La tecnologia JavaServer Faces è un component framework lato-server per costruire applicazioni web basate su Java.

JavaServer Faces consiste di:

* Un’API usata per rappresentare i componenti e gestire il loro stato, gestire eventi, validazione lato-server e conversione dati, definizione della navigazione delle pagine, supporto all’internazionalizzazione e all’accessibilità e per offrire estensibilità a tutte queste funzioni.
* Librerie di tag per aggiungere componenti alle pagine web e connetterli a oggetti lato-server.

La tecnologia JavaServer Faces offre un modello di programmazione definito e molte librerie di tag.

Le librerie di tag contengono tag handlers che implementano le tag dei componenti.

Queste funzionalità semplificano significativamente il lavoro per costruire e manutenere le applicazioni web con interfacce utente lato-server (UIs).

Rendono possibile completare i seguenti compiti con poco sforzo:

* Creare una pagina web
* Trascinare componenti su una pagina web aggiungendo component tags
* Unire i componenti di una pagina ai dati lato-server
* Legare eventi generati dai componenti con il codice dell’applicazione lato-server
* Salvare e ripristinare lo stato dell’applicazione oltre la vita delle richieste al server
* Riutilizzare e estendere i componenti mediante la customizzazione

Questo capitolo offre una panoramica sulla tecnologia JavaServer Faces.

## 7.1 Cos’è un’appicazione JavaServer Faces?

Le funzionalità offerte da un’applicazione JavaServer Faces sono assimilabili a quelle di ogno altra applicazione web di Java.

Una tipica applicazione JavaServer Faces include le seguenti parti:

* Un set di pagine web in cui i componenti sono impostate.
* Un set di tags per aggiungere componenti alla pagina web.
* Un set di managed beans (che sono POJO gestiti dal container). In un’applicazione JavaServer Faces i managed beans funzionano come beans di sostegno che definiscono proprietà e funzioni per i componenti UI di una pagina.
* Un web deployment descriptor (web.xml)
* Opzionalmente uno o più **application configuration resource files**, ad esempio il file *faces-config.xml*, che possono essere utilizzati per definire le regole di navigazione all’interno della pagina e per configurare beans e altri oggetti e componenti personalizzati.
* Opzionalmente un set di oggetti personalizzati che possono includere custom components, validators, converters o listeners, creati dallo sviluppatore dell’applicazione
* Opzionalmente un set di tag personalizzate per rappresentare gli oggetti personalizzati sulla pagina.

La figura qua sotto mostra l’interazione tra il client e il server in una tipica applicazione JavaServer Faces



In risposta a una richiesta del client la pagina web viene renderizzata dal web container che implementa JavaServer Faces.

La pagina web *myfacelet.xhtml* è costruita usando le tag dei componenti JavaServer Faces.

I component tags sono utilizzati per aggiungere componenti alla view (rappresentata da *myview* nell’immagine), che è la rappresentazione lato-server della pagina. In aggiunta ai componenti la pagina web può avere riferimenti a oggetti quali:

* Ogni event listener, validatore e convertitore che sia registrato sui componenti
* I componenti JavaBeans che catturano i dati e processano le funzionalità specifiche dell’applicazione dei componenti

Su richiesta del client la view è renderizzata come una risposta. Il rendering è il processo con cui, basandosi sulla view lato-server, il web container genera output che possono essere letti dal client tramite il browser (come HTML o XHTML).

## 7.2 I benefici della tecnologia JavaServer Faces

Uno dei più grandi vantaggi della tecnologia JavaServer Faces è che offre una divisione pulita tra il comportamento e la rappresentazione delle applicazioni web.

Un’applicazione JavaServer Faces può mappare le richieste HTTP proveniente dalla gestione di eventi specifici dei componenti e gestire componenti come oggetti con stato (ad esempio i carrelli della spesa) sul server.

La tecnologia JavaServer Faces permette di costruire applicazioni web che implementano una fine separazione tra comportamento e rappresentazione che tradizionalmente è offerta dalle architetture UI lato-client.

La separazione tra la logica e la rappresentazione permette anche a ogni membro del team di sviluppo di un’applicazione web di focalizzarsi su un singolo pezzo del processo di sviluppo e offre un modello di programmazione semplice per unire i pezzi.

Un altro importante obiettivo di questa tecnologia è di influenzare familiar component e concetti di web-tier senza limitarsi a una particolare tecnologia di scripting o linguaggio di mark-up.

La tecnologia JavaServer Faces è a livello più alto della Servlet API, come mostrato in figura:



Questo livellaggio delle API mette a disposizione alcuni importanti casi di utilizzo dell’applicazione, come utilizzare diverse tecnologie di rappresentazione, creare i propri componenti personalizzati direttamente dalle classi dei componenti e generare output per diversi dispositivi client.

La tecnologia Facelets, disponibile come parte della tecnologia JavaServer Faces è la migliore tecnologia di rappresentazione per costruire applicazioni web basate su JavaServer Faces.

