JSF 2 の魅力: 複合コンポーネントのベスト・プラクティス

拡張可能なカスタム・コンポーネントを実装する

David Geary
President
Clarity Training, Inc.

2011年 1月 11日

連載「JSF 2 の魅力」の今回の記事では、Java™Server Faces の複合コンポーネントを実装する際の 5 つのベスト・プラクティスについて学びます。開発者はこれらのガイドラインに従うことで、ページ作成者でも簡単に拡張できるようなカスタム・コンポーネントを実装することができます。

このシリーズの他の記事を見る

この連載について

連載「JSF 2 の魅力」では、David Geary が JSF 2 について紹介する 3 回からなる同名の連載記事の続編として、皆さんが JSF 2 フレームワークのスキルを開発して磨きをかけ、カンフー・マスター並みにこのフレームワークの達人になるためのお手伝いをします。この連載では JSF 2 フレームワークとそれを取り巻くエコシステムの詳細を探るとともに、このフレームワークの外部にも目を向け、CDI (Contexts and Dependency Injection) などの Java EE 技術を JSF に統合する方法についても紹介します。

JSF はコンポーネント・ベースのフレームワークです。これは、開発者が独自のコンポーネントを実装するのに必要なインフラストラクチャーを提供することを意味します。さらに JSF 2 では、複合コンポーネントを用いて簡単にカスタム・コンポーネントを実装できるようになっています。

これまで、この連載ではいくつかの複合コンポーネントの実装を紹介してきました (「テンプレート機能と複合コンポーネント」、「Ajax components」、「後から追加する Ajax 複合コンポーネント」を参照)。今回は複合コンポーネントの実装というトピックのまとめとして (そして、連載「JSF 2 の魅力」のまとめとして)、JSF 2 で複合コンポーネントを実装する際の 5 つのベスト・プラクティスを紹介します。具体的には、以下のベスト・プラクティスです。

1. コンポーネントは `DIV` の中にラップすること
2. JavaScript と Ajax を組み込むこ
3. ページに含まれる複数のコンポーネントをサポートするには、JavaScript クロージャーを使用すること
4. ページの作成者がコンポーネントをカスタマイズできるようにすること

5. コンポーネントを国際化対応にすること

この 5 つのベスト・プラクティスを具体的に説明するために、1 つの単純な複合コンポーネントの実装にそれぞれのベスト・プラクティスをどのように適用するのかを見ていきます。

編集可能な入力用複合コンポーネント

この記事で使用するサンプル・コンポーネントは、編集可能な入力用の複合コンポーネントです。図 1 に示すアプリケーションは、2 つの編集可能な入力コンポーネントを使用しています。1 つはファーストネーム用、もう 1 つはラストネーム用です。

