JSF 2 の魅力: 第 3 回 イベント処理、JavaScript、そして Ajax

JSF 2 の新機能をさらに活用して複合コンポーネントを拡張する

David Geary
President
Clarity Training, Inc.

2009年 7月 14日

JSF (Java™Server Faces) 2 Expert Group のメンバー、David Geary が JSF 2 の新機能を紹介する 3 回の連載は、今回が最終回です。この連載の締めくくりとして、このフレームワークの新しいイベント・モデルと Ajax の組み込みサポートによって、再利用可能なコンポーネントをなお一層のこと強力にする方法を説明します。

このシリーズの他の記事を見る

JSF の最大のセールスポイントの 1 つは、これがコンポーネント・ベースのフレームワークであるということです。つまり、自分や他の開発者が再利用できるコンポーネントを実装することができます。しかし JSF 1 ではコンポーネントの実装がかなり困難であったため、この強力な再利用のメカニズムはほとんど重要視されていませんでした。

それが JSF 2 になると、連載第 2 回で説明したように、複合コンポーネントという新しい機能を使うことで、Java コードを作成したり、構成を行ったりしなくても、簡単にコンポーネントを実装できるようになっています。JSF コンポーネントの可能性をついに現実のものにするこの機能は、JSF 2 の最も重要な部分となるはずです。

JSF 2 を紹介する連載の最終回となるこの第 3 回では、JSF 2 を最大限に活用するための次の 3 つのヒントに沿って、JSF 2 で同じく新しく導入された Ajax のサポート機能およびイベント処理機能を使用して複合コンポーネント機能を拡張する方法を説明します。

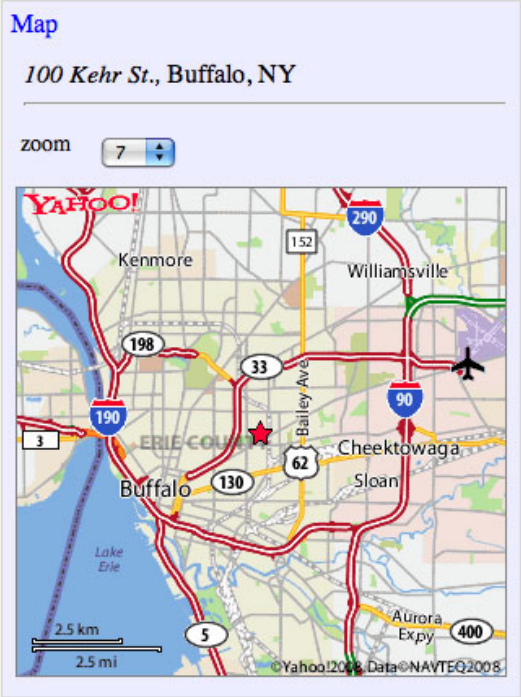
- ヒント 1: コンポーネント化する
- ヒント 2: Ajax 化する
- ヒント 3: 進行状態を示す

最初のヒントのなかで、第 2 回で詳しく説明した 2 つのコンポーネントについて簡単に復習します。続く 2 つのヒントでは、Ajax とイベント処理を使用してこの 2 つのコンポーネントを変換する方法を説明します。

ヒント 1: コンポーネント化する

第 1 回で導入した Places アプリケーションには、今やいくつもの複合コンポーネントが含まれています。そのうちの 1 つは、指定された住所の地図を表示する `map` コンポーネントです。この複合コンポーネントには、ズーム・レベルのプルダウン・メニューが備わっています (図 1 を参照)。