La tecnologia Facelets offre diversi vantaggi:

* Il codice può essere riutilizzato e esteso per i componenti tramite il templating e caratteristiche composite dei componenti.
* Si possono usare annotazioni per registrare automaticamente i managed beans come risorse disponibili per le applicazioni JavaServer Faces. Inoltre le regole di navigazione implicite permettono agli sviluppatori di configurare velocemente la navigazione delle pagine. Tali caratteristiche riducono il processo di configurazione manuale delle applicazioni.
* La tecnologia JavaServer Faces offri una ricca architettura per gestire lo stato dei componenti, processarne i dati, validare gli imput dell’utente e gestire eventi.

## 7.3 Una semplice applicazione JavaServer Faces

JavaServer Faces offre un semplice processo per la creazione di applicazioni web. Per sviluppare una semplice applicazione JavaServer Faces bisogna semplicemente eseguire i seguenti compiti (già descritti nell’esempio hello1):

* Creare pagine web utilizzando i tag dei componenti
* Sviluppare managed beans
* Mappare le istanze FacesServlet

L’esempio hello1 include un managed bean e due pagine web Facelets, quando un client accede alla prima pagina questa chiede all’utente il nome e la seconda pagina risponde offrendo il greeting.

Ogni applicazione web ha un ciclo di vita. I compiti comuni quali gestire le richieste, decodificare i parametri, modificare e salvare lo stato e renderizzare le pafine web al browser sono tutti compiuti durante il ciclo vitale dell’applicazione. Alcuni framework nasconondono i dettagli del ciclo vitale in modo che si possa modificare o fare azioni diverse se i requisiti dell’applicazione lo permettono.

Il ciclo vitale di un’applicazione JavaServer Faces inizia e finisce con le seguenti attività: il client fa la richiesta per la pagina web e il server risponde con la pagina.

Il ciclo vitale consiste di queste due fasi: esecuzione (Execute) e renderizzazione (Render).

Drante la fase di esecuzione possono esserci diverse azioni:

* La view dell’applicazione è costruita o ripristinata.
* I parametri della richiesta vengono applicati.
* Viene effettuata la conversione e la validazione per i valori dei componenti.
* I managed beans vengono aggiornati con i valori dei componenti.
* Viene invocata la logica dell’applicazione.

Nella prima richiesta (initial) viene costruita solo la view. Per le richieste successive (postback) alcune o tutte le altre azioni possono essere eseguite.

Nella fase di Render la view richiesta viene renderizzata come risposta al client.

La seguente descrizione mostra quali azioni accadono nel ciclo di vita dell’applicazione hello1:

1. Quando l’applicazione è costruita (**built**) e importata sul Server GlassFish l’applicazione è in uno stato di non inizializzazione.
2. Quando un client fa una richiesta **initial** per la pagina *index.xhtml,* viene compilata l’applicazione Facelets *hello1*.
3. L’applicazione compilata viene eseguita e un nuovo **component tree** viene costruito per l’applicazione *hello1* e posto in un ***FacesContext***.
4. Il component tree viene riempito con le proprietà dei component e managed beans a lui associati, rappresentati dall’espressione EL *hello.name*.
5. Viene costruita una nuova view, basata sul component tree.
6. La view viene renderizzata al client richiedente come risposta.
7. Il component tree viene distrutto automaticamente.
8. Nelle richieste seguenti (**postback**) il component tree viene ricostruito e il saved state viene applicato.

## 7.4 User Interface Component Model

In aggiunta alla descrizione del ciclo vitale una panoramica dell’architettura JavaServer Faces può rendere più facile capire questa tecnologia.

I componenti JavaServer Faces sono i blocchi di costruzione di una view JavaServer Faces.

Un componente può essere un componente con interfaccia utente (UI) o senza.

I componenti JavaServer Faces UI sono elementi configurabili e riutilizzabili che compongono l’interfaccia utente di un’applicazione JavaServer Faces. Un componente può essere semplice, come un bottone, o composto, come una tabella con molti componenti.

JavaServer Faces offre un’architettura ricca e flessibile che include:

* Un set di classi *javax.faces.component.UIComponent* per specificare lo stato e il comportamento dei componenti UI
* Un modello di rendering che definisce come fare il render in vari modi
* Un modello di conversione che definisce come registrare i dati convertiti sui componenti
* Un modello di eventi e listener che definisce come gestire gli eventi dei componenti
* Un modello di validazione che definisce come registrare i validatori su un componente

Questo capitolo descrive ognuno di questi pezzi dell’architettura dei componenti.

## 7.4.1 User Interfaces Component Classes

JavaServer Faces offre un set di classi di componenti UI e le interfacce di comportamento associate che specificano tutte le funzionalità dei componenti UI, come conservare lo stato dei componenti, manutenzione dei riferimenti agli oggetti, gestione guidata degli eventi e del rendering per un set di componenti standard.

Le classi component sono completamente estensibili e permettono a chi scrive i componenti di creare i propri componenti personalizzati.

La classe astratta di base per tutti i componenti UI è *javax.faces.component.UIComponent.*

Le classi dei componenti JavaServer Faces UI estende la classe *UIComponentBase* (una sottoclasse di *UIComponent*) che definisce lo stato di default e il comportamento di un componente.

