# 16 Configuring JavaServer Faces Application

Il processo di costruzione e distribuzione di semplici applicazioni JavaServer Faces è descritto nei capitoli 6, 8, 13 e 14. Quando crei applicazioni larghe e complesse, tuttavia, sono richiesti vari incarichi aggiuntivi. Tra questi troviamo:

* Registrare i managed bean con l’applicazione così che tutte le parti dell’applicazione abbia accesso ad essi.
* Configurare i managed bean e i model beans così che siano istanziati con il valore appropriato quando una pagina fa riferimento a loro.
* Definire le regole di navigazione per ogni pagina nell’applicazione così che abbia un flusso liscio di pagina, se è necessaria la navigazione non di default
* Impacchettare l’applicazione per includere tutte le pagine, risorse, e altri files così che l’applicazione possa essere distribuita su qualsiasi container compatibile.

## 16.1 Annotations to Configure Managed Beans

Le Bean Annotation possono essere usate per configurare le applicazioni JavaServer Faces.

L’annotazione *@Named* (*javax.inject.Named*) in una classe, insieme ad un’annotazione scope, registra automaticamente questa classe come risorsa con l’implementazione JavaServer Faces. Un bean che usa queste annotazioni è un managed bean CDI.

Ecco mostrato l’utilizzo delle annotazioni *@Named* e *@SessionScoped* in una classe:

@Named(“cart”)

@SessionScoped

Public class ShoppingCart ... { ... }

Questo frammento di codice mostra un bean gestito (“managed”) dall’implementazione JavaServer Faces ed è disponibile per la durata della sessione.

Tutte le classi saranno esaminate per le annotazioni all’inizializzazione finché l’elemento *faces-config* nel file *faces-config.xml* non ha l’attributo *metadata-complete* impostato su *true*.

Le annotazioni sono inoltre disponibili per altri elementi, come componenti, convertitori, validatori, renderers, per essere usate al posto dell’inserimento di un file di configurazione della risorsa.

### 16.1.1 Using Managed Bean Scopes

Puoi usare le annotazioni per definire lo scope in cui il bean sarà memorizzato. Puoi specificare uno dei seguenti scope per una classe bean:

* **Application** (*javax.enterprise.context.ApplicationScoped*): L’Application scope persiste attraverso tutte e le interazioni degli utenti con un’applicazione web.
* **Session** (*javax.enterprise.context.SessionScoped*): Il Session scope persiste attraverso richieste HTTP multiple in un’applicazione web.
* **Flow** (*javax.faces.flows.FlowScoped*): Il Flow scope persiste durante un’interazione dell’utente con un flusso specifico dell’applicazione web.
* **Request** (*javax.enterprise.context.RequestScoped*): Il Request scope persiste durante una singola richiesta HTTP nell’applicazione web.
* **Dependent** (*javax.enterprise.context.Dependent*): Indica che il bean dipende da qualche altro bean.

Potresti voler usare *@Dependent* quando un managed bean fa riferimento ad un altro managed bean. Il secondo bean non dovrebbe essere in uno scope (*@Dependent*) se si suppone che sia stato creato solo una volta referenziato. Se definisci un bean come @*Dependent*, il bean viene istanziato di nuovo ogni volta che viene referenziato, così non viene salvato in alcuno scope.

Se il managed bean viene referenziato dall’attributo *binding* di un component tag, dovresti definire un bean con un request scope. Se invece ponessi il bean in un session scope o un application scope, il bean dovrebbe prendere precauzioni per assicurare la sicurezza del thread, poiché le istanze di *java.faces.component.UIConponent* dipendono dall’esecuzione dentro un singolo thread.

Se stai configurando un bean che consente agli attributi di essere associati con la view, puoi usare il view scope. Gli attributi persistono finché l’utente non naviga verso un’altra view.

## 16.2 Application Configuration Resource File

La tecnologia JavaServer Faces fornisce un formato di configurazione transportabile (come un documento XML) per configurare le risorse dell’applicazione. Uno o più documenti XML, chiamati *application configuation resource flies*, può usare questo formato per registrare e configurare gli oggetti e le risorse e per definire le regole di navigazione per l’applicazione. Il file di configurazione di un’applicazione è di solido chiamato *faces-config.xml*.

Hai bisogno del file di configurazione delle risorse nei seguenti casi:

* Per specificare gli elementi di configurazione per la tua applicazione che non sono disponibili attraverso le annotazioni dei managed bean, come i messaggi localizzati e le regole di navigazione.
* Per fare l’override delle annotazioni del managed bean quando l’applicazione viene distribuita.

Il file di configurazione delle risorse dell’applicazione deve essere conforme allo schema XML allocato [*http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_2.xsd*](http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd).

In aggiunta, ogni file deve includere le seguenti informazioni, nel seguente ordine:

* Il numero della versione di XML, in genere con l’attributo *encoding*:

*<?xml version=”1.0” encoding=’UTF-8’?>*

* Un tag *faces-config* in chiusura di tutte le altre dichiarazioni:

*<faces-config version=”2.2” xmlns=”http://xmlns.jcp.org/xml/ns/javaee”*

*xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”*

*xsi:schemaLocation=”http://xmlns.jcp.org/xml/ns/javaee*

*http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_2.xsd”>*

*...*

*</faces-config>*

Puoi avere più di un file di configurazione delle risorse dell’applicazione. L’implementazione JavaServer Faces trova il file di configurazione o i files cercando:

* Una risorsa chiamata */META-INF/faces-config.xml* in qualsiasi file JAR nella directory dell’applicazione web */WEB-INF/lib* e nei caricatori delle classi parent. Se esiste una risorsa con questo nome, viene caricata come risorsa di configurazione. Questo metodo è pratico per una packaged library contenente alcuni componenti e renderers. In aggiunta, anche qualsiasi file con un nome che finisce con *faces-config.xml* è considerato una risorsa di configurazione ed è caricato come tale.
* Un parametro di inizializzazione del contesto, *javax.faces.application.CONFIG\_FILES*, nel file web deployment descriptor che specifica uno o più percorsi (delimitati da virgole) a files di configurazione multipli per l’applicazione web. Questo metodo è più spesso usato per le applicazioni enterprise-scale che delega a separati gruppi la responabilità della manutenzione del file per ogni porzione di una grande applicazione.
* Una risorsa chiamata *faces-config.xml* nella directory /WEB-INF/ dell’applicazione. Le applicazioni web semplici rendono disponibili i files di configurazione in questo modo.

Per accedere alle risorse registrate con l’applicazione, uno sviluppatore può usare un’istanza della classe *javax.faces.application.Application,* che viene automaticamente creata per ogni applicazione. L’istanza *Application* agisce come una fabbrica centralizzata per le risorse definite nel file XML.

Quando l’applicazione viene inizializzata, l’implementazione JavaServer Faces crea una singola istanza della classe *Application* e la configura con le informazione fornite nel file di configurazione dell’applicazione.

### 16.2.1 Configuring Eager Application-Scoped Managed Beans

I managed beans JavaServer Faces (specificati o nel file *faces-config.xml* o annotati con *javax.faces.bean.ManagedBean*) sono istanziati pigramente (Vai a vedere Lazy Initialization <https://it.wikipedia.org/wiki/Lazy_initialization>). In questo modo, sono istanziati quando viene fatta una richiesta dall’applicazione.

Per forzare l’application-scoped bean ad essere istanziato e allocato nell’application scope appena viene fatta partire l’applicazione e prima che qualsiasi request venga fatta, l’attributo *eager* del managed bean dovrebbe essere settato su *true*, come mostrato nel seguente esempio.

La dichiarazione del file *faces-config.xml* come segue:

*<managed-bean eager=”true”>*

L’annotazione come segue:

@ManagedBean(eager=true)

@ApplicationScoped

### 16.2.2 Ordering of Application Configuration Resource Files

Poiché la tecnologia JSF consente l’uso di file di configurazione delle risorse dell’applicazione multipli memorizzati in diverse posizioni, l’ordine in cui sono caricati dall’implementazione diventa importante in certe situazioni (per esempio, quando si usano oggetti a livello dell’applicazione). Questo ordine può essere definito attraverso un elemento *ordering* e i suoi sottoelementi nel file di configurazione stesso. L’ordine dei files di configurazione delle risorse può essere assoluto o relativo.

L’ordine assoluto viene definito da un elemento *absolute-ordering* nel file. Con ordine assoluto, l’utente specifica l’ordine in cui i file di configurazione delle risorse verrà caricato. Il seguente esempio mostra l’ingresso per un ordine assoluto.

Il file *my-faces-config.xml* contiene i seguenti elementi:

<faces-config>

<name>myJSF</name>

<absolute-ordering>

<name>A</name>

<name>B</name>

<name>C</name>

</absolute-ordering>

</faces-config>

In questo esempio, A, B e C sono diversi file di configurazione delle risorse e stanno per essere caricati nell’ordine della lista.