図 1. 編集可能なテキスト・コンポーネント

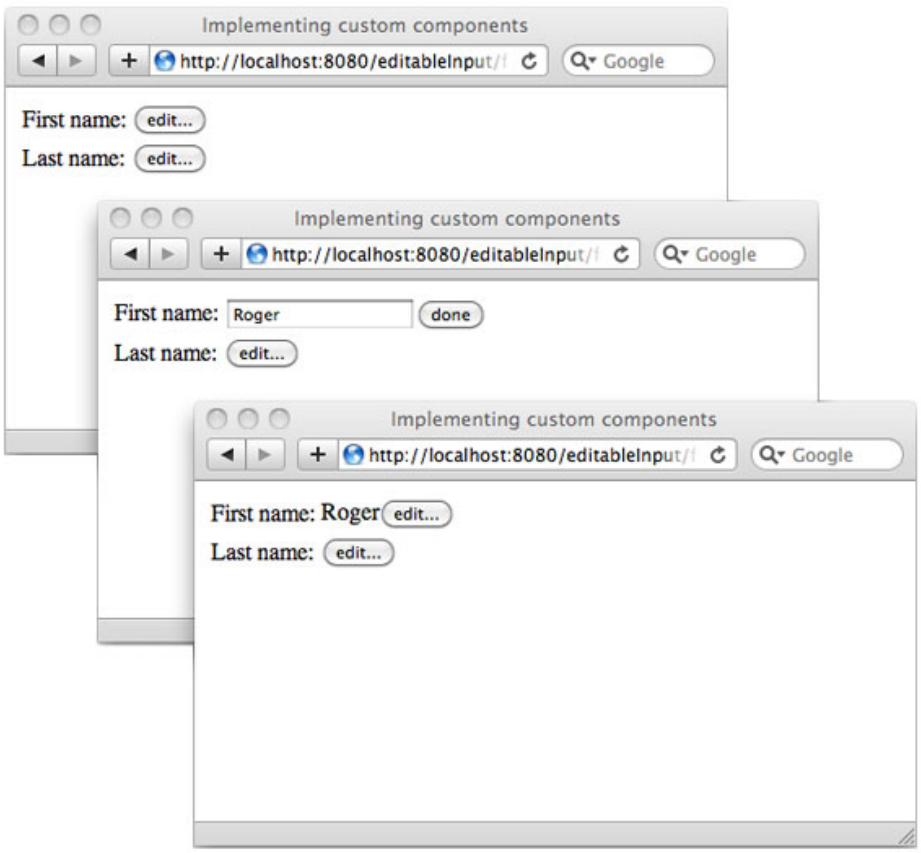


図 1 に示す 3 つの画面は、上から下の順に、ファーストネームを編集するシーケンスを示しています。

- 一番上のスクリーンショットは、このアプリケーションの初期画面です。画面には、「First name: (ファーストネーム:)」と「Last name: (ラストネーム:)」というラベルがあり、それぞれのラベルの右側には「edit... (編集…)」ボタンがあります。
- 中央のスクリーンショットは、ユーザーが「First name: (ファーストネーム:)」の隣にある「edit... (編集…)」ボタンをクリックした後、テキスト入力域に「`Roger`」という名前を入力したときの画面を示しています。テキスト入力域の右側には、「done (完了)」ボタンが表示されています。

- 一番下のスクリーンショットが示しているのは、ユーザーが「done (完了)」ボタンをクリックした後のアプリケーションの画面です。「First name: Roger (ファーストネーム: Roger)」と表示されている右側に「edit... (編集…)」ボタンがあります。

次のセクションでは、この編集可能な入力コンポーネントの使い方を説明し、それに続いてこのコンポーネントがどのように実装されているのかを解説します。その後、コンポーネントの実装に関する 5 つのベスト・プラクティスのそれぞれを説明します。

コンポーネントを使用する方法

この編集可能な入力コンポーネントを使用する方法は、他の JSF 複合コンポーネントを使用する場合のステップとまったく同じです。つまり、適切な名前空間を宣言し、JSF がこの複合コンポーネント用に生成したタグを使用します。図 1 に示したページに対応するマークアップを使った場合、この 2 つのステップはリスト 1 のようになります。

リスト 1. `<util:inputEditable>` を使用する

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:util="http://java.sun.com/jsf/composite/util">
   <h:head>
      <title>Implementing custom components</title>
   </h:head>
   <h:body>
      <h:form>
         <h:panelGrid columns="2">
            First name:
<util:inputEditable id="firstName"
                            value="#{user.firstName}"/>

            Last name:
<util:inputEditable id="lastName"
                            value="#{user.lastName}"/>
        </h:panelGrid>
      </h:form>
   </h:body>
</html>
```

完全を期して、リスト 2 にリスト 1 で参照されている `user` Bean の実装を記載します。

リスト 2. `user` Bean

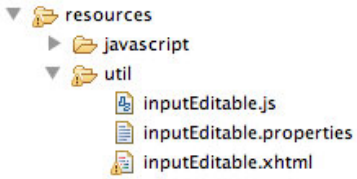
```
package com.corejsf;

import java.io.Serializable;
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;

