*

*if ( (convertedValue.contains("-")) || (convertedValue.contains(" "))) {*

*char[] input = convertedValue.toCharArray();*

*StringBuilder builder = new StringBuilder(input.length);*

*for (int i = 0; i < input.length; ++i) {*

*if ((input[i] == '-') || (input[i] == ' ')) {*

*} else {*

*builder.append(input[i]);*

*}*

*}*

*convertedValue = builder.toString();*

*}*

*return convertedValue;*

*}*

Durante la fase di Apply Request Values, quandoi metodi *decode* del componente vengono processati, JavaServer Faces controlla il valore locale del componente nella richiesta e chiama il metodo *getAsObject*.

Quando questo metodo viene chiamato l’implementazione di JavaServer Faces passa nell’istanza corrente di *FacesContext* il componente che necessita della conversione e il valore locale come *String*.

Il metodo quindi scrive il valore locale in un array di caratteri, toglie punteggiatura e spazi banchi, aggiunge il resto dei caratteri a una stringa e ritorna una *String*.

Per definire come i dati siano convertiti dalla model view alla presentation view, l’implementazione di *Converter* deve implementare il metodo *getAsString(FacesContext, UIComponent, Object)* dall’interfaccia *Converter* (come mostrato sotto).

*@Override*

*public String getAsString(FacesContext context,*

*UIComponent component, Object value)*

*throws ConverterException {*

*String inputVal = null;*

*if ( value == null ) {*

*return "";*

*}*

*// value must be of a type that can be cast to a String.*

*try {*

*inputVal = (String)value;*

*} catch (ClassCastException ce) {*

*FacesMessage errMsg = new FacesMessage(CONVERSION\_ERROR\_MESSAGE\_ID);*

*FacesContext.getCurrentInstance().addMessage(null, errMsg);*

*throw new ConverterException(errMsg.getSummary());*

*}*

*// insert spaces after every four characters for better*

*// readability if they are not already present.*

*char[] input = inputVal.toCharArray();*

*StringBuilder builder = new StringBuilder(input.length + 3);*

*for (int i = 0; i < input.length; ++i) {*

*if ((i % 4) == 0 && (i != 0)) {*

*if ({input[i] != ' ') || (input[i] != '-')){*

*builder.append(" ");*

*// if there are any "-"'s convert them to blanks.*

*} else if (input[i] == '-') {*

*builder.append(" ");*

*}*

*}*

*builder.append(input[i]);*

*}*

*String convertedValue = builder.toString();*

*return convertedValue;*

*}*

Durante la fase Render Response, in cui i metodi *encode* vengono chiamati, L’implementazione di JavaServer Faces chiama il metodo *getAsString* in modo da generare l’output appropriato.

Quando chiama questo metodo, passa nel *FacesContext* corrente il *UIComponent* il cui valore necessita di essere convertito e il valore del bean da convertire.

Dato che questo convertitore fa una conversione *String*-to-*String*, questo metodo può castare il valore del bean come una *String*.

Se il valore non può essere convertito in una *String* il metodo legge la stringa in un array di caratteri e va in loop lungo tutto l’array, aggiungendo uno spazio ogni 4 caratteri.

Si può anche creare un convertitore personalizzato con l’annotazione *@FacesConverter* che specifica l’attributo *forClass*, come mostrato sotto:

*@FacesConverter(forClass=Guardian.class, value="guardian")*

*public class GuardianConverter extends EntityConverter implements Converter { ...*

L’attributo *forClass* registra il convertitore come quello di default per la classe *Guardian*.

Ogni volta che la classe è specificata da un attributo *value* di un componente input, il convertitore verrà invocato automaticamente.

Una classe convertitore può una classe Java POJO’s separata.

Se lo è necessita dell’accesso agli oggetti definiti in una classe managed bean, però può anche essere una sottoclasse di un managed bean, come nell’esercizio *address-book* in cui i convertitori usano un enterprise bean che è iniettato in un managed bean.

### 15.10.2 Usare un convertitore personalizzato

Per applicare la conversione dei dati eseguita da un convertitore personalizzato a un valore di un particolare componente bisogna fare una delle seguenti cose:

* Referenziare il convertitore dall’attributo *converter* del tag del componente
* Annidare un tag *f:converter* dentro il tag del componente e riferirsi al convertitore personalizzato da uno degli attributi del tag *f:converter*.