図 1. Places アプリケーションの `map` コンポーネント



リスト 1 に、`map` コンポーネントのコード (省略バージョン) を記載します。

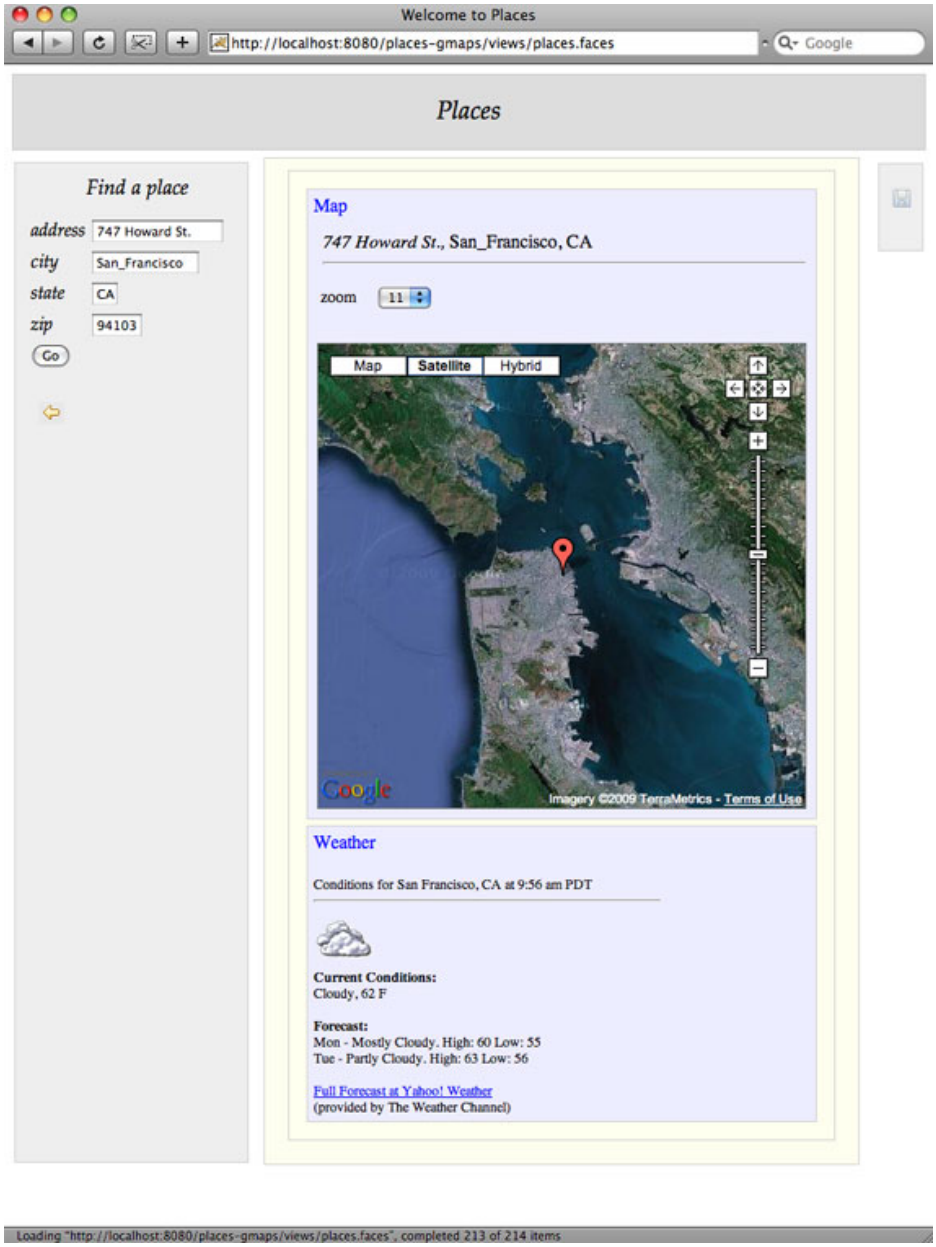
リスト 1. `map` コンポーネント

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<html xmlns="http://www.w3.org/1999/xhtml"
    ...
    xmlns:composite="http://java.sun.com/jsf/composite"
    xmlns:places="http://java.sun.com/jsf/composite/components/places">

   <!-- INTERFACE -->
   <composite:interface>
     <composite:attribute name="title"/>
   </composite:interface>

   <!-- IMPLEMENTATION -->
   <composite:implementation">
     <div class="map">
       ...
       <h:panelGrid...>
         <h:panelGrid...>
            <h:selectOneMenu onchange="submit()"
                 value="#{cc.parent.attrs.location.zoomIndex}"
                 valueChangeListener="#{cc.parent.attrs.location.zoomChanged}"
                 style="font-size:13px;font-family:Palatino">

              <f:selectItems value="#{places.zoomLevelItems}"/>
```

```
            </h:selectOneMenu>
          </h:panelGrid>
        </h:panelGrid>

        <h:graphicImage url="#{cc.parent.attrs.location.mapUrl}"
          style="border: thin solid gray"/>
       ...

     </div>
   ...