Se c’è un elemento *absolute-ordering* nel file, solo i file elencati nei sotto-elementi *name* sono processati. Per processare qualsiasi altro file di configurazione, è necessario un sotto-elemento *others*. In assenza di sotto-elementi *other*,tutti gli altri file non elencati saranno ignorati al momento del caricamento.

L’ordine relativo è definito da un elemento *ordering* e i suoi sotto-elementi *before* e *after*. Con l’ordine relativo, l’ordine in cui saranno caricati i file di configurazione delle risorse è calcolato considerando l’ordine di ingressi dai diversi files. Il seguente esempio mostra alcune di queste considerazioni. Nel seguente esempio, *config-A*, *config-B*, e *config-C* sono diversi files di configurazione.

Il file *config-A* contiene i seguenti elementi:

<faces-config>

<name>config-A</name>

< ordering>

<before>

<name>config-B</name>

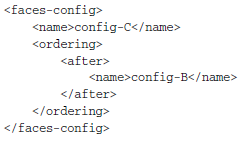
</before>

</ordering>

</faces-config>

Il file *config-B* (non mostrato qui) non contiene elementi *ordering*.

Il file *config-C* contiene i seguenti elementi:



Basato sull’ingresso del sotto-elemento *before*, il file *config-A* sarà caricato prima del file *config-B*. Basato sull’ingresso del sotto-elemento *after*, il file *config-C* sarà caricato dopo il file *config-B*.

In aggiunta, un sotto-elemento *other* può anche essere annidato nei sotto-elementi *before* e *after*. Se l’elemento *other* è presente, il file specificato potrebbe ricevere preferenza più alta o più bassa tra entrambi i file di configurazione elencati e non elencati.

Se non è presente un elemento *ordering* in un file di configurazione, allora questo file sarà caricato dopo tutti i file che contengono elementi *ordering*.

### 16.3 Using Faces Flows

La caratteristica Faces Flows della tecnologia JSF permette di creare un set di pagine con uno scope, *FlowScoped*, migliore che il request scope ma meno del session scope. Per esempio, potresti voler creare una serie di pagine per il processo di checkout in un deposito online. Potresti creare un set di pagine auto-contenute trasferibili da un deposito all’altro come necessario.

Faces Flows è qualcosa di simile alle sub-routines nella programmazione procedurale, nei seguenti modi.

* Come una sub-routine, un flusso ha un punto d’ingresso, un lista di parametri e un valore di ritorno ben definito. Tuttavia, diversamente da una sub-routine, un flusso può restituire valori multipli.
* Come una sub-routine, un flusso ha uno scope, permettendo alle informazioni di essere disponibili solo durante l’invocazione del flusso. Tali informazioni non sono disponibili al di fuori dello scope del flusso e non consuma altre risorse una volta che il flow dà un risultato.
* Come una sub-routine, un flusso può richiamare altri flussi prima di dare un risultato. L’invocazione di flussi è conservata in una call stack: un nuovo flusso genera una spinta sulla stack, e un ritorno causa un pop.

Un’applicazione può avere un qualsiasi numero di flussi. Ogni flusso include un set di pagine e, di solito, uno o più managed beans nell’ambito del flusso. Ogni flusso ha un punto di partenza, chiamato nodo di partenza (start node), e un punto di uscito, chiamato nodo di ritorno (return node).

I dati in un flusso sono solo nell’ambito di questo flusso, ma puoi trasferire i dati da un flusso ad un altro specificando i parametri e richiamando l’altro flusso.

I flussi possono essere annidati, così che se chiami un flusso da un altro e poi esci dal secondo, torni al flusso chiamante anziché al nodo di ritorno del secondo flusso.

Puoi configurare un flusso programmaticamente, creando una classe annotata con @*FlowDefinition*, o puoi configurare un flusso usando un file di configurazione. Il file di configurazione può essere limitato a un flusso, o puoi usare il file *faces-config.xml* per inserire tutti i flow in un unico posto, se si hanno diversi flussi in un’applicazione. La configurazione programmatica alloca il codice più vicino al resto del flusso e agevola la modularizzazione dei flussi.

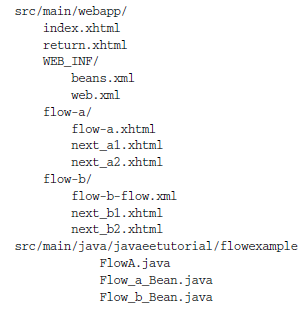
La figura mostra due flussi e le loro interazioni:



In questa figura, il Flow A ha un nodo di partenza chiamato *flow-a* e due pagine aggiuntive, *next\_a1* e *next\_a2*. Da *next\_a2*, un utente può uscire dal flusso usando il nodo di ritorno definito, *taskFlowReturn1*, o chiamare il Flow B, passando due parametri. Il Flow A definisce inoltre due parametri in entrata che può accettare dal Flow B. Il Flow B è identico al Flow A eccetto per il nome del flusso e per i files. Ogni flusso ha anche un managed bean associato; i beans sono *Flow\_a\_Bean* e *Flow\_b\_Bean*.

### 16.3.1 Packaging Flows in an Application

Tipicamente, si impacchettano i flussi in un’applicazione web usando un struttura di directory che modularizza i flussi. Nella directory *src/main/webapp* di un progetto Maven, per esempio, dovresti allocare i files Facelets esterni al flusso al livello più alto. Poi i files *webapp* per ogni flusso dovrebbero essere in directory separate, e i files Java dovrebbero essere sotto *src/main/java*. Per esempio, i files per l’applicazione mostrata in figura dovrebbe essere così:



In questo esempio, *flow-a* è definito programmaticamente in *FlowA.java*, mentre *flow-b* è definito nel file di configurazione *flow-b-flow.xml*.

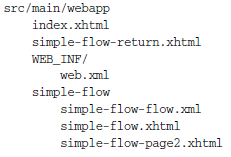
### 16.3.2 The Simplest Possible Flow: The simple-flow Example Application

L’applicazione esempio *simple-flow* dimostra la i blocchi di costruzione più basici di un’applicazione Faces Flows e illustra alcune convenzioni che rendono semplice la partenza con lo sviluppo iterativo usando i flussi. Potresti voler partire con un semplice e costruire su questo.

Questo esempio fornisce una **definizione di flusso implicita** che include un file di configurazione vuoto. Un file di configurazione che contenuto, o una classe annotata @*FlowDefinition*, fornisce una **definizione esplicita di flusso**.

Il codice sorgente per quest’applicazione è nella directory ***tut-install****/example/web/jsf/simple-flow/.*

Il layout dei file dell’esempio *simple-flow* è così:



L’esempio *simple-flow* ha un file di configurazione vuoto, convenzionalmente chiamato ***flow-name****-flow.xml*. Il flusso non richiede alcuna configurazione per le seguenti ragioni.

* Il flusso non chiama altri flussi, né passa parametri ad altri flussi.
* Il flusso usa i nomi di default per la prima pagina del flow, ***flow-name****.xhtml*, e la pagina di ritorno, ***flow-name****-return.xhtml*.

Questo esempio ha solo quattro pagine Facelets.

* *Index.xhtml*, la pagina di partenza, che contiene quasi nulla tranne un bottone che naviga alla prima pagina del flow:



* *simple-flow.xhtml* e *simple-flow-page2.xhtml*, due pagine dello stesso flusso. In assenza di una definizione esplicita di flusso, la pagina il cui nome è lo stesso del nome del flusso è considerata il nododi partenza del flusso. In questo caso, il flusso è denomiinato *simple-flow*, quindi la pagina denominata *simple-flow.xhtml* è considerata il nodo di partenza del flusso. Il nodo di partenza è il nodo a cui si naviga per l’ingresso nel flusso. Può essere immaginato come la pagina home del flusso.

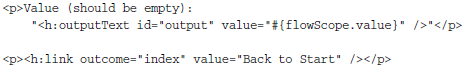
La pagina *simple-flow.xhtml* chiede di inserire un valore dell’ambito del flusso e fornisce un bottone che naviga verso la pagina successiva del flusso:



La seconda pagina, che può avere qualsiasi nome, mostra il valore dell’ambito dello scope e fornisce un bottone per navigare alla pagina di ritorno:

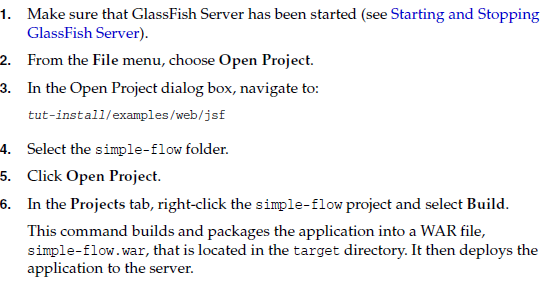


* *simple-flow-return.xhtml*, la pagina di ritorno. La pagina di ritorno, che è chiamata convenzionalmente ***flow-name****-return.xhtml*, deve essere allocata fuori dal flusso. Questa pagina mostra il valore dell’ambito dello scope, per mostrare che non ha valore fuori dal flusso, e fornisce un link per navigare alla pagina *index.xhtml*:



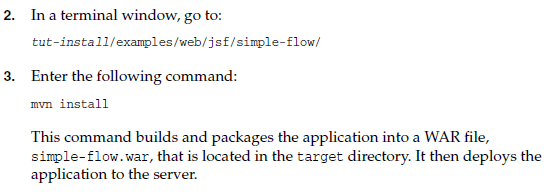
Le pagine Facelets usano solo dati dell’ambito del flusso, così l’esempio non necessita di un managed bean.