Il seguente set di classi componenti è incluso nella tecnologia JavaServer Faces:

* **UIColumn:** Rappresenta una singola colonna di dati in un componente **UIData**.
* **UICommand:** Rappresenta un controllo che compie qualcosa quando attivato.
* **UIData:** Rappresenta dei dati uniti a una collezione di dati rappresentati da un’istanza *javax.faces.model.DataModel*
* **UIForm:** Rappresenta un form di input da presentare agli utenti I componenti figli rappresentano (tra le altre cose) i campi input da includere quando il form è sottoscritto. Questo componente è l’analogo del tag *formin HTML.*
* **UIGraphic:** Mostra un’immagine.
* **UIInput:**Prende i dati inseriti dall’utente. E’ una sottoclasse di *UIOutput*.
* **UIMessage:** Mostra un messaggio di errore.
* **UIMessages:**Mostra un set di messaggidi errore.
* **UIOutcomeTarget:**Mostra un link come link o bottone.
* **UIOutput:**Mostra dati di output su una pagina.
* **UIPanel:**Gestisce il layout dei suoi componenti figli.
* **UIParameter:**Rappresenta dei parametri.
* **UISelectBoolean:**Permette all’utente di settare un valore booleano su un controllo selezionandolo o deselezionandolo. E’ una sottoclasse della classe *UIInput*.
* **UISelectItem:**Rappresenta un singolo oggetto in un set di oggetti.
* **UISelectItems:**Rappresenta un set di oggetti.
* **UISelectMany:**Permette a un utente di selezionare oggetti multipli da un gruppo di oggetti. E’ una sottoclassa di *UInput*.
* **UISelectOne:**Permette a un utentedi selezionare un oggetto da un gruppo di oggetti. E’ una sottoclasse di *UIInput*.
* **UIViewParameter:**Rappresenta i parametri query di una richiesta. E’ una sottoclasse di *UIInput*.
* **UIViewRoot:**Rappresenta la root del component tree.

Oltre a estendere *UIComponentBase* le classi componenti implementano una o più interfacce di comportamento, ognuna delle quali definisce un certo comportamento per un set di componenti le cui classi implementano quest’interfaccia.

Queste interfacce di comportamento, definite nel package *javax.faces.component,* sono le seguenti:

* **ActionSource:** Indica che un componente può fare un’azione in un evento. Questa interfaccia è usata per versioni di JavaServer Faces precedenti alla 1.1\_01. E’ stata deprecata in JavaServer Faces 2.
* **ActionSource2:** Estende *ActionSource* e offre le stesse funzionalità. Inoltre permette ai componenti di utilizzare l’Expression Language (EL) quando si riferiscono a metodi che gestiscono azioni su eventi.
* **EditableValueHolder:** Estende *ValueHolder* e specifica funzionalità aggiuntive per componenti modificabili (editable) come la validazione e l’emissione di eventi che modificano valori (value-change events).
* **NamingContainer:** Specifica che ogni componente incluso in questo componente ha un unico ID.
* **StateHolder:** Denota che un componente ha un stato che deve essere salvato tra una richiesta e l’altra.
* **ValueHolder:** Indica che un componente mantiene un valore locale così come l’opzione per accedere ai dati nel model tier.
* **javax.faces.event.SystemEventListenerHolder:** mantiene una lista di istanze *javax.faces.event.SystemEventListener* per ogni tipo di *javax.faces.event.SystemEvent* definito in quella classe.
* **javax.faces.component.behavior.ClientBehaviorHolder:** Aggiunge la capacità di attaccare istanze *javax.faces.component.behavior.ClientBehavior* come uno script riutilizzabile.

***UiCommand*** implementa *ActionSource2* e *StateHolder*.

***UiOutput*** e le classi di componenti che lo estendono implementano *StateHolder* e *ValueHolder*.

***UIInput*** e le classi di componenti che lo estendono implementano *EditableValueHolder, StateHoldere* ValueHolder*.*

**UIComponentBase** implementa *StateHolder.*

Solo i componenti **writers** hanno bisogno di utilizzare le classi dei componenti e le interfacce direttamente. Gli autori delle pagine e gli sviluppatori dell’applicazione possono usare un componente standard tramite una tag che lo rappresenta sulla pagina. Molti dei componenti possono essere renderizzati in modi diversi sulla pagina. Ad esempio un componente *UICommand* può essere renderizzato come bottone o link.

Il prossimo capitolo spiega come funziona il modello di rendering e come l’autore della pagina può decidere di renderizzare i componenti tramite le tag appropriate.

## 7.4.2 Component Rendering Model

L’architettura dei componenti JavaServer Faces è disegnata perché le funzionalità dei componenti siano definite dalle classi dei componenti mentre la loro visualizzazione possano essere definite da classi di rendering separate. Questo design ha molti benefici inclusi i seguenti:

* I **component writers** possono definire il comportamento di un componente una volta e creare molte visualizzazioni, ognuna delle quali definisce un modo diverso di visualizzare il componente nello stesso client o in client diversi.
* Gli autori delle pagine e gli sviluppatori dell’applicazione possono cambiare l’aspetto di un componente su una pagina selezionando il tag che rappresenta la combinazione appropriata di componenti da visualizzare.