  </composite:implementation>
</html>
```

コンポーネントの素晴らしいところは、関連する機能にまったく影響を与えることなく、さらに強力なコンポーネントに置き換えられるという点です。例えば図 2 では、リスト 1 の `image` コンポーネントを GMaps4JSF 提供の Google マップ・コンポーネントに置き換えています (「参考文献」を参照)。

図 2. GMaps4JSF による地図の画像



`map` コンポーネントを更新した後のコード (省略バージョン) をリスト 2 に記載します。

リスト 2. 地図の画像と GMaps4JSF コンポーネントとの置換

```
<h:selectOneMenu onchange="submit()"value="#{cc.parent.attrs.location.zoomIndex}"
              valueChangeListener="#{cc.parent.attrs.location.zoomChanged}"
              style="font-size:13px;font-family:Palatino">

  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>

...

<m:map id="map" width="420px" height="400px"
     address="#{cc.parent.attrs.location.streetAddress}, ..."
```

```
    zoom="#{cc.parent.attrs.location.zoomIndex}"
    renderOnWindowLoad="false">

  <m:mapControl id="smallMapCtrl"
              name="GLargeMapControl"
          position="G_ANCHOR_TOP_RIGHT"/>

  <m:mapControl  id="smallMapTypeCtrl" name="GMapTypeControl"/>
  <m:marker id="placeMapMarker"/>

</m:map>
```

GMaps4JSF コンポーネントを使用するために、`<h:graphicImage>` タグを GMaps4JSF コンポーネント・セットに含まれる `<m:map>` タグに置換しました。また、GMaps4JSF コンポーネントをズーム・プルダウンに接続するのも簡単で、`<m:map>` タグの `zoom` 属性に適切な Backing Bean プロパティーを指定するだけのことです。

ズーム・レベルについて言うと、ユーザーがズーム・レベルを変更するとフォームがサブミットされるようにしています。これには、リスト 1 で部分的に太字になっている最初の行に示すように、`<h:selectOneMenu>` の `onchange` 属性を使用しています。フォームのサブミットによって JSF ライフサイクルがトリガーされ、最終的にはズーム・レベルの値が、親複合コンポーネントに保存された `location` Bean の `zoomIndex` プロパティーに入れられます。この Bean のプロパティーが、リスト 2 の最初の行にある入力コンポーネントにバインドされています。

ズーム・レベルの変更に関連付けられたフォームのサブミットにはナビゲーションを指定していないので、JSF はリクエストを処理した後に同じページを更新表示し、地図の画像を再描画して新しいズーム・レベルが反映されるようにします。しかしこのページの更新表示によって、変更されたのは地図の画像だけであるにも関わらず、ページ全体が再描画されてしまうことになります。Ajax を使用して、ズーム・レベルの変更に応答して地図の画像のみを再描画するように変更する方法については、「ヒント 2: Ajax 化する」で説明します。

`login` コンポーネント

Places アプリケーションで使用されているコンポーネントには、login コンポーネントもあります。図 3 に、実際の `login` コンポーネントを示します。

図 3. `login` コンポーネント

リスト 3 に、図 3 の `login` コンポーネントを生成したマークアップを記載します。

リスト 3. 必要な属性のみで構成される最小限の `login` コンポーネント

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:util="http://java.sun.com/jsf/composite/components/util">

  <util:login loginAction="#{user.login}"
              managedBean="#{user}"/>

</ui:composition>
```

`login` コンポーネントに必要な属性は、以下の 2 つだけです。

- `loginAction`: ログイン・アクション・メソッド
- `managedBean`: 名前およびパスワードのプロパティーを持つ管理対象 Bean

リスト 4 に、リスト 3 で指定されている管理対象 Bean を記載します。

リスト 4. User.groovy

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped

@ManagedBean()
@SessionScoped

public class User {
  private final String VALID_NAME     = "Hiro"
  private final String VALID_PASSWORD = "jsf"

  private String name, password;

  public String getName() { name }
  public void setName(String newValue) { name = newValue }

  public String getPassword() { return password }
  public void setPassword(String newValue) { password = newValue }

  public String login() {
    "/views/places"
  }

  public String logout() {
    name = password = nameError = null
    "/views/login"
  }
}
```

リスト 4 の管理対象 Bean は Groovy Bean です。この場合、Java 言語の代わりに Groovy を使用しても、セミコロンと return 文の単調な作業から解放される以外にはそれほどメリットがありません。その一方、ヒント 2 の「検証」セクションで説明しますが、User 管理対象 Bean には Groovy を使用せざるを得ない理由があります。

大抵は、図 4 のようにプロンプトとボタン・テキストを完備した login コンポーネントを構成したいと思うはずです。

図 4. 完全に構成された `login` コンポーネント

リスト 5 に、図 4 の `login` コンポーネントを生成したマークアップを記載します。

リスト 5. `login` コンポーネントの構成