@Named("user")
@SessionScoped
public class UserBean implements Serializable {
   private String firstName;
   private String lastName;

   public String getFirstName() { return firstName; }
   public void setFirstName(String newValue) { firstName = newValue; }

   public String getLastName() { return lastName; }
   public void setLastName(String newValue) { lastName = newValue; }
}
```

編集可能な入力コンポーネントの使い方は以上のとおりです。次は、このコンポーネントがどのように実装されているのかを解説します。

コンポーネントの実装

編集可能な入力コンポーネントは、resources/util ディレクトリー内の inputEditable.js、inputEditable.properties、および inputEditable.xhtml ファイルに実装されています。図 2 に、ファイルシステムの階層を示します。

図 2. サンプル・コンポーネントのファイル構成



inputEditable.xhtml ファイルの内容は、リスト 3 のとおりです。

リスト 3. `inputEditable` コンポーネントのマークアップ (inputEditable.xhtml)

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

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

    <composite:interface>
       <composite:attribute name="text"/>
       <composite:editableValueHolder name="text" targets="editableText" />
       <composite:actionSource name="editButton" targets="editButton" />
       <composite:actionSource name="doneButton" targets="doneButton" />
       <composite:clientBehavior name="edit" event="action" targets="editButton"/>
       <composite:clientBehavior name="done" event="action" targets="doneButton"/>
       <composite:facet name="textMessage"/>
    </composite:interface>

    <composite:implementation>
      <h:outputScript library="javascript" name="prototype.js" target="head"/>
      <h:outputScript library="javascript" name="scriptaculous.js" target="head"/>
      <h:outputScript library="javascript" name="effects.js" target="head"/>
      <h:outputScript library="util" name="inputEditable.js" target="head"/>

      <div id="#{cc.clientId}">
         <h:outputText id="text" value="#{cc.attrs.text}"/>
         <h:commandButton id="editButton" type="button"
            value="#{cc.resourceBundleMap.editButtonText}"
            onclick="this.startEditing()"/>

         <h:inputText id="editableText" value="#{cc.attrs.text}" style="display: none"/>

         <h:commandButton id="doneButton"
            value="#{cc.resourceBundleMap.doneButtonText}" style="display: none">

             <f:ajax render="text textMessage" execute="editableText"
               onevent="ajaxExecuting"/>
```

```
         </h:commandButton>

         <h:panelGroup id="textMessage">
            <composite:renderFacet name="textMessage"/>
         </h:panelGroup>
      </div>

      <script> com.clarity.init('#{cc.clientId}'); </script>

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

このマークアップは 4 つのコンポーネントを作成しますが、そのうち初期画面で表示されるのは、テキストと編集ボタンの 2 つだけです。ユーザーが編集ボタンをクリックすると、アプリケーションが JavaScript 関数 `com.clarity.startEditing()` を呼び出します。この関数は、リスト 4 に実装されています。

リスト 4. **`inputEditable`** コンポーネントの JavaScript (inputEditable.js)

```
package com.corejsf;
   var com = {};

if (!com.clarity) {
   com.clarity = {
init: function (ccid) {
         var mydiv = document.getElementById(ccid);
         mydiv.editButton = $(mydiv.id + ':editButton');
         mydiv.text = $(mydiv.id + ':text');
         mydiv.editableText = $(mydiv.id + ':editableText');
         mydiv.doneButton = $(mydiv.id + ':doneButton');
         mydiv.doneButton.offsetLeft = mydiv.editButton.offsetLeft;

         mydiv.editButton.startEditing = function() {
           mydiv.text.fade( { duration: 0.25 } );
           mydiv.editButton.fade( { duration: 0.25 } );

           window.setTimeout( function() {
              mydiv.editableText.appear( { duration: 0.25 } );
              mydiv.doneButton.appear( { duration: 0.25 } );