#### 16.3.2.1 To Build, Package, and Deploy the simple-flow Example Using NetBeans IDE

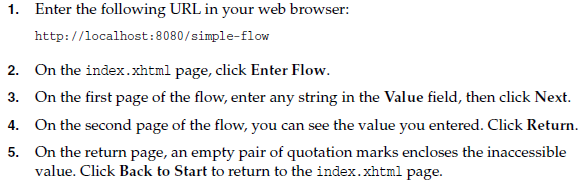


#### 16.3.2.2 To Build,Package, and Deploy the simple-flow Example Using Maven





#### 16.3.2.3 To Run the simple-flow Example



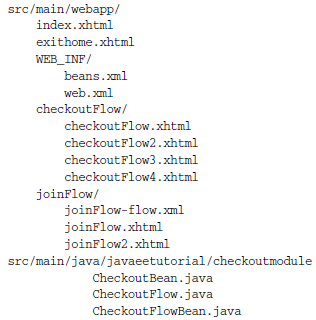
### 16.3.3 The checkout-module Example Application

L’applicazione esempio *checkout-module* è considerabilmente più complessa di *simple-flow*. Questo mostra come potresti usare la caratteristica di Faces Flows per implementare un modulo di checkout per un negozio online.

L’esempio contiene due flussi, ognuno dei quali può chiamare l’altro. Entrambi i flussi hanno una definizione di flusso esplicita. Un flow, *checkoutFlow*, è specificata programmaticamente. L’altro flusso, *joinFlow*, è specificato in un file di configurazione.

Il codice sorgente per l’applicazione è nella directory ***tut-install****/examples/web/jsf/checkout-module/*

Per l’applicazione *checkout-module*, la struttura della directory è come segue (c’è anche una directory *src/main/webapp/resources* con uno stylesheet e un’immagine):



Per l’esempio, *index.xhtml* è la pagina iniziale per l’applicazione così come il nodo di ritorno per il flusso di checkout. La pagina *exithome.xhtml* è il nodo di ritorno per il flusso di collegamento.

Il file di configurazione *joinFlow-flow.xml* definisce il flusso di collegamento, e il file sorgente *CheckoutFlow.java* definisce il flusso di checkout.

Il flusso di checkout contiene quattro pagine Facelets, mentre il flusso ne contiene due.

I managed beans in scope ad ogni flusso sono *CheckoutFlowBean.java* e *JoinFlowBean.java,* mentre *CheckoutBean.java* è il bean di sostegno per la pagina *index.html*.

#### 16.3.3.1 The Facelets Pages for the checkout-module Example

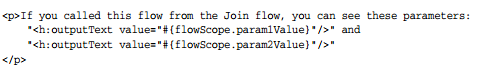
La pagina di partenza per l’esempio, *index.xhtml*, riassume i contenuti dell’ipotetico carrello della spesa. Questo consente all’utente di cliccare uno dei bottoni per entrare in uno dei due flussi:



Questa pagina è inoltre il nodo di ritorno per il flusso di checkout.

La pagina Facelets *exithome.xhtml* è il nodo di ritorno per il flusso di collegamento. Questa pagina ha un bottone che consente di tornare alla pagina *index.xhtml*.

Le quattro pagine Facelets all’interno del flusso di checkout, partendo con *checkoutFlow.xhtml* e terminando con *checkoutFlow4.xhtml*, consentono di procedere alla pagine successiva o, in alcuni casi, di ritornare dal flusso. La pagina *checkoutFlow.xhtml* consente di accedere ai parametri passati dal flusso di collegamento attraverso lo scope del flusso. Questi appaiono come vuoti simboli interrogativi se non è stato richiamato il flusso di checkout dal flusso di collegamento.



Solo *checkoutFlow2.xhtml* ha un bottone per tornare alla pagina precedente, ma muoversi tra le pagine è generalmente permesso all’interno del flusso. Ecco i bottoni per *checkoutFlow2.xhtml*:



L’azione *returnFromCheckoutFlow*è definita nel codice del file di configurazione, *CheckoutFlow.java*.

La pagina finale del flow di checkout, *checkoutFlow4.xhtml*, contiene un bottone che richiama il flow di collegamento:



L’azione *calljoin* è anche definita nel codice del file di configurazione, *CheckoutFlow.java*. Questa azione inserisce nel flusso, passando due parametri dal flusso di checkout.

Le due pagine del flusso di collegamento, *joinFlow.xhtml* e *joinFlow2.xhtml*, sono simili alle pagine del flusso di checkout. La seconda pagina ha un bottone per richiamare il flusso di checkout come uno per ritornare dal flusso di collegamento:



Per questo flusso, le azione *callcheckoutFlow* e *returnFromJoinFlow* sono definite nel file di configurazione *joinFlow-flow.xml*.

#### 16.3.3.2 Using a Configuration File to Configure a Flow

Se usi un file di configurazione delle risorse dell’applicazione per configurare un flusso, dev’essere denominato ***flowName****-flow.xml*. in questo esempio, il flusso di collegamento usa un file di ocnfigurazione denominato *joinFlow-flow.xml*. Questo file è un file *faces-config* che specifica un elemento *flow-definition*. Questo elemento deve definire il nome del flusso usando l’attributo *id*. Sotto l’elemento *flow-definition*, ci deve essere un elemento *flow-return* che specifica il punto di ritorno per il flusso. Qualsiasi parametro in entrata è specificato con l’elemento *inbound-parameter*. Se il flusso richiama un altro flusso, l’elemento *call-flow* deve usare l’elemento flow-reference per nominare il flow chiamato e può usare l’elemento *outbound-parameter* per specificare qualsiasi parametro in uscita.

Il file di configurazione per il flusso di collegamento è così:



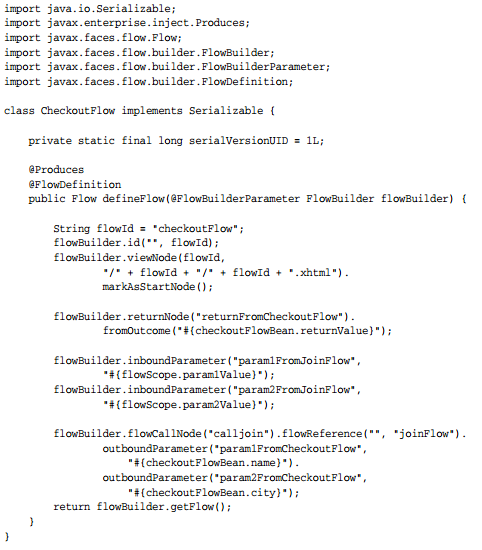
L’attributo *id* dell’elemento *flow-definition* definnisce il nome del flusso come *joinFlow*. Il valore dell’attributo *id* dell’elemento *flow-return* identifica il nome del nodo di ritorno, e il suo valore è definito nell’elemento *from-outcome* come la proprietà *returnValue* del managed bean per il flusso di collegamento, *JoinFlowBean*.

I nomi e i valori dei parametri in entrata sono recuperati dallo scope del flusso in ordine (*flowScope.paramValue*, *flowScope.param2Value*), basato sul modo in cui sono definiti nella configurazione del flusso di checkout.

L’elemento *flow-call* definisce come il flusso di collegamento richiama il flusso di checkout. L’attributo *id* di un elemento, *callcheckoutFlow*, definisce l’azione di richiamare il flusso. All’interno dell’elemento *flow-call*, l’elemento *flow-reference* definisce il nome effettivo del flusso da richiamare, *checkoutFlow*. L’elemento *outbound-parameter* definisce il parametro da passare quando viene chiamato *checkoutFlow*. Qui sono solo stringhe arbitrarie.

#### 16.3.3.3 Using a Java Class to Configure a Flow

Se usi una classe Java per configurare un flusso, deve avere il nome del flusso. La classe per il flusso di checkout è chiamata *CheckoutFlow.java*.



La classe esegue azioni circa identiche a quelle eseguite dal file di configurazione *joinFlow-flow.xml*. questo contiene un singolo metodo, *defineFlow*, come metodo produttore con il qualificatore *@FlowDefinition* che restituisce una classe *javax.faces.flow.Flow*. Il metodo *defineFlow* prende un parametro, un *FlowBuilder* con il qualificatore *@FlowBuilderParameter*, che viene passato dall’implementazione JSF. Il metodo poi richiama i metodi dalla classe *javax.faces.flow.Builder.FlowBuilder* per configurare il flusso.