```
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:ui="http://java.sun.com/jsf/facelets"
  xmlns:util="http://java.sun.com/jsf/composite/components/util">

  <util:login loginPrompt="#{msgs.loginPrompt}"
              namePrompt="#{msgs.namePrompt}"
          passwordPrompt="#{msgs.passwordPrompt}"
         loginButtonText="#{msgs.loginButtonText}"
             loginAction="#{user.login}"
             managedBean="#{user}"/>

</ui:composition>
```

リスト 5 では、プロンプト用のストリングとログイン・ボタンのテキストを、リソース・バンドルから取得しています。

`login` コンポーネントの定義は、リスト 6 のとおりです。

リスト 6. `login` コンポーネントの定義

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!-- Usage:

  <util:login loginPrompt="#{msgs.loginPrompt}"
              namePrompt="#{msgs.namePrompt}"
          passwordPrompt="#{msgs.passwordPrompt}"
         loginButtonText="#{msgs.loginButtonText}"
             loginAction="#{user.login}"
             managedBean="#{user}">

    <f:actionListener for="loginButton"
                      type="com.clarity.LoginActionListener"/>

  </util:login>

  managedBean must have two properties: name and password.

  The loginAction attribute must be an action method that takes no
  arguments and returns a string. That string is used to navigate
  to the page the user sees after logging in.
```

```
  This component's loginButton is accessible so that you can
  add action listeners to it, as depicted above. The class specified
  in f:actionListener's type attribute must implement the
  javax.faces.event.ActionListener interface.

 -->

<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
    xmlns:composite="http://java.sun.com/jsf/composite">

  <!-- INTERFACE -->
  <composite:interface>

    <!-- PROMPTS -->
    <composite:attribute name="loginPrompt"/>
    <composite:attribute name="namePrompt"/>
    <composite:attribute name="passwordPrompt"/>

    <!--  LOGIN BUTTON -->
    <composite:attribute name="loginButtonText"/>

    <!-- loginAction is called when the form is submitted -->
    <composite:attribute name="loginAction"
             method-signature="java.lang.String login()"
                     required="true"/>

    <!-- You can add listeners to this actionSource:  -->
    <composite:actionSource name="loginButton" targets="form:loginButton"/>

    <!-- BACKING BEAN -->
    <composite:attribute name="managedBean" required="true"/>
  </composite:interface>

  <!-- IMPLEMENTATION -->
  <composite:implementation>
    <div class="prompt">
      #{cc.attrs.loginPrompt}
    </div>

    <!-- FORM -->
    <h:form id="form">
      <h:panelGrid columns="2">

        <!-- NAME AND PASSWORD FIELDS -->
        #{cc.attrs.namePrompt}
        <h:inputText id="name"
                  value="#{cc.attrs.managedBean.name}"/>

        #{cc.attrs.passwordPrompt}
        <h:inputSecret id="password" size="8"
          value="#{cc.attrs.managedBean.password}"/>

      </h:panelGrid>

      <p>
        <!-- LOGIN BUTTON -->
        <h:commandButton id="loginButton"
          value="#{cc.attrs.loginButtonText}"
          action="#{cc.attrs.loginAction}"/>
      </p>
    </h:form>
  </composite:implementation>
</html>
```

`map` コンポーネントと同じく、`login` コンポーネントも Ajax を使用するようにアップグレードすることができます。`login` コンポーネントに Ajax による検証を追加する方法については、次のヒントの「検証」セクションで説明します。

ヒント 2: Ajax 化する

Ajax には通常、Ajax 以外の HTTP リクエストでは行わない 2 つのステップが必要になります。1 つはサーバー上でのフォームの部分的処理、そしてもう 1 つは、その後のクライアント上での DOM (Document Object Model) の部分的レンダリングです。

部分的処理とレンダリング

JSF 2 は、JSF ライフサイクルを実行とレンダリングという 2 つの論理上の部分に分けることによって、部分的な処理と部分的なレンダリングをサポートします。図 5 で強調表示しているのは、実行の部分です。