Un **render kit** definisce come vengono associate le classi dei componenti con i relativi tag approprati per un particolare client. L’implementazione di JavaServer Faces include un render kit standard HTML per la visualizzazione su un client HTML.ù

Il render kit definisce un set di classi *javax.faces.render.Renderer* per ogni componente che lo supporta.

Ogni classe *Rendered* definisce un modo diverso di visualizzare un componente specifico nell’output definito dal render kit.

Ad esempio un componente *UISelectOne* ha tre differenti visualizzazioni possibili (gruppo di opzioni, combo box o list box) o un componente *UICommand* può essere visualizzato come link o come bottone tramite le *tag h:commandLink* o *h:commandButton*.

La parte *command* di ogni tag corrisponde alla classe *UICommand* più la parte che specifica la funzionalità di fare qualcosa, la parte *Button* o *Link* corresponde a due classi di visualizzazioni diverse che definiscono come tale componente sarà visualizzato sulla pagina.

Ogni tag personalizzata definiti nello stesso render kit standard HTML è composto dalle funzionalità (definite dalla classe *UICommand*) e dagli attributi di visualizzazione (definiti dalla classe *Rendered*).

Il capitolo 10.2 fornisce una lista di tutti i tag supportati e illustra come utilizzarli.

L’implementazione di JavaServer Faces offre una libreria di tag personalizzate per la visualizzazione dei componenti in HTML.

## 7.4.3 Conversion Model

Un’applicazione JavaServer Faces può (opzionalmente) associare un componente con un oggetto dati lato-server.

Questo oggetto è un componente JavaBean, come un managed bean.

Un’applicazione prende e setta (get e set) l’oggeto dati per un componente tramite la chiamata delle proprietà dell’oggetto appropriato per il detto componente.

Quando un componente è legato a un oggetto l’applicazione ha due views dei dati del componente:

* La **model view** in cui i dati sono rappresentati come tipi di dati (ad esempio *int* o *long*).
* La **presentation view** in cui sono rappresentati in un modo che può essere letto o modificato dall’utente (esempio un *java.util.Date* verrà visualizzato nel formato mm/dd/yyyy o come un set di tre stringhe).

L’implementazione di JavaServer Faces converte automaticamente i dati dei componenti tra queste due view quando la proprietà del bean associato con il componente è di uno dei tipi supportati dai dati del componente stesso (ad esempio se un componente *UISelectBoolean* è associato a una proprietà del bean di tipo *java.lang.Boolean* i dativerranno automaticamente convertiti da *String* a *Boolean*.

Inoltre alcuni dati del componente devono essere legati a proprietà di un tipo particolare (ad esempio un componente *UISelectBoolean* deve essere collegato a una proprietà di tipo *boolean* o *java.lang.Boolean*).

A volte si potrebbe voler convertire i dati di un componente in un tipo che non sia quello standard o si potrebbe voler convertire il formato dei dati. Per semplificare questa operazione la tecnologia JavaServer Faces permette di registrare un’implementazione di *javax.faces.convert.Converter*  sui componenti *UIOutput* e sottoclassi.

Registrando l’implementazione di tale *Converter* su un componente i dati verranno da lui convertiti tra le due views.

## 7.4.4 Event and Listene Model

Il modello Java Server Faces event and listener è similare all’event model dei JavaBeans dato che ha classi evento fortemente tipicizzate e interfacce listener che un’applicazione può usare per gestire gli eventi generati dai componenti.

La specifica di JavaServer Faces definisce tre tipi di eventi: application events, system events e data-model events.

Gli **application events** sono legati a una particolare applicazione e generati da un *UIComponent*.

Rappresentano gli eventi standard disponibili nelle versioni precedenti di JavaServer Faces.

Un oggetto evento identifica i componenti che generano l’evento e immagazzina le informazioni su di esso.

Per essere messa al corrente di un evento l’applicazione deve fornire un’implementazione della classe listener e registrarla sul componente che genera l’evento.

Quando un utente attiva un componente, ad esempio cliccando un bottone, l’evento viene eseguito.

Ciò causa l’invocazione del metodo listener che processa l’evento.

JavaServer Faces supporta due tipi di application events: action events e value-change events.

Un **action event** (*javax.faces.event.ActionEvent*) succede quando un utente attiva un componente che implementa *ActionSorce*. Questi componenti includono bottoni e links.

Un **value-change event** (javax.faces.event.ValueChangeEvent) capita quando l’utente cambia il valore di un componente rappresentato da *UIInput* per una delle sue sottoclassi.

Ad esempio un’azione che cambi il valore della selezione di una check box a true.

I tipi di componenti che possono generare questo tipo di eventi sono i componenti *UIInput, UISelectOne, UISelectMany* e *UISelectBoolean.*

Gli eventi value change sono eseguiti solo se non vengono scopert errori di validazione.

A seconda del valore della proprietà **immediate** (vedi capitolo 10.2.1.2) del componente che crea l’evento, gli **action event** possono essere processati durante la fase di invocazione dell’applicazione o la fase di Apply Request Values e gli eventi **value-change** possono essere processati durante la fase Process Validations o quella di Apply Request Values.