               window.setTimeout( function() {
                  mydiv.editableText.focus();
               }, 300);
            }, 300);
         };
      },

toggleDisplay: function(element) {
         element.style.display = element.style.display == "none" ? "" : "none";
      },

ajaxExecuting: function(data) {
         var mydiv = $(data.source.parentNode);

         if (data.status == 'complete') {
            toggleDisplay(mydiv.editableText);
            toggleDisplay(mydiv.doneButton);
            toggleDisplay(mydiv.text);
            toggleDisplay(mydiv.editButton);
         }
      }
   }
}
```

`startEditing()` 関数は、Scriptaculous フレームワーク (「参考文献」を参照) の `fade()` および `appear()` メソッドを使用します。また、フェードと表示が正しい順序で行われるように 2 つのタイマーを使用します。さらに、`startEditing()` 関数は最終的にフォーカスをテキスト入力に渡す点にも注目してください。

リスト 5 に、inputEditable.properties ファイルの内容を記載します。

リスト 5. `inputEditable` コンポーネントのプロパティー・ファイル (inputEditable.properties)

```
editButtonText=edit...
doneButtonText=done
```

ここからは、この編集可能な入力コンポーネントの実装について、複合コンポーネントを実装する際の 5 つのベスト・プラクティスの観点から説明します。

コンポーネントは div の中にラップすること

JSF は複合コンポーネントを作成するときに、ネーミング・コンテナーと呼ばれるものを作成します。ここに、複合コンポーネントに含まれるすべてのコンポーネントが格納されます。ただし、マークアップを生成するのはネーミング・コンテナーではありません。代わりに JSF が、複合コンポーネント内のコンポーネントごとにマークアップを生成します。その結果、デフォルトで複合コンポーネントにはコンポーネント ID が付けられないため、ページのマークアップはコンポーネント ID を使って複合コンポーネントを参照することができません。

ページ作成者が複合コンポーネントを参照できるようにするには、コンポーネントを実装するときに、`div` の中にコンポーネントをラップする必要があります。例えば、Ajax 呼び出しの一環として、ページのマークアップにこの編集可能な入力コンポーネントを参照させたいとします。それには、以下に示すように、マークアップにファーストネームの入力を処理する Ajax ボタンを追加します。このボタンをクリックするとサーバーに対して Ajax 呼び出しが行われ、サーバーでファーストネームの入力が処理されます。そして Ajax 呼び出しに対するレスポンスが返されると、JSF が入力をレンダリングします。

```
<h:form>
   <h:panelGrid columns="2">
      First name:
      <util:inputEditable id="firstName"
                      value="#{user.firstName}"/>

      Last name:
      <util:inputEditable id="lastName"
                      value="#{user.lastName}"/>

<h:commandButton value="Update first name">
         <f:ajax execute="firstName" render="firstName">
      </h:commandButton>
  </h:panelGrid>
</h:form>
```

上記のマークアップに追加された Ajax ボタンが機能する理由は、リスト 3 では以下のように、コンポーネントが `div` の中にラップされているためです。

```
<div id="#{cc.clientId}">

   ...

</div>
```

div に使用される ID は、複合コンポーネント自体のクライアント ID です。したがって、複合コンポーネントを `div` に含めることで、上記の例で説明した Ajax ボタンと同じく、ページ作成者は複合コンポーネントを 1 つのコンポーネントとして参照できるようになるというわけです。

JavaScript と Ajax を組み込むこと

この記事に示すスクリーンショットでは静的なためわかりませんが、ユーザーが「edit... (編集…)」ボタンをクリックすると、この編集可能な入力コンポーネントはフェード・アニメーションを実行します。フェードを行うのは結局のところ、Scriptaculous フレームワークです。リスト 3 では `<h:outputScript>` タグを使って Scriptaculous に必要な JavaScript を出力し、リスト 4 で Scriptaculous フレームワークの `fade()` および `appear()` メソッドを使用して目的のアニメーションを実現しています。