Prima, il metodo definisce l’*id* del flusso come *checkoutFlow*. Poi, definisce esplicitamente il nodo di partenza per il flusso. Di default, questo è il nome del flusso con suffisso *.xhtml*

Il metodo poi definisce il nodo di ritorno analogamente a come fa il file di configurazione. Il metodo *returnNode* imposta il nome del nodo di ritorno come *returnFromCheckoutFlow* e il metodo collegato *fromOutcome* specifica il suo valore come la proprietà *returnValue* del managed bean per il flusso di checkout, *CheckoutFlowBean*.

Il metodo *inboundParameter* imposta i nomi e i valori dei parametri in entrata dal flusso di collegamento, che vengono recuperati dallo scope del flusso in ordine (*flowScope.paramValue*, *flowScope.param2Value*), in base al modo in cui sono definiti nella configurazione del flusso di collegamento.

Il metodo *flowCallNode* definisce come il flusso di checkout richiama il flusso di collegamento, l’argomento, *calljoin*, specifica l’azione di richiamo del flusso. Il metodo collegato *flowReference* definisce il nome effettivo del flusso da richiamare, *joinFlow*, poi richiama i metodi *outboundParameter* per definire i parametri da passare quando viene richiamato *joinFlow*. Qui ci sono i valori dal managed bean *CheckoutFlowBean*.

Infine, il metodo *defineFlow* richiama il metodo *getFlow* e restituisce il risultato.

#### 16.3.3.4 The Flow-Scoped Managed Bean

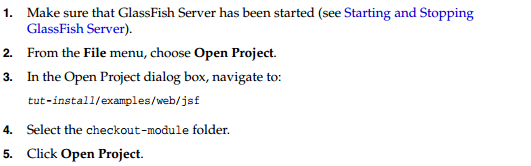
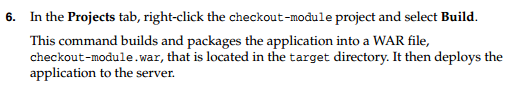
Ognuno dei due flussi ha un managed bean che definisce le proprietà per le pagine interne al flusso. Per esempio, il *CheckoutFlowBean* definisce le proprietà i cui valori sono inseriti dall’utente su entrambe le pagine *checkoutFlow.xhtml* e *checkoutFlow3.xhtml*.

Ogni managed bean ha un metodo *getReturnValue* che imposta il valore del nodo di ritorno. Per il *CheckoutFlowBean*, il nodo di ritorno è la pagina *index.xhtml*, specificata usando la navigazione implicaita:



Per il *JoinFlowBean*, il nodo di ritorno è la pagina *exithome.xhtml*.

#### 16.3.3.5 To Build, Package, and Deploy the checkout-module Example Using NetBeans IDE



#### 16.3.3.6 To Build, Package, and Deploy the checkout-module Example Using Maven

#### 

#### 16.3.3.7 To Run the checkout-module Example

#### 

## 16.4 Configuring Managed Beans

Quando una pagina fa riferimento a un managed bean per la prima volta, l’implementazione JSF la inizializza basandosi o sull’annotazione *@Names* e l’annotazione di scope nella classe bean, o in base alla sua configurazione nel file di configurazione delle risorse.

Puoi usare o annotazioni o il file di configurazione delle risorse per istanziare un managed bean usato in un’applicazione JSF e per memorizzarlo in uno scope. La struttura della creazione del managed bean è configurata nel file di configurazione usando l’elemeno XML *managed-bean* per definire ogni bean. Il file viene processato all’apertura dell’applicazione.

I managed beans creati nel file di configurazione sono managed bean JSF, non managed bean CDI.

Con la struttura della creazione di managed bean, puoi

* creare beans in un file centralizzato disponibile all’intera applicazione, oltre che condizionalmente istanziare i beans durante l’applicazione
* personalizzare una proprietà del bean senza codice aggiuntivo
* personalizzare i valori di una proprietà del bean direttamente dall’interno del file di configurazione così che venga inizializzata con questi valori quando viene creata.
* Usare l’elemento *value*, impostare una proprietà di un managed bean per essere il risultato della valutazione dell’espressione di un altro valore.

### 16.4.3 Using the managed-bean Element

Un managed bean viene istanziato nel file di configurazione usando un elemento *managed-bean*, che rappresenta un’istanza di una classe bean che deve esistere nell’applicazione. Al runtime, l’implementazione JSF processa l’elemento *managed-bean*. Se una pagina fa riferimento al bean e non esiste l’istanza del bean, l’implementazione JSF istanzia un bean come specificato dalla configurazione dell’elemento.

Ecco la configurazione di un managed bean d’esempio dal Duke’s Bookstore case study:



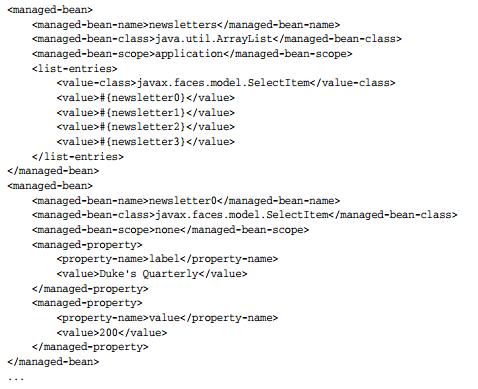
L’elemento *managed-bean-name* definisce la chiave sotto cui il bean sarà memorizzato nello scope. Per un valore del componente da mappare a questo bean, l’attributo *value* del tag del componente deve coincidere con i *managed-bean-name* fino al primo punto.

L’elemento *managed-bean-class* definisce il nome completo della classe del componente JavaBeans usato per istanziare il bean.

L’elemento *managed-bean* contiene zero o più elementi *managed-property*, ognuno corrispondente a una proprietà definita in una classe bean. Questi elementi sono usati per inizializzare i valori delle proprietà del bean. Se non vuoi inizializzare una particolare proprietà con un valore quando viene istanziato il bean, non includere la definizione *managed-property* per questa nel file di configurazione.

Se un elemento *managed-bean* non contiene altri elementi *managed-bean*, può contenere un elemento *map-entries* o un elemento *list-entries*. L’elemento *map-entries* configura un set di beans che sono istanze di *Map*. L’elemento *list-entries* configura un set di beans che sono istanze di *List*.

Nel seguente esempio, il managed bean *newsletters*, che rappresenta un componente *UISelectItems*, è configurato come un *ArrayList* che rappresenta un set di oggetti *SelectItem*. Ogni oggetto *SelectItem* è configurato uno alla volta, come un managed bean con le proprietà:

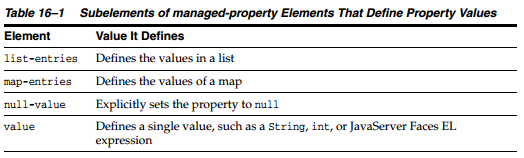


Questo approccio potrebbe essere utile per una creazione sbrigativa di una lista di oggetti prima che il team di sviluppatori abbia avuto il tempo di creare questa lista da un database. Attenzione che ognuno dei beans della newsletter ha un impostazione *managed-bean-scope* su *none*, quindi non si allocheranno in alcuno scope.

Per mappare una proprietà definita da un elemento *managed-property*, devi assicurarti che la parte dell’espressione *value* del tag di un componente dopo il punto coincida con l’elemento *managed-property* dell’elemento *property-name*.

### 16.4.2 Initializing Properties Using the managed-property Element

Un elemento *managed-property* deve contenere un elemento *property-name*, che deve coincidere con il nome della proprietà corrispondente del bean. Un elemento *managed-bean* deve inoltre contenere uno dei set di elementi che definisce il valore della proprietà. Questo valore dev’essere dello stesso tipo di quello definito per la proprietà nel bean corrispondente. Quale elemento usi per definire il valore dipende dal tipo di proprietà definito nel bean. La tabella elenca tutti gli elementi usati per inizializzare un valore.



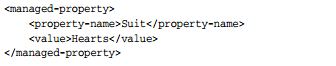
[Using the managed-bean Element](#_16.4.3_Using_the) include un esempio di inizializzazione di una proprietà *int* (tipo primitivo) usando il sotto-elemento *value*. Puoi usare il sotto-elemento *value* ancheper inizializzare *String* e altri tipi di riferimenti. Il resto della sezione descrive come usare il sotto-elemento *value* e altri sotto-elementi per inizializzare proprietà di tipi Java *Enum*, *Map*, *array*, e *Collection*, come i parametri di inizializzazione.