図 5. JSF ライフサイクルの実行部分

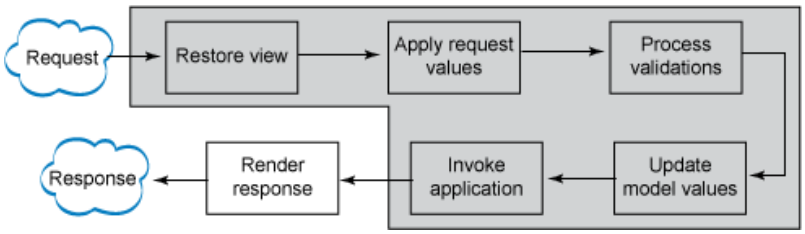
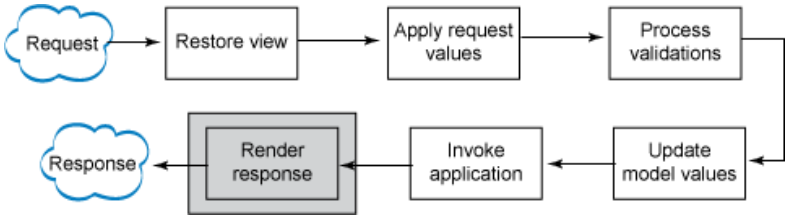


図 6 では、JSF ライフサイクルのレンダリング部分を強調表示します。

図 6. JSF ライフサイクルのレンダリング部分



ライフサイクルの実行部分とレンダリング部分の背後にある概念は単純で、JSF がサーバー上で実行 (処理) するコンポーネントと、Ajax 呼び出しから戻ったときに JSF がレンダリングするコンポーネントを指定できるようにするというものです。そしてそのための手段が、JSF 2 に用意された新しいタグ、`<f:ajax>` です (リスト 7 を参照)。

リスト 7. Ajax を使用した zoom メニュー

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax event="change" execute="@this" render="map"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>

<m:map id="map"...>
```

リスト 7 は、リスト 2 の最初の行に記載されたメニューを変更したものです。リスト 2 の `onchange` 属性を削除して、`<f:ajax>` タグを追加しました。`<f:ajax>` タグは、以下の項目を指定します。

- Ajax 呼び出しをトリガーするイベント
- サーバー上で実行するコンポーネント
- クライアント上でレンダリングするコンポーネント

ユーザーが zoom メニューから項目を選択すると、JSF がサーバーに対して Ajax 呼び出しを行います。続いて JSF は、ライフサイクルの実行部分にメニューを渡し (`@this` は、`<f:ajax>` を囲んでいるコンポーネントを表します)、ライフサイクルのモデル値更新 (Update model values) フェーズでメニューの `zoomIndex` を更新します。Ajax 呼び出しから戻った時点で JSF は map コンポーネントをレンダリングします。その際、地図を再描画するために使用されるのは、(新しく設定された) ズーム·インデックスです。このように 1 行の XHTML を追加することによって、zoom メニューが Ajax 化されました。

しかし、このコードはまだまだ単純化することができます。JSF は、`event` 属性と `execute` 属性のデフォルト値を提供するためです。

それぞれの JSF コンポーネントには、`<f:ajax>` タグがコンポーネント・タグの内側に組み込まれている場合に Ajax 呼び出しをトリガーするデフォルト・イベントがあります。メニューの場合、これに該当するイベントは `change` イベントです。つまり、リスト 7 の `<f:ajax>` で指定している `event` 属性は省くことができます。また、`<f:ajax>` の `execute` 属性のデフォルトは `@this` で、これは `<f:ajax>` を囲んでいるコンポーネントを表します。この例で言うと、これに該当するコンポーネントはメニューなので、`execute` 属性も同じく省くことができるというわけです。

`<f:ajax>` にデフォルト属性値を使用すると、リスト 7 のコードはリスト 8 のように簡潔になります。

リスト 8. Ajax を使用した zoom メニューの簡易化バージョン

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax render="map"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

</h:selectOneMenu>

<m:map id="map"...>
```

JSF 2 では、このようにいとも簡単に Ajax をコンポーネントに追加することができます。もちろん、この例はかなり単純なもので、ユーザーがズーム・レベルを選択したときに、ページ全体ではなく地図だけを再描画するようにしているにすぎません。これよりも複雑な例として、次はフォーム内の個々のフィールドを検証するケースについて取り組みます。

検証

ユーザーがタブでフィールドから移動するときには、フィールドの有効性を検証し、すぐにフィードバックを提供するのが良策です。例えば図 7 では、Ajax を使用して Name フィールドを検証しています。