I **System events** sono generati da un oggetto invece che da componenti *UIComponent*.

Sono generati durante l’esecuzione di un’applicazione in momenti predefiniti.

Sono applicabili all’intera applicazione invece che a uno specifico componente.

Un **data-model event** capita quando viene selezionata una nuova riga di un componente *UIData*.

Esistono due modi per causare un reazione da parte dell’applicazione a un action event o a un value-change event che sono generati da un componente standard:

* Implementare una classe event listener per gestire l’evento e registrarla sul componente annidando un tag *f:valueChangeListener* o un tag *f:actionListener* dentro il tag del componente.
* Implementare un metodo di un managed bean per gestire l’evento e riferirsi al metodo con un’espressione dall’attributo appropriato del tag del componente.

## 7.4.5 Validation Model

La tecnologia JavaServer Faces supporta un meccanismo per validare i dati locali di componenti modificabili (quali i campi di testo). Questa validazione viene compiuta prima che il model data corrispondente effettui l’update per correlare i valori locali.

Come il conversion model, il validation model definisce un set di classi standard per performare controlli di validazione di dati comunemente usati. La libreria di tag del core del JavaServer Faces definisce un set di tag che corrispondono all’implementazione standard di *javax.faces.validator.Validator* (vedere capitolo 11.3 per la lista).

La maggior parte dei tag ha un set di attributi per configurare le proprietà del validatore, ad esempio i valori minimi e massimi consentiti per tale dato.

L’autore della pagina registra il validatore su un componente annidando le tag del validatore nel tag del componente.

Oltre ai validatori registrati sui componenti si può dichiarare un validatore di default registrato su tutti i componenti *UIInput* dell’applicazione.

Il modello di validazione permette inoltre di creare il proprio validatore personalizzato e i propri tag per effettuare la validazione. Il validation model offre due modi per implementare un validatore personalizzato:

* Implementare un’interfaccia Validator che effettua la validazione
* Implementare un metodo di un managed bean che effettua la validazione.

Se si implementa l’interfaccia si deve anche:

* Registrare l’implementazione del Validator nell’applicazione
* Creare la propria tag personalizzata o usare un tag *f:validator* per registrare il validatore sul componente.

## 7.5 Navigation Model

Il navigation model rende più semplice definire la navigazione della pagina e gestire ogni processo addizionale utile per scegliere la sequenza di caricamento delle pagine.

La navigazione è un set di regole per scegliere la prossima pagina o view che deve essere visualizzata dopo un’azione.

La navigazione può essere implicita o definita dall’utente.

Quando si aggiunge un componente quale *commandButton* a una pagina Facelets e si assegna un’altra pagina come valore per la proprietà azione, il gestore di navigazione di default proverà a abbinare un pagina adatta in automatico.

Nel seguente esempio il gestore di navigazione di default proverà a localizzare una pagina nominata *response.xhtml* nell’applicazione e navigare verso la stessa:

*<h:commandButton value="submit" action="response">*

Le regole di navigazione definite dall’utente sono dichiarate in nessun o più file di configurazione dell’applicazione, come ad esempio *faces-config.xml* usando un set di elementi XML.

La struttura di default delle regole di navigazione è la seguente:

*<navigation-rule>*

*<description></description*

*<from-view-id></from-view-id>*

*<navigation-case>*

*<from-action></from-action>*

*<from-outcome></from-outcome>*

*<if></if>*

*<to-view-id></to-view-id>*

*</navigation-case>*

*</navigation-rule>*

La navigazione definita dall’utente è gestita in questo modo:

* Definizione delle regole nel file di configurazione dell’applicazione.
* Riferimento a una String di uscita dall’attributo di azione di un bottone o link.

Esempio:

<navigation-rule>

<from-view-id>/greeting.xhtml</from-view-id>

<navigation-case>

<from-outcome>success</from-outcome>

<to-view-id>/response.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

Questa regola esplicita che quando un componente di comando (*h:commandButton* o *h:commandLink*) è attivato su *greeting.xhtml* l’appligazione navigherà da tale pagina alla pagina *response.xhtml* se l’outcome a cui si riferisce il tag del bottone è *success*.

Qua sotto c’è un tag *h:commandButton* da *greeting.xhtml* che specifica ciò:

*<h:commandButton id="submit" value="Submit" action="success"/>*

Come dimostra l’esempio ogni elemento che regola la navigazione definisce come prendere da una pagina (specificata dall’elemento *from-view-id*) ad altre pagine dell’applicazione.

Gli elementi *navigation-rule* possono contenere un qualunque numero di elementi *navigation-case* ognuno dei quali definisce la pagina da aprire successivamente (definita da *to-view-id*) basato su un outcome logico (definito da *from-outcome*).

In applicazioni più complesse l’outcome logico può anche arrivare dal valore di ritorno di un metodo di azione di un managed bean. Tale metodo performa alcuni processamenti per determinare l’outcome.

Ad esempio il metodo può controllare se la password che l’utente ha inserito è corretta, se lo fa il metodo tornerà *success* altrimenti ritornerà *failure*.