#### 16.4.2.1 Referencing a Java Enum Type

Una proprietà di un managed bean può anche essere di tipo *Enum*. In questo caso, l’elemento *value* dell’elemento *managed-property* deve essere una *String* che coincide con uno dei limiti *String* per l’*Enum*. In altre parole, la *String* dev’essere uno dei valori validi che possono essere restituiti se viene chiamato *valueOf* (*Class*, *String*) di *enum*, dove *Class* è una classe *Enum* e *String* è il contenuto del sotto-elemento *value*. Per esempio, supponiamo che la proprietà del managed bean sia la seguente:



Ipotizzando che si voglia configurare questa proprietà nel file di configurazione, l’elemento *managed-property* corrispondente sarebbe:



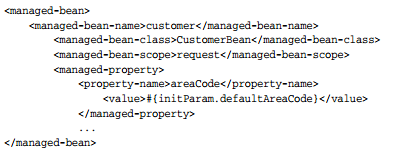
Quando il sistema si imbatte in questa proprietà, itera per ogni membro dell’*enum* e richima *toString()* su ogni membro finché non ne trova uno che combacia esattamente con il valore dall’elemento *value*.

#### 16.4.2.2 Referencing a Context Initialization Parameter

Un'altra caratteristica potente della struttura della creazione del managed bean è l’abilità di referenziare oggetti impliciti dalla proprietà di un managed bean.

Supponiamo che hai una pagina che accetta dati da un cliente, incluso l’indirizzo del cliente. Supponiamo anche che la maggior parte dei tuoi clienti viva in una particolare area. Puoi far rendere al componente dell’area il codice dell’area salvandolo in un oggetto implicito facendovi riferimento quando viene renderizzata la pagina.

Puoi salvare il codice dell’area come valore iniziale di default nell’oggetto implicito del contesto *initParam* aggiungendo un parametro di contesto alla tua applicazione web e impostando il suo valore nel deployment descriptor. Per esempio, per impostare un parametro di contesto chiamato *defaultAreaCode* a *650*, aggiungi un elemento *context-param* al deployment descriptor e dai al parametro il nome *defaultAreaCode* e valore *650*.

Poi, scrivi una dichiarazione *managed-bean* che configura una proprietà che fa riferimento al parametro:

Per accedere al codice dell’area al momento del rendering della pagina, fai riferimento alla proprietà dall’attributo *value* del tag del componente:

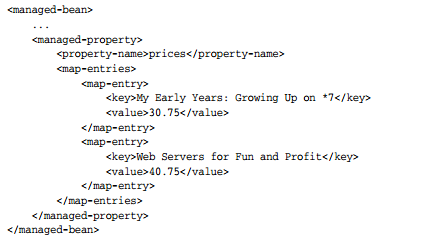


I valori sono recuperati da un altro oggetto implicito in modo analogo.

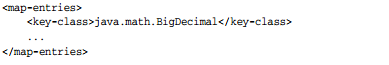
#### 16.4.2.3 Initializing Map Properties

L’elemento *map-entries* è usato per inizializzare i valori di una proprietà del bean con un tipo *Map* se l’elemento *map-entries* è isato all’interno dell’elemento *managed-property*. Un elemento *map-entries* contiene un elemento opzionale *key-class*, un elemento opzionale *value-class*, e zero o più elementi *map-entry*.

Ognuno degli elementi *map-entry* deve contenere un elemento *key* e o un elemento *null-value* o un elemento *value*. Ecco un esempio che usa l’elemento *map-entries:*

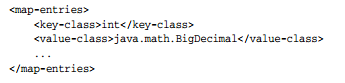


La mappatura creata da questo tag *map-entries* contiene due ingressi. Di default, tutte le chiavi e i valori sono convertiti in *String*. Se vuoi specificare un tipo diverso per la chiave nella mappatura, inserisci l’elemento *key-class* appena dentro l’elemento *map-entries*:



Questa dichiarazione convertirà tutte le chiavi in *java.math.BigDecimal*. Ovviamente, devi assicurarti che le chiavi possano essere convertite nel tipo specificato. La chiave dell’esempio in questa sezione non può essere convertita in un *BigDecimal*, poiché è una *String*.

Se vuoi specificare un tipo differente per ogni valore nella mappatura, includi l’elemento *value-class* dopo l’elemento *key-class*:



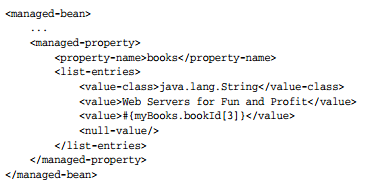
Nota che questo tag imposta solo il tipo di tutti i sotto-elementi *value*.

Ogni *map-entry* nel precedente esempio include un sotto-elemento *value*. Questo sotto-elemento *value* definisce un singlolo valore, che sarà convertito nel tipo specificato nel bean.

Anziché usare un elemento *map-entries*, è possinbile anche assegnare l’intera mappatura usando un elemento *value* che specifica un’espressione di tipo map.

#### 16.4.2.4 Initializing Array and List Properties

L’elemento *list-entries* è usato per inzializzare i valori di una proprietà array o *List*. Ogni valore individuale dell’array o della *List* è inizializzato usando un elemento *value* o *null-value*. Ecco un esempio:



Questo esempio inizializza un array o una *List*. Il tipo della proprietà corrispondente nel bean determina quale struttura di dati è stata creata. L’elemento *list-entries* definisce una lista di valori nell’array o nella *List*. L’elemento *value* specifica un singolo valore nell’array o nella *List* e provocherà la chiamata del metodo *setBook* con un argomento *null*. Una proprietà *null* non può essere specificata per una proprietà il cui tipo di dati è un primitivo Java, come un *int* o un *boolean*.

#### 16.4.2.5 Initializing Managed Bean Properties

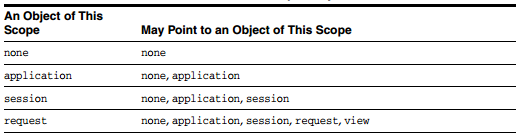
A volte potresti voler creare un bean che fa anche riferimento ad altri managed beans così da costruire un grafico o un albero di beans. Per esempio, supponiamo che te voglia creare un bean che rappresenta le informazioni di un cliente, inclusi l’indirizzo mail e l’indirizzo di casa, ognuno dei quali è anche un bean. La seguente dichiarazione *managed-bean* crea un’istanza *CustomerBean* che ha due proprietà *AddressBean*: una per l’indirizzo mail, l’altra per l’indirizzo di casa. Questa dichiarazione risulta in un albero di beans con *CustomerBean* come radice e i due oggetti bean *AddressBean* come figli.



La prima dichiarazione *CustomeBean* (con il *managed-bean-name* di *costumer*) crea un *CustomerBean* nel request scope. Questo bean ha due proprietà, *mailingAddress* e *streetAddress*. Queste proprietà usano l’elemento *value* per far riferimento al bean denominato *addressBean*.

La seconda dichiarazione managed-bean definisce un *AddressBean* ma non lo crea, poiché il suo elemento *managed-bean-scope* definisce come scope *none*. Ricorda che uno scope *none* significa che il bean è creato solo quando qualcos’altro gli fa riferimento. Poiché entrambe le proprietà *mailingAddress* e *streetAddress* fanno riferimento a *addressBean* usando l’elemento *value*, due istanze di *AddressBean* vengono create quando viene creato *CustomerBean*.

Quando crei un oggetto che punta ad un altro oggetto, non provare a puntare ad un oggetto con un ciclo di vita più corto, poiché potrebbe essere impossibile recuperare la risorsa di quello scope quando sarà sparita. Un oggetto session-scoped, per esempio, non può puntare a un oggetto request-scoped. E gli oggetti con scope *none* non hanno un ciclo di vita effettivo gestito dal framework, quindi possono puntare solo ad altri oggetti *none*-scoped. La tabella sottolinea tutte le connessioni consentite.



Assicurati di non consentire riferimenti ciclici tra gli oggetti. Per esempio, nessuno degli oggetti *AddressBean* nel precedente esempio dovrebbe puntare indietro all’oggetto *CustomerBean*, poiché *CustomerBean* punta già agli oggetti *AddressBean*.

### 16.4.3 Initializing Maps and Lists

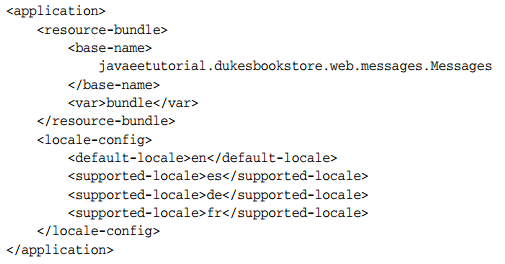
In aggiunta per la configurazione delle proprietà *Map* e *List*, puoi anche configurare una *Map* e una *List* direttamente così che puoi farvi riferimento dal tag piuttosto che far riferimento ad una proprietà che circonda una *Map* o una *List.*

## 16.5 Registering Application Messages

Gli application messages possono includere stringhe mostrate all’utente come i messaggi d’errore personalizzati (che sono mostrati dai tags *message* e *messages*) per i tuoi convertitori e validatori personalizzati. Per rendere un messaggio disponibile al momento dell’apertura dell’applicazione, fai una delle seguenti cose:

* Accoda un messaggio individuale all’istanza *javax.faces.context.FacesContext* progammaticamente.
* Registra tutti i messaggi con l’applicazione usando il file di configurazione delle risorse.