```
mydiv.editButton.startEditing = function() {
  mydiv.text.fade( { duration: 0.25 } );
  mydiv.editButton.fade( { duration: 0.25 } );

  window.setTimeout( function() {
     mydiv.editableText.appear( { duration: 0.25 } );
     mydiv.doneButton.appear( { duration: 0.25 } );

      window.setTimeout( function() {
         mydiv.editableText.focus();
      }, 300);
   }, 300);
};
```

上記の JavaScript にネストされているタイマーが、アニメーションのすべての動作がタイミング良く行われることを確実にします。例えば、入力が画面上に完全に表示されるまで、入力テキストにフォーカスが移りません。このようにしなければ、入力が表示される前に `focus()` を呼び出しても、この呼び出しは入力が表示されるまで待機しないことになってしまいます。

Scriptaculous や JQuery などのサード・パーティーの JavaScript フレームワークを JSF で使用するのは難しいことではありません。ページに適切な JavaScript を出力した後に、JavaScript コードで目的のフレームワークを使用すればよいのです。

この編集可能な入力コンポーネントはさらに、Ajax も使用します。ユーザーが「done (完了)」ボタンをクリックしたときには、Ajax を使用してサーバーへの呼び出しを行います。

```
<h:commandButton id="doneButton"
   value="#{cc.resourceBundleMap.doneButtonText}" style="display: none">

   <f:ajax render="text textMessage" execute="editableText"
      onevent="ajaxExecuting"/>

</h:commandButton>
```

上記のマークアップでは、ユーザーが「done (完了)」ボタンをクリックした時点で Ajax 呼び出しを行うために JSF の `<f:ajax>` タグを使用していました。この Ajax 呼び出しは、入力されたテキストをサーバーに送信してサーバー上で実行させます。Ajax 呼び出しに対するレスポンスが返されると、テキストとテキスト・メッセージが更新されます。

JavaScript クロージャーを使用すること

複合コンポーネントを実装する際には、1 つのページに複数のコンポーネントがあることを考慮しなければなりません。1 つのコンポーネントのすべてのインスタンスが同じ JavaScript を共有する場合、ユーザーが現在対話しているコンポーネントだけに操作の対象を制限するように注意する必要があります。

1 つのページで複数のコンポーネントをサポートするには、いくつかの方法があります。その 1 つは、Oracle のエンジニアである Jim Driscoll が、この記事のサンプルと同様の編集可能な入力コンポーネントについてのブログ・エントリーで説明しているように、コンポーネント ID の名前空間を使用するという方法です (「参考文献」を参照)。それとは別の方法として、JavaScript クロージャーを使用することもできます。

```
com.clarity = {
   init: function (ccid) {
      var mydiv = document.getElementById(ccid);
      mydiv.editButton = $(mydiv.id + ':editButton');
      mydiv.text = $(mydiv.id + ':text');
      mydiv.editableText = $(mydiv.id + ':editableText');
      mydiv.doneButton = $(mydiv.id + ':doneButton');
      mydiv.doneButton.offsetLeft = mydiv.editButton.offsetLeft;

      mydiv.editButton.startEditing = function() {
        mydiv.text.fade( { duration: 0.25 } );
        mydiv.editButton.fade( { duration: 0.25 } );

        window.setTimeout( function() {
           mydiv.editableText.appear( { duration: 0.25 } );
           mydiv.doneButton.appear( { duration: 0.25 } );

            window.setTimeout( function() {
               mydiv.editableText.focus();
            }, 300);
         }, 300);
      };
   },
```

上記は、リスト 4 に記載されている、編集可能な入力コンポーネントの JavaScript を抜粋したものです。リスト 3 の終わりを見るとわかるように、`init()` 関数はすべての編集可能な入力コンポーネントに対して呼び出されます。コンポーネントをラップしている `div` のクライアント ID を指定すれば、その特定の `div` への参照を取得することができます。さらに、JavaScript は動的言語であり、実行時にプロパティーとメソッドをオブジェクトに追加できることから、後で `div` 自体へのコールバック関数で必要となるすべての要素への参照を追加してあります。

コンポーネントの編集ボタンには、さらに `startEditing()` 関数を追加します。ユーザーが「edit... (編集…)」ボタンをクリックしたときに、この関数が呼び出されるようにするためです。