Se si usa l’attributo *converter* del tag del componente, questo attributo deve riferirsi all’identificatore dell’implementazione *Converter* o il fully-qualified class name del converter.

L’identificatore per la classe converter della carta di credito è *ccno*, il valore specificato nell’annotazione *@FacesConverter*:

*@FacesConverter("ccno")*

*public class CreditCardConverter implements Converter {*

Quindi l’istanza di *CreditCardConverter* può essere registrata sul componente *ccno* come mostrato qui:

*<h:inputText id="ccno"*

*size="19"*

*converter="ccno"*

*value="#{cashierBean.creditCardNumber}"*

*required="true"*

*requiredMessage="#{bundle.ReqCreditCard}">*

*...*

*</h:inputText>*

Settando l’attributo *converter* di un tag di un componente con l’identificativo del convertitore o il suo nome di Classe, si fa in modo che il valore locale del componente venga automaticamente convertito in accordo con le regole specificate nell’implementazione di *Converter*.

Invece di riferirsi al convertitore dall’attributo *converter* del tag del componente, ci si può riferire a lui dal tag *f:converter* innestato all’interno del tag del componente.

Ciò si fa in questo modo:

* Settare l’attributo *converterId* del tag *f:converter* all’identificatore definito nell’annotazione *@FacesConverter* o nel file di configurazione risorse dell’applicazione.

*<h:inputText id="ccno"*

*size="19"*

*value="#{cashierBean.creditCardNumber}"*

*required="true"*

*requiredMessage="#{bundle.ReqCreditCard}">*

*<f:converter converterId="ccno"/>*

*<f:validateRegex*

*pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}"/>*

*</h:inputText>*

* Legare l’implementazione *Converter* a una proprietà del managed bean usando l’attributo *binding* del tag *f:converter* come descritto nel capitolo 15.13.

Se si utilizza l’attributo *converter* tale attributo deve riferirsi all’identificativo o al e fully-qualified class name dell’implementazione di *Converter* del converter.

Il capitolo 15.10 spiegava come implementare il convertitore personalizzato.

L’identificativo per la classe convertitore è *ccno*, il valore specificato nell’annotazione *@FacesConverter*:

*@FacesConverter("ccno")*

*public class CreditCardConverter implements Converter {*

Quindi l’istanza di *CreditCardConverter* può essere registrata sul componente *ccno* come mostrato sotto:

*<h:inputText id="ccno"*

*size="19"*

*converter="ccno"*

*value="#{cashierBean.creditCardNumber}"*

*required="true"*

*requiredMessage="#{bundle.ReqCreditCard}">*

*...*

*</h:inputText>*

Settando l’attributo *converter* del tag di un componente all’identificatore del convertitore o il suo nome di classe, si fa in modo che il valore locale del componente sia automaticamente convertito in base alle regole specificato nell’implementazione di *Converter*.

Invece di riferirsi al convertitore dall’attributo *converter* del tag del componente si può far riferimento al convertitore da un tag *f:converter* innestato dentro il tag del componente.

Per riferirsi al convertitore personalizzato usando il tag *f:converter* di fa uno dei seguenti:

* Settare l’attributo *converterId* del tag *f:converter* all’identificatore dell’implementazione definito nell’annotazione *@FacesConverter* o nel file di configurazione delle risorse dell’applicazione. Questo metodo è mostrato in *bookcashier.xhtml*:

*<h:inputText id="ccno"*

*size="19"*

*value="#{cashierBean.creditCardNumber}"*

*required="true"*

*requiredMessage="#{bundle.ReqCreditCard}">*

*<f:converter converterId="ccno"/>*

*<f:validateRegex*

*pattern="\d{16}|\d{4} \d{4} \d{4} \d{4}|\d{4}-\d{4}-\d{4}-\d{4}"/>*

*</h:inputText>*

* Legare l’implementazione di *Converter* a una proprietà del managed bean usando l’attributo *binding* del tag *f:converter*, come descritto nel capitolo 15.13.

L’implementazione di JavaServer Faces chiama il metodo *getAsObject* del convertitore per togliere gli spazi e la punteggiatura dal valore di input.