Un outcome di *failure* potrebbe far ricaricare la pagina di logon mentre quello di *success* potrebbe far mostrare alla pagina le attività della carta di credito dell’utente.

Se si vuole far ritornare l’outcome da un metodo o un bean bisogna riferirsi al metodo udando un’espressione con l’attributo *action*, come mostrato sotto:

*<h:commandButton id="submit" value="Submit"*

*action="#{cashierBean.submit}" />*

Quando l’utente clicca il bottone rappresentato in questo tag, il componente corrispondente genera l’evento, l’evento viene gestito dall’ istanza di default di *javax.faces.event.ActionListener* che chiama il metodo a cui si riferisce il componente che ha innescato l’evento.

Il metodo ritorna l’outcome logico al listener che lo passa con un riferimento al metodo dell’azione che produce l’outcome al *javax.faces.application.NavigationHandler* che seleziona la pagina da visualizzare successivamente andando a comparare l’outcome o il riferimento al metodo dell’azione con le regole di navigazione nel file di configurazione dell’applicazione seguendo questo processo:

1. Il *NavigationHandler* seleziona la regola di navigazione corrispondente alla pagina attualmente visualizzata.
2. Lo compara con l’outcome o il riferimento al metodo dell’azione ricevuto dal *javax.faces.event.ActionListener* tra quelli definiti nei casi di navigazione (navigation cases).
3. Prova a comparare sia il riferimento al metodo che l’outcome con lo stesso navigation case.
4. Se il precedente punto fallisce il gestore di navigazione tenta di comparare l’outcome.
5. Infine, se il gestore di navigazione tenta di comparare il riferimento al metodo di navigazione se i due punti precedenti falliscono.
6. Se nessun navigation case corrisponde mostra di nuovo la stessa view.

*Quando il* NavigationHandler *trova una corrispondenza inizia la fase di* ***Render Response****.*

Durante questa fase la pagina selezionata dal *NavigationHandler* viene visualizzata.

Il seguente esempio utilizza le regole di navigazione nel metodo business che gestisce la creazione, la modifica e la cancellazione dell’utente dell’applicazione.

Il form per creare uno studente ha il seguente tag per il bottone:

*<h:commandButton id="submit"*

*action="#{adminBean.createStudent(studentManager.newStudent)}"*

*value="#{bundle['action.submit']}"/>*

L’action event chiama il metodo *dukestutoring.ejb.AdminBean.createStudent*:

*public String createStudent(Student student) {*

*em.persist(student);*

*return "createdStudent";*

*}*

Il valore di ritorno di *createdStudent* ha un caso di navigazione corrispondente nel file di configurazione *faces-config.xml*:

<navigation-rule>

<from-view-id>/admin/student/createStudent.xhtml</from-view-id>

<navigation-case>

<from-outcome>createdStudent</from-outcome>

<to-view-id>/admin/index.xhtml</to-view-id>

</navigation-case>

</navigation-rule>

Dopo la creazione dello studente l’user viene ritornato alla pagina indice dell’amministratore.

## 7.6 Il ciclo vitale di un’applicazione JavaServer Faces

Il ciclo vitale di un’applicazione fa riferimento a vari passaggi del processamento, dall’inizializzazione alla conclusione. Tutte le applicazioni hanno un ciclo vitale.

Durante il ciclo vitale di un’applicazione web vengono eseguiti determinati compiti che sono:

* Gestione di richieste in entrata
* Decodifica dei parametri
* Modifica e salvataggio dello stato
* Visualizzazione di pagine web su browser

Il framework per le applicazioni web JavaServer Faces gestisce le fasi del ciclo vitale in maniera automatica per semplici applicazioni o permette allo sviluppatore di gestirlo manualmente per quelle più complesse se necessario.

Le applicazioni che utilizzano funzionalità avanzate potrebbero richiedere l’interazione con il ciclo vitale in certe fasi. Ad esempio le applicazione Ajax usano funzionalità di processamento parziali del ciclo vitale.

Una chiara comprensione delle fasi del ciclo vitale è requisito fondamentale per creare componenti ben disegnato.

## 7.6.1 Overview of the JavaServer Faces Lifecycle

Il ciclo vitale di un’applicazione JavaServer Faces inizia quando il client fa una richiesta HTTP per una pagina e termina quando il server risponde con la pagina tradotta in HTML.

Il ciclo vitale può essere diviso in due fasi principali: **Execute** e **Render**.

La fase di esecuzione è inoltre divisa in sottofasi che supportano l’albero dei componenti (component tree).

Questa struttura richiede che i dati dei componenti siano convertiti e validati, gli eventi gestiti e che i dati del componente propagati tramite beans in maniera ordinata.

Una pagina JavaServer Faces è rappresentata da un albero dei componenti, chiamato **view**.

Durante il ciclo vitale, l’implementazione di JavaServer Faces deve costruire la view mentre considera lo stato salvato da una precedente submission della pagina.

Quando il client richiede una pagina l’implementazione di JavaServer Faces porta a termine diversi compiti quali la validazione dei dati da input dei componenti nella view e la conversione dei dati input in tipi specificati a lato-server.