図 7. Ajax を使用した検証

リスト 9 に、Name フィールドのマークアップを記載します。

リスト 9. Name フィールド

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:panelGrid columns="2">
  #{cc.attrs.namePrompt}
  <h:panelGroup>
    <h:inputText id="name" value="#{cc.attrs.managedBean.name}"
       valueChangeListener="#{cc.attrs.managedBean.validateName}">

       <f:ajax event="blur" render="nameError"/>

     </h:inputText>

     <h:outputText id="nameError"
       value="#{cc.attrs.managedBean.nameError}"
       style="color: red;font-style: italic;"/>
  </h:panelGroup>
  ...
</h:panelGrid>
```

ここでもまた `<f:ajax>` を使用しますが、この場合には入力のデフォルト・イベントである `change` では役に立ちません。そこで、Ajax 呼び出しをトリガーするイベントとして `blur` を指定します。ユーザーがタブで Name フィールドを離れると、JSF はサーバーに対して Ajax 呼び出しを行い、ライフサイクルの実行部分で `name` 入力コンポーネントを実行します。つまり、JSF はライフサイクルのプロセス検証 (Process validations) フェーズで、リスト 9 に指定された `name` 入力の値変更リスナーを起動するということです。リスト 10 に、値変更リスナーを記載します。

リスト 10. `validateName()` メソッド

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import javax.faces.event.ValueChangeEvent
import javax.faces.component.UIInput

@ManagedBean()
```

```
@SessionScoped

public class User {
  private String name, password, nameError;

  ...

  public void validateName(ValueChangeEvent e) {
    UIInput nameInput = e.getComponent()
    String name = nameInput.getValue()

    if (name.contains("_"))   nameError = "Name cannot contain underscores"
    else if (name.equals("")) nameError = "Name cannot be blank"
    else                      nameError = "
  }

  ...
}
```

この値変更リスナー (`user` 管理対象 Bean の `validateName()` メソッド) は、Name フィールドを検証し、`user` 管理対象 Bean の `nameError` プロパティーを更新します。

Ajax 呼び出しから戻ると、リスト 9 に記載した `<f:ajax>` タグの `render` 属性に基づき、JSF が `nameError` 出力をレンダリングします。その出力には、`user` 管理対象 Bean の `nameError` プロパティーが表示されます。

複数フィールドの検証

前のサブセクションで説明したのは単一のフィールドで Ajax 検証を実行する方法ですが、複数のフィールドを同時に検証しなければならない場合もあります。一例として、図 8 に Name フィールドと Password フィールドを同時に検証する Places アプリケーションを示します。

図 8. 複数フィールドの検証

この例では、ユーザーがフォームをサブミットする際に Name フィールドと Password フィールドを同時に検証するので、Ajax は必要ありません。代わりに JSF 2 の新しいイベント・システムを使用します (リスト 11 を参照)。

リスト 11. `<f:event>` の使用

```
<h:form id="form" prependId="false">

  <f:event type="postValidate"
       listener="#{cc.attrs.managedBean.validate}"/>
  ...
</h:form>

<div class="error" style="padding-top:10px;">
  <h:messages layout="table"/>
</div>
```

リスト 11 では、`<f:event>` を使用しています。このタグは `<f:ajax>` と同じく JSF 2 に用意された新しいタグですが、簡単に使えるという点でも `<f:event>` と `<f:ajax>` は似ています。

`<f:event>` タグをコンポーネント・タグの内側に組み込んだ場合、指定されたイベント (`type` 属性によって指定) がそのコンポーネントに対して発生すると、JSF が `listener` 属性で指定されたメソッドを呼び出します。したがってリスト 11 の `<f:event>` タグが指示する内容を説明すると、フォームを検証した後、ユーザーがこの複合コンポーネントに渡した管理対象 Bean 上で `validate()` メソッドを呼び出すということです。このメソッドをリスト 12 に記載します。