Il metodo *getAsString* viene evocato quando la pagina *bookcashier.xhtml* viene ri visualizzata; nel caso dell’esempio succede quando l’utente spende più di 100$.

Nell’esercizio Duke’s Tutoring ogni convertitore viene registrato come il convertitore di una classe specifica.

Il convertitore viene automaticamente evocato ogni volta che la propria classe è specificata dall’attributo *value* di un componente input.

Nel seguente esempio l’attributo *itemValue* (evidenziato in **grassetto**) chiama il convertitore della classe *Guardian*:

*<h:selectManyListbox id="selectGuardiansMenu"*

*title="#{bundle['action.add.guardian']}"*

*value="#{guardianManager.selectedGuardians}"*

*size="5"*

*converter="guardian">*

*<f:selectItems value="#{guardianManager.allGuardians}"*

*var="selectedGuardian"*

*itemLabel="#{selectedGuardian.name}"*

***itemValue="#{selectedGuardian}" />***

*</h:selectManyListbox>*

## 15.11 Creare e usare un validatore personalizzato

Se un validatore standard o bean non esegue il tipo di validazione di cui si ha bisogno, si può costruire un validatore personalizzato.

Come spiegato nel capitolo 7.4.5, ci sono due modi per implementare il codice del validatore:

* Implementare un metodo del managed bean che esegua la validazione
* Offrire un’implementazione dell’interfaccia *javax.faces.validator.Validator* per eseguire la validazione.

Il capitolo 12.3.2 spiegava come implementare un metodo di un managed bean per eseguire la validazione.

Questo capitolo spiegherà come implementare l’interfaccia *javax.faces.validator.Validator*.

Se si sceglie di implementare l’interfaccia e si vuole consentire all’autore della pagina di configurare gli attributi del validatore dalla pagina, bisogna specificare un tag personalizzato per registrare il validatore sul componente.

Se si preferisce configurare gli attributi nell’implementazione di *Validator* si può rinunciare a creare un tag personalizzato e invece lasciare modo all’autore di registrare il validatore su un componente usando il tag *f:validator*.

Si può inoltre creare una proprietà di un managed bean che accetti e ritorni l’implementazione di *Validator* che si ha creato.

Si può usare l’attributo *binding* del tag *f:validator* per legare l’implementazione di *Validator* alla proprietà del managed bean.

In genere si vuole visualizzare un messaggio di errore quando i dati falliscono la validazione.

In tal caso è necessario archiviare tali messaggi di errore in un bundle di risorse.

Dopo aver creato il resoource bundle, ci sono due modi di rendere il messaggio disponibile all’applicazione.

Si possono accodare i messaggi di errore sopra il *FacesContext* in maniera programmatica o si possono registrare i messaggi di errore nel file di configurazione delle risorse dell’applicazione.

Esempio di un tag personalizzato di un validatore per una carta di credito:

*<mystore:formatValidator*

*formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>*

I dati inseriti saranno considerati validi se sono:

* 16 cifre senza spazi
* 16 cifre con uno spazio tra ogni 4 cifre
* 16 cifre con una barretta tra ogni 4 cifre

Il tag *f:validateRegex* rende un validatore personalizzato inutile in questa situazione.

Comunque il resto del capitolo descrive come questo validatore dovrebbe essere implementato e come specificare un tag personalizzato in modo che l’autore della pagina possa registrare il validatore su un componente.

### 15.11.1 Implementare l’interfaccia Validator

Un’implementazione di *Validator* deve contenere un costruttore, un set di metodi accessori per ogni attributo del tag e un metodo *validate,* che faccia l’override del metodo *validate* dell’interfaccia *Validator*.

La classe ipotetica *formatValidator* definisce anche metodi accessori per settare l’attributo *formatPatterns* che specifica il formato del pattern accettabile per l’input nei campi.

Il metodo setter chiama il metodo *parseFormatPatterns* che separa i componenti della stringa pattern in un array di stringhe, *formatPatternList*.

*public String getFormatPatterns() {*

*return (this.formatPatterns);*

*}*

*public void setFormatPatterns(String formatPatterns) {*

*this.formatPatterns = formatPatterns;*

*parseFormatPatterns();*

*}*

Oltre a definire i metodi accessori per gli attributi, la classe esegue l’override del metodo *validate* dell’interfaccia *Validator*.