Ecco la sezione del file *faces-config.xml* che registra i messaggi per l’applicazione Duke’s Bookstore case study:



Questo set di elementi provoca all’applicazione di essere popolata da messaggi contenuti in specifici insiemi di risorse.

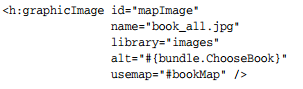
L’elemento *resource-bundle* rappresenta un set di messaggi localizzati. Deve contenere il percorso preciso dell’insieme di risorse che contiene i messaggi localizzati (in questo caso, *dukesbookstore.web.messages.Messages*). l’elemento *var* definisce il nome EL da cui gli autori della pagina si riferiscono all’insieme delle risorse.

L’elemento *locale-config* elenca la località di default e le altre località supportate. L’elemento *locale-config* agevola il sistema a trovare la corretta località basata sulle impostazioni della lingua del browser.

I tags *supported-locale* e *default-locale* accettano i codici di lettere minuscole definite dall’ISO 639-1. Assicurati che il tuo insieme di risorse contenga effettivamente i messaggi per le località specificate nei seguenti tags.

Per accedere al messaggio localizzato, lo sviluppatore dell’applicazione fa solo riferimento alla chiave del messaggio dall’insieme delle risorse.

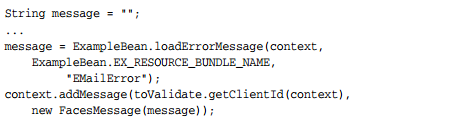
Puoi inserire testo localizzato in un tag *alt* per le immagini grafiche, come nel seguente esempio:



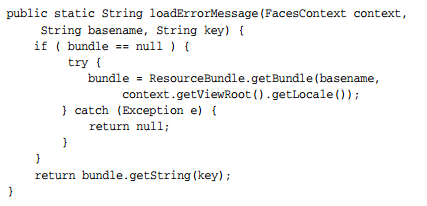
L’attributo *alt* può accettare espressioni di valore. In questo caso, l’attributo *alt* fa riferimento al testo localizzato che sarà incluso nel testo alternativo dell’immagine renderizzata da questo tag.

### 16.5.1 Using FacesMessage to Create a Message

Anziché registrare i messaggi nel file di configurazione delle risorse, puoi accedere al *java.util.ResourceBundle* direttamente dal codice del managed bean. Il frammento di codice qui sotto individua un messaggio d’errore email:



Queste righe richiamano il metodo *loadErrorMessage* del bean per ottenere il messaggio dal *ResourceBundle*. Ecco il metodo *loadErrorMessage*:



### 16.5.2 Referencing Error Messages

Una pagina JavaServer Faces usa i tags *message* o *messages* per accedere ai mesaggi d’errore.

I messaggi d’errore a cui questi tags accedono includono:

* I messaggi standard d’errore che accompagnano i convertitori e i validatori standard che sono inviati con le API.
* I messaggi d’errore personalizzati contenuti in insiemi di risorse registrati con l’applicazione dall’architetto dell’applicazione usando l’elemento *resource-bundle* nel file di configurazione.

Quando un convertitore o un validatore viene registrato su un componente di input, il messaggio d’errore appropriato viene automaticamente accodato al componente.

L’autore di una pagina può fare l’override dei messaggi d’errore accodati su un componente usando i seguenti attributi del tag del componente:

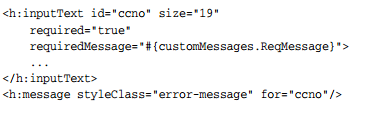
* *converterMessage*: fa riferimento al messaggio d’errore da mostrare quando i dati sul componente racchiuso non possono essere convertiti dal convertitore registrato su questo componente.
* *requiredMessage*: fa riferimento al messaggio d’errore da mostrare quando non è stato inserito alcun valore nel componente racchiuso.
* *validatorMessage*: fa riferimento al messaggio d’errore da mostrare quando i dati sul componente racchiuso non possono essere convalidati dal validatore registrato sul componente.

Tutti e tre gli attributi sono abilitati a ricevere valori letterali e espressioni di valore. Se un attributo usa un’espressione di valore, questa espressione fa riferimento al messaggio d’errore in un insieme di risorse. Questo insieme di risorse deve essere reso disponibile all’applicazione in uno dei seguenti modi:

* Dall’architetto dell’applicazione usando l’elemento *resource-bundle* nel file di configurazione
* Dall’autore della pagina usando il tag *f:loadBundle*

Al contrario, l’elemento *resource-bundle* deve essere usato per rendere disponibile all’applicazione quegli insiemi di risorse contenenti i messaggi d’errore personalizzati che vengono accodati al componente come risultato di un convertitore o validatore personalizzato registrato sul componente.

I seguenti tags mostrano come specificare l’attributo *requiredMessage* usando un’espressione di valore per fare riferimento al messaggio d’errore:

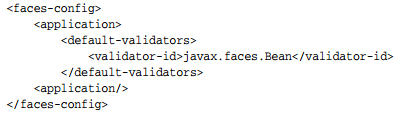


L’espressione di valore usata da *requiredMessage* in questo esempio fa riferimento al messaggio d’errore con la chiave *ReqMessage* nell’insieme di risorse *customMessages*.

Questo messaggio rimpiazza il messaggio corrispondente accodato al componente e sarà mostrato dovunque il tag *message* o *messages* è allocato sulla pagina.

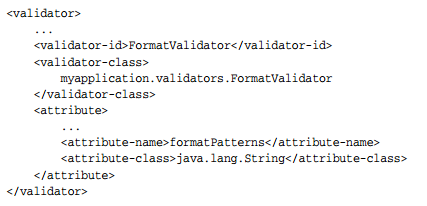
## 16.6 Using Default Validators

In aggiunta ai validatori che dichiari sui componenti, puoi inoltre specificare zero o più validatori di default nel file di configurazione delle risorse dell’applicazione. Il validatore di default applica a tutte le istanze *javax.faces.component.UIInput* in una view o albero del componente ed è apposto dopo il validatore locale definito. Ecco un esempio di un validatore di default registrato in un file di configurazione:



## 16.7 Registering a Custom Validator

Se lo sviluppatore dell’applicazione fornisce un’implementazione dell’interfaccia *javax.faces.validator.Validator* per eseguire la validazione, devi registrare questo validatore personalizzato o usando l’annotazione *@FacesValidator*, o usando l’elemento XML *validator* nel file di configurazione:



Gli attributi specificati nel tag *validator* fa override di qualsiasi impostazione nell’annotazione *@FacesValidator.*

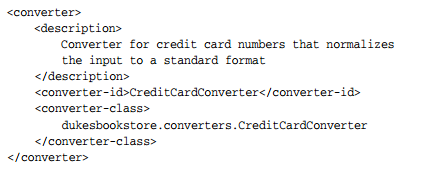
Gli elementi *validator-id* e *validator-class* sono sotto-elementi richiesti. L’elemento *validator-id* rappresenta l’identificatore sotto cui la classe *Validator* dovrebbe essere registrato. Questo ID è usato dalla classe del tag corrispondente al tag del *validator* personalizzato.

L’elemento *validator-class* rappresenta il nome completo della classe *Validator*.

L’elemento *attribute* identifica un attributo associato con l’implementazione del *Validator*. Questo richiede i sotto-elementi *attribute-name* e *attribute-class*. L’elemento *attribute-name* fa riferimento al nome dell’attributo come appare nel tag *validator*. L’elemento *attribute-class* identifica il tipo Java del valore associato all’attributo.

## 16.8 Registering a Custom Converter

Come nel caso del validatore personalizzato, se lo sviluppatore dell’applicazione crea un convertitore personalizzato, devi registrarlo con l’applicazione o usando l’annotazione *@FacesConverter,* o usando l’elemento XML *converter* nel file di configurazione. Ecco un ipotetica configurazione di *converter* per *CreditCardConverter* dal Duke’s Bookstore case study:



Gli attributi specificati nel tag *converter* fanno override di qualsiasi impostazione nell’annotazione *@FacesConverter*.

L’elemento *converter* rappresenta un’implementazione *javax.faces.convert.Converter* e contiene gli elementi richiesti *converter-id* e *converter-class*.

L’elemento *converter-id* identifica un ID usato dall’attributo *converter* del tag di un componente UI da applicare il convertitore ai dati del componente.

L’elemento *converter-class* identifica l’implementazione di *Converter.*

## 16.9 Configuring Navigation Rules

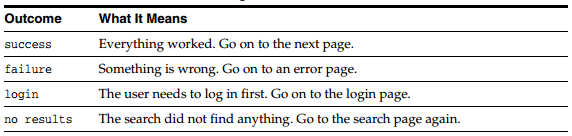
La navigazione tra pagine differenti di un’applicazione JSF, come richiedere che sia mostrata la pagina successiva dopo che un componente bottone o link sia stato cliccato, è definito da un set di regole. Le regole di navigazione possono essere implicite, o possono essere definite esplicitamente nel file di configurazione delle risorse dell’applicazione.