リスト 12. `validate()` メソッド

```
package com.clarity

import javax.faces.context.FacesContext
import javax.faces.bean.ManagedBean
import javax.faces.bean.SessionScoped
import javax.faces.event.ValueChangeEvent
import javax.faces.component.UIInput

@ManagedBean()
@SessionScoped

public class User {
  private final String VALID_NAME     = "Hiro";
  private final String VALID_PASSWORD = "jsf";

  ...

  public void validate(ComponentSystemEvent e) {
    UIForm form = e.getComponent()
    UIInput nameInput = form.findComponent("name")
    UIInput pwdInput = form.findComponent("password")

    if ( ! (nameInput.getValue().equals(VALID_NAME) &
        pwdInput.getValue().equals(VALID_PASSWORD))) {

      FacesContext fc = FacesContext.getCurrentInstance()
      fc.addMessage(form.getClientId(),
        new FacesMessage("Name and password are invalid. Please try again."))
      fc.renderResponse()
    }
  }

  ...
}
```

JSF がリスト 12 の `validate()` メソッドをコンポーネント・システム・イベントに渡すと、メソッドはそのイベントから、イベントが適用されるコンポーネントへの参照、つまりログイン・

フォームの参照を取得します。このフォームから、`findComponent()` メソッドを使用して名前とパスワードのコンポーネントを取得します。これらのコンポーネントの値がそれぞれ Hiro と jsf でない場合には、Faces コンテキストにメッセージを保存し、JSF に対して即時、ライフサイクルのレスポンス・レンダリング (Render response) フェーズに進むように指示します。このようにして、モデル値更新 (Update model values) フェーズで不正な名前とパスワードがモデルに渡されないようにしています (図 5 を参照)。

お気づきかもしれませんが、リスト 10 とリスト 12 の検証メソッドはいずれも Groovy で作成されています。リスト 4 では、Groovy を使用するメリットはセミコロンと return 文から解放されることだけでしたが、それとは異なり、リスト 10 とリスト 12 の Groovy コードには、キャストが不要になるというメリットがあります。例えばリスト 10 では、`ComponentSystemEvent.getComponent()` と `UIComponent.findComponent()` が返す型はどちらも `UIComponent` です。Java 言語を使用するとしたら、この 2 つのメソッドの値をキャストしなければならなくなります。Groovy なら、代わりにキャストしてくれます。

ヒント 3: 進行状態を示す

「Ajax 化する」セクションでは、`map` コンポーネントの zoom メニューを Ajax 化して、ユーザーがズーム・レベルを変更したときに、Places アプリケーションがページの地図の部分だけを再描画するようにしました。よくあるもう 1 つの Ajax の使用例は、ユーザーに Ajax イベントが進行中であるというフィードバックを提供することです (図 9 を参照)。

図 9. プログレス・バー

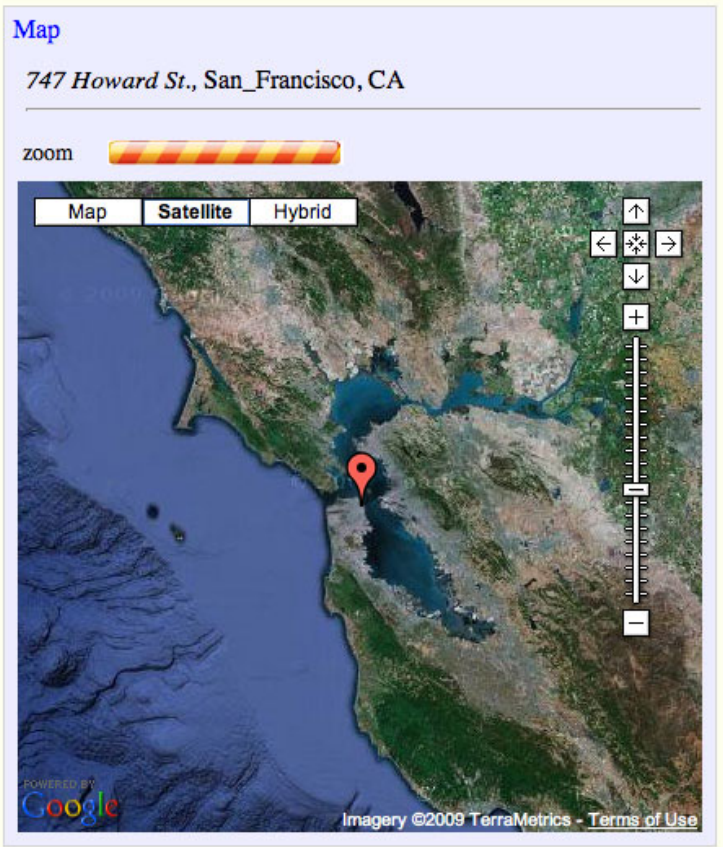


図 9 では、Ajax 呼び出しの進行中に zoom メニューの代わりに動画 GIF が表示されるようにしています。Ajax 呼び出しが完了すると、このプログレス・バーの画像を zoom メニューに置き換えます。リスト 13 に、その方法を示します。