Questo metodo convalida l’input e accede ai messaggi di errore personalizzati da visualizzare quando una stringa non è valida.

Il metodo *validate* esegue la validazione dei dati.

Questo metodo prende l’istanza *FacesContext,* il componente che necessita di essere convalidato, e il valore che necessita di essere convalidato.

Un validatore può convalidare solo dati di un componente che implementi *javax.faces.component.EditableValueHolder*.

Sotto è riportata l’implementazione del metodo *validate*:

*@FacesValidator*

*public class FormatValidator implements Validator, StateHolder {*

*...*

*public void validate(FacesContext context, UIComponent component,*

*Object toValidate) {*

*boolean valid = false;*

*String value = null;*

*if ((context == null) || (component == null)) {*

*throw new NullPointerException();*

*}*

*if (!(component instanceof UIInput)) {*

*return;*

*}*

*if ( null == formatPatternsList || null == toValidate) {*

*return;*

*}*

*value = toValidate.toString();*

*// validate the value against the list of valid patterns.*

*Iterator patternIt = formatPatternsList.iterator();*

*while (patternIt.hasNext()) {*

*valid = isFormatValid(*

*((String)patternIt.next()), value);*

*if (valid) {*

*break;*

*}*

*}*

*if ( !valid ) {*

*FacesMessage errMsg =*

*new FacesMessage(FORMAT\_INVALID\_MESSAGE\_ID);*

*FacesContext.getCurrentInstance().addMessage(null, errMsg);*

*throw new ValidatorException(errMsg);*

*}*

*}*

*}*

L’annotazione *@FacesValidator* registra la classe *formatValidator* come un validatore tramite l’implementazione di JavaServer Faces.

Il metodo *validate* prende il valore locale del componente e lo converte in *String*.

Poi fa l’iterazione sulla lista *formatPatternsList* che è la lista dei patterns accettabili che viene analizzata dall’attributo *formatPatterns* del tag del validatore personalizzato.

Mentre effettua l’iterazione sulla lista, questo metodo controlla il pattern del valore locale del componente con i patterns della lista.

Se il pattern del valore locale non corrisponde a nessuno della lista il metodo genera un messaggio d’errore.

Poi crea un *javax.faces.application.FacesMessage* e lo accoda nel *FacesContext* per visualizzare, usando una *String* che rappresenta la chiave nel file *Properties*:

*public static final String FORMAT\_INVALID\_MESSAGE\_ID =*

*"FormatInvalid";*

*}*

Infine il metodo passa il messaggio al costruttore di *javax.faces.validator.ValidatorException*.

Quando il messaggio di errore viene visualizzato, il formato del pattern sarà sostituito per il {0} nel messaggio di errore:

*Input must match one of the following patterns: {0}*

Si potrebbe voler salvare e ripristinare lo stato del proprio validatore, anche se solitamente non è necessario.

Per farlo bisogna implementare l’interfaccia *StateHolder,* insieme a *Validator*, e implementare i suoi 4 metodi*: saveState(FacesContext), restoreState(FacesContext, Object), isTransient*, e *setTransient(boolean)*.

### 15.11.2 Specificare un tag personalizzato

Si si ha implementato l’interfaccia *Validator* invece che implementare un metodo di un managed bean che esegue la validazione, si ha bisogno di uno dei seguenti:

* Permettere all’autore della pagina di specificare l’implementazione di *Validator* da usare con il tag *f:validator*. In questo caso l’implementazione di *Validator* deve definire le proprie proprietà.
* Specificare un tag personalizzato che fornisce gli attributi per configurare le proprietà del validatore dalla pagina

Per creare untag personalizzato si ha bisogno di aggiungere il tag al tag library descriptor dell’applicazione, *bookstore.taglib.xml*:

*<tag>*

*<tag-name>validator</tag-name>*

*<validator>*

*<validator-id>formatValidator</validator-id>*

*<validator-class>*

*dukesbookstore.validators.FormatValidator*

*</validator-class>*

*</validator>*

*</tag>*

L’elemento *tag-name* definisce il nome del tag come deve essere usato in una pagina Facelets.

L’elemento *validator-id* specifica il validatore personalizzato.

L’elemento *validator-class* unisce il tag personalizzato alla sua classe di implementazione.