Tutti questi compiti vengono eseguiti dall’implementazione di JavaServer Faces in una serie di step nel ciclo vitale del request-response. La figura seguente mostra questi passaggi:



Il ciclo di vita del request-response gestisce due tipi di richieste: **initial requests** e **postbacks**.

Un’initial request è quando un utente fa una richiesta per una pagina per la prima volta mentre il postback è quando un utente fa il submit di un form contenuto in una pagina precedenemente caricata come risultato di un’initial request.

Quando il ciclo vitale gestisce un’initial request, esegue solo le fasi di **Restore View** e di **Rendere Response** perché non c’è input da parte dell’utente o azione da processare mentre quando gestisce un postback esegue tutte le fasi.

Solitamente la prima richiesta per una pagina viene da un client come risultato del click su un link o un bottone su una pagina JavaServer Faces.

Per visualizzare una risposta che sia un’altra pagina JavaServer Faces l’applicazione crea una nuova view e la immagazzina in un’istanza *javax.faces.context.FacesContext* che rappresenta tutte le informazioni associate al processamento della richiesta in entrata e alla creazione della risposta.

Poi l’applicazione acquisisce i riferimenti all’oggetto richiesto dall view e chiama il metodo *FacesContext.renderResponse* che forza la visualizzazione immediata della view saltando la fase di Render Response del ciclo vitale, come mostrato dal’imagine sopra.

A volte un’applicazione può necessitare di reindirizzare a una risorsa di un’applicazione web diversa, come un servizio web, o generare risposte che non contengano componenti JavaServer Faces. In tali situazioni lo sviluppatore deve saltare la fase di Render Response chiamando il metodo *FacesContext.responseComplete*.

La situazione più comune è che un componente JavaServer Faces sottometta una richiesta da un’altra pagina JavaServer Faces, in tal caso l’implementazione di JavaServer Faces gestisce le richieste e passa automaticamente da tutte le fasi del ciclo vitale per effettuare ogni conversione, validazione e model update necessario a generare la risposta.

Esiste un’unica eccezione al ciclo vitale descritto in questo capitolo: quando un attributo ***immediate*** è settato su true, in questo caso la validazione, conversione e gli eventi associati a questi componenti sono processati durante la fase di Apply Request Values invece che nelle fasi successive.

La fase **Execute** del ciclo vitale di un’applicazione JavaServer Faces contiene le seguenti sottofasi:

* Fase di **Restore View**
* Fase di **Apply Request Values**
* Fase di **Process Validations**
* Fase di **Update Model Values**
* Fase di **Invoke Application**
* Fase di **Render Response**

## 7.6.2 Fase di Restore View

Quando viene avanzata la richiesta di una pagina JavaServer Faces, solitamente da un’azione come il click su un bottone, inizia la fase del Restore View.

Durante questa fase l’implementazione di JavaServer Faces costruisce la view della pagina, lega i gestori degli eventi e i validatori ai componenti della view e salva la view in un’istanza *FacesContext* che contiene tutte le informazioni di cui si ha bisogno per processare una singola richiesta.

Tutti i componenti dell’applicazioni, gli event handler, i convertitori e i validatori hanno accesso a tale istanza.

Se la richiesta è una **initial request** l’implementazione crea una vista vuota durante questa fase e il ciclo avanza alla fase di Render Response durante la quale la vista vuota viene riempita con i componenti a cui si fa riferimento tramite le tag della pagina.

Se invece la richiesta è un **postback** la vista corrispondente esiste già nell’istanza *FacesContext* e quindi l’implementazione ripristina la vista utilizzando le informazioni di stato salvate sul client o sul server.

## 7.6.3 Apply Request Values Phase

Dopo aver ripristinato l’albero dei componenti durante una richiesta postback ogni componente estrai il proprio nuovo valore dai parametri della richiesta utilizzando il proprio metodo *processDecodes()*.

Il valore viene quindi immagazzinato localmente per ogni componente.

Se un metodo di decode o un event listener ha chiamato il metodo *renderResponse* sull’istanza corrente di *FacesContext* l’implementazione salta alla fase di *Render Response*.

Se degli eventi sono in coda durante questa fase, l’implementazione di JavaServer Faces trasmette l’evento sui listeners interessati.

Se qualche componente sulla pagina ha degli attributi ***immediate*** settati su true le validazioni, conversioni e gli eventi associati vengono processati durante questa fase. Se qualche conversione fallisce viene generato un messaggio di errore che viene messo in coda sul *FacesContext*, tale messaggio sarà visualizzato durante la fase di Render Response insieme a ogni errore di validazione risultante dalla fase di Process Validations.

A questo punto se l’applicazione necessita di reindirizzare a una risorsa di un’altra applicazione web o genera una risposta che non contiene alcun componente JavaServer Faces, può chiamare il metodo *FacesContext.responseComplete*.

Alla fine di questa fase i componenti sono settati al loro nuovo valore e i messaggi e gli eventi sono stati messi in coda.

Se la richiesta corrente è una partial request il contesto parziale viene recuperato dal *FacesContext* e il metodo di processamento parziale viene applicato.