```
<h:commandButton id="editButton" type="button"
   value="#{cc.resourceBundleMap.editButtonText}"
   onclick="this.startEditing()"/>
```

`startEditing()` 関数が呼び出される際に、`mydiv` 変数は `init()` 関数が呼び出された時点での値を保持しています。これが、JavaScript クロージャーの賢いところです (そして Java の内部クラスがいくらか賢いところでもあります)。`init()` が呼び出されてから `startEditing()` が呼び出されるまでにどれだけの時間が経過していようと、また、その間に `init()` が何度も呼び出されたとしても関係ありません。`startEditing()` が呼び出されたときにこの関数に含まれる `mydiv` の値は、まさにそのコンポーネントの `init()` 関数が呼び出された時点での値そのものなのです。

このように、JavaScript クロージャーは内部の関数に含まれる変数の値を保持するため、JavaScript クロージャーを使用することで、それぞれの `startEditing()` 関数がそのコンポーネントの該当する `div` に確実にアクセスすることができます。

ページ作成者がカスタマイズできるようにすること

通常は、コンポーネント定義の中に 1 行か 2 行の XML を記述するだけで、ページ作成者がコンポーネントをカスタマイズできるようになります。複合コンポーネントをカスタマイズする主な方法には、以下の 3 つがあります。

- バリデーター、コンバーター、およびリスナーを追加する
- ファセットを使用する
- Ajax を使用する

バリデーター、コンバーター、およびリスナー

ページ作成者が複合コンポーネントに含まれるコンポーネントにバリデーター、コンバーター、リスナーを追加できるようにするには、これらの内部コンポーネントを公開するようにします。例えば、図 3 の編集可能な入力コンポーネントには検証機能が追加されています。

図 3. ファーストネーム・フィールドの検証

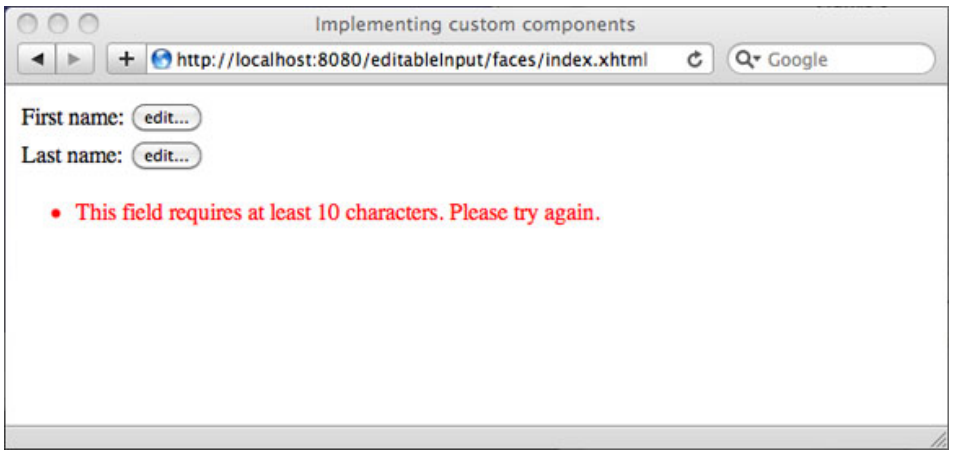


図 3 に表示されているエラー・メッセージは、フィールドに入力する文字は 10 文字以上でなければならないと通知しています。このページ作成者がファーストネームの編集可能な入力コンポーネントのテキストに追加したのは、以下のようなバリデーターです。