### 15.11.3 Usare un validatore personalizzato

Per registrare un validatore personalizzato su un componente bisogna fare una delle seguenti cose:

* Annidare il tag personalizzato del validatore all’interno del componente di cui si vuole convalidare il valore.
* Annidare il tag standard *f:validator* all’interno del tag del componente e riferirsi all’implementazione del *Validator* personalizzato dal tag *f:validator*.

Sotto è riportato un ipotetico tag personalizzato *formatValidator* per il campo Credit Card Number, annidato all’interno del tag *h:inputText*.

*<h:inputText id="ccno" size="19"*

*...*

*required="true">*

*<mystore:formatValidator*

*formatPatterns="9999999999999999|9999 9999 9999 9999|9999-9999-9999-9999"/>*

*</h:inputText>*

*<h:message styleClass="validationMessage" for="ccno"/>*

Questo tag convalida l’imput del campo *ccno* con il pattern definito nell’attributo *formatPatterns*.

Si può utilizzare lo stesso validatore personalizzato per ogni componente simile semplicemente innestando il tag personalizzato del validatore all’interno del tag del componente.

Se lo sviluppatore dell’applicazione che ha creato il validatore personalizzato preferisce configurare gli attributi nell’implementazione *Validator* invece che permettere di farlo dalla pagina, lo sviluppatore non deve creare il tag personalizzato da usare col validatore.

In questo caso, l’autore della pagina deve annidare il tag *f:validator* all’interno del tag del componente i cui dati necessitano della validazione e poi deve fare una di queste cose:

* Settare l’attributo *validatorId* del tag *f:validator* all’ID del validatore definito nel file di configurazione delle risorse dell’applicazione.
* egare l’implementazione personalizzata di *Validator* a una proprietà di un managed bean usando l’attributo *binding* del tag *f:validator*.

Il seguente tag registra un validatore ipotetico su un componente usando un tag *f:validator* che si riferisce all’ID del validatore:

*<h:inputText id="name" value="#{CustomerBean.name}"*

*size="10" ...>*

*<f:validator validatorId="customValidator" />*

*...*

*</h:inputText>*

## 15.12 Binding Component Values and Instances to Managed Bean Properties

Un tag di un componente può legare i propri dati a un managed bean con uno dei seguenti metodi:

* Legare il valore del proprio componente a una proprietà del bean.
* Legare l’istanza del proprio componente a una proprietà del bean.

Per legare il valore del componente a una proprietà del managed bean, l’attributo *value* del tag del componente utilizza una value expression EL.

Per legare l’istanza del componente a una proprietà del bean, l’attributo *binding* del tag del componente utilizza una value expression.

Quando un’istanza viene collegata a una proprietà di un managed bean, la proprietà prende il valore locale del componente mentre quando si lega il valore del componente la proprietà prende il valore immagazzinato nel managed bean.

Questo valore è sovrascritto con il valore locale durante la fase Model Values del ciclo vitale.

Entrambi i metodi hanno alcuni vantaggi:

Legare l’istanza del componente alla proprietà del bean:

* Il managed bean può modificare programmaticamente gli attributi del componente.
* Il managed bean può istanziare i componenti invece di lasciarlo fare all’autore della pagina.

Legare il valore del ccomponente a una proprietà del bean ha i seguenti vantaggi:

* L’autore della pagina ha più controllo sugli attributi del componente.
* Il managed bean non ha dipendenze sull’API JavaServer Faces (come le classi componenti) e permette una migliore separazione del presentation layer dal model layer.
* L’implementazione JavaServer Faces può effettuare conversioni di dati basate sul tipo della proprietà del bean senza che lo sviluppatore debba applicare un convertitore.

Nella maggior parte dei casi si legherà il valore del componente invece che la sua istanza.

Potrebbe essere utile legare il componente solo quando si necessita di cambiare gli attributi dinamicamente.

Ad esempio se un’applicazione renderizza un componente solo a certe condizioni, si può impostare la proprietà *rendered* in maniera conseguente per accedere alla proprietà a cui il componente è legato.

Quando ci si riferisce alla proprietà tramite l’utilizzo dell’attributo *value* del componente, bisogna usare la sintassi appropriata.