Ogni regola di navigazione specifica come navigare da una pagina ad un’altra pagina o ad un set di pagine. L’implementazione JSF sceglie la regola di navigazione appropriata a seconda di quale pagina è mostrata al momento.

Dopo che viene selezionata la regola di navigazione appropriata, la scelta della pagina da accedere successivamente da quella corrente dipende da due fattori:

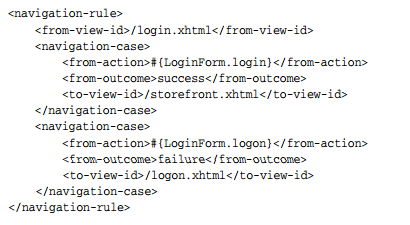
* Il metodo action invocato quando il componente viene cliccato
* Il risultato logico referenziato dal tag del componente o restituito dal metodo action

Il risultato può essere qualsiasi cosa scelta dallo sviluppatore, ma la tabella elenca alcuni risultati comunemente usati nelle applicazioni web



Solitamente, il metodo action esegue alcune elaborazioni sui dati form della pagina corrente. Per esempio, il metodo potrebbe controllare se il nome e la password inseriti dall’utente nel form corrispondono al nome e password sul file. Se corrispondono, il metodo restituisce il risultato *success*. Altrimenti, restituisce il risultato *failure*. Come questo esempio dimostra, entrambi i metodi usati per processare l’action e il rislutato restituito sono necessari per determinare la pagina corretta a cui accedere.

Ecco la regola di navigazione che potrebbe essere usata con l’esempio appena descritto:



Questa regola di navigazione definisce i modi possibili di navigare da *login.xhtml*. Ogni elemento *navigation-case* definisce un percorso di navigazione possibile da *login.xhtml*. il primo *navigation-case* dice che se *LoginForm.login* restituisce come risultato *success*, allora avverrà l’accesso a *storefront.xhtml*. la seconda *navigation-rule* dice che *login.xhtml* sarà ri-renderizzata nel caso *LoginForm.login* restituisca *failure*.

La configurazione del flusso di una pagina dell’applicazione consiste in un set di regole di navigazione. Ogni regola è definita nell’elemento *navigation-rule* nel file *faces-config.xml*

Ogni elemento *navigation-rule* corrisponde a un identificatore dell’albero del componente definito dall’elemento opzionale *from-view-id*. ciò significa che ogni regola definisce tutti i modi possibili di navigare da una particolare pagina dell’applicazione. Se non ci sono elementi *from-view-id*, le regole di navigazione definite nell’elemento *navigation-rule* si applicano a tutte le pagine dell’applicazione. L’elemento *from-view-id* riconosce anche i percorsi di corrispondenza jolly. Per esempio, questo elemento *from-view-id* dice che la regola di navigazione va applicata a tutte le pagine nella directory *books*:



Un elemento *navigation-rule* può contenere zero o più elementi *navigation-case*. L’elemento *navigation-case* definisce un set di criteri di corrispondenza. Quando questi criteri vengono soddisfatti, l’applicazione navigherà alla pagina definita dall’elemento *to-view-id* contenuto nello stesso elemento *navigation-case*.

I criteri di navigazione sono definiti negli elementi opzionali *from-outcome* e *from-action*. L’elemento *from-outcome* definisce un risultato logico, come *success*. L’elemento *from-action* usa un’espressione di metodo per far riferimento al metodo action che restituisce una *String*, che è il risultato logico. Il metodo esegue alcuni ragionamenti per determinare il risultato e restituisce il risultato.

Gli elementi *navigation-case* sono controllati a confronto con il risltato e l’espressione di metodo nel seguente ordine.

1. Casi che specificano entrambi i valori *from-outcome* e *from-action*. Entrambi gli elementi possono essere usati se il metodo action restituisce risultati differenti in base al risultato del processo che esegue.
2. Casi che specificano solo un valore *from-outcome*. L’elemento *from-outcome* deve corrispondere o al risultato definito dall’attributo *action* del componente *javax.faces.component.UICommand* o al risultato restituito dal metodo referenziato dal componente *UICommand*.
3. Casi che specificano solo un valore *from-action*. Questo valore deve corrispondere all’espressione *action* specificata dal tag del componente.

Quando uno di questi casi corrisponde, l’albero del componente definito dall’elemento *to-view-id* sarà selezionato per essere renderizzato.

## 16.10 Registering a Custom Renderer with a Render Kit

Quando lo sviluppatore di un’applicazione crea un renderer personalizzato, devi registrarlo usando un render kit appropriato. Poiché l’applicazione image map implementa una image map HTML, le classi *AreaRenderer* e *MapRenderer* nel Duke’s Bookstore case study dovrebbe eseere registrate usando un render kit HTML.

Si registra un renderer o usando l’annotazione *@FacesRenderer*, o usando l’elemento *render-kit* del file di configurazione. Ecco un ipotetica configurazione di *AreaRenderer*:



Gli attributi specificati in un tag *renderer* fa override di qualsiasi impostazione nell’annotazione *@FacesRenderer.*

L’elemento *render-kit* rappresenta un’implementazione javax.faces.render.*RenderKit.* Se non è specificato un *render-kit-id*, è preso di default il render kit HTML. L’elemento *renderer* rappresenta un’implementazione *javax.faces.render.Renderer.* Annidando l’elemento *renderer* all’interno dell’elemento *render-kit*, stai registrando il renderer con l’implementazione *RenderKit* associata con l’elemento *render-kit*.

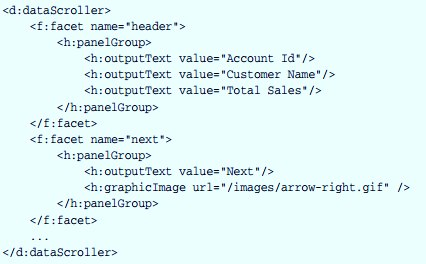
La *renderer-class* è il nome completo della classe di *Renderer*.

Gli elementi *component-family* e *renderer-type* sono usati da un componente per cercare i renderers in grado di renderizzarlo. L’identificatore *component-family* deve corrispondere a ciò che viene restituito dal metodo *getFamily* della classe del componente. La famiglia del componente rappresenta un componente o un set di componenti che un particolare renderer è in grado di renderizzare. Il *renderer-type* deve corrispondere a ciò che viene restituito dal metodo *getRendererType* della classe che gestisce il tag.

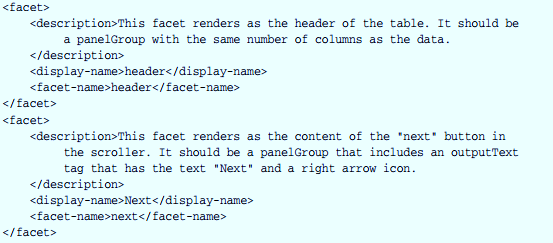
Usando la famiglia del componente e un tipo di renderer per cercare i renderers per il componente, l’implementazione JSF permette a un componente di essere rederizzato da renderers multipli e permette a un renderer di renderizzare componenti multipli.

Ognuno dei tags *attribute* specifica un attributo dipendente dal render e dal suo tipo. L’elemento *attribute* non interviene nell’esecuzione al runtime della tua applicazione. Piuttosto, fornisce informazioni per lavorare riguardo agli attributi che il *Renderer* supporta.

L’oggetto responsabile per il rendering di un componente (che può essere il componente stesso o un renderer che il componente delega per il rendering) può usare le facets come sostegno nel processo di rendering. Queste facets permettono allo sviluppatore del component personalizzato di controllare alcuni aspetti del rendering del componente. Considera questo esempio tag di un componente personalizzato:



Il tag del componente *dataScroller* include un componente che renderizzerà l’header e un componente che renderizzerà il bottone **Next**. Se il renderer associato a questo componente renderizza le facets, puoi includere i seguenti elementi *facet* nell’elemento *renderer*:



Se un componente che supporta facets fornisce il proprio rendering e vuoi includere elementi *facets* nel file di configurazione, hai bisogno di inserirli nella configurazione del componente piuttosto che nella configurazione del renderer.

## 16.11 Registering a Custom Component

In aggiunta alla registrazione di renderers personalizzati, puoi anche registrare i componenti personalizzati che solitamente sono associati con i renderers personalizzati. Puoi usare o l’annotazione *@FacesComponent*, o l’elemento *component* del file di configurazione.

Ecco un ipotetico elemento *component* per il file di configurazione che registra *AreaComponent*:



Gli attributi specificati nel tag *component* fanno override di qualsiasi impostazione nell’annotazione *@FacesComponent*.

L’elemento *component-type* indica il nome sotto cui il componente dovrebbe essere registrato. Gli altri oggetti che fanno riferimento a questo componente usano questo nome. Per esempio, l’elemento *component-type* nella configurazione per *AreaComponent* definisce un valore di *DemoArea*, che corrisponde al valore restituito dal metodo *getComponentType* della classe *AreaTag*.