## 7.6.4 Process Validations Phase

Durante questa fase l’implementazione di Java Server Faces processa tutti i validatori registrati sui componenti dell’albero utilizzando il suo metodo di validazione (processValidators) che esamina gli attributi dei componenti che specificano le regole per la validazione e comparano le stesse al valore locale immagazzinato per il componente. Inoltre JavaServer Faces completa la conversione per i componenti input che non hanno l’attributo ***immediate*** settato su true.

Se un valore locale non valido o una conversione fallisce l’implementazione di JavaServer Faces aggiunge un messaggio di errore all’istanza di *FacesContext* e il ciclo avanza direttamente alla fase di Render Response in modo che la pagina venga visualizzata di nuovo mostrando il messaggio di errore. Se ci sono errori di conversione dalla fase di Apply Request Values vengono mostrati anche quei messaggi di errore.

Se un metodo di validazione o un event listener hanno chiamato il metodo *renderResponse* su *FacesContext* corrente, si salta alla fase di Render Response.

A questo punto, se l’applicazione necessita di reindirizzare a una risorsa di un’altra applicazione web o genera una risposta che non contiene componenti JavaServer Faces, può chiamare il metodo *FecesContext.responseComplete*.

Se degli eventi sono stati messi in coda in questa fase, vengono trasmessi ai listeners interessati.

Se la richiesta corrente è una partial request, il contesto parziale viene recuperato da *FacesContext* e il metodo di processamento parziale applicato.

## 7.6.5 Update Model Values Phase

Dopo che l’implementazione di JavaServer Faces determina che i dati sono validi, essa scala l’albero dei componenti e setta le proprietà dell’oggetto server-side corrispondente ai valori locali dei componenti.

L’implementazione di JavaServer Faces sovrascrive solo le proprietà del bean che puntano al valore di un attributo di un componente input.

Se i dati locali non possono essere convertiti nei tipi specificati dalle proprietà del bean il ciclio vitale avanza direttamente alla fase di Render Response in modo che la pagina sia visualizzata di nuovo mostrando gli errori come accadeva con gli errori di validazione.

Se qualche metodo *updateModels* o qualche listener chiama il metodo *renderResponse* sull’istanza di *FacesContext* corrente si salta alla fase di Render Response.

A questo punto, se l’applicazione necessita di reindirizzare a una risorsa di un’altra applicazione web o genera una risposta che non contiene componenti JavaServer Faces, può chiamare il metodo FecesContext.responseComplete.

Se degli eventi sono stati messi in coda in questa fase, vengono trasmessi ai listeners interessati.

Se la richiesta corrente è una partial request, il contesto parziale viene recuperato da FacesContext e il metodo di processamento parziale applicato.

## 7.6.6 Invoke Application Phase

Durante questa fase JavaServer Faces gestisce ogni evento application-level, come il submitting di un form o il linking a un’altra pagina.

A questo punto, se l’applicazione necessita di reindirizzare a una risorsa di un’altra applicazione web o genera una risposta che non contiene componenti JavaServer Faces, può chiamare il metodo FecesContext.responseComplete.

Se la vista che viene processata è recostruita dalle informazioni di stato di una richiesta precedente e se un componente ha eseguito un evento questi eventi vengono trasmessi ai listeners interessati.

Infine l’implementazione di JavaServer Faces trasferisce il controllo alla fase Render Response.

## 7.6.7 Render Response Page

Durante questa fase JavaServer Faces costruisce la view e delega l’autorità alle risorse appropriate per la visualizzazione delle pagine.

Se questa è un’ initial request i componenti rappresentati nella pagina saranno aggiunti all’albero dei componenti, se no sono già nell’albero e quindi non necessitano di esservi reintrodotti.

Se la richiesta è un postback e ci sono errori riscontrati nella fasi Apply Request Values, Process Validations o Update Model Values la pagina originale viene visualizzata di nuovo durante questa fase. Se le pagine contengono tag *h:message* o *h:messages*ogni messaggio di errore in coda viene visualizzato sulla pagina.

Dopo che il contenuto della view viene renderizzato viene salvato lo stato della risposta in modo che le richieste seguenti possano accedervi. Lo stato salvato è disponibile per la fase di Restore View.

## 7.7 Partial Processing and Partial Rendering

Il ciclo di vita di JavaServer Faces abbraccia tutti i processi di esecuzione e visuaizzazione di un’applicazione.

E’ inoltre possibile processare e visualizzare solo parti di un’applicazione, ad esempio un singolo componente.

Ad esempio il framework JavaServer Faces Ajax può generare richieste contenenti informazioni su come un componente specifico deve essere processato e quale componente deve essere renderizzato al client.

Una volta che una richiesta parziale entre nel ciclo di vita di JavaServer Faces, l’informazione viene identificata e processata da un oggetto *javax.faces.context.PartialViewContext*.

Il ciclo di vita di JavaServer Faces continua a essere consapevole di queste richieste Ajax e modifica di conseguenza il proprio component tree.

Gli attributi *execute* e *render* del tag *f:ajax* sono utilizzati per identificare quali componenti devono essere eseguiti e renderizzati, per maggiori informazioni vedere il capitolo 13.