Ad esempio supponiamo che un managed bean chiamato *MyBean* abbia questa proprietà *int*:

*protected int currentOption = null;*

*public int getCurrentOption(){...}*

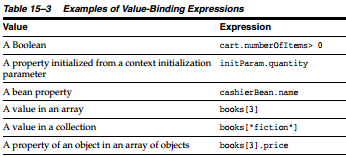
*public void setCurrentOption(int option){...}*

L’attributo *value* che si riferisce a tale proprietà deve avere questa espressione che lega il valore:

*#{myBean.currentOption}*

Oltre a legare il valore di un componente a una proprietà di un bean, l’attributo *value* può specificare un valore letterale o può associare i dati del componente a qualsiasi tipo primitivo (tipo *int*), struttura (un array ad esempio) o collezione (come una *list*) indipendentemente da un componente JavaBeans.

La tabella sotto riportata elenca alcuni esempi di espressioni value-binding che si possono usare con un attributo *value*:



### 15.12.1 Binding a Component Value to a Property

Per legare un valore del componente a una proprietà del managed bean, si specifica il nome del bean e la proprietà usando l’attributo *value*.

Ciò significa che la prima parte della value-expression EL deve combaciare con il nome del managed bean fino al punto (.) e la parte dopo il punto deve combaciare con la proprietà del managed bean.

Esempio:

*<h:dataTable id="books"*

***value="#{store.books}"***

*var="book"*

*headerClass="list-header"*

*styleClass="list-background"*

*rowClasses="list-row-even, list-row-odd"*

*border="1"*

*summary="#{bundle.BookCatalog}">*

Il valore è ottenuto chiamando il metodo *getBooks* del backing bean che chiama il metodo *getBooks* del session bean *BookRequestBean*.

Se si usa il file di configurazione delle risorse per configurare il managed bean invece di definirlo nella classe del managed bean, il nome del bean nella value expression deve combaciare con l’elemento *managed-bean-name* della dichiarazione del managed bean fino al punto e la parte seguente deve essere il nome specificato nell’elemento *property-name* del file di configurazione, ad esempio (preso dall’esempio *Duke’s Bookstore*):

*<managed-bean eager="true">*

*...*

*<managed-bean-name>Book201</managed-bean-name>*

*<managed-bean-class>dukesbookstore.model.ImageArea</managed-bean-class>*

*<managed-bean-scope>application</managed-bean-scope>*

*<managed-property>*

*...*

*<property-name>shape</property-name>*

*<value>rect</value>*

*</managed-property>*

*<managed-property>*

*...*

*<property-name>alt</property-name>*

*<value>Duke</value>*

*</managed-property>*

*...*

Questo esempio configura un bean chiamato *Book201* che ha diverse proprietà, una delle quali è chiamata *shape*.

Sebbene i tags *bookstore:area* della pagina *index.xhtml* non siano legati a una proprietà di *ImageArea* (sono legati direttamente al bean), ci si può riferire alla proprietà usando una value expression dall’attributo *value* del tag del componente:

*<h:outputText value="#{Book201.shape}" />*

### 15.12.2 Binding a Component Value to an Implicit Object

Una fonte dati esterna a cui l’attributo *value* può riferirsi è un oggetto implicito.

La pagina *bookreceipt.xhtml* del Duke’s Bookstore ha un riferimento a un oggetto implicito:

*<h:outputFormat title="thanks"*

*value="#{bundle.ThankYouParam}">*

*<f:param value="#{sessionScope.name}"/>*

*</h:outputFormat>*

Questo tag prende il nome dell’utente dal session scope e lo inserisce nel messaggio parametrizzato nel punto chiave *ThankYouParam* dal rsource bundle.

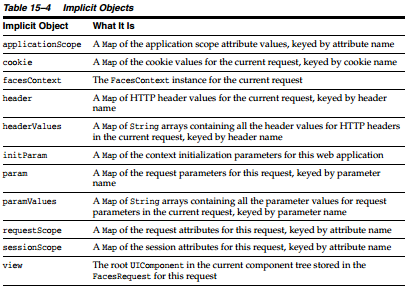
Ad esempio se il nome dell’utente è *Gwen Canigetit* questo tag visualizzerà:

*Thank you, Gwen Canigetit, for purchasing your books from us*.

Il recupero dei valori da altri oggetti impliciti viene effettuato in modo simile all’esempio mostrato in questo capitolo.

La tabella sottostante elenca gli oggetti impliciti a cui un attributo *value* può riferirsi.