```
<util:inputEditable id="firstname" text="#{user.firstName}">
   <f:validateLength minimum="10" for="text"/>
</util:inputEditable>
```

<f:validateLength> タグの for 属性に注目してください。この属性は JSF に対し、バリデーターの対象が、編集可能な入力コンポーネント内部のテキストであると指示しています。JSF が該当するテキストを認識できるのは、私が複合コンポーネントの実装で、この入力コンポーネントを公開しているからです。

```
<composite:interface>
   ...
   <composite:editableValueHolder name="text" targets="editableText" />
   ...
</composite:interface>

<composite:implementation>
   ...
   <h:inputText id="editableText" value="#{cc.attrs.text}" style="display: none"/>
   ...
</composite:implementation>
```

ファセット

図 3 では、エラー・メッセージがページ下部に表示されています。なぜこの位置に表示されるかと言うと、私が使用しているのはプロジェクト・ステージが開発ステージのものなので、このステージでは JSF が自動的に検証エラーを下部に追加するからです。しかし、これではエラーがどちらの編集可能な入力コンポーネントに関連しているのかがわかりません。このようにするよりも、問題となっているコンポーネントの隣にエラー・メッセージを表示するほうが親切です (図 4 を参照)。

図 4. ファセットの使用

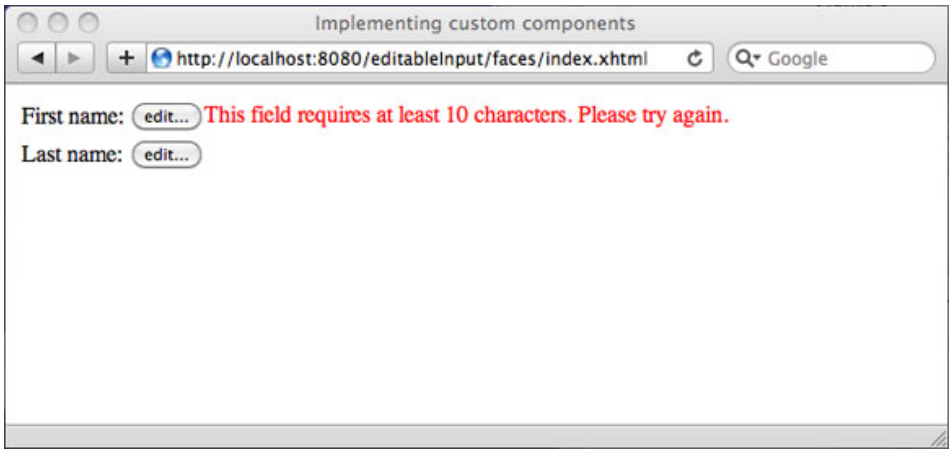


図 4 のようにするには、ページ作成者は以下のように、当該コンポーネントにファセットを追加します。

```
<util:inputEditable id="firstname" text="#{user.firstName}">
   <f:validateLength minimum="10" for="text"/>
   <f:facet name="textMessage">
      <h:message for="editableText" style="color: red"/>
   </f:facet>
</util:inputEditable>
```

コンポーネントはこのファセットを、以下のようにしてサポートします。

```
<composite:interface>
   <composite:facet name="textMessage"/>
</composite:interface>