リスト 13. Ajax リクエストのモニタリング

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
<h:selectOneMenu id="menu"
     value="#{cc.parent.attrs.location.zoomIndex}"
     style="font-size:13px;font-family:Palatino">

  <f:ajax render="map" onevent="zoomChanging"/>
  <f:selectItems value="#{places.zoomLevelItems}"/>

  ...
</h:selectOneMenu>
...
<h:graphicImage id="progressbar" style="display: none"
              library="images" name="orange-barber-pole.gif"/>
```

リスト 13 では、プログレス・バーの画像を最初は非表示の状態で追加し、`<f:ajax>` には `onevent` 属性を指定しています。この属性が参照するのは、リスト 13 で開始される Ajax 呼び出しの進行中に JSF が呼び出す JavaScript 関数です (リスト 14 を参照)。

リスト 14. Ajax リクエストに応答する JavaScript

```
function zoomChanging(data) {
  var menuId = data.source.id;
  var progressbarId = menuId.substring(0, menuId.length - "menu".length)
      + "progressbar";

  if (data.name == "begin") {
    Element.hide(menuId);
    Element.show(progressbarId);
  }
  else if (data.name == "success") {
    Element.show(menuId);
    Element.hide(progressbarId);
  }
}
```

JSF がリスト 14 の関数に渡すオブジェクトに含まれる情報は、イベントを起動したコンポーネントのクライアント識別子 (この例の場合、該当するコンポーネントはズーム・レベル・メニュー)、そして `name` という不適切な名前のプロパティーによって表された Ajax リクエストの現在の状況などです。

リスト 14 に記載された `zoomChanging()` 関数は、プログレス・バーの画像のクライアント識別子を計算した後、プロトタイプの `Element` オブジェクトを使用して、Ajax 呼び出しの間に該当する HTML 要素の非表示/表示を切り替えます。

まとめ

長年の間、JSF 1 は使いにくいフレームワークという評判を生んできました。多くの点で、その評判は当然と言えます。JSF 1 は象牙の塔で開発され、実際の使用に即した十分な後知恵が盛り込まれることもありませんでした。その結果、JSF は当初の予定よりも遙かにアプリケーションおよびコンポーネントを実装するのが難しいフレームワークとなったからです。

その一方、JSF 2 は、JSF 1 をベースに実際にオープンソース・プロジェクトを実装した開発者たちが経験した試練のなかから生まれました。この実際の経験による反省が、Ajax を利用した堅牢なアプリケーションを簡単に実装できる、遙かに実際的なフレームワークを生む結果となったわけです。

この連載をとおして、とりわけ優れた JSF 2 の機能として、構成に置き換わるアノテーションと規約、単純化されたナビゲーション、リソースのサポート、複合コンポーネント、組み込み Ajax、そして拡張イベント・モデルを紹介してきました。けれども JSF 2 にはその他にも豊富な機能が備わっています。この連載では取り上げませんでしたが、ビューおよびページのスコープ、ブックマーク可能なページのサポート、プロジェクト・ステージなどを含めたすべての機能が、JSF 2 にその前身を大きく上回る改善をもたらしています。

ダウンロード

内容	ファイル名	サイズ
Source code for the article examples	j-jsf2-fu-3.zip	7.7MB

著者について

David Geary

著者、講演者、コンサルタントとして活躍する David Geary は、Clarity Training, Inc. の社長です。彼はこの会社で、開発者に JSF および GWT (Google Web Toolkit) を使用した Web アプリケーションの実装を指導しています。JSTL 1.0 および JSF 1.0/2.0 Expert Group のメンバー、そして Sun の Web 開発者認定試験の共同制作者としての経験を持つ彼は、Apache Struts や Apache Shale などのオープンソース・プロジェクトにも貢献しています。彼の著書『Graphic Java Swing』は Java 関連の本のなかでは史上に残るベスト・セラーの 1 つで、『Core JSF』(Cay Horstman との共著) は JSF 関連のベストセラー本となっています。コンファレンスやユーザー・グループで定期的に講演を行っている他、2003年以来 NFJS ツアーの常連で、Java University では講座を受け持っています。彼は JavaOne ロック・スターに 2 回選ばれました。