L’elemento *component-class* indica il nome completo della classe del componente. Gli elementi *property* specificano le proprietà dei componenti i loro tipi.

Se il componente personalizzato può includere facets, puoi configurare le facets nella configurazione del componente usando gli elementi *facets*, che sono permessi dopo gli elementi *component-class*.

## 16.12 Basic Requirements of a JavaServer Faces Application

In aggiunta alla configurazione della tua applicazione, devi soddisfare altri requisiti delle applicazioni JSF, incluso un impacchettamento appropriato di tutti i files necessari e la fornitura di deployment descriptor. Questa sezione descrive come eseguire questi compiti amministrativi.

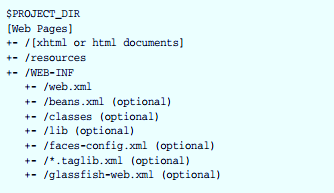
Le applicazioni JSF possono essere impacchettate in file WAR, che sono i più conformi ai requisiti specifici per l’esecuzione attraverso diversi containers. Come minimo, un file WAR per un’applicazione JSF dovrebbe contenere:

* Un deployment descriptor, chiamato *web.xml*, per configurare le risorse richieste dall’applicazione (richiesto)
* Un set specifico di files JAR contenenti le classi essenziali (opzionale)
* Un set di classi, pagine JSF, e altre risorse richieste, come files di immagini.

Un file WAR dovrebbe inoltre contenere:

* Un file di configurazione delle risorse, che configura le risorse dell’applicazione
* Un set di files tag library descriptor

Per esempio, un file WAR di un’applicazione web JSF che usa Facelets generalmente ha la seguente struttura di directory:



Il file *web.xml* (o web deployment descriptor), il set di files JAR, e il set di files dell’applicazione devono essere contenuti della directory *WEB-INF* del file WAR.

### 16.12.1 Configuring an Application with a Web Deployment Descriptor

Le applicazioni web sono comunemente configurate usando gli elementi contenuti nel deployment descriptor dell’applicazione web, *web.xml*. Il deployment descriptor per un’applicazione JSF deve specificare alcune configurazioni, comprese:

* La servlet usata per processare le richieste JSF
* La mappatura servlet per la servlet di elaborazione
* Il percorso al file di configurazione delle risorse, se esiste e non è posto nella locazione di default

Il deployment descrptor può anche includere altre configurazioni opzionali, come

* Specificare dove è stato salvato lo stato del componente
* Criptare lo stato salvato sul client
* Comprimere lo stato salvato sul client
* Restringere l’accesso alle pagine che contengono i tags JSF
* Attivare la validazione XML
* Specificare il Project Stage
* Verificare gli oggetti personalizzati

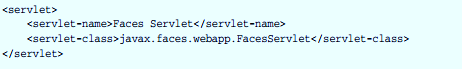
Questa sezione spiega più nei dettagli queste configuraioni. Dove appropriato, descrive inoltre come puoi fare queste configurazioni usando NetBeans IDE.

#### 16.12.1.1 Identifying the Servlet for Lifecycle Processing

Un requisito di un’applicazione JSF è che tutte le richieste all’applicazione che fanno riferimento ai componenti JSF salvati in precedenza devono passare attraverso *javax.faces.webapp.FacesServlet*. un’istanza *FacesServlet* gestisce il ciclo di vita che processa le richieste per l’applicazione web e inizializza le risorse richieste dalla tecnologia JSF.

Prima che un’applicazione JSF possa lanciare la prima pagina web, il web container deve invocare l’istanza *FacesServlet* in ordine per far partire l’elaborazione del ciclo di vita dell’applicazione.

Il seguente esempio mostra la configurazione di default di *FacesServlet*:



Dovrai fornire un ingresso alla configurazione della mappatura per assicurarti che l’istanza *FacesServlet* sia invocata. La mappatura a *FacesServlet* può essere una mappatura di prefissi, come */faces/\**, o una mappatura di estensioni, come *\*.xhtml*. La mappatura è usata per identificare una pagina come un contenuto JSF. Per questo, l’URL della prima pagina dell’applicazione deve includere la mappatura del percorso URL.

I seguenti elementi specificano una mappatura di prefissi:



I seguenti elementi, usati negli esempi del tutorial, specificano una mappatura di estensioni:



Quando usi questo meccanismo, gli utenti accedono all’applicazione come mostrato nel seguente esempio:



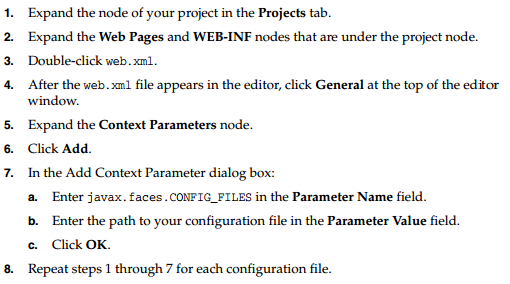
Nel caso della mappatura di estensioni, se una richiesta viene dal server per la pagina con un’estensione *.xhtml*, il container invierà la richiesta all’instanza *FacesServlet*, che si aspetterà l’esistenza di una pagina corrispondente con lo stesso nome contenente il contenuto.

Se stai usando NetBeans IDE per creare la tua applicazione, un web deployment descriptor viene creato automaticamente per te con le configurazioni di default. Se hai creato la tua applicazione senza un IDE, puoi creare un deployment descriptor.

#### 16.12.1.2 To Specify a Path to an Application Configuration Resource File

Come spiegato in [Application Configuration Resource File](#_16.2_Application_Configuration), un’applicazione può avere files di configurazione delle risorse multipli. Se questi files non sono posizionati nelle directories che l’implementazione ricerca di default o i files non sono denominati *faces-config.xml*, hai bisogno di specificare i percorsi a questi files.

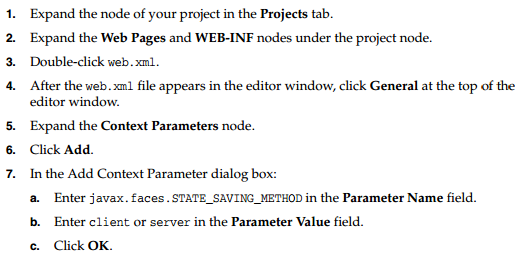
Per specificare questi percorsi usando NetBeans



#### 16.12.1.3 To Specify Where State is Saved

Per tutti i componenti in un’applicazione web, puoi specificare nel deployment descriptor dove vuoi che lo stato venga salvato, se sul client o sul server. Lo fai impostando un parametro di contesto nel tuo deployment descriptor. Di default, lo stato viene salvato sul server, quindi devi specificare questo parametro di contesto sono se vuoi che venga salvato sul client.

Per specificare dove deve essere salvato lo stato usando NetBeans IDE



Se lo stato viene salvato sul client, lo stato dell’intera vista viene renderizzato a un campo nascosto sulla pagina. L’implementazione JSF salva lo stato sul server di default. Duke’s Forest salva il proprio stato sul client.

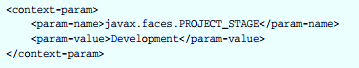
### 16.12.2 Configuring Project Stage

Project Stage è un parametro di contesto che identifica lo stato di un’applicazione JSF nel ciclo di vita di un software. Lo stage di un’applicazione interviene sul comportamento dell’applicazione. Per esempio, i messaggi d’errore possono essere mostrati durante il Development stage ma sono soppressi durante il Production stage.

I possibili valori di Progect Stage sono:

* *Development*
* *UnitTest*
* *SystemTest*
* *Production*

Il Project Stage viene configurato attraverso un parametro di contesto nel deployment descriptor. Ecco un esempio:



Se non è definito un Project Stage, lo stage di default è *Production*. Puoi inoltre personalizzare gli stages in base alle tue necessità.

### 16.12.3 Including the Classes, Pages, and Other Resources

Quando si impacchettano le applicazione web usando i build scripts inclusi, noterai che gli scripts impacchettano le risorse nei seguenti modi:

* Tutte le pagine web sono allocate al livello più alto del file WAR.
* Il file *faces-config.xml* e il file *web.xml* sono impacchettati nella directory *WEB-INF.*
* Tutti i packages sono memorizzati nella directory *WEB-INF/classes/*
* Tutti i file delle risorse sono o sotto la root della directory */resources* dell’applicazione o nel percorso delle classi, la directory *META-INF/resources/****resourceIdentifier***.

Quando impacchetti la tua applicazione, puoi usare NetBEans IDE o puoi usare i files XML come quelli creati da Maven. Puoi modificare i files XML per adeguarli alla situazione. Tuttavia, puoi continuare a impacchettare i files WAR usando la struttura delle directory descritta in questa sezione, poiché questa tecnica si attiene alla pratica comune accettata per l’impacchettamento delle applicazioni web.

## 