<div id="#{cc.clientId}">
   <h:panelGroup id="textMessage">
      <composite:renderFacet name="textMessage"/>
   </h:panelGroup>
   ...
</div>
```

Ajax

「後から追加する Ajax 複合コンポーネント」でも説明したように、ページ作成者が複合コンポーネントに Ajax 機能を追加できるようにするには、<composite:clientBehavior> タグを使用します。図 5 に示すダイアログ・ボックスは、ユーザーが「edit... (編集…)」ボタンをクリックすると行われる Ajax 呼び出しをモニターします。

図 5. Ajax リクエストのモニター

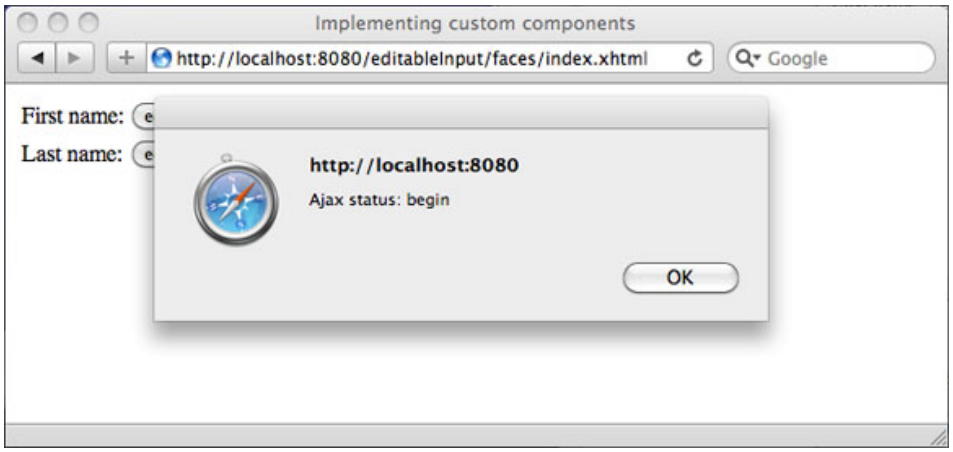


図 5 のようにするには、ページ作成者は以下のように、`<f:ajax>` タグをコンポーネントに追加して、Ajax 関数を指定します。そして、`<f:ajax>` タグの `onevent` 属性を Ajax 呼び出しの処理中に呼び出す JavaScript 関数 (ダイアログ・ボックスを表示する関数) に設定すればよいのです。

```
<util:inputEditable id="firstname" text="#{user.firstName}">
   <f:validateLength minimum="10" for="text"/>
   <f:facet name="textMessage">
      <h:message for="editableText" style="color: red"/>
   </f:facet>

<f:ajax event="edit" onevent="monitorAjax"/>
</util:inputEditable>
```

ページ作成者が `<f:ajax>` タグをコンポーネントに追加できる理由は、私がそのクライアントの振る舞いをコンポーネントの実装ですでに公開しているからです。

```
<composite:interface>
   <composite:clientBehavior name="edit" event="action" targets="editButton"/>
</composite:interface>
```

国際化対応にすること

編集可能な入力コンポーネントは、2 つのボタンにテキスト (「edit... (編集…)」と「done (完了)」) を表示します。コンポーネントが複数のロケールで使用される場合には、テキストを国際化対応にして、ローカライズする必要があります。

コンポーネントのテキストをローカライズするには、コンポーネントと同じディレクトリー内にプロパティー・ファイルを追加し、そのプロパティー・ファイルのキーに、`#{cc.resourceBundleMap.KEY}` (`KEY` は、プロパティー・ファイルのキー) という表現でアクセスすればよいのです。リスト 3 とリスト 5 を見るとわかるように、編集可能な入力コンポーネントのボタンは、この方法でローカライズしています。

まとめ

JSF 1 はコンポーネントの実装を難しくしていたことから、ほとんどの JSF 開発者はコンポーネントを実装するのを控えていました。けれども JSF 2 により、もはやカスタム・コンポーネントはカスタム・コンポーネント・フレームワーク開発者だけの領域ではなくなっています。この記事では、複合コンポーネントを実装する際のベスト・プラクティスを取り上げました。この記事で説明したほんのわずかな作業で、ページ作成者でも簡単に拡張できるコンポーネントを作成することができます。

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『グラフィック Java2 〈Vol.2〉 Swing 編』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、これまでに 3 回 Java University の講師になり、JavaOne ロック・スターに 2 回選ばれました。