Tutti gli oggetti impliciti ad eccezione degli oggetti *scope* sono di sola lettura e quindi non possono essere usati come valori per i componenti di input.



### 15.12.3 Binding a Component Instance to a Bean Property

Un’istanza di un componente può essere legata a una proprietà di un bean usando una value expression con l’attributo *binding* del tag del componente.

Solitamente si effettua questo tipo di binding invece che quello sul *value* se il bean deve cambiare dinamicamente gli attributi del componente.

Sotto ci sono due tags presi a *bookcashier.xhtml* le legano componenti a proprietà del bean:

*<h:selectBooleanCheckbox id="fanClub"*

*rendered="false"*

*binding="#{cashierBean.specialOffer}" />*

*<h:outputLabel for="fanClub"*

*rendered="false"*

*binding="#{cashierBean.specialOfferText}"*

*value="#{bundle.DukeFanClub}"/>*

*</h:outputLabel>*

Il tag *h:selectBoolean* renderizza un checkbox e lega il componente *UISelectBoolean fanClub* alla proprietà *specialOffer* del *cashierBean*.

Il tag *h:outputLabel* lega il componente rappresentante la label del checkbox alla proprietà *specialOfferText* del *cashierBean*.

Se il *locale* dell’applicazione è English, il tag *h:outputLabel* renderizza:

*I'd like to join the Duke Fan Club, free with my purchase of over $100*

Gli attributi *rendered* di entrambi i tags sono settati su false per non far visualizzare il checkbox e il suo label.

Questi tags utilizzano il binding al componente invece che al *value* perché il managed bean necessita di settare dinamicamente il valore della proprietà *rendered*.

Se facessero il contrario, il managed bean non avrebbe accesso diretto ai componenti e avrebbe avuto bisogno di codice addizionale per accedervi dall’istanza *FacesContext* per cambiare i valori di *rendered*.

## 15.13 Binding Converters, Listeners, and Validators to Managed Bean Properties

Come già detto nei capitoli precedenti l’autore della pagina può legare convertitori, listeners e validatori alle proprietà dei managed beans attraverso l’attributo *binding* del tag che viene usato per registrare l’implementazione sui componenti.

Questa tecnica ha vantaggi simili al binding delle istanze dei componenti spiegato poco fa.

In particolare, legare un convertitore, listener o validatore a una proprietà di un managed bean fornisce i seguenti vantaggi:

* Il managed bean può istanziare l’implementazione invece di lasciarlo fare all’autore della pagina.
* Il managed bean può modificare programmaticamente gli attributi dell’implementazione.

nel caso di un’implementazione personalizzata, l’unico altro modo di farlo all’infuori della classe di implementazione sarebbe quello di creare appositamente un tag personalizzato e richiedere all’autore della pagina di settare i valori dell’attributo dalla pagina stessa.

Il processo per legare convertitori, validatori e listeners è lo stesso:

* Si annida il tag del convertitore/validatore/listener all’interno del componente appropriato.
* Ci si assicura che il managed bean abbia una proprietà che accetti e ritorni la classe d’implementazione del convertitore/validatore/listener che si vuole legare alla proprietà.
* Ci si riferisce alla proprietà usando una value expression sull’attributo *binding*  del tag del convertitore/validatore/listener

Ad esempio supponiamo di vuole legare un convertitore standard *DateTime* a una proprietà del managed bean perché si vuole settare il pattern di formattazione dell’input dell’utente nel managed bean invece che sulla pagina Facelets.

Per prima cosa la pagina registra il convertitore sul componente annidando il tag *f:convertDateTime* all’interno del tag del componente.

Poi la pagina fa riferimento alla proprietà tramite l’attributo *binding* del tag:

*<h:inputText value="#{loginBean.birthDate}">*

*<f:convertDateTime binding="#{loginBean.convertDate}" />*

*</h:inputText>*

La proprietà *convertDate* sarà qualcosa di simile a questo:

*private DateTimeConverter convertDate;*

*public DateTimeConverter getConvertDate() {*

*...*

*return convertDate;*

*}*

*public void setConvertDate(DateTimeConverter convertDate) {*

*convertDate.setPattern("EEEEEEEE, MMM dd, yyyy");*

*this.convertDate = convertDate;*